

Патер Сейбен



КОДЕРЫ ЗА РАБОТОЙ

РАЗМЫШЛЕНИЯ
О РЕМЕСЛЕ ПРОГРАММИСТА

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-188-2, название «Кодеры за работой. Размышления о ремесле программиста». Идеальная фотография со вспышкой» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Coders at Work

Reflections on the Craft
of Programming

Peter Seibel

Apress®

ПРОФЕССИОНАЛЬНО

Кодеры за работой

Размышления
о ремесле программиста

Питер Сейбел



Санкт-Петербург — Москва
2011

Серия «Профессионально»

Питер Сейбел

Кодеры за работой

Размышления о ремесле программиста

Перевод А. Коробейникова и В. Петрова

Главный редактор

А. Галунов

Зав. редакцией

Н. Макарова

Выпускающий редактор

П. Щеголев

Научные редакторы

И. Сагалаев,

С. Тепляков

Редактор

Т. Темкина

Корректор

О. Макарова

Верстка

К. Чубаров

Сейбел П.

Кодеры за работой. Размышления о ремесле программиста. – Пер. с англ. – СПб: Символ-Плюс, 2011. – 544 с., ил.

ISBN 978-5-93286-188-2

Программисты – люди не очень публичные, многие работают поодиночке или в небольших группах. Причем самая важная и интересная часть их работы никому не видна, потому что происходит у них в голове. Питер Сейбел, писатель-программист, снимает покров таинственности с этой профессии. Он взял интервью у 15 величайших профессионалов: Кена Томпсона, создателя UNIX, Берни Козелла, участника первой реализации сети ARPANET, Дональда Кнута, Гая Стила, Саймона Пейтон-Джонса, Питера Норвига, Джошуа Блоха, Брэда Фицпатрика, создателя Живого Журнала, и других. Все они «подсели» на программирование еще в школе. Тогда, на заре зарождения отрасли, лишь в немногих учебных заведениях читались курсы по компьютерным наукам. Поэтому будущим гуру приходилось покорять профессиональные вершины самостоятельно, но всех их отличает творческое горение и полная самоотдача любимому делу.

Вы узнаете, что они думают о будущем программирования и как сами научились программировать, как, по их мнению, нужно проектировать ПО, как выбор языка программирования влияет на продуктивность и можно ли облегчить выявление труднонаходимых ошибок.

ISBN 978-5-93286-188-2

ISBN 978-1-4302-1948-4 (англ)

© Издательство Символ-Плюс, 2011

Authorized translation of the English edition © 2009 Apress Inc. This translation is published and sold by permission of Apress Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 15.02.2011. Формат 70×90 1/16. Печать офсетная.

Объем 34 печ. л. Тираж 1500 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»

199034, Санкт-Петербург, 9 линия, 12.

Амелии

Оглавление

Об авторе	8
Благодарности	9
Введение	11
Глава 1. Джейми Завински	15
Глава 2. Брэд Фицпатрик	55
Глава 3. Дуглас Крокфорд	93
Глава 4. Брендан Аик	129
Глава 5. Джошуа Блох	159
Глава 6. Джо Армстронг	191
Глава 7. Саймон Пейтон-Джонс	219
Глава 8. Питер Норвиг	255
Глава 9. Гай Стил	283
Глава 10. Дэн Ингаллс	321
Глава 11. Питер Дойч	357
Глава 12. Кен Томпсон	391
Глава 13. Фрэн Аллен	421
Глава 14. Берни Козелл	451
Глава 15. Дональд Кнут	491
Библиография	526
Алфавитный указатель	529

Об авторе

Питер Сейбел – писатель-программист или программист-писатель. Получив высшее филологическое образование и какое-то время проработав журналистом, пленился Сетью. В начале 1990-х программировал на Perl для журнала «Mother Jones» и портала Organic Online. Участвовал в революции Java как сотрудник WebLogic, позже преподавал программирование на Java на заочных курсах при Калифорнийском университете Беркли. В 2003 году оставил работу архитектора транзакционной системы сообщений, основанной на Java, планируя за год освоить язык Лисп. Вместо этого два года писал книгу «Practical Common Lisp» (Common Lisp на практике), получившую премию Jolt Productivity Award. С тех пор работает «главной обезьяной» Gigamonkeys Consulting: учится учить, заниматься тай-цзи и быть отцом. Проживает в Беркли (Калифорния) с женой Лили, дочерью Амелией и собакой Мелани.

Благодарности

Прежде всего я хотел бы поблагодарить всех своих собеседников, которые любезно согласились потратить свое время и без которых эта книга была бы просто брошюрой с вопросами без ответов. Кроме того, я благодарен Джо Армстронгу и Берни Козеллу и их семьям за то, что предложили мне остановиться у них в Стокгольме и Виргинии. Еще одна особая благодарность – Питеру Норвигу и Джейми Завински, которые не только наговорили собственные интервью на мой диктофон, но и помогли мне связаться с другими будущими героями этих интервью.

Пока я путешествовал по всему свету, проводя интервью, меня приютили еще несколько семей: благодарю за гостеприимство Дэна Уэйнреба и Черил Моро (Бостон), Гарета и Эмму Маккоуэнов (Кембридж, Англия), а также моих родителей, которые предоставили мне отличный плацдарм для действий в Нью-Йорке. Кристофер Родс помог мне заполнить паузу между интервью экскурсией по Кембриджскому университету; благодаря ему и Дэйву Фоксу тот вечер закончился обедом и походом по кембриджским пабам.

Дэн Уэйнреб не только приютил меня в Бостоне, но и был самым усердным рецензентом на протяжении всей работы над этой книгой с той самой поры, когда я только выбирал потенциальных собеседников. Зак Бин, Люк Горри, Дэйв Уолден и моя мама тоже читали рукопись и своевременно меня подбадривали. Зак к тому же – что для моих книг уже традиция – является автором некоторых слов на обложке, на этот раз подзаголовка книги. Алан Кэй внес замечательное предложение включить Дэна Ингаллса и Питера Дойча. Скотт Фальман рассказал мне много полезного о начале карьеры Джейми Завински, а Дэйв Уолден прислал материалы по истории корпорации BBN Technologies, чтобы я смог подготовиться к интервью с Берни Козеллом. Если же я кого-то забыл, примите мою благодарность и вместе с нею извинения.

Спасибо издательству Apress, особенно Гари Корнеллу, который и предложил написать эту книгу, Джону Вакка и Майклу Бэнксу за их предложения, а также моему редактору Кэндейс Инглиш, которая исправила бесчисленные ошибки.

Наконец, самые искренние слова благодарности моей семье – и большой, и малой. Обе мои мамы – родная и теща – приходили присмотреть за ребенком, чтобы я мог еще немного поработать; мои родители на неделю приютили моих жену и малышку, позволив мне сделать последний мощный рывок. И главная благодарность – жене и дочери. Лили и Амелия, хотя иногда мне требуется некоторое время на работу, без вас, девочки, она не имела бы смысла. Я люблю вас.

Введение

Если не считать работу, проделанную Адой Лавлейс – аристократкой XIX века, которая придумала программы для так и не законченной «Аналитической машины» Чарльза Бэббиджа, – компьютерное программирование как область человеческой деятельности появилось совсем недавно: Конрад Цузе представил свой электромеханический компьютер Z3, первую работающую вычислительную машину общего назначения, в 1941 году, всего 68 лет назад. И всего 64 года прошло с тех пор, как шесть женщин – Кей Антонелли, Джин Бартик, Бетти Холбертон, Мартин Мельцер, Фрэнсис Спенс и Рут Тейтельбаум, – служивших в американских «вычислительных войсках» и составлявших вручную баллистические таблицы, были привлечены к созданию программ для ЭНИАК – первого электронного цифрового компьютера общего назначения. Среди ныне живущих многие – старшие представители поколения «бэби-бума»¹ и все родители «бэби-бумеров» – увидели свет, когда в мире не было ни одного программиста.

Теперь, разумеется, все иначе. Программисты заполонили все вокруг. Согласно данным американского Бюро трудовой занятости, в 2008 году в США примерно один из каждых 106 работников – всего более 1,25 млн человек – был разработчиком программного обеспечения или инженером-программистом. Это не считая профессиональных программистов за пределами США, студентов и программистов-любителей, а также тех, кто официально занимается чем-то другим, но тратит сколько-то времени – порой даже много времени – на то, чтобы подчинить компьютер своей воле.

И хотя написанием программ занимались и занимаются миллионы человек, хотя миллиарды, даже триллионы строк кода уже написаны, кажется, будто само понятие «программист» непрерывно уточняется с течением времени. Все еще идут споры о том, к какой области относится программирование – к математике или к инженерной деятельности. Ремесло, искусство или наука? Конечно же, идут споры, зачастую оже-

¹ Поколение «бэби-бума» (Baby Boom Generation) – жители Соединенных Штатов, рожденные в 1945–1964 гг. Окончание Второй мировой войны ознаменовалось ростом рождаемости, в результате чего на свет появилось почти 80 млн человек. – *Прим. науч. ред.*

сточенные, по поводу лучших способов программирования: Интернет полон сообщений в блогах и форумах, посвященных тому или иному способу написания кода. Книжные магазины набиты книгами о новых языках программирования, новых методах, новых попытках осмыслить задачи программирования.

Эта книга предлагает читателю нестандартный подход к понятию программирования. Она следует традиции, заложенной журналом «Paris Review», когда-то пославшим двух преподавателей проинтервьюировать романиста Э. М. Форстера; впоследствии все подобные интервью были объединены в сборник «Writers at Work» (Писатели за работой).

Я беседовал с пятнадцатью в полной мере состоявшимися программистами, имеющими большой опыт работы: с такими гениями кода, как Кен Томпсон, создатель операционной системы UNIX, и Берни Козелл, участник первой реализации сети ARPANET; с программистами, сочетающими выдающиеся академические заслуги с невероятными практическими навыками, – Дональдом Кнутом, Гаем Стилом и Саймоном Пейтон-Джонсом; промышленными исследователями – Фрэн Аллен из IBM, Джо Армстронгом из Ericsson, Питером Норвигом из Google; с выпускниками научно-исследовательского центра Херох PARC Дэном Ингаллсом и Л. Питером Дойчем; с разработчиками ранних версий Netscape Джейми Завински и Бренданом Айком; с участниками проектирования и реализации языков программирования, применяемых сегодня для построения веб-приложений, – тем же Айком, Дугласом Крокфордом и Джошуа Блохом; и наконец с Брэдом Фицпатриком, создателем Живого Журнала (Live Journal) и одним из способнейших программистов эпохи Интернета.

Я расспрашивал каждого из них о программировании: как они научились этому, что открыли, создавая программы, и что думают о будущем программирования. В частности, я старался направить разговор на темы, вечно актуальные для программистов: как нужно проектировать программное обеспечение? как выбор языка программирования влияет на продуктивность и позволяет избегать ошибок? можно ли облегчить выявление труднонаходимых ошибок?

Поскольку однозначные решения всех этих проблем пока не найдены, неудивительно, что ответы оказались довольно разными. Джейми Завински и Дэн Ингаллс подчеркивали важность запуска кода сразу после его написания, а Джошуа Блох говорил о проектировании API и тестов, помогающих кодировать до этапа реализации. Что касается Дональда Кнута, то он поведал, что сделал эскиз полной версии системы компьютерной верстки TeX еще до того, как написал первую строку кода. А вот разные мнения относительно языка Си. Фрэн Аллен считает, что язык Си – причина снижения интереса к компьютерным наукам в последние

два десятилетия, а Берни Козелл называет Си «самой большой угрозой для безопасности современных компьютеров». В то же время Кен Томпсон утверждает, что угрозы для безопасности создают сами программисты, а не языки, а Дональд Кнут считает указатели в Си «одним из самых потрясающих усовершенствований в нотации», виденных им. Некоторые из моих собеседников высмеивали утверждение, будто формальные доказательства корректности помогают улучшить качество программ, но Гай Стил прекрасно проиллюстрировал их силу и в то же время ограниченность возможностей.

Кое с чем, однако, согласны все. Так, почти каждый настаивает на важности написания хорошо читаемого кода; большинство считают самыми трудноуловимыми ошибки в коде с параллельными вычислениями; никто не думает, что все проблемы программирования решены, — многие все еще ищут новые пути к разработке программного обеспечения с помощью автоматического анализа кода, улучшения организации совместной работы программистов или путем использования (или создания) более эффективных языков программирования. Однако практически все согласны, что вездесущие многоядерные процессоры серьезно повлияют на процесс разработки программ.

Эти разговоры отражают состояние данной области на определенном этапе, и, конечно, отдельные вопросы, затронутые в книге, с течением времени станут представлять лишь исторический интерес. Но даже в такой молодой области, как программирование, история сможет многому нас научить. Кроме того, думаю, все поднятые в книге темы раскрывают глубинный смысл того, что же такое разработка программного обеспечения, как мы можем лучше создавать его и что будет полезно программисту сегодня и через несколько поколений.

В заключение о названии книги: мы назвали ее «Кодеры за работой» по аналогии с уже упомянутой серией «Писатели за работой» журнала «Paris Review» и книгой «Founders at Work» (Учредители за работой) издательства Apress, которая применяет к основанию технологической компании примерно тот же подход, какой эта книга пытается применить к компьютерному программированию.

Я понимаю, что «кодирование» — лишь часть более широкого понятия «программирование». И всегда верил, что нельзя быть хорошим кодером, не являясь при этом хорошим программистом, или быть хорошим программистом, не являясь хорошим проектировщиком, общительным и думающим человеком. Разумеется, на страницах книги поднимаются все эти, а также многие другие вопросы. Не сомневаюсь, что обсуждения, которые вы собираетесь прочитать, прекрасно это отражают. Приятного чтения!

1

Джейми Завински

Лисп-хакер¹, один из первых разработчиков Netscape и владелец ночного клуба, Джейми Завински (Jamie Zawinski, JWZ) принадлежит к той избранной группе хакеров, которых узнают как по настоящим именам, так и по трехбуквенным инициалам.

Завински начал программировать еще подростком – поступил в лабораторию искусственного интеллекта при университете Карнеги–Меллона в качестве программиста на языке Лисп. Пробыв в колледже ровно столько, чтобы его возненавидеть, он около десяти лет проработал в мире Лиспа и искусственного интеллекта, удивительным образом погрузившись в умирающую хакерскую субкультуру, пока другие программисты, его сверстники, вырастали на микрокомпьютерах.

Он работал в Калифорнийском университете в Беркли с Питером Норвигом, который описал его как «одного из лучших программистов, которых он когда-либо нанимал», а затем в Лисп-компании Lucid, где участвовал в разработке редактора Lucid Emacs, позже переименованного в XEmacs, которая в итоге пришла к большому расколу проекта Emacs – одному из самых известных в истории свободного ПО.

¹ Хакер – ИТ-специалист высшей квалификации, в совершенстве изучивший работу компьютерных систем и ПО. Хакерами часто называют взломщиков программ, что в общем случае неверно. – *Прим. науч. ред.*

В 1994 году он наконец покинул компанию Lucid и мир Лиспа и перешел в Netscape, где стал одним из первых разработчиков UNIX-версии браузера Netscape Navigator, а позже – почтового клиента Netscape.

В 1998 году Завински, как и Брендан Айк, стал одним из основателей mozilla.org – организации, которая занималась переводом браузера Netscape в проект с открытым исходным кодом. Однако год спустя, разочарованный в отсутствии прогресса на пути к выпуску продукта, он покинул проект и приобрел ночной клуб DNA Lounge в Сан-Франциско, которым владеет до сих пор. В настоящее время Завински отдает все свои силы борьбе с Калифорнийским управлением по контролю за потреблением алкоголя, рассчитывая превратить клуб в место встречи любителей живой музыки всех возрастов.

В этом интервью мы среди прочего говорим о том, почему C++ – отвратительный язык программирования, о радости от того, что миллионы людей используют созданные тобой программы, а также о том, как важно «пинать» подающих надежды программистов.

Сейбел: Как вы научились программировать?

Завински: О, это было так давно, что я едва помню. Впервые я использовал компьютер для программирования, наверное, в восьмом классе. У нас было несколько компьютеров TRS-80, и мы дурачились с Бейсиком. По-моему, это вообще были внеклассные занятия. Помнится, невозможно было даже сохранять программы, так что мы просто вводили те, что печатались в журналах и так далее. Затем я прочел стопку книг. Помню, я читал книги по языкам, которые не мог использовать на наших компьютерах, и писал на бумаге программы на языках, о которых только прочитал.

Сейбел: Что это были за языки?

Завински: Один из них – APL. Я прочел статью про него и подумал, что это классный язык.

Сейбел: Ну да, не нужно иметь навороченную клавиатуру.¹ А в старших классах были компьютерные уроки?

¹ Язык программирования APL отличается очень короткой нотацией (большинство операций обозначаются одним-двумя символами), что делает программы крайне непонятными для чтения, но достаточно простыми для записи. – *Прим. науч. ред.*

Завински: В старших классах мы изучали Фортран. И все.

Сейбел: Как же вы пришли к Лиспу?

Завински: Я читал много фантастики. Мне казалось, что искусственный интеллект – это действительно классно, что компьютеры скоро покорят мир. И я решил почитать что-нибудь еще на эту тему. У меня в старших классах был приятель, Дэн Зигмонд, мы обменивались книгами и поэтомы оба выучили Лисп. Однажды он пошел на встречу группы пользователей компьютеров Apple в Университете Карнеги–Меллон – там можно было просто обменяться программами, и Дэн надеялся добыть кое-что бесплатно. Он разговорился с каким-то студентом колледжа, и тот сказал: «Ого, пятнадцатилетний парень, который знает Лисп, – это фантастика. Спроси Скотта Фальмана насчет работы». Дэн так и поступил. Фальман взял его на работу. Тогда Дэн сказал: «Возьмите и моего друга», – то есть меня. Так что в итоге мы оба оказались у Фальмана. Думаю, он размышлял примерно так: «Раз уж эти школьники интересуются такими штуками, не будет большого вреда, если они поболтаются в лаборатории». Нам поручили самую простую грязную работу – надо было что-то перекомпилировать, когда выходила новая версия компилятора, и попробуй пойми, как это делается. Просто жуть. Представьте: двое ребят-школьников, а вокруг аспиранты занимаются языками программирования и искусственным интеллектом.

Сейбел: Значит, запустить программу на Лиспе вам впервые удалось в Карнеги–Меллоне?

Завински: Пожалуй, да. Какое-то время мы дурачились с языком XLISP, который использовался на Маках. Хотя, думаю, это было позже. Я там по-настоящему научился программировать (на рабочих станциях PERQ, установленных для проекта Spice) на языке Spice Lisp, который затем стал языком CMU Common Lisp. Странное это было место. Мы узнавали, что такое разработка программного обеспечения, на ежедневных собраниях – просто слушали и все. Но встречались и действительно забавные персонажи, например парень, который был кем-то вроде нашего менеджера – присматривал за нами. Звали его Скеф Хоули – этакий белобрысый детина довольно дикого вида. Выглядел просто устрашающе. Говорил он мало. Сколько раз, бывало, сижу я в своем отсеке – работаю, что-то делаю, пишу программу на Лиспе. И тут, шаркая, подходит босой Скеф с кружкой пива и становится позади меня. Я ему: «Привет», – а он в ответ буркнет что-то или вообще промолчит. Просто стоит и смотрит, как я жму на клавиши. Я что-то делаю, и тут он вдруг: «Пффф, ошибка», – и уходит. А я остаюсь в полном недоумении. Такой дзэнский подход: учитель стукнул меня палкой – значит, надо помедитировать.

Сейбел: Я связался по электронной почте с Фальманом, и он написал, что вы были талантливы и быстро всему учились. Но он также припомнил вашу недисциплинированность. «Мы потихоньку пытались научить его работать в команде, научить писать такой код, чтобы вы или кто угодно еще мог его понять и через месяц». Вы помните эти уроки?

Завински: Не помню, чтобы я учился чему-то такому. Конечно, очень важно писать код, к которому потом сможешь вернуться. Но мне сейчас 39, а тогда было 15, и кое-что уже забылось.

Сейбел: В каком году это началось?

Завински: В 1984 или 1985. Думаю, летом, когда я из 10 класса перешел в 11. Занятия в школе заканчивались часа в четыре или около того, я отправлялся туда и оставался там до восьми-девяти вечера, хотя и не каждый день. Так или иначе, я проводил там немало времени.

Сейбел: И после школы вы ненадолго пришли в Университет Карнеги–Меллона?

Завински: Да. Дело в том, что я ненавидел учиться в старших классах. Это было самое жуткое время в моей жизни. И перед выпуском я спросил Фальмана, не возьмет ли он меня на полный день. Он ответил: «Нет, но у меня есть друзья, которые начинают новое дело. Поговори с ними». Это была фирма Expert Technologies (ETI). Кажется, Фальман был в деле. Они разрабатывали систему автоматической пагинации «желтых страниц» на Лисп, и я уже знал там кое-кого, кто работал с Фальманом. Они взяли меня, и все было хорошо, но через год я запаниковал: «Господи, я же устроился на эти две работы совершенно случайно, такое больше не повторится. Что будет, когда я уйду отсюда?» Без диплома колледжа мне светил разве что Макдональдс. Значит, надо было добывать диплом.

План был такой: я работаю на полставки в ETI, а остальное время учусь, тоже в половинном режиме. На деле же вышло, что работать и учиться пришлось по полной. Так продолжалось недель шесть, может, даже девять. Знаю только, что успел пропустить выбор курсов на семестр, так что деньги было уже не вернуть. Но не успел получить какие-нибудь оценки. В общем, учусь я или нет, непонятно.

Это было ужасно. В школе тебе говорят: мол, у нас тут один отстой и тесты, но в колледже все будет лучше. Поступаешь в колледж, и первый год там все то же самое. И тебе говорят: в магистратуре будет лучше. Все тот же отстой, только в другом месте – не для меня. Вставать в восемь, зубрить. Мне не разрешили пропустить курс «Введение в вычислительную технику», где объясняли, как пользоваться мышью. «Я полтора года работаю в этом университете, – говорил я, – и знаю, как пользоваться мышью». «Нет, мы не можем позволить вам, – отвечали мне. –

У нас такой порядок». И все в таком же духе. Я не выдержал и бросил колледж. И рад, что сделал это.

Потом я работал в ЕТИ года четыре или около того, пока компания не стала разваливаться. Мы работали на Лисп-машинах¹ серии TI Explorer, и я – кроме того, что работал над экспертными системами, – тратил массу времени на возню с пользовательским интерфейсом и на понимание того, как они вообще работают, сверху донизу. Я любил их, любил копаться в операционной системе и понемногу осознавать, как это все устроено.

Я написал кучу кода и разместил в нескольких группах новостей объявление о том, что ищу работу, предлагая при этом взглянуть на фрагмент моего кода. Питер Норвиг увидел его и назначил мне собеседование. Моя тогдашняя подруга переехала сюда, в Калифорнию, чтобы учиться в Университете Беркли, и я переехал вместе с ней.

Сейбел: Норвиг тогда был в Беркли?

Завински: Да. Это была очень странная работа. Там был целый выводок практикантов, которые исследовали понимание людьми естественных языков: это были лингвисты по образованию, которые слегка занимались программированием. И нужен был тот, кто собрал бы написанные ими куски и обрывки кода и сляпал бы из них что-то работающее.

Это было невероятно трудно, потому что у меня не хватало подготовки, чтобы понять, что, черт возьми, они там делают. То и дело получалось так: я остолбенело смотрю на что-то и не знаю, что это значит и в каком направлении двигаться, что читать, чтобы понять это. И я спросил Питера. Он внимательно выслушал меня и сказал: «В общем ясно, что пока это для тебя непонятно. Во вторник сядем, и я тебе все объясню». Значит, делать мне было пока нечего. Я углубился в работу с оконными системами, скринсейверами и другими штуками, связанными с пользовательским интерфейсом, которыми раньше занимался для забавы.

После шести-семи месяцев я почувствовал, что трачу свое время впустую. Я не делал для них ничего серьезного, это было похоже на каникулы. Позже не раз, действительно много работая, я оглядывался и спрашивал себя: «Зачем ты оставил эту работу, похожую на каникулы? Что тебе не нравилось? Тебе платили за разработку скринсейверов!»

¹ Речь о компьютерах, аппаратно оптимизированных для выполнения приложений на Лиспе, в отличие от обычных компьютеров, оптимизированных для ассемблерного кода. Такие компьютеры широко применялись для исследования задачи искусственного интеллекта, поскольку компьютеры общего назначения с ними просто не справлялись. См. http://en.wikipedia.org/wiki/Lisp_machine. – Прим. науч. ред.

В конце концов я устроился в компанию Lucid – одну из двух оставшихся Лисп-компаний. По-настоящему меня заставило уволиться чувство, что в Беркли я ничего не достигну. Вокруг меня были сплошь лингвисты, кое с кем я до сих пор дружу, они хорошие ребята – но не программисты. Абстрактные понятия им намного интереснее решения реальных задач. Я хотел делать что-то такое, чтобы можно было ткнуть пальцем и сказать: «Смотри, какую классную штуку я сделал».

Сейбел: Именно работая в Lucid, вы начали заниматься графическим редактором XEmacs. А когда вы туда пришли, вы что-нибудь писали на Лиспе?

Завински: Да, в одном из первых проектов, над которым я работал. Я, правда, не помню, что это был за компьютер, но это точно был 16-процессорный компьютер с поддержкой параллельных вычислений, на котором мы использовали собственную реализацию языка Common Lisp¹ с управляющими структурами, позволяющими распараллеливать задачи на разные процессоры.

Я немного поработал над задачей уменьшения накладных расходов при создании потоков, чтобы, например, выгоды от применения параллельного вычисления чисел Фибоначчи не сводились на нет накладными расходами создания стека для каждого потока. Мне это действительно нравилось. Я впервые имел дело с таким замысловатым компьютером.

А до этого я поднимал Лисп на новых типах машин. Обычно это означало, что кто-то уже написал компилятор под новую архитектуру железа и скомпилировал загрузчик Лиспа. Затем я брал бинарный, вроде бы работающий код и расшифровывал формат загрузчика новой машины, чтобы затем написать небольшую программу на Си, которая бы загрузила бинарные файлы на страницу памяти, сделала ее исполняемой и передала ей управление. После чего, вполне возможно, вы получали командную строку Лиспа и могли вручную загружать другие программы.

Из-за отсутствия нормальной документации этот процесс для каждой новой архитектуры был непростым делом. Приходилось компилировать код на Си, а затем просматривать и редактировать его в Emacs байт за байтом. Давайте-ка посмотрим, что же произойдет, если вот этот бит установить в ноль... Рухнет или нет?

¹ Речь идет о коммерческой реализации языка Common Lisp компании Lucid, получившей название Lucid Common Lisp. Позднее права перепродавались от одной компании к другой, пока не перешли к компании LispWorks, которая и продает эту реализацию под маркой Lucid Common Lisp. – *Прим. науч. ред.*

Сейбел: Когда вы говорите, что не было нормальной документации, это значит, что документация была неточной или что ее не было вовсе?

Завински: Нет, документация была, но зачастую она не отвечала действительности. Возможно, ошибка вкралась несколькими версиями раньше – кто знает? Но в определенный момент ты изменяешь этот бит, и машина уже не воспринимает твою программу как исполняемый модуль, и тебе приходится выяснять, что же произошло.

Сейбел: Ну, такое случается сплошь и рядом, начиная от низкоуровневого системного программирования и заканчивая высокоуровневым API, когда всё начинает работать совсем не так, как ты ожидаешь, или не так, как написано в документации. Как вы справлялись с этим?

Завински: Да просто начинаешь ожидать этого. Чем раньше поймешь, что сбился с пути, тем раньше сможешь выяснить, где именно. Лично я пытался создать исполняемый файл. Я знал, что компилятор Си может создавать исполняемые файлы. Поэтому алгоритм работы был такой: берешь хороший исполняемый файл и начинаешь его ковырять, пока он не превратится в плохой. Это основной механизм обратной разработки (reverse engineering).

Думаю, именно в компании Lucid я исправил самый сложный компьютерный баг. Я дошел до момента выполнения исполняемого файла, когда тот пытался загрузить интерпретатор Лиспа, но после выполнения 500 инструкций процесс загрузки падал. Тогда я начал выполнять процесс загрузки пошагово, чтобы выяснить, где же он падает. Хотя это было бессмысленно, создавалось впечатление, что процесс падал каждый раз в другом месте. Я стал исследовать ассемблерный код компьютерной архитектуры, о которой имел лишь смутное представление. Наконец до меня дошло. «Господи, при пошаговом выполнении он делает что-то не то. Возможно проблема связана с временными задержками». В итоге я понял, что происходило: дело в том, что это была одна из первых машин с упреждающим исполнением команд¹. В этом случае вы-

¹ Упреждающее исполнение команд (Speculative Execution), или исполнение команд по предположению, – это совокупность методов, позволяющая ЦП с конвейерной архитектурой обрабатывать команды без уверенности в том, что они реально будут исполняться в программе (например, в случае условного перехода). Если предположение оказывается верным, то исполнение команд продолжается и выигрывается время, а если нет (misspeculation), результаты упреждающего исполнения аннулируются. – *Прим. науч. ред.*

полнялись обе ветви кода¹. Но GDB² при пошаговой отладке выполнял только одну из ветвей. Так что баг был в GDB.

Сейбел: Здорово.

Завински: Точно. Но это меня подкосило. «Господи! Мне придется отлаживать GDB, который я первый раз вижу». Чтобы обойти ошибку отладчика, нужно остановить выполнение процесса перед инструкцией ветвления, задать точки останова в обеих ветвях и продолжить выполнение. Именно таким способом мне удалось воспроизвести ситуацию и понять, что же происходит на самом деле. Затем я потратил около недели на исправление GDB, но так и не смог понять, в чем же дело. Я предполагал, что из-за проблем с одним из регистров отладчик считал, что всегда выполняется одна из ветвей условия или что-то в этом роде. Поэтому я изменил команду пошагового выполнения инструкций, чтобы определить, когда оно дойдет до инструкции ветвления, и там сказать: «Стоп, это не делай». Теперь я мог просто пошагово выполнять программу. Выполнение в конце концов останавливалось, я вручную задавал точку останова и продолжал выполнение. Когда что-то отлаживаешь, понимая, что не только путь выбран неверный, так еще и инструмент никуда не годится, — что может быть хуже.

Разработка систем на Лиспе была особенно запутанной, поскольку GDB был совершенно неприменим к Лисп-коду, который не содержал никакой отладочной информации. Причина была в том, что интерпретатор Лиспа был разработан с помощью компилятора, о котором GDB не имел ни малейшего понятия. Думаю, для некоторых платформ создавались стековые фреймы, которых отладчик GDB просто не понимал. На этом этапе GDB был способен лишь на пошаговый прогон ассемблерного кода. Поэтому мы хотели избавиться от него как можно скорее.

Сейбел: А потом у вас появился отладчик Лиспа, и вы оказались во всеоружии.

Завински: Ага.

Сейбел: И примерно в то же время компания Lucid сменила курс, решив создавать интегрированную среду разработки для C++.

¹ Теоретически, при наличии условия, должна выполняться только одна ветвь программы, но благодаря упреждающему исполнению команд выполнялись обе ветви, хотя результаты одной из них затем отбрасывались. — *Прим. науч. ред.*

² GDB (GNU Project Debugger) — переносимый отладчик проекта GNU, который работает на многих UNIX-подобных системах и умеет выполнять отладку многих языков программирования, включая Си, C++ и Фортран. — *Прим. науч. ред.*

Завински: Это началось еще до меня: когда я пришел туда, интегрированная среда разработки уже создавалась. Люди начали переходить с Лиспа на Energize – так называлась эта среда разработки. Это был отличный продукт, но он появился на два-три года раньше, чем нужно. Никто – по крайней мере, никто среди пользователей UNIX – не думал, что им вообще это нужно. Сейчас все используют такие возможности, но тогда мы тратили кучу времени, объясняя, почему эта штука лучше, чем vi¹ и GCC. Еще я делал кое-что для Emacs. Кажется, тогда я уже переписал компилятор байт-кода². Зачем мне это было нужно? Правильно, потому что я делал что-то вроде адресной книги.

Сейбел: Базу данных для Большого Брата?

Завински: Ага. Но работала она ужасно медленно. Я стал выяснять, почему, и понял, что компилятор ни к черту не годится. Я переписал компилятор, что и привело к моей первой ссоре с непримиримым Стеллменом. Тогда я много чего узнал о Emacs.

Сейбел: А изменения в компиляторе коснулись формата байт-кода или только процесса компиляции?

Завински: Я сделал несколько изменений: внес исправления в интерпретатор байт-кода, написанного на Си, а также добавил несколько новых инструкций для повышения производительности. Но компилятор мог быть сконфигурирован, чтобы генерировать байт-код в старом формате или использовать преимущества нового.

Так вот, я написал новый компилятор, а Стеллмен заявил: «Не вижу необходимости в этих изменениях». А я ему в ответ: «Да что вы? Теперь генерируется более быстрый код». А он: «О'кей, тогда пришли мне все изменения исходников и объясни каждую измененную строку». Тогда я ответил: «Нет, я не буду этого делать. Я полностью переписал старый компилятор, потому что он был дерьмовым». Это ему не понравилось. Мой компилятор был добавлен только потому, что я выпустил его, тысячи людей начали им пользоваться, компилятор им понравился, и они надоедали Стеллмену два года. Вот он и добавил мой компилятор, чтобы его больше не доставали.

¹ vi (сокр. от visual) – серия текстовых редакторов операционных систем семейства UNIX, которые применялись совместно с компиляторами GCC для разработки ПО. – *Прим. науч. ред.*

² В зависимости от реализации некоторые компиляторы могут преобразовывать исходный текст на языке ЛИСП сразу же в машинный код (native code), а некоторые вначале компилируют исходный код в промежуточный (байт-код), который уже затем интерпретируется в машинный код конкретной машины. Для Emacs были реализованы компилятор и интерпретатор. – *Прим. науч. ред.*

Сейбел: Вы подписали передачу авторских прав на этот компилятор компании Free Software Foundation?

Завински: Да, я сделал это сразу же. По-моему, это было первое, о чем говорилось в том электронном письме. Там было что-то вроде: «Пришли мне изменения и подпиши это». Я подписал и сказал: «Остальное я выполнить не могу. Я не могу прислать вам изменения. Это просто смешно. Тут все описано, взгляните сами». Вряд ли он смотрел на это хоть раз.

Толкуют, будто были какие-то судебные разборки между Lucid и FSF. Это полная ерунда. Мы подписывали бумаги о передаче авторских прав на все, что делали для FSF. Им было зачем-то нужно утверждать, будто мы делали это не всегда. Бывало, мы подписывали одни и те же бумаги несколько раз, потому что они говорили, будто потеряли их. Кажется, подобная шумиха была с подписыванием бумаг по XEmacs, но это было позже, когда я уже ушел.

Сейбел: Итак, вы начали с Лиспа, но явно не заикнулись на нем на всю жизнь. Что было потом?

Завински: Следующим языком, на котором я создавал что-то серьезное, был Си. Казалось, будто вернулись времена программирования на ассемблере под Apple II: это был ассемблер PDP-11, который считал себя настоящим языком. Программирование на Си, как вы наверное знаете, малоприятное занятие, поэтому я старался как можно дольше держаться подальше от всего этого. А C++ – просто гадость. С ним все не так. Так что я старался как можно дольше держаться от него подальше и в Netscape писал на Си. Это было просто – ведь мы были нацелены на небольшие компьютеры, на которых нельзя было нормально использовать программы на C++, потому что код распухал до безумия, как только были задействованы библиотеки. И потом компиляторы для C++ постоянно менялись, что приводило к массе проблем с несовместимостью. Поэтому мы с самого начала выбрали для себя ANSI C, и он хорошо служил нам. После всего этого Java отчасти воспринимался как возврат к Лиспу – в том смысле, что этот язык не лезет из кожи вон, желая отпугнуть вас. Им удобно пользоваться.

Сейбел: В каком смысле?

Завински: Автоматическое управление памятью. Функции выглядят именно как функции, а не как подпрограммы. Очень много сделано для обеспечения модульности. В коде на Си всегда есть искушение применить оператор `goto` только потому, что это просто.

Сейбел: Значит, сейчас вы в основном используете языки Си и Perl?

Завински: Ну, я вообще сейчас не много программирую. В основном пишу глупые маленькие скрипты на Perl для поддержания работы

моих серверов. Я написал кучу дурацких программ поиска картинок для моих МРЗ-файлов или нечто вроде того. Простенькие одноразовые программки.

Сейбел: Вам нравится Perl или он просто всегда под рукой?

Завински: Терпеть его не могу, ужасный язык. Но он установлен абсолютно везде. Садись за компьютер – и не нужно никого просить установить Perl, чтобы выполнить свой скрипт: Perl там уже есть. Это единственный аргумент в его пользу.

У него неплохая коллекция библиотек. Часто есть библиотека, позволяющая делать именно то, что тебе нужно. Пусть библиотеки иногда неважно работают, но это уже что-то. Не то, что с Java, когда пишешь что-нибудь на этом языке и пытаешься понять, что вышло. Я сам с трудом установил Java на своем компьютере. Это ужасно. Мне кажется, Perl – противный язык. Если научиться использовать его хоть немного, можно сделать его похожим на Си или, скорее, на JavaScript. Сумасшедший синтаксис, непонятные структуры данных. Немного хорошего можно сказать о Perl.

Сейбел: Но он не так плох, как C++.

Завински: Нет, конечно, нет. Они созданы для разных задач. Есть задачи, которые намного проще реализовать на Perl (или подобном ему языке), чем на Си, только потому, что все так называемые скриптовые языки ориентированы на работу с текстом. Вот чего я действительно не понимаю, так это различия между «программированием» и «написанием скриптов». По-моему, чепуха все это. Но если основная твоя работа заключается в обработке текста или запуске программ (например, запустить `wget`¹, получить от нее какой-то HTML и сопоставить его с образцом), то гораздо легче это сделать на Perl, чем даже на Emacs Lisp.

Сейбел: Не говоря о том, что Emacs Lisp не слишком удобен для работы с утилитами командной строки.

Завински: Ну да, хотя я все время писал разные мелкие утилиты с помощью Emacs. Было время, на раннем этапе работы в Netscape, когда часть процесса сборки включала запуск скриптов с помощью команды `emacs -batch` для работы с некоторыми файлами. Это никому не нравилось.

Сейбел: Представляю... А как насчет XScreenSaver² – все еще работаете над ним?

¹ `wget` – программа для загрузки файлов по сети.

² XScreenSaver – коллекция из более чем двухсот различных заставок (`screen saver`) для UNIX и Mac OS. Создана Джейми Завински в 1992 году и до сих пор им поддерживается. – *Прим. науч. ред.*

Завински: Я до сих пор пишу новые скринсейверы – только ради развлечения и только на Си.

Сейбел: Применяете ли вы какую-нибудь интегрированную среду разработки?

Завински: В основном Emacs. Правда, недавно я портировал XScreen-Saver на OS X. Я сделал это так: реализовал заново Xlib на базе Cocoa (графической основе Маков), поэтому мне не пришлось переписывать код всех скринсейверов. Они все еще обращаются к Xlib, но я реализовал все соответствующие методы. Сделано это было на Objective-C, который оказался отличным языком программирования, и работа доставила мне огромное удовольствие. Он определенно напоминает Java в лучших его проявлениях, но также напоминает и Си. То есть в основном это Си, и можно напрямую использовать Си-код, вызывая нужные функции без лишних усилий.

Сейбел: Работая в компании Lucid, что вы узнали по технической части кроме политической составляющей разработки Emacs?

Завински: Работая там, я точно стал более хорошим программистом. Во многом потому, что был окружен действительно очень умными людьми. Все, кто там работал, были великолепны. Просто здорово находиться в коллективе, в котором, если кто-то скажет: «Это чепуха» или «Нужно сделать это вот так», – можно просто верить на слово, не сомневаясь, что он знает, о чем говорит. Это было на самом деле здорово. Не скажу, что раньше я не бывал среди умных людей, но это был именно коллектив высококлассных специалистов в равной степени.

Сейбел: А насколько велика была команда разработчиков?

Завински: В компании было человек 70 – точно не знаю, и около 40 из них разработчики. В команде Energize было 20–25 человек. Все разработчики делились по направлениям. Кто-то работал над компиляторами, кто-то над серверной частью базы данных. Кто-то работал над пользовательским интерфейсом, не связанным с Emacs. Я и еще двое трое занимались интегрированием Emacs с внешним окружением. Так получилось, что я работал в основном над Emacs, пытаюсь сделать так, чтобы нашим редактором Emacs 19 можно было пользоваться, чтобы он не падал то и дело и чтобы под ним запускались все пакеты, которые должны запускаться.

Сейбел: То есть вы хотели, чтобы Emacs, который являлся составной частью вашего продукта, был полнофункциональной версией Emacs.

Завински: Изначально мы не собирались включать Emacs в наш продукт. Идея была такая: на вашей машине уже стоит Emacs, вы берете наш продукт, и они совместно работают. Например, на вашей машине установлен компилятор GCC и наш продукт, и они совместно работают.

Кажется, одним из первых кодовых названий нашего продукта было что-то вроде Hitchhiker (попутчик), так как идея была в том, что он будет брать и интегрировать все имеющиеся у вас инструменты – заставит их «общаться» между собой, предоставив им необходимый уровень коммуникации.

Но это не сработало. Мы стали выпускать собственные версии GCC и GDB, потому что не успевали за изменениями этих систем, по крайней мере, успевали не всегда. То же самое было и с Emacs. Поэтому мы выпустили все целиком. В конце концов мы пошли таким путем: «Так, мы заменили Emacs. Черт. Нам пришлось это сделать, поэтому нужно заставить его работать нормально». Только на режим эмуляции vi я потратил уйму времени.

Сейбел: И это несколько недель вашей жизни, которые вам никогда не хотелось бы пережить снова.

Завински: И не говорите. Это было настоящее испытание. Кажется, в результате все заработало. Настоящая проблема была не в том, что режим эмуляции vi работал плохо, а в том, что пользователи были вынуждены постоянно выходить и перезапускать vi. И что бы я ни делал, эту проблему никак не удавалось решить. Пользователь говорил: «Я думал, она будет запускаться за полсекунды, а она запускается за четырнадцать. Это просто смешно. Я не могу этим пользоваться».

Сейбел: Почему вы ушли из компании Lucid?

Завински: Lucid разваливалась. Людей то и дело отправляли в неоплачиваемый отпуск. Я разослал письма своим знакомым: «Привет, кажется, мне скоро будет нужна новая работа». Одним из них был Марк Андрессен. Он сказал: «Забавно, что ты об этом упомянул, ведь на прошлой неделе мы как раз создали компанию». Так я нашел работу.

Сейбел: Итак, вы ушли в Netscape. Над чем вы работали?

Завински: Я практически сразу начал работать над версией броузера для UNIX. К тому моменту было написано совсем немного кода – результат нескольких дней работы. Немногим больше было сделано для версий под Windows и Mac. Модель системы состояла из крупного куска бизнес-логики, независимой от платформы, и небольшой части кода, отвечающего за представление для каждой платформы.

Сейбел: Это был полностью новый код?

Завински: Полностью новый. Большинство основателей компании Netscape были разработчиками броузера NCSA/Mosaic. Они разработали разные версии, которые по сути представляли собой три разные программы. И все шестеро, делавших это, оказались в Netscape. Они не

использовали повторно старый код, но уже решали подобную задачу ранее.

Сейбел: То есть они просто начали писать код с чистого листа?

Завински: Именно так. Я никогда не видел код броузера Mosaic (кстати, я до сих пор его не видел). Как раз в это время у нас были судебные разбирательства: университет утверждал, что мы использовали их код, но мы, кажется, как-то уладили этот вопрос. Ходили слухи, будто мы действительно использовали их код, но мы этого не делали.

И зачем? Каждый хочет попробовать еще раз заново, верно? Во время разработки программы находишь ответы на многие вопросы, и, получив шанс выкинуть все и начать с нуля, конечно же, воспользуешься этим шансом. Во второй раз все должно получаться гораздо лучше. И действительно получается. Например, общепринятая архитектура не предоставляла возможность параллельной загрузки изображений. А ведь это действительно важная возможность. Поэтому мы разработали лучшую архитектуру.

Сейбел: Но это похоже на классический случай синдрома второй системы.

Завински: Вот именно.

Сейбел: Так как же всем вам удалось его избежать?

Завински: Самым святым для нас были сроки. Или мы выпустим новый продукт за полгода, или умрем, пытаюсь это сделать.

Сейбел: Как вам удалось справиться со сроками?

Завински: Мы осмотрелись и поняли, что если не сделаем это за полгода, кто-нибудь опередит нас. Поэтому мы решили, что справимся с этим за шесть месяцев.

Сейбел: Учитывая, что вначале была определена дата, вам нужно было жертвовать либо возможностями системы, либо качеством. Как вы поступили?

Завински: Мы долго обсуждали возможности системы. Ну, на самом деле не так и долго, но нам казалось именно так, потому что тогда каждый день был как неделя. В конце концов мы определили все возможности. У нас была белая доска, на которой мы писали наши идеи, связывая их друг с другом. Всего нас было человек шесть-семь, точно не помню. Группа умных, самовлюбленных людей, которые сидят в одной комнате и орут друг на друга неделю или около того.

Сейбел: Шесть-семь человек – это вся команда разработчиков Netscape или только версии для UNIX?

Завински: Это была вся команда разработки клиентской части. Были еще ребята, отвечавшие за серверную часть, которые в основном занимались разработкой собственной версии Apache. Мы мало общались с ними, так как были заняты. Мы завтракали вместе, но не более того. Мы выяснили, что должно быть в системе, и распределили работу так, чтобы над каждой частью проекта работало не более двух человек. Я занимался UNIX-частью, а Лу Монтулли делал основную работу по сетевой составляющей серверной части. Предварительная версия команды была следующей: Эрик Бина занимался общей компоновкой системы, Джон Миттельхаузер и Крис Хаук занимались пользовательским интерфейсом под Windows, Алекс Тотич и Марк Ланетт – пользовательским интерфейсом под Мак. Потом эти команды слегка увеличились. Мы совещались, потом расходились по своим отсекам и стучали по клавиатуре 16 часов в сутки, пытаюсь сделать так, чтобы хоть что-то работало.

Обстановка была отличная, мне правда нравилось. Поскольку каждый считал себя правым, мы постоянно спорили, но быстро понимали друг друга. Кто-нибудь заглядывает в твой отсек и говорит: «Какого хрена ты сохранил этот код? Так нельзя делать, это же дерьмо собачье. Ты идиот». Отвечаешь ему: «Да пошел ты!», смотришь на свой код, исправляешь ошибки и сохраняешь его. Мы были весьма резкими парнями, но зато не тратили лишнее время, потому что никто ни с кем не церемонился, не объяснял подолгу, почему считает что-то неправильным. Можно было просто сказать: «Да это куча дерьма, я не буду это использовать». И вопрос тут же решался. Да, атмосфера была напряженная, но мы сделали все довольно быстро.

Сейбел: То есть для быстрого создания программы требовалась много-часовая интенсивная работа?

Завински: Да уж, не курорт. Я знаю, что мы поступали именно так, и это работало. Ответить на этот вопрос можно так: а есть ли примеры того, как кто-то быстро и качественно создал здоровенную систему, обедая дома и регулярно высыпаясь? Бывало ли такое? Не знаю. Может, и бывало.

Но не всегда нужно делать все как можно быстрее. Нужно не перегореть за два года, а быть в состоянии проработать еще десять. А при 80-часовой рабочей неделе это невозможно.

Сейбел: Чем вы больше всего гордитесь из того, над чем работали?

Завински: На самом деле я горжусь самим фактом того, что мы выпустили это. Всю систему. Я был полностью сконцентрирован на своей части – создании пользовательского интерфейса для UNIX. Но горжусь

я именно тем, что мы вообще выпустили систему и людям она понравилась. Пользователи сразу же стали переходить с NCSA Mosaic, говоря: «Ух ты, это лучшая система, которую мы когда-либо видели». У нас на панели инструментов была кнопка для страницы What's Cool (Что есть классного), чтобы показать миру все эти безумные веб-сайты, созданные к тому времени, – штук двести, наверное! Не сказать, чтобы я сильно гордился кодом, скорее именно тем, что дело сделано. Код во многих отношениях был не очень хорош – слишком я торопился. Но все было закончено. Мы выпустили систему – это было главное.

В ту ночь, выпустив бета-версию .96, мы все сидели по разным углам комнаты и смотрели, как идет закачка с прикрученным звуковым сигналом. Это было потрясающе. А через месяц программой, которую написал я, пользовались два миллиона человек. Это было невероятно. Это определенно стоило потраченных усилий – повлиять на жизнь людей, сделать так, чтобы их времяпрепровождение стало веселее, или приятнее, или удобнее благодаря тому, что мы сделали.

Сейбел: После этой бешеной гонки в какой-то момент надо было вернуться к качеству полученного кода. Как вы, ребята, справились с этим?

Завински: Честно говоря, не очень. Никогда не было времени начать все сначала и переписать код. Да и вообще не слишком это здорово – начинать все заново и переписывать код.

Сейбел: На каком-то этапе вы также работали и над почтовым клиентом, да?

Завински: Когда мы работали над версией 2.0, Марк зашел в мой отсек и сказал: «Нам нужен почтовый клиент». Я ответил: «Отлично, я уже работал над почтовыми клиентами». Я тогда жил в Беркли и несколько недель почти не появлялся в офисе – сидел часами в кафе и набрасывал что-то, пытаюсь выяснить, чего же хочу от почтовой программы. Я делал списки, что-то из них вычеркивал, думая, когда же приду к чему-нибудь. Как должен выглядеть пользовательский интерфейс?

И вот я вернулся на работу и стал писать код. Потом Марк снова зашел и сказал: «Мы тут взяли еще одного парня, он занимался почтовыми клиентами. Будете работать вместе». Это был Терри Вейссман, просто фантастический человек – мы прекрасно сработались. И совсем в другом ключе, по сравнению с работой над броузером на ранней стадии.

Мы совсем не кричали друг на друга. Просто не представляю, как можно было так работать, да и вообще, работал ли так хоть кто-нибудь еще. Мы распределили свою работу следующим образом. Я делал набросок дизайна и начинал понемногу писать код, и раз в несколько дней мы

смотрели на список задач и говорили друг другу: «Я буду работать над этим. – А я над этим». И расходились.

Мы сохраняли код в репозитории и встречались снова. Он говорил: «Ладно, я с этим разобрался, а ты что делаешь?!» – «Работаю вот над этим». – «Хорошо, тогда я займусь вон тем». Так мы в некотором роде делили задачи между собой, и все шло превосходно.

Бывали и разногласия. Я предложил добавить фильтрацию на уровне папок, поскольку у нас не было времени сделать это как следует. Он сказал: «Нет-нет, я правда думаю, что мы должны сделать все как надо». А я: «Да у нас же нет времени!» Но он все сделал той же ночью.

Еще кое-что: мы с Терри виделись редко – он жил в Санта-Крус, я в Беркли. До работы нам было ехать примерно одинаково, но с разных сторон. А поскольку по работе нам нужно было общаться только между собой, то мы договаривались так: «Если ты согласен, чтобы я не приезжал, то и я согласен, чтобы ты не приезжал». – «Давай».

Сейбел: Вы вдвоем много переписывались по электронной почте?

Завински: Да, постоянно. Это было еще до эры мгновенных сообщений (Instant messenger, IM) – сейчас, конечно, мы бы прибегли именно к ним, потому что постоянно слали друг другу электронные письма. И переговаривались по телефону.

Итак, мы выпустили версию 2.0 с почтовым клиентом, хорошо принятую пользователями. Потом мы работали над версией 2.1, и я уже начал думать, что работа подходит к концу, но это оказался один из случаев, когда мы так и не смогли выпустить версию с первого раза. Мы с Терри были уже на полпути, когда зашел Марк и сказал: «Мы покупаем такую-то компанию. И там разрабатывают почтовый клиент, похожий на ваш». Я ответил: «Да, хорошо, но у нас уже есть такая программа». Он пояснил: «Понятно, но мы растем очень быстро, и нам все труднее нанимать на работу хороших людей. Иногда можно их нанять, купив другую, хорошо знакомую компанию». – «Ладно. Чем они будут заниматься?» – «Работать над вашим проектом». – «Черт, тогда я займусь чем-нибудь другим».

И как только Netscape купила эту компанию (которая называлась Collabra), нас с Терри отдали под их руководство. Эта компания выпускала почтовый клиент, во многом очень похожий на наш, за исключением того, что он работал только под Windows и потерпел полный крах на рынке.

А потом им крупно повезло – их купила компания Netscape. После чего Netscape передала бразды правления этой компании. Но вместо того чтобы отвечать только за почтовую программу, они оказались во

главе всего подразделения по разработке клиентской части. Мы с Терри работали над Netscape 2.1, когда это произошло и начался процесс переписывания. Тогда стало понятно, что Netscape 3.0 будет выпущен слишком поздно, и наша версия 2.1 превратилась в версию 3.0, потому что нужно было выпустить на рынок основную версию.

А потом версия 3.0, над которой они начали работу, стала версией 4.0, которая, как вы знаете, стала одной из самых больших неудач в области программных разработок. Главным образом это и погубило компанию. Она умирала долго, но это было неотвратимо. Процесс переписывания был инициирован новоприобретенной компанией, которая ничего особенного не добилась, пренебрегла всей нашей работой и нашими достижениями, а сразу же начала страдать синдромом второй системы и потянула нас ко дну.

Они полагали, что, оказавшись у руля, просто обязаны поступать по-своему. Но когда они делали это по-своему в своей компании, они провалились. И когда люди, добивавшиеся успеха, говорили им: «Послушайте, правда, не используйте C++, не используйте потоки», они отвечали: «О чем вы говорите? Вы ничего не понимаете».

Именно такие решения, как отказ от C++ и использования потоков, позволили нам выпустить продукт вовремя. Другой важной составляющей было то, что мы всегда выпускали версии под все платформы одновременно. Это решение они тоже считали глупым: «У 90% пользователей установлена Windows, так что мы сосредоточим усилия на работе версии под Windows, а позже портируем ее под остальные платформы». Так поступали многие компании, потерпевшие крах. Если вы собираетесь выпускать кроссплатформенный продукт, то история показывает, как именно не следует поступать. Если вы действительно хотите выпускать кроссплатформенное решение, то разрабатывать все нужно одновременно. А портирование приводит к паршивому результату на второй платформе.

Сейбел: Версия 4.0 создавалась с нуля?

Завински: Ну, не то чтобы совсем с нуля, но в итоге они переписали каждую строку кода. И использовали C++ с самого начала. Я боролся против этого изо всех сил и, черт возьми, был прав. Все стало распухать, появились проблемы с совместимостью, потому что когда пишешь на C++, невозможно прийти к согласию, какие именно 10% языка можно безопасно использовать. Кто-нибудь говорит: «Мне нужны шаблоны», а затем выясняется, что нет двух компиляторов, которые реализуют шаблоны одинаково.

А когда весь опыт написания кода говорит о том, что мультиплатформенный код означает работу и под Windows 3.1, и под Windows 95,

вы даже не представляете, насколько это важно. Поэтому разработка версии под UNIX (к счастью, это была уже не моя проблема) стала настоящим кошмаром, как и разработка версии для Мака. Это означало, что стал невозможным выпуск под старые версии Windows, такие как Win16. Пришлось сокращать количество поддерживаемых платформ. Может быть и пришло время так поступить, но причины были ни к черту. В этом не было необходимости.

Может показаться, что это мой горький, сугубо личный взгляд на вещи, будто мы с Терри создали великолепную штуку и теперь наказаны за свой успех, а в качестве наказания нами управляют идиоты. То время в Netscape стало несчастьем для меня. Начался период, когда я оставался в компании, ожидая, когда Netscape перейдет в другие руки.

Сейбел: И вы проработали там пять лет?

Завински: Да. И еще один год после продажи Netscape, потому что незадолго до этого стартовал проект mozilla.org, и снова стало интересно. Так что я остался из-за него.

Сейбел: Вам все-таки пришлось иметь дело с C++?

Завински: Ну, скорее с Java. В какой-то момент мы решили переписать браузер на Java. Мысли у нас были такие: «Ура! Мы выкинем весь код версии 4.0, который грозит потопить нашу компанию, и это точно работает, ведь мы знаем, что делаем!»

Но не работало.

Сейбел: Не работало, потому что язык Java был еще сырым?

Завински: Нет. Нас снова разбили на четко определенные группы. Трое работали над почтовым клиентом. И мы сделали его. Мы сделали действительно отличный почтовый клиент – быстрый, со многими удобными функциями, он лучше сохранял ваши данные и никогда не тормозил при записи больших файлов. Мы воспользовались многими преимуществами механизма многопоточности языка Java, работа с которым оказалась не такой мучительной, как я ожидал. Работать на самом деле было приятно. С помощью разработанного нами API мы видели все направления, в которых он мог бы развиваться.

Кроме одной вещи, с которой он не справлялся, – отображения сообщений. Он генерировал HTML, а для его отображения нужен был слой отображения HTML, который тогда не был готов и не был готов никогда. Работа группы планирования пошла насмарку, и именно они были причиной отмены всего проекта.

Сейбел: Значит, вам приходилось бороться с сырыми на тот момент библиотеками Java для построения пользовательского интерфейса.

Завински: Нет, я бы так не сказал. Все работало. Просто в середине окна был большой белый прямоугольник, где мог отображаться только обычный текст. Они подходили к проекту очень академично, оперировали такими понятиями, как объектная модель документа (Document Object Model, DOM) и описание типа документа (Document Type Definition, DTD). «Нам нужно создать вот тут еще один уровень абстракции и создать здесь делегирование для делегирования вон того делегирования. И может быть, на экране наконец появится буква».

Сейбел: По-моему, вас сильно раздражает перепроектирование.

Завински: Да. К концу дня доделай эту чертову хрень! Хорошо, конечно, переписывать код, чистить его, чтобы с третьего раза он-таки получился красивым. Но суть-то не в этом, ты сидишь здесь не для того, чтобы писать красивый код, а для того, чтобы выпускать продукт.

Сейбел: Любители перепроектирования часто говорят: «Ну, все пойдет как по маслу, после того как мы прикрутим эту библиотеку. На самом деле мы сэкономим кучу времени».

Завински: Это чистая теория.

Сейбел: Иногда она оправдывает себя на практике, если человек мыслит здраво, а библиотека не слишком переусложнена. Тогда это действительно экономит время. Можете ли вы четко обозначить свою позицию?

Завински: Я знаю, что это банально, но всегда оказывается верным принцип «чем хуже – тем лучше». Если тратишь время на создание совершенной библиотеки, которая будет делать то, что тебе хочется, и позволит сопровождать версии от 1.0 до 5.0, все прекрасно. Но знаете что: пока три года создаешь версию 1.0, конкурент создаст аналогичный продукт за полгода – и ты вне игры. Ты никогда не выпустишь версию 1.0, потому что кто-то успел раньше.

В версии твоего конкурента, выпущенной за полгода, код – полное дерьмо, и они собираются переписать его за два следующих года, но, видите ли, они могут себе это позволить, потому что ты уже без работы.

Сейбел: Иногда, когда время поджигает, приходится выкидывать большой кусок кода, потому что кажется, что проще написать его заново.

Завински: Да, порою кое-что приходится списывать в утиль. Мне никогда не нравился такой подход, но если тебе достается чужой код, бывает проще написать все заново, чем использовать старый. Ведь нужно потратить время, чтобы понять тот код, выяснить, как им пользоваться, и понять его настолько, чтобы ты смог его отлаживать. Начать с нуля получится быстрее. Он может выполнять только 80% от того, что тебе нужно, но, может быть, именно эти 80% тебе и нужны.

Сейбел: А разве не та же логика — когда кто-то приходит и говорит: «Я не могу понять эту ерунду, я просто перепишу ее заново» — приводит к бесконечному переписыванию, что вам так не нравится в разработке программ с открытым исходным кодом?

Завински: Да. Но есть и другой аспект, помимо вопроса эффективности: намного интереснее писать свой код, чем пытаться разобраться в чужом. Так что совершенно понятно, почему так бывает. Но вся возня с Linux/GNOME — это постоянное метание между чьим-то хобби и настоящим продуктом. Что это? Исследовательский проект, с помощью которого мы экспериментируем и пытаемся понять, как должна выглядеть графическая среда пользователя? Или конкуренция с компанией Apple? Трудно заниматься и тем, и другим.

Но даже это утверждение, которое предполагает, что есть некто, принимающий решение, совершенно не соответствует истине. Все это — просто стечение обстоятельств. Вот и получается, что все постоянно переписывается, но так и не доводится до конца. Все просто прекрасно, если компьютер для тебя — это всего лишь игрушка, с которой всегда интересно повозиться, а не средство для достижения цели, не инструмент, с помощью которого доводишь интересное для себя дело до конца.

Сейбел: Кстати, насчет «интересно повозиться»: вы все еще получаете удовольствие от программирования?

Завински: Иногда. Сейчас я занимаюсь сисадминской работой, которую терпеть не могу, да и никогда не любил. Мне нравится работать над XScreenSaver, потому что в некотором роде скринсейвер (настоящий скринсейвер, а не библиотека XScreenSaver) — это совершенная программа: все делается с нуля и никаких версий 2.0. И ошибки в такой программе бывают редко. А если она падает (ай-ай-ай, произошло деление на ноль), просто ее исправляешь.

И никто никогда не попросит о новой функции для скринсейвера: «Хочу, чтобы он был более желтым». Ведь скринсейвер такой, какой есть. Вот почему для собственного удовольствия я пишу именно их. Это четкий результат, о котором не надо думать слишком много. Он не преследует тебя.

Сейбел: И вам нравятся головоломки из математики, геометрических построений и графики?

Завински: Да. Если повернуть это маленькое уравнение вот этак, что получится? Как сделать, чтобы эти шарики крутились более естественно и менее механически, чем это свойственно компьютеру? И все такое. Как сделать так, чтобы эти гармонические волны больше походили на чьи-то прыжки?

А затем я стал писать все эти мелкие глупые сценарии командной оболочки для самозащиты. Я знаю, что могу сделать это вручную, щелкнув на одной из 30 000 страниц, но почему бы не написать сценарий и не сэкономить время? Мне ничего не стоит это запрограммировать. А непрограммистам это кажется волшебством.

Мне очень понравилось портировать библиотеку XScreenSaver для Мака. Пришлось написать немало нового кода, обдумать API и всю общую структуру.

Сейбел: Вы проектировали API? Как вы структурировали код?

Завински: Я одновременно рассматривал существующие API и пытался найти лучший способ создать слой между миром X11 и миром Мака. Как мне все это структурировать? Какие API Мака подойдут лучше? Впервые за долгое время я сделал что-то и подумал: «Здорово! Пожалуй, тут я еще кое-что могу».

И это было незабываемое ощущение, потому что индустрия разработки ПО вымотала меня. Я не мог больше выносить всю эту политику как в мире корпораций, так и в среде свободных разработчиков. Я был сыт этим по горло. Я хотел работать над чем-то, где не было бы споров по всяким мелочам, и не хотел видеть, как мой продукт уничтожается бюрократическим решением, перед которым я бессилён.

Сейбел: А не было искушения вернуться и поработать над Mozilla?

Завински: Нет. У меня не было никакого желания снова погружаться в споры и эти долбанные противостояния по Bugzilla. Это совсем не весело. Но это неизбежно при создании больших продуктов. Если для работы над проектом требуется больше одного человека (что естественно для таких проектов, как Mozilla), по-другому не получится. Но я не хочу больше этих баталий, за многие годы это желание полностью выбили из меня. Как программист, я могу пойти работать где-то еще. Но мне это не нужно, да я и не хочу. Как только меня что-то достанет, я сразу уйду. А если я организую собственную компанию, то не смогу там быть программистом, поскольку придется ею руководить.

Сейбел: Что вам нравится в программировании, кроме того что два миллиона человек пользуются вашим продуктом?

Завински: Трудно сказать. Думаю, поиск решения задачи. Это не совсем то же, что головоломка, да я и небольшой любитель головоломок. Просто пытаешься понять, как попасть из точки А в точку Б, и думаешь, как заставить машину выполнять то, что тебе нужно. Удовольствие от программирования заключается главным образом в этом.

Сейбел: Есть ли для вас понятие красоты кода? Помимо возможности поддержки, существует ли эстетическая составляющая?

Завински: Да, конечно. Когда что-то выражено очень верно – лаконично и по сути, – вроде хорошо сформулированного афоризма или мгновенной карикатуры, выполненной одним росчерком пера и очень похожей на оригинал. Что-то в этом духе.

Сейбел: Как вы думаете, программирование и писательская деятельность – это похожие интеллектуальные занятия?

Завински: В каком-то смысле да. Программирование все же гораздо строже. Но они очень близки во всем, что касается способности выражать собственные мысли. Никакого беспорядка: все, что приходит в голову и требует перевода в слова, выражаешь максимально четко. По-моему, именно этим программирование очень близко к сочинению прозы.

Мне кажется, здесь задействованы одни и те же отделы мозга, но выразить это словами нелегко. Я часто читаю код и чувствую, когда в нем что-то не так. Как и в большинстве договоров. Отсутствие гибкости, множество повторений. Смотрю на него и думаю: почему бы не разбить его на подпрограммы (которые мы называем параграфами). И то, что обычно вначале идут такие и такие определения, которые используются бла-бла-бла...

Сейбел: Поговорим о буднях программирования. Как вы проектируете код? Как структурируете его? Может быть, ваша недавняя работа над портированием XScreenSaver под OS X послужит примером?

Завински: Сначала я для затравки создаю маленькие демонстрационные программы, которые больше никем и никогда не используются. Делается это только для того, чтобы выяснить, как расположить окно на экране, и тому подобное. Поскольку я реализую протокол X11, то прежде всего беру один из скринсейверов и составляю список всех вызовов X11, которые он делает.

Потом я делаю для каждого из них заглушку и начинаю понемногу их заполнять, постепенно выясняя, как буду реализовывать тот или другой вызов.

На другом уровне, со стороны Мака, начинаем все с самого начала. Как расположить окно на экране? Затем в какой-то момент приходится прибегать к Xcode. При этом сложнее всего понять, как поднять и заставить работать систему сборки нормальным образом. Приходится экспериментировать, крутить то так, то сяк. Потом думаешь, может поместить этот кусок кода выше, чтобы он обращался вот к этому куску? А может быть, стоит вывернуть это наизнанку? Приходится перетасовывать немало кода, пока в голове не сформируется разумный поток управления. Потом я чищу код, перемещаю его в более подходящие файлы, так чтобы вот этот кусок кода был вместе с вот этим куском.

Это как рисовать жирные стрелки на схеме. Потом я перехожу к следующему скринсейверу, а он использует другие три функции, которых не было в предыдущем, поэтому мне их тоже нужно реализовать. Каждая из этих задач достаточно проста. Но над некоторыми из них приходится попотеть, потому что в API для X11 есть миллион настроек для отображения текста на экране или для поворота прямоугольника. Постепенно этот кусок кода становится все сложнее. Но большинство задач довольно просты.

Сейбел: Итак, для каждого обращения к X11 вы пишете свою реализацию. А вам никогда не казалось, что у вас набирается куча практически одинакового кода?

Завински: Конечно. После двух-трех раз вырезания и вставки похожего кода думаешь: ага, пора остановиться и поместить этот код в подпрограмму.

Сейбел: Представим, что вы снова работаете над проектом такого же масштаба, что и почтовый клиент. Вы говорили о том, что пишете несколько абзацев текста и список функциональных возможностей. Это практически все, что вы сделаете до начала работы над кодом?

Завински: Да. Может, добавлю небольшое описание различий между библиотекой и клиентской частью. А может, и нет. Если бы я работал один, то я бы с этим не заморачивался, потому что для меня это очевидные вещи. Далее, первое, что я бы сделал, – это решил, сверху или снизу начинать. Можно начать так: расположить на экране окно с несколькими кнопками, а уже потом зарываться в детали и писать логику работы этих кнопок. А можно начать с другого конца – с разбора и сохранения почтовых сообщений. Можно начать с любой стороны или одновременно с двух сторон и затем встретиться посередине.

Я заметил, что получение чего-либо на экране как можно раньше помогает мне лучше сосредоточиться на задаче и понять, что делать дальше. Ведь если смотришь на огромный список задач и не знаешь, за что взяться, то на самом деле не важно, за что возьмешься в первую очередь. Но если есть на что конкретно смотреть, пусть даже это вывод отладочной информации синтаксического анализатора почтовых сообщений, – совсем другое дело! Это уже что-то: куда будем двигаться дальше? Ладно, вместо отображения этой древовидной структуры можно заняться генератором HTML или чем-то в этом духе. Или сделать более детальный разбор заголовков. Просто ищешь очередную задачу, которую нужно решить.

Сейбел: Вы применяете рефакторинг для сохранения целостности внутренней структуры кода? Или с самого начала способны четко представить, как различные части кода будут взаимодействовать между собой?

Завински: Обычно я достаточно хорошо представляю это. Очень мало таких случаев, когда я говорил себе: «Да я же сделал все шиворот-навыворот! Придется все переставить». Но иногда это происходит.

Когда пишу первую версию программы, я стараюсь поместить все в один файл. Потом я начинаю видеть в этом файле структуру. Вот эти блоки очень похожи. Вот в нем уже тысяча строк, так почему бы не переместить что-то в другой файл? Да и API в этом случае получается естественнее. Проектирование – это непрерывный процесс, никогда не знаешь, насколько проект хорош, пока программа не будет готова. Поэтому я предпочитаю как можно скорее снять пробу, получить что-нибудь на экране и посмотреть на это со всех сторон.

И потом, начав писать код, внезапно понимаешь: «Нет, это глупая идея. И почему я решил, что этот модуль сделать легко, когда на самом деле это гораздо сложнее?» Это такие вещи, которые не понять, пока не начнешь писать код и не почувствуешь, как они от тебя ускользают.

Сейбел: Каковы признаки того, что какие-то вещи от тебя ускользают?

Завински: Когда погружаешься во что-нибудь с мыслью «Так, здесь я за полдня напишу кусок кода такой-то длины», а потом приступаешь к работе и постепенно начинаешь чуять недоброе: «Ага, нужен еще кусок, надо бы и за него взяться. Да тут работы выше крыши!»

Сейбел: Я заметил, чем хороший программист отличается от плохого: хороший программист легко переходит от одного уровня абстракции к другому, он способен не смешивать эти уровни при внесении изменений и точно определяет уровень, на котором эти изменения нужно внести.

Завински: В том, где и что размещается, нужно придерживаться определенного стиля, это очень важно во всех отношениях. Решить какую-нибудь проблему на уровне, более близком к пользователю, или внести какое-то, возможно, более крупное изменение, которое отразится на всей системе? Любое из этих решений может быть правильным, и очень сложно понять, какое из них какое. Мне нужно сделать какое-то изменение, но является ли оно единичным частным случаем или таких случаев будет 12?

Думаю, одна из самых важных вещей, по крайней мере для меня, когда создаешь что-то с нуля, заключается как раз в том, чтобы как можно скорее довести программу до состояния, когда ты, программист, сможешь ее использовать. Хотя бы немного. Это подскажет, что делать дальше, – ты буквально почувствуешь, что нужно делать. Когда что-то уже есть на экране и есть кнопка, связанная с некоторой функцией, появляется ощущение, какая кнопка будет следующей. Конечно, это GUI-центричное описание того, что я имею в виду.

Сейбел: Мы немного говорили об ужасных ошибках, с которыми вы сталкивались, например та история с GNB. Но давайте еще немного поговорим о процессе отладки. Для начала, какие инструменты вы предпочитаете? Инструкции печати отладочной информации? Отладчики исходного кода (symbolic debugger)¹? Формальные доказательства корректности?

Завински: За последние годы многое изменилось. Когда я работал на Лиспе, процесс отладки заключался в запуске программы, ее остановке и последующем изучении данных. Был специальный инструмент, который позволял копаться в памяти, и я изменил его таким образом, что все эти функции стали доступны через цикл Чтение-Вычисление-Печать (то есть через Lisp Listener). Когда этот инструмент выводил содержимое объекта, появлялось контекстное меню, щелкнув на котором, можно было получить значение этого объекта. Все это упрощало следование по цепочкам объектов и всякое такое. Я уже тогда думал о подобных вещах: погрузиться в середину кода, искать, экспериментировать.

Позже, уже работая на Си и используя GNB в Emacs, я попытался сделать то же самое. Именно исходя из этой модели мы создавали Energize. Но мне кажется, что он так никогда нормально и не заработал. Со временем я вообще прекратил пользоваться такими инструментами, а просто вставлял инструкции печати и запускал все снова. И так раз за разом, пока не исправишь ошибку. Интересно, что при переходе на все более и более примитивные среды, такие как JavaScript, Perl, это становится единственным вариантом, поскольку там нет никаких отладчиков.

В те дни люди вообще слабо представляли, что такое отладчик. «Да зачем он тебе нужен? Что он делает – добавляет за тебя инструкции печати? Не понимаю. Что за странные слова ты говоришь?» В те дни отладка в основном заключалась в инструкциях печати.

Сейбел: Имела ли тут значение разница между Лиспом и Си? Помимо инструментов одно из отличий в том, что в Лиспе можно тестировать маленькие кусочки. Можно вызвать небольшую функцию, если в правильности работы есть сомнения, остановить ее на середине и посмотреть, что происходит. А в Си запускаешь программу целиком, во всем ее величии и сложности, и задаешь точку останова в каком-нибудь месте.

¹ Сейчас все уже пользуются отладчиками, позволяющими проходить по исходному тексту программ, но так было не всегда. В то время, о котором идет речь, не все среды разработки имели подобные отладчики. – *Прим. науч. ред.*

Завински: Лисп и аналогичные языки позволяют в этом отношении больше, чем Си. Perl, Python и им подобные в этом смысле немного более похожи на Лисп, но все равно мало кто так делает.

Сейбел: Но ведь GNB дает возможность заглянуть внутрь. Чем он вам не подходит?

Завински: Мне он всегда казался неприятным. Отчасти из-за того, что имеет отношение к Си. Я анализирую массив и вдруг вижу кучу чисел, и нужно лезть туда и вернуть все к нормальному виду. Он никогда не работал правильно, как мог бы работать с нормальным языком.

Сейбел: В то время как в Лиспе, если вы смотрите на массив, он выводится как нужно, поскольку отладчик знает, что есть что.

Завински: Вот именно. Мне всегда казалось, что GNB прыгает вниз-вверх, и в стеке все оказывается перепутанным. Идешь вверх по стеку, а нижняя часть стека из-за проблем в GDB изменяется. Или думаешь, что в регистре должно быть одно значение, но поскольку находишься в другом кадре стека, оно оказывается совсем другим.

Такое чувство, что я не могу по-настоящему доверять сведениям отладчика. Он что-то вывел, посмотрим – так, это число. Правильное оно или нет? Неизвестно. Зачастую вообще оказываешься без всякой отладочной информации. Берешь кадр стека, и кажется, будто у этой функции нет аргументов. Минут десять пытаешься вспомнить, через какой регистр передается нулевой аргумент. Потом сдаешься, заново компонуешь программу и добавляешь инструкцию печати.

Похоже, со временем инструменты отладки становятся все хуже и хуже. С другой стороны, люди наконец-то начинают понимать, что распределение памяти вручную – тупиковый путь. Сегодня это уже неактуально, поскольку наиболее сложные ошибки, когда приходится закапываться в структуры данных, происходят редко, ведь в Си они обычно были связаны с проблемами повреждения памяти.

Сейбел: Вы используете операторы утверждений (assertions) или другой более-менее формальный способ документирования или проверки инвариантов?

Завински: Мы тогда ходили вокруг да около, не зная, как приступить к операторам утверждений в базовом коде Netscape. Очевидно, что добавление операторов утверждений – всегда хорошая идея как для отладки, так и, как вы говорили, для документирования. Этим выражается намерение. Мы добавили множество таких операторов. Но вопрос в том, что произойдет при нарушении утверждения в финальных (не отладочных) версиях? Что тогда? Мы склонились к мысли возвращать нулевое значение в надежде, что программа будет продолжать работать.

Ведь если браузер падает, это действительно плохо, гораздо хуже, чем возврат к циклу ожидания, большие утечки памяти или что-то в этом роде, поскольку все это меньше расстраивает пользователей.

Многие программисты инстинктивно говорят: «Выдавайте сообщение об ошибке!» Нет, не нужно. Никого это не волнует. С такими проблемами гораздо легче справиться в языках, поддерживающих исключения, таких как Java. В таких языках просто перехватываешь все исключения на самом верхнем уровне и готово. И не нужно беспокоить пользователя сообщением о том, что какое-то значение равно нулю.

Сейбел: Вы когда-нибудь просто проходили программу пошагово – для отладки или, как некоторые советуют, для проверки написанного кода?

Завински: Нет, не совсем. Обычно я использую пошаговое выполнение для отладки. Пожалуй, иногда для проверки правильности написанного кода. Но не часто.

Сейбел: Так что же вы делаете при отладке?

Завински: Сначала пробегаю глазами код, читаю его, пока не подумаю: «Так, этого не может быть, все должно работать правильно». Тогда я добавляю некоторый код для проверки и разрешения этого противоречия. Либо если, читая код, я не вижу никаких проблем, то запускаю его на выполнение, останавливаю где-нибудь в середине и смотрю, что происходит. Сложно говорить об этом вообще. Ситуации бывают разные.

Сейбел: Что касается операторов утверждений – насколько формально вы к ним подходите? Кто-то использует стандартное утверждение: «Я считаю, что в этом месте кода некоторое условие должно быть истинным». А кто-то мыслит более формально: у функций есть предусловия, постусловия и есть глобальные инварианты. Какова ваша позиция?

Завински: Я точно не думаю об этом в контексте математического доказательства корректности. Скорее я за стандартные утверждения. Конечно же, всегда полезно, получив входные данные в функции, хотя бы приблизительно понимать, какие у них ограничения. Может ли это быть пустая строка? Что-то в таком духе.

Сейбел: Тестирование – тема, весьма близкая к отладке. В Netscape была специальная группа обеспечения качества или вы все тестировали сами?

Завински: И то и другое. Мы все время запускали свои программы – это лучший способ проверки качества на месте. Но была и группа обеспечения качества, у которой были формальные тесты. И каждый раз перед выпуском новой версии они все проверяли по списку. Перейти на такую-то страничку, щелкнуть там-то. Вы должны увидеть это. Или не должны увидеть.

Сейбел: А как насчет тестов на уровне разработчика, таких как модульные тесты?

Завински: Нет, у нас такого не было. Для некоторых вещей я делал нечто похожее. У синтаксического анализатора дат для почтовых заголовков был огромный набор сценариев тестирования. В те дни мало кто обращал внимание на стандартизацию, поэтому в заголовках могла приходиться всякая ерунда. И что бы там ни было, пользователям не нравится, когда их почта сортируется неправильно. Поэтому я собрал целую кучу примеров, создал тесты и получил громадный список плоховато отформатированных дат и чисел, в которые, как я считал, эти даты должны преобразовываться. Каждый раз, внося изменения в код, я запускал тесты, и некоторые из них падали. Так должен я соглашаться с ними или нет?

Сейбел: Переросло ли это в какую-то форму автоматического тестирования?

Завински: Нет, когда я писал такие модульные тесты для своего кода, они запускались в основном лишь тогда, когда я сам запускал их. Потом мы немного занимались этим, когда работали над проектом Grendel, переписывая почтовый клиент на Java, поскольку там гораздо легче писать модульные тесты при разработке новых классов.

Сейбел: Оглядываясь назад, как вы считаете, ваша команда пострадала от этого или нет? Упростилась бы или ускорилась разработка, если бы вы были дисциплинированнее в вопросах тестирования?

Завински: Не думаю. Мне кажется, это бы только замедляло нас. Можно долго говорить о том, как все делать правильно с первого раза. Первоначально мы были зациклены на скорости. Нам нужно было выпустить продукт, пусть даже несовершенный. Мы могли бы выпустить его позже, и его качество было бы выше, но к тому времени нас мог бы кто-нибудь опередить.

Довольно часто говорят, что дело пойдет быстрее при наличии модульных тестов, меньших модулей и тому подобное. Все это хорошо звучит в теории. При неспешном темпе разработки это действительно отличный вариант. Но когда тебе говорят: «Нам нужно сделать это с нуля за шесть недель», – приходится чем-то жертвовать, иначе не справиться. И я выбрасываю то, что не является абсолютно критичным. Модульные тесты не критичны. Пользователь не будет жаловаться на то, что у вас нет модульных тестов. Это ваше личное дело.

Не хочу, чтобы все решили, будто я считаю тесты ерундой. Нет. Это всего лишь вопрос приоритетов. Вы пытаетесь написать хорошую программу или пытаетесь закончить к следующей неделе? Сделать то и другое одновременно не получится. У нас в Netscape была такая шутка: «Мы

стопроцентно преданы качеству. Мы собираемся выпустить продукт самого высокого, насколько у нас получится, качества к 31 марта».

Сейбел: Отсюда другая тема – сопровождение программного обеспечения. Как вы разбираетесь в коде, который не сами писали?

Завински: Я просто беру его и читаю.

Сейбел: С чего вы начинаете? Читаете его последовательно, с первой страницы?

Завински: Когда как. Чаще всего приходится изучать, как использовать новую библиотеку или набор инструментов. Если повезет, будет даже какая-нибудь документация. Есть некий API. Разбираешься с отдельным фрагментом, чтобы понять, как он работает, или выясняешь, как что-то реализовано. Так и прокладываешь себе путь через все это. С такими инструментами, как Emacs, можно начать снизу. Из чего состоят списочные ячейки (cons cells)? Как это выглядит? От этого можно танцевать. Иногда знакомство с системой построения позволяет понять, как все это сочетается. Мне всегда казалось, что, для того чтобы погрузиться в кусок кода, нужно взять некоторую задачу и попытаться ее решить.

С такими инструментами, как Emacs, можно взять существующий модуль и распотрошить его. Так, вот кусок кода. Вычленим часть, которая действительно что-то делает, и получаем шаблон. Теперь мы знаем, на что похож компонент этой системы, и можем вставлять туда свои куски кода. Это как разобрать его по косточкам.

Сейбел: В Emacs вы в конце концов переписали компилятор байт-кода и части байт-кода виртуальной машины. Мы уже говорили о том, что значительно приятнее что-то переписать заново, чем исправлять, но это не всегда подходит. Не понимаю, как вы перешли эту черту? Вы решили переписать компилятор целиком, потому что это проще, чем исправить только некоторые его части? Или просто подумали: «Ух ты! А здорово будет написать компилятор!»

Завински: Так уж вышло, что в результате компилятор был переписан полностью. Я начал с внесения исправлений и попыток оптимизации. Но в конце концов в нем не осталось ничего от первоначального варианта. Я использовал тот же API, пока в нем не отпала необходимость. По-моему, компилятор байт-кода вышел отлично, отчасти потому, что это был изолированный модуль с единственной «точкой входа»: скомпилировать и сохранить.

Конечно, многое из того, что я добавил в Lucid Emacs, было не столь полезным. На самом деле, многие мои действия объяснялись желанием приблизить все это к Лисп-машинам и к более привычным для меня версиям Emacs и средам разработки под Лисп. И я приложил много

усилий, чтобы Лисп в Emacs был более совершенным: например, чтобы использовались объекты-события, а не списки чисел. Использовать списки чисел вместо объектов-событий – примитив и безвкусица. Теперь понятно, что эти изменения были одной из главных проблем, поскольку приводили к несовместимости с библиотеками сторонних разработчиков.

Сейбел: И вы, конечно, не знали о скором появлении двух разновидностей Emacs¹.

Завински: Разумеется. Но и без того уже были две версии Emacs – 18 и 19. Так или иначе, проблема совместимости все равно бы возникла. Оглядываясь назад, я понимаю, что если бы осознавал последствия внесенных мной изменений, то сделал бы все по-другому. Или потратил бы больше времени на то, чтобы старый вариант тоже работал. Как-то так.

Сейбел: Вы уже упоминали о написании легко читаемого кода, что непосредственно касается сопровождения. Так что же делает код легко читаемым?

Завински: Конечно же комментарии. Запись предположений и того, что код делает. Если речь идет о создании структуры данных, то описание ее устройства. Я много раз убеждался, что это полезно. Это особенно полезно при написании кода на Perl, где все обстоит примерно так: это хеш-таблица, а значения представляют собой ссылки на списки, поскольку структуры данных в Perl – такая вот ерунда. Нужно ли использовать стрелку вправо (->), чтобы добраться до этого? Думаю, такие примеры не помешают.

Я всегда советовал писать больше комментариев, но меня раздражает, когда комментарий представляет собой перефразированное имя функции. Функция называется `push_stack` (добавить элемент в стек), а комментарий – «добавление элемента в стек». Большое спасибо.

В комментарии можно сказать то, что неочевидно из кода. Иначе зачем он нужен? Это должно быть высокоуровневое или низкоуровневое описание в зависимости от того, что важнее. Иногда самое важное – для чего вообще предназначен данный элемент. Зачем я его использовал? А иногда самое важное – диапазон допустимых входных значений.

Длинные имена переменных. Я не сторонник венгерской нотации, но предпочитаю использовать для описаний реальные английские фразы (кроме переменных цикла, там и так все ясно). По возможности, чем больше в них слов, тем лучше.

¹ Имеется в виду раскол 1991 года, когда ряд разработчиков GNU Emacs отделились для развития собственной ветки под названием XEmacs. – *Прим. науч. ред.*

Сейбел: А что насчет организации кода? В конце концов, организация кода является линейной, но программы по своей сути нелинейны. Вы пишете сверху вниз или снизу вверх?

Завински: Обычно я располагаю элементы нижнего уровня в верхней части файла и стараюсь придерживаться этого порядка. Также в самое начало файла я обычно помещаю то, что нужно для API: высокоуровневые точки входа этого файла, описание данного модуля и так далее. В объектных языках все это делает за вас язык программирования. А с Си приходится больше действовать самостоятельно. На Си я стараюсь, чтобы для каждого .c-файла был соответствующий .h-файл, который содержит все внешние объявления. Все, что не экспортируется из .h-файла, является статическим. Бывает, я возвращаюсь и думаю: «Стоп, мне надо вызвать вот это», – и вношу соответствующее изменение. Но это делается намеренно, а не просто случайно.

Сейбел: Вы помещаете элементы нижнего уровня в начало файла. А код пишете так же? Начиная снизу?

Завински: Не всегда. Иногда я начинаю сверху, иногда снизу, когда как. Иногда я знаю, какие строительные блоки мне понадобятся, и сначала собираю именно их. А иногда сначала вижу решение только в общих чертах и лишь потом начинаю углубляться в детали. Бывает и так, и так.

Сейбел: Предположим, чисто теоретически, что вы собираетесь вернуться к программированию и набираете команду разработчиков. Как вы организуете их работу?

Завински: Думаю, нужно разбить всех на группы, максимум по три-четыре человека, которые будут работать в тесном ежедневном сотрудничестве. Этот вариант отлично масштабируется. Допустим, есть проект, который можно разделить на двадцать пять реально изолированных модулей. Значит, у вас может быть двадцать пять небольших команд – хотя это уже многовато. Скажем, десять команд. Пока они могут координировать работу друг с другом, не думаю, что здесь так уж много ограничений для роста. Просто постепенно это начинает походить скорее на несколько проектов, чем на один.

Сейбел: Итак, у вас несколько команд максимум из четырех человек каждая. Как вы будете координировать их действия? С помощью главного архитектора, который будет управлять зависимостями и выступать посредником между этими командами?

Завински: Здесь требуется согласовать интерфейс между модулями. Для того чтобы такой модульный подход работал, интерфейс между модулями должен быть простым и понятным. Тогда есть надежда, что согласование интерфейса пройдет без лишних воплей и будет проще

выполнять обязательства по модулям. Думаю, лучший способ добиться нормального взаимодействия между модулями – сделать это взаимодействие максимально простым. В этом случае будет меньше возможностей для ошибок.

А сам процесс разделения на модули целиком зависит от проекта. Для некоторых типов веб-приложений первая ваша команда будет работать над пользовательским интерфейсом, вторая – над базой данных, третья – над кодом, выполняемым на сервере, четвертая – над кодом, который выполняется на клиентской машине. То же самое для настольного приложения. Форматы файлов, графический интерфейс пользователя, базовая структура команд.

Сейбел: Как вы распознаете таланты?

Завински: Даже не знаю. Я никогда не был тем, кто принимает людей на работу. А если и участвовал в собеседованиях, то мне всегда казалось, что у меня нет никаких идей по этому поводу. После собеседования я мог сказать, смогу или нет работать с этим человеком, но сказать, хороший это работник или нет, всего лишь поговорив с ним, я не мог. Мне всегда казалось, что это трудно сделать.

Сейбел: А если работник плохой? Есть ли надежный способ определить это?

Завински: Когда как. Например, я всегда старался избегать ярких сторонников шаблонов C++. Возможно, зря. Может быть, в том контексте, где они использовались, это было хорошим решением. Что касается тех, с кем я работал, умение отстаивать свою позицию очень ценилось, потому что все мы были большими любителями поспорить. В той обстановке это сильно помогало. Разумеется, на способность программировать это никак не влияет. Все это из области межличностных отношений.

Сейбел: А в другой команде это умение может оказаться очень вредным.

Завински: Само собой.

Сейбел: Складывается впечатление, что в Netscape вы, ребята, разделили все так, что разные люди владели разными частями системы. Одни считают, что это очень важно. Другие полагают, что для команды предпочтительнее коллективно владеть кодом. Что вы об этом думаете?

Завински: Со мной бывало и так, и так. Оба подхода имеют свои достоинства. Не думаю, что идея коллективного владения кодом так уж практична, поскольку кода бывает слишком много. Людям нужна специализация, иногда необходимы эксперты в конкретных вопросах. Обычно так и получается. Всегда оказывается, что есть код, с которым ты знаком лучше других, так уж вышло, что ты работал над этим модулем больше, чем кто-либо еще. Или оказывается, что какие-то части

системы дороги тебе больше всего. Конечно же, хорошо, когда кто-то еще знаком с твоим кодом, если только ты не собираешься поддерживать свою часть системы вечно. Так или иначе, приходится кому-то доверить свой код. Хорошо, когда знание распространяется. Но также хорошо, когда есть кого винить. Если за что-то отвечают сразу все, значит, ответственного нет.

Сейбел: Вы когда-либо были руководителем?

Завински: Не совсем. Когда я работал над Emacs в Lucid, в Lucid Emacs было много модулей, написанных другими ребятами. Формально они на меня не работали, но в каком-то смысле приходилось руководить. Многим из них явно не хватало опыта, они просто делали то, что хотели, а мои указания сводились к следующему: «Так, мне нужно вот это, но вначале мне нужно вот это и это и отсюда вот это».

Сейбел: И вы давали им свободу действий? Говорили им, что вам нужно А, Б и В, а они уже сами решали, как это сделать?

Завински: Да. Когда мне нужно решить, включать данный модуль в конечный продукт или нет, я формулирую требования к нему. Главное, чтобы эта чертова штука работала. И я мог дать им совет: «Думаю, у вас будет больше шансов, если вы поступите так, а не иначе». Но я хотел: 1) чтобы все работало и 2) без моего участия. Они могли применить какой-нибудь безумный способ, но если он сработал, меня это устраивало, поскольку выполнялся второй пункт: мне не нужно было писать это самому. Собственно, мои советы касались лишь того, будет ли это работать и насколько это разумно.

Сейбел: С другой стороны, когда вы были неопытным программистом, что полезного делали ваши наставники?

Завински: Думаю, важнее всего было то, что они понимали, когда мне пора переходить на следующий уровень. Когда я пришел на работу к Фальману, мне дали какую-то глупую несложную работу. Но постепенно задания становились все серьезнее, хотя на самом деле по-настоящему серьезными они не были.

Сейбел: Это вы о Скефе, который только нависал над вами и говорил: «Ошибка»? Наверное, его компенсировал кто-нибудь, кто относился к вам более заботливо?

Завински: Ну, он не был совсем уж троглодитом. Мог и что-то полезное рассказать. Я же читал много кода и задавал кучу вопросов. Думаю, очень важно не бояться собственного незнания. Если не понимаешь, как что-то работает, спроси у того, кто понимает. Многие боятся. А что тут такого? Ведь чего-то не знаешь не потому, что тупой, а просто пока не знаешь.

Сейбел: Вы читали код, потому что работали над ним или просто хотели понять, как он работает?

Завински: Мне всегда было интересно выяснить, как что-то работает. Именно со стремления вникнуть в детали для человека начинается путь в эту профессию.

Сейбел: Вы не из тех, кто в детстве курочил тостеры?

Завински: Да. Я сделал телефон и выяснил, как набирать номер телеграфным ключом, который сделал из консервной банки. В детстве у меня были эти старые книжки, с барахолки или еще откуда, вроде «Науки для мальчиков», издания 1930-х годов; помнится, они сильно меня раззадорили. В 1920–1930-х существовала настоящая «хакерская» культура: советы, как протянуть телеграф от дома к сараю, как изготовить лейденскую банку...

Сейбел: Тогда другой стандартный вопрос: как программист, вы считаете себя ученым, инженером, художником, ремесленником или кем-то еще?

Завински: Точно не ученым и не инженером, поскольку эти понятия имеют очень четкие определения. Я не занимаюсь математикой, не создаю проекты, не провожу доказательств. Скорее, что-то среднее между ремесленником и художником, зависит от проекта. Я часто пишу скринсейверы – это ближе к искусству, к созданию красивых картин. Что-то из той же области.

Сейбел: Как считаете, вы овладели компьютерной наукой или только программированием?

Завински: За годы я почерпнул многое из компьютерных наук. Но целью было научиться программировать, заставить машину что-то выполнять. Компьютерная наука была лишь инструментом для этого.

Сейбел: Вы когда-нибудь считали это упущением? Не было желания получить более систематическое образование?

Завински: Да, были времена, особенно в Lucid, когда я осознавал, что все, о чем говорят эти парни, покрыто для меня мраком, потому что это никогда мне не требовалось и прошло мимо. Потом я освоил терминологию, разобрался в общих чертах, о чем они говорят, и немного читал, когда что-то нужно было узнать. Конечно же, иногда, особенно в молодости, я говорил себе: «Господи, я же ничего не знаю». Просто минутная слабость, но был и испуг. Мальчишка среди всех этих парней с учеными степенями – «Ааааа, я ничего не знаю! Я идиот! Что я здесь делаю?»

Моя жизнь была бы иной, если бы я проучился подольше. Но в тот момент я мог поступить только так, как поступил.

Сейбел: А бывало наоборот – когда вы чувствовали, что в настоящем программировании понимаете гораздо лучше окружающих вас компьютерных ученых?

Завински: Да, причем довольно часто. Но без этих мыслей вроде «Вы ребята выбрали не тот путь» или «Нас просто интересуют разные вещи». Я не хочу быть математиком, но математиков критиковать не собираюсь.

Весьма странно, что люди часто путают эти профессии. Они думают, что разработчик с огромной теоретической базой размышляет точно так же, как разработчик настольных приложений. У них мало общего.

Сейбел: Вы по сути самоучка. Что вы можете посоветовать программистам-самоучкам?

Завински: Тут все не так просто, поскольку мир сильно изменился. Я всегда странно себя чувствую, когда заходит разговор на тему «Как это было у меня». Я не знаю, правильно ли я тогда поступил. А люди всегда воспринимают это в смысле «Делай, как я».

У меня это вышло случайно. Так получилось, вот и все. Я принял решения, которые привели к другим решениям, и вот результат.

Я то и дело получаю электронные письма примерно такого содержания: «Я хочу стать программистом, что мне делать?» Или: «Идти мне в колледж или нет?» Что я могу ответить? В 1986 году я бы дал однозначный ответ. Но сегодня никто не сможет пойти по моему пути, потому что его больше нет.

Десять лет назад я бы говорил, что перво-наперво следует изучить язык ассемблера. Нужно осознать, как работает компьютер. Так ли важно это сейчас? Не знаю. Может быть, да. А может, и нет. Может, через десять лет все программное обеспечение будет представлять собой веб-приложения или распределенный код, выполняемый на арендованных кластерах, части которого, находясь на десятке серверов Google, порождают собственные копии и воссоединяются после получения результатов. Нужно ли будет для этого знать ассемблер? Или все настолько абстрагируется, что это уже будет неважно? Я не знаю.

Я был поражен, когда узнал, что люди получают дипломы программистов, не написав ни строки на Си. Начав с Java, они там и остались. Это кажется неестественным и неправильным. Но откуда мне знать, правильно это или нет? Может, дико думать: «А мы в свое время программировали с помощью девятивольтовой батареи и твердой руки!»

Сейбел: А что скажете о книгах? Есть ли книги по компьютерным наукам или программированию, которые должен прочесть каждый?

Завински: Я читал их не так много. Я всегда рекомендую книгу «Structure and Interpretation of Computer Programs»¹, которая многих пугает, потому что там много о Лиспе. Но я думаю, что она действительно учит программированию, а не конкретному языку. Мне кажется, что многие курсы для начинающих делают упор на синтаксис. Я видел это в старших классах и на вводных курсах, когда недолго учился в Карнеги–Меллоне.

Они обучают не программированию, а тому, куда поставить точку с запятой. Это отталкивает людей от программирования, потому что это не самая интересная его часть. Даже для тех, кто знает, что делает.

Была еще книга – как же она называлась? – про отладку, ее написал кто-то из Microsoft. О том, как эффективно использовать утверждения. Помню, мне она показалась стоящей, но не потому, что я узнал из нее много нового, а потому, что именно эту книгу хочется дать прочесть идиоту-коллеге.

Была и еще одна книга, которую все считали величайшим трудом своего времени: «Design Patterns»². По-моему, отстой. Там учили программировать методом «вырезания и вставки». Вместо того чтобы подумать над своей задачей, берешь сборник рецептов, ищешь там что-то более-менее похожее и просто пытаешься это воспроизвести. Это не учебник программирования, а книжка-раскраска. Но, похоже, многим она нравилась. И при встрече они перекидывались словечками из этой книги – такой паттерн, сякой паттерн. А вы о чем – о цикле? Ясно.

Сейбел: Есть ли ключевые навыки, необходимые каждому программисту?

Завински: Любопытство, желание разобрать что-то по косточкам. Стремление узнать, что там внутри происходит на самом деле. Думаю, это основа программирования. Без этого далеко не пойдешь. Это основной путь получения знаний. Чтобы научиться создавать что-то свое, нужно разобраться в чем-то уже готовом, посмотреть, как оно устроено. По крайней мере, для меня это так. Я очень мало читал компьютерной литературы. Я учился, копаясь в исходниках и читая документацию. У меня была цель, для достижения которой я должен был знать, что делает эта штука и вот эта. Я просто шел наугад, пока не понимал, куда мне нужно.

¹ Абельсон Х. «Структура и интерпретация компьютерных программ». – М.: Добросвет, 2006.

² Гамма Э. и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – СПб.: Питер, 2007.

Сейбел: Вы читали «The Art of Computer Programming» Кнута¹?

Завински: Нет. Это одна из тех книг, которую действительно стоило бы прочесть. Но я этого так и не сделал.

Сейбел: Читать ее достаточно сложно. Для ее понимания требуется хорошая математическая подготовка.

Завински: А я совсем не математик.

Сейбел: Довольно интересно, что многие программисты выходят из математиков и компьютерная наука прочно опирается на математические теории. А вы являетесь живым доказательством того, что это не обязательно. Какой объем математических знаний необходим, чтобы стать хорошим программистом?

Завински: Это зависит от того, где провести границу, определяющую, что относится к математике, а что нет. Сопоставление с образцом (Pattern matching) – это математика или нет? Понимание порядка величин и основ комбинаторики должно быть на интуитивном уровне. Но я полностью уверен, что элементарный математический тест я бы провалил. Ведь я давно не занимался чем-то настолько формальным.

Разве что в старших классах на уроках математики. Алгебра, немного матанализа. У меня не очень-то хорошо получалось. Я осилил это, но по-настоящему своим так и не ощутил. В старших классах была физика и основы механики. На лабораторных работах мы двигали бруски по наждачной бумаге и все такое. На уроках физики я выглядел просто ужасно, чувствуя себя полным идиотом, а ведь мне все это нравилось. Лабораторные работы я делал отлично, у меня здорово получалось. После этого на математику меня просто не хватало.

Я нашел решение, которое, как я догадывался, было абсолютно неверным. Я сдал работу, так и не поняв, в чем ошибка. Я получил зачет, потому что данные были собраны правильно и кое-что я пересдал позже. Просто математика никогда не была моей сильной стороной.

Конечно, я не стану утверждать, что программисту это вообще не нужно. Есть разные виды программирования. Если бы все были такими, как я, никакого программирования вообще бы не было. Но лично мне кажется, что программирование ближе к литературе, чем к математике. Словно пишешь рассказ, пытаешься передать его смысл очень тупому парню с ограниченным словарем, то есть компьютеру. У тебя есть некоторая идея, которую нужно объяснить, и ограниченные средства для объяснения. Какие подберешь слова, как будет выглядеть введение, заключение? Что-то в этом роде.

¹ Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008.

Здесь важен еще и вкус. Можно описать что-нибудь правильно, а можно не только правильно, но и красиво. Так и с программированием. Можно просто решить какую-то задачу, а можно сделать это с умом и красиво.

Сейбел: Зачем это нужно? Для собственного удовольствия? Или красивый код имеет какие-то практические преимущества ?

Завински: По большому счету, понятия «красоты» и «простоты сопровождения» эквивалентны. Или, по крайней мере, тесно связаны. Одна из составляющих красивого кода – это легкая для понимания структура. Ключевая информация собрана в начале или она разбросана повсюду? Возвращаясь к параллели с литературой: можно ли найти какой-то эпизод, перелистывая роман? «Это где-то ближе к середине, потому что именно там об этом шла речь». Или об этом говорится по всей книге. С программированием во многом так же.

Сейбел: Считаете ли вы, что сегодня, для того чтобы стать успешным программистом, требуются другие навыки?

Завински: Разумеется, сегодня уже нельзя написать программу полностью с нуля без каких-либо внешних зависимостей. Из-за бурного роста инструментов, библиотек, фреймворков и тому подобного сегодня даже самая элементарная программа нуждается во всем этом. Все как-то разом изменилось. Сегодня все что угодно должно быть веб-приложением. В мое время этого не было.

Так что умение разобраться в чужом коде и понять, как им пользоваться, стало еще важнее. Поэтому подход, вроде «я не понимаю этого, так что напишу-ка это заново», остался в прошлом. Тогда ты мог сделать это, и неважно, была ли эта идея хороша или нет. Сейчас с этим намного сложнее.

Сейбел: А как насчет стремления разобрать все по косточкам и понять, как это устроено, – так ли важна его роль в наши дни? Если сегодня захочешь проанализировать каждый кусок кода, с которым работаешь, то увязнешь в этом навсегда. Видимо, сейчас надо занимать такую позицию: «В целом я понимаю, как это работает, и пусть пока так и будет, а если очень понадобится, разберусь в этом поподробнее».

Завински: Да. Когда появились такие тенденции, первая моя мысль была о том, что у нас вырастает поколение программистов, которые ничего не будут знать об эффективности и о том, куда расходится память. Когда такой программист наконец заметит: «Ого, моя программа стала просто огромной», что он сможет с этим сделать? Он даже не сможет понять, с чего начать. Вот о чем я сразу подумал, но ведь я – пещерный человек. Может, это и неважно, ведь теперь достаточно просто добавить памяти и все в порядке.

Сейбел: А может, люди будут глубже изучать те или иные вещи. Например, неважно, выделили мы на это четыре байта или шесть, но важно, сможет ли эта штука выполняться на одном узле кластера или будет задействован также и второй.

Завински: Да, вы правы. В этом смысле программирование сильно изменилось. Раньше внимание уделялось другим вещам. Когда-то я считал каждый байт и задавался вопросами вроде: «Какого размера будут мои объекты? Может, сделать это как-то по-другому, а то этот заголовок массива уже можно просчитать». Что-то в таком духе. Теперь уже никто об этом не думает. Трюк с сохранением в одном слове двух указателей с помощью операции XOR стал для всех темным лесом. Зачем кому-то использовать что-то подобное? Это безумие. Сегодня на передний план выходят другие навыки. Сегодня очень важно уметь разбираться в API и понимать, что там нужно, а что нет.

Сейбел: Если бы вам было 13 лет, вы бы увлеклись программированием – тем, каким оно стало сегодня?

Завински: Трудно сказать. Я не знаю сегодняшних тринадцатилетних, не знаю, как для них выглядит этот мир. Во всем этом стало сложнее разбираться. Нынешний десятилетний мальчишка не станет курочить свой мобильник, чтобы понять устройство микрофона, как я ребенком делал это со своим телефоном. Сегодня уже ничего не починишь самостоятельно.

Думаю, я пришел к программированию благодаря исследованиям вроде этого: снять крышку магнитофона и посмотреть, как крутятся шестеренки. А сегодня есть только наборы LEGO для создания роботов. Пожалуй, этого мало, чтобы пройти подобный путь. Но, как уже сказал, возможно, я ошибаюсь, ведь я не знаю, чем живут сегодняшние тринадцатилетние. Я не знаю современных игрушек. Видеоигры, радиоуправляемые машинки... Конструкторов хороших я не видел. Грустно все это.

Сейбел: С другой стороны, программирование стало гораздо доступнее. Вам уже не нужно осваивать все премудрости ассемблера, прежде чем заставить компьютер делать что-то классное.

Завински: Да. Наверное, современные ребята, которые сегодня учатся программированию, начинают с создания веб-приложений, разработки плагинов для Facebook и тому подобного. Брэд Фицпатрик, создатель Живого Журнала (LiveJournal), – мой друг. LiveJournal родился, когда он просто валял дурака и написал скрипт на Perl, чтобы вместе с друзьями сообщать друг другу что-то вроде «Я собираюсь пообедать». Все началось с небольшого скрипта на Perl, который он затем поместил на веб-сервер. Вероятно, это направление будет развиваться и дальше.

2

Брэд Фицпатрик

Брэд Фицпатрик – самый молодой из моих собеседников и единственный, кто никогда не жил в мире без Интернета или персональных компьютеров. Родился в 1980 году, очень рано занялся программированием – уже в пять лет стал учиться программировать на самодельной копии Apple II. Когда Брэд был подростком, интернет-революция уже развернулась в полную мощь, и он принял в ней активное участие: в старших классах разработал свой первый коммерческий веб-сайт, а летом перед поступлением в колледж начал работу над популярным сообществом LiveJournal (Живой Журнал).

Работа с Живым Журналом, популярность которого стремительно росла, заставила Фицпатрика освоить трудную науку создания расширяемых веб-сайтов. Попутно он вместе с другими программистами основанной им компании Danga Interactive разработал несколько программ с открытым кодом, таких как memcached, Perlbal и MogileFS, которые сегодня применяются на серверах, обслуживающих самые загруженные в мире сайты.

Фицпатрик – типичный (пусть и добившийся необычайных успехов) программист рубежа двух столетий: его основными языками программирования были Perl и Си, хотя при необходимости он работал и с такими языками, как Java, C++, Python, JavaScript и C#. Так или иначе, с сетями связано почти все, что он программирует, – более совершенная инфраструктура веб-сайта, новые протоколы и ПО для лучшего учета записей при обновле-

нии блогов, свой мобильный телефон на автоматическое открытие гаражной двери во время мотоциклетного заезда.

Мы говорили о том, как это – учиться программировать, когда еще читаешь книжки про Красного щенка Клиффорда, и почему Брэд рад, что остался учиться в колледже, одновременно работая над Живым Журналом, и о том, как он научился не бояться читать чужой код.

Сейбел: Как вы стали программистом?

Фицпатрик: Мой отец работал в компании Mostek, выпускавшей компьютерную память, и был увлечен компьютерами. Он собрал Apple II практически из запасных частей. Помню, как они с мамой месяцами сидели перед телевизором, собирая его по кускам. Отец мог принести с работы ПЗУ, которые компания не собиралась продавать, потому что в них один или несколько битов залипли в ноль или единицу. Потом им как-то удалось достать ПЗУ Apple II, и они припаивали их на те мертвые чипы, пока наконец не получили работающую память, в которой залипшие биты принимали правильные значения. В результате отцу и многим его коллегам удалось собрать самодельные компьютеры Apple II. Я играл с ним лет с двух или около того и смотрел, как отец на нем программирует.

Сейбел: Он занимался программированием или разработкой железа?

Фицпатрик: Он инженер-электротехник, а программирование – его хобби. Он научил меня программировать, когда мне было пять лет, и шутил, что я заткнул его за пояс лет в шесть или семь. Мама говорит, что я читал библиотечное руководство по программированию Apple II одновременно с книжками про Красного щенка Клиффорда. Вместо «переменные» я говорил «драгоценные». Программирование вместе с отцом – одно из первых воспоминаний в моей жизни. Как он затащил меня в кухню, написал на листке бумаги какой-то код и спросил: «Как думаешь, что делает эта программа?» Там было что-то вроде 10 PRINT HELLO, 20 GOTO 10.

Сейбел: Итак, вы начинали с Бейсика?

Фицпатрик: Да, это был Бейсик. Я не мог работать с мышью, графическими режимами с высокими разрешением и цветами, пока один из друзей нашей семьи не познакомил меня с языком Си и не установил для меня Turbo C. Мне было лет восемь или десять. Отец перешел в Intel в 1984, и мы переехали в Портленд. Он участвовал в разработке процессоров i386 и i486. Отец до сих пор в Intel, и у нас всегда новые классные компьютеры.

Сейбел: Программировали ли вы когда-либо на ассемблере?

Фицпатрик: Да, я немного писал на ассемблере для калькулятора, вроде тех калькуляторов Texas Instruments с процессорами Z80.

Сейбел: Помните, чем вас привлекло программирование?

Фицпатрик: Не знаю. Это всегда было весело. Мама выгоняла меня из-за компьютера, выдавала мне купоны для работы за компьютером, лишь бы я шел на улицу поиграть с друзьями. Друзья говорили: «Опять Брэд за компьютером, вот зануда». А мама: «Иди погуляй».

Сейбел: Помните первую интересную программу, написанную вами?

Фицпатрик: У нас был принтер фирмы Epson, а к нему прилагались огромные толстые руководства с разделом для программистов в конце. Я писал что-то (еще на компьютере Apple) для рисования графики с высоким разрешением, и когда моя программа заканчивала рисовать линии, узоры или еще что-то, я нажимал Ctrl-C. Все это в фоновом режиме сохранялось в неотображаемом кадровом буфере, после чего я загружал другую свою программу, которая читала изображение с экрана и распечатывала его.

А еще раньше я написал программу, которая позволяла мне печатать, как на пишущей машинке: при каждом нажатии клавиши головка принтера перемещалась, а при нажатии backspace двигалась назад.

Одна из первых моих программ делала примерно следующее: сравниваю значение переменной K со следующим введенным символом. Если $K = a$, вывожу на экран значение a, если $K = b$ — значение b. Я добавил условия для каждой буквы и для большинства знаков препинания. Но в какой-то момент понял: «Стоп, я же могу просто написать: „вывести на экран значение переменной“, — и заменил 40 строк кода одной. Я подумал: «Черт, это же здорово!» Неплохое обобщение для шестилетнего.

Вот то, что запомнилось из ранних программ. Потом, в средней школе, я уже писал игры, графические редакторы и редакторы уровней для своих друзей, а друзья делали графику для разных уровней, и мы продавали эти игры своим одноклассникам. Помню, что делал игры, которые определяли режим монитора, EGA или VGA. Если игра не могла использовать режим VGA, то переходила обратно в EGA и использовала другой набор изображений, подходящих для этого режима, так что нам везде требовалось два набора графики. Школьники покупали это дело баксов за пять и устанавливали у себя, а оно не работало, и их родители звонили моим родителям и орали: «Твой сын содрал с моего ребенка пять долларов за эту дрянь, которая не работает». Мама отвозила меня туда и ждала в каком-нибудь тупике, пока я занимался отладкой и исправлял ошибки.

Сейбел: В то время вы посещали какие-нибудь занятия по программированию?

Фицпатрик: Если честно, нет. Была пара книг из библиотеки, ну, и возня с компьютером. У меня не было ни форумов, ни Интернета. Как-то я зашел на BBS¹, но там не было ничего полезного. Она не была подключена к Сети, там только играли в настольные игры.

Сейбел: В вашей школе был факультатив по информатике или что-то подобное?

Фицпатрик: Нет, но у нас были уроки информатики. Вел занятия один парень, а потом я потихоньку стал вести что-то вроде продвинутых курсов. Они до сих пор используют графический редактор и графическую библиотеку, которую я написал; конечной целью курса была разработка игры. Я иногда встречаю этого учителя информатики – моя семья с ним дружит, и мы видимся на футбольных матчах, в которых участвует мой брат. И он говорит мне: «Мы до сих пор используем твою библиотеку».

Я сдал тест повышенного уровня по информатике. Это был последний год, когда тесты сдавали на Паскале, потом все перешли на Си, а еще через год – на Java или что-то вроде того. Я не знал Паскаль, поэтому ходил в соседнюю школу, в которой был факультатив по программированию. Я посещал три-четыре вечерних курса, потом нашел книгу и изучил этот язык. Большую часть времени я рисовал с помощью Паскаля звезды, потому что только что изучил тригонометрию. «О, синус и косинус, забавно. Можно растягивать, сжимать и все такое».

Сейбел: Что вы получили за этот тест?

Фицпатрик: О, я получил пятерку. Мне нужно было создать классы больших целых чисел. Теперь это одно из заданий, которое я даю на собеседованиях: «Создайте класс для произвольных манипуляций с большими целыми, включая умножение и деление». Если я сделал это, будучи старшеклассником, то и кандидаты должны справиться.

Сейбел: Летом, перед началом обучения в колледже, вы работали в компании Intel. Вы работали программистом в конце школы?

Фицпатрик: Да, я работал какое-то время в компании Tektronix. Перед тем как получить официальную работу, у меня была хостинговая учет-

¹ BBS (Bulletin Board System) – электронная доска объявлений. Когда кабельные компьютерные сети были редкостью, пользователи компьютеров широко применяли этот способ общения посредством коммутируемых телефонных сетей. – *Прим. науч. ред.*

ная запись. Но AOL¹ ее прикрыла за создание ботов, зафлуживание их чатов и просто за назойливость. Я тогда написал сценарий, запускающий AOL-клиент из другой Windows-программы. Еще я написал бот, чтобы постоянно заполнять онлайн-форму ввода для получения от них компакт-дисков. Я использовал разные варианты своего имени, чтобы их программа защиты от дублирования не отправила мне всего лишь один диск, поскольку каждый из них предоставлял 100 или 5000 часов бесплатной работы. Я ввел данные несколько тысяч раз, так что где-то неделю почтальон приносил мне пачки дисков.

Мама уже начала говорить: «Черт возьми, Брэд, у тебя будут неприятности!» А я говорил: «А вот и нет, эта долбаная компания сама виновата». Однажды дома раздался звонок, я поднял трубку, чего обычно не делал, – звонили из AOL. Этот тип заорал: «Прекратите присылать нам эти заявки!» Я обычно туго соображаю, но тут сразу нашелся и проорал в ответ: «Зачем вы шлете мне эту хрень?! Каждый день почтальон тащит кучу дисков!» Он даже опешил: «Извините, это больше не повторится». Я украсил ими свою комнату в колледже. Они до сих пор лежат в коробке в гараже. Не могу от них избавиться, потому что все еще помню, как они когда-то классно смотрелись на стенах.

После этого мою учетную запись в AOL заблокировали, а я зарегистрировал учетную запись у одного из местных shell-провайдеров. Тогда я, по сути, и научился работать с UNIX. Я не мог запускать CGI-скрипты, но мог загружать данные на сервер по FTP, поэтому я запускал разную ерунду, написанную на Perl на домашнем компьютере, которая создавала сайт целиком и затем загружала его на сервер. Потом я получил временную летнюю работу в Tektronix. Я хорошо знал Perl и всякие веб-штуки, но никогда не работал с динамическими веб-страницами. Это был 1994 или 1995 год – Сеть тогда была делом совсем новым.

Итак, я пришел в Tektronix. В первый же день они поставили меня перед чем-то и говорят: «Вот твой компьютер». Это была большая рабочая станция SPARCstation или что-то вроде того, где крутились X и Motif. А потом: «Вот твой броузер». Кажется, это был Netscape 2, точно не помню. И еще: «Все свои CGI-скрипты записывай в этот каталог». Помню, как взял какой-то элементарный CGI-скрипт в три строки, чтобы посмотреть его вечером, и подумал: «Черт, здорово-то как!» На следующий день я был на работе уже в шесть утра, просто тронулся на этих CGI.

¹ AOL – America Online, американская медиакомпания, поставщик онлайн-сервисов. – *Прим. науч. ред.*

Потом я занялся динамическим веб-программированием для себя. Где-то в то же время я нашел веб-сервер под Windows с поддержкой CGI. Я убедил своего провайдера – я был на хорошем счету, по крайней мере, вел себя достаточно умно, чтобы мне начали доверять, – и мне сказали: «Ладно, запустим твои CGI, но сначала их проверим». Они просмотрели их и разместили в своем каталоге. Так и начался мой скрипт Voting Booth, где можно создать тему, например «Твой любимый фильм», добавлять нужную информацию и голосовать. Он становился все популярнее. Потом я еще пару лет занимался им параллельно основной работе.

Сейбел: Так был создан сайт FreeVote?

Фицпатрик: Да, он стал сайтом FreeVote после того, как мой компьютер перестал справляться с нагрузкой. В то время реклама на баннерах была уже весьма популярна (или стремительно становилась популярной), и я получал все больше и больше денег, заключал все более выгодные контракты с более высокой платой за щелчок. Я дошел до 27 центов за щелчок на одном баннере, а это и по сегодняшним меркам до нелепости много. Так что я получал до 25–27 тысяч в месяц за дурацкие щелчки на баннерах.

Все это было в старших классах – я занимался этим одновременно с учебой. На Intel я работал в течение двух летних каникул, а в последнее лето перед колледжем приступил к работе над Живым Журналом. На первом курсе колледжа я продал FreeVote приятелю за бесценок – где-то за 11 тысяч, – просто потому, что хотел избавиться от него и от юридической ответственности за него.

Сейбел: Когда вы начали работать с UNIX, это сильно изменило ваш подход к программированию?

Фицпатрик: Да. Но не скажу, что я был от него без ума. Я не мог понять, что происходит в Windows. Видели, наверное, Windows API: по двадцать параметров на каждую функцию, и все это флаги, половина из которых равны нулю. Совершенно непонятно, что происходит. И нельзя заглянуть внутрь, если что-то волшебным образом не работает.

Сейбел: Велика ли разница между вашим первоначальным подходом к программированию, или стилем программирования, и сегодняшним взглядом на эти вещи?

Фицпатрик: Я прошел через множество стилей: сначала объектно-ориентированный, потом функциональный, а затем непонятная смесь того и другого. Вот почему я люблю Perl. С его уродливым синтаксисом, с накопившимся за много лет багажом и изъянами, он никогда не парит мне мозги, задавая стандарт оформления кода. Любой стиль, который вам нравится, хорош. Можно сделать свой код красивым и последова-

тельным, но нет никакого стандарта, определяемого самим языком. Только после перехода в Google я перестал много писать на Perl.

После работы над Живым Журналом я много занимался тестированием. Особенно когда начал работать с другими людьми. Однажды я осознал, что написанный код никуда не исчезает, и мне придется сопровождать его до конца своих дней. Я получаю комментарии к постам в блоге, написанным мною 10 лет назад. «Привет, я нашел этот код и нашел в нем ошибку». И мне внезапно приходится исправлять этот код.

Сейчас я сопровождаю огромный объем кода, над которым работают и другие люди, и если в нем будет хоть какая-то заумь, наверняка найдутся те, кто не сможет понять некоторые мои идеи. Поэтому каждый раз, когда пишу что-то сложное, я обязательно создаю тест, который громко сообщит пользователю о том, что тот запутался. Мне приходилось заставлять многих писать тесты, в основном тех, кто работал на меня. Я пишу тесты, чтобы защитить свой собственный код от поломок, поэтому когда код написан, я говорю: «Вы в самом деле уверены, что он работает? Напишите тест. Докажите мне это». И в какой-то момент человек понимает, что все это окупится, особенно при последующем сопровождении.

Сейбел: Когда вы начали работать с другими людьми?

Фицпатрик: Где-то под конец учебы в колледже. Тогда я стал нанимать других, особенно в Портленде, куда вернулся после колледжа.

Первые работники занимались только технической поддержкой, то есть никакого кода не писали. Затем понемногу я начал брать на работу и программистов. Первым был мой приятель по Сети Брэд Уиттакер: у каждого из нас был сайт с названием вроде BradleyLand или BradleyWorld, так что мы нашли сайты друг друга. Я опережал его в области веб-программирования на год или два, и он спрашивал меня: «Слушай, как ты это сделал?», – шла ли речь об HTML, фреймах, CGI или Perl. Потом мне начали предлагать проекты, и те, что мне не нравились, я передавал ему. И вот однажды нам достался такой крупный проект, что ни один из нас самостоятельно его бы не потянул, и мы сказали заказчику: «Для этого проекта нужны двое». И он отправил нас в Пенсильванию. В Питтсбург? Я вообще не знаю восточное побережье, ведь сам я с западного. Или в Филадельфию? В общем, туда, где любят чизстейки.

Сейбел: Это Филадельфия.

Фицпатрик: Да. Мы там встретились впервые, в каком-то дешевом отеле, но у меня было такое чувство, будто я его уже знаю. Он был очень общительный. Пошел в туалет в моем номере и даже не закрыл дверь, хотя я стоял рядом. Я сказал что-то вроде: «Отлично, будь как дома».

Казалось, будто мы знаем друг друга лет пять, хотя на самом деле никогда не встречались. И мы начали работать над проектом вместе.

Он перебрался в мою просторную спальню, мы выкинули из кухни почти все, поставили компьютерные столы и там работали. Мы просыпались в 10–11 утра, работали до полудня, сидя в трусах, потом немного смотрели телевизор, после чего трудились в непрерывном режиме до 3–4 часов ночи. Потом к нам на лето присоединился еще один мой приятель – он учился в Университете Вашингтона. Летом после первого курса колледжа мы работали втроем. Мой третий друг жил в центре города. Он приезжал утром на метро, от станции катил до нас на скейтборде. Сидел на скамеечке рядом с домом и работал по Wi-Fi, пока мы не просыпались и не выпускали его.

Когда нас стало трое, места оказалось уже маловато, и я предложил: «Давайте снимем офис». Мы сняли офис и сказали друг другу: «Столько места! Давайте возьмем еще людей». Мы понемногу росли, и через пару лет нас стало уже 12 человек. Живой Журнал становился все популярнее, но и нагрузка повысилась, так как я занимался персоналом.

Моя мама тоже занималась персоналом, и мы с ней постоянно ссорились, потому что она работала на меня. Мне даже пришлось выработать правила: «Мама, если ты звонишь мне, это должно быть по личному делу либо по работе. Либо то, либо это. Нельзя перескакивать с личных дел на работу и обратно». Я просто бросал трубку, когда она так делала. А когда она перезванивала, я говорил ей: «Все, хватит». Так что нервозотрепки хватало. Мама была счастлива, когда я продал это дело, – она больше на меня не работала, и мы перестали ссориться.

Сейбел: Ваша компания тогда работала и по контрактам или занималась только Живым Журналом?

Фицпатрик: В основном Живым Журналом. Мы также пытались запустить фотохостинг, но Flickr нас уделал. Видимо, наш сервис был переусложнен – с прекрасными абстракциями и возможностью встраивания куда угодно. Создавая новый инфраструктурный компонент для Живого Журнала, мы спрашивали себя: «Как это будет работать с FotoBilder?», так что все начали делать абстрактным. Memcached был абстрактным, потому что не было смысла завязывать его с Живым Журналом. Потом мы создали файловую систему наподобие GFS¹ и очередь заданий. Мы создавали эти компоненты, чтобы они могли работать с любым из наших продуктов, но еще и потому, что чем меньше

¹ GFS (Google File System) – распределенная файловая система компании Google, кластерная система, оптимизированная для работы с большими блоками данных по 64 Мбайт и обладающая повышенной защитой от сбоев. – *Прим. науч. ред.*

в системе запутанных взаимозависимостей, тем легче ее поддерживать. Даже если это требует больше работы, убрать часть зависимостей – уже большое дело. Поэтому мы начали создавать все эти обобщенные компоненты.

Сейбел: Интересно было бы услышать о развитии Живого Журнала, о том, как вы начинали и как в процессе усваивали необходимые уроки.

Фицпатрик: Все начиналось с одного компьютера с UNIX. На нем одновременно работали несколько пользователей, но постепенно Живой Журнал вытеснил их всех.

Сейбел: Выполнялся как набор CGI-скриптов?

Фицпатрик: Да. То был, кажется, CGI в самом буквальном смысле – пожертвуй всем и умри. Тот провайдер приставил ко мне одного парня. У меня были проблемы, потому что сервер все время сдыхал, и я сказал ему: «Я плачу десять долларов в месяц. Почему он не работает?» Парень ответил: «Ага, сделай-ка вот это». Вскоре я освоил UNIX и понял, что происходит на самом деле.

Затем я переделал все под FastGCI, настроил Apache и вырубил обратный поиск по DNS. После того как все эти этапы пройдены, упираешься в ограничения ввода/вывода или в ресурсы процессора. Потом я получил собственный выделенный сервер, но он был только один, и когда он умирал, у меня начинались проблемы с ресурсами. Я дал доступ на него своим друзьям и просто оставил страницу регистрации открытой. Потому друзья пригласили своих друзей, которые, в свою очередь, пригласили своих друзей, хотя сайт не задумывался как общедоступный. Страница регистрации осталась открытой случайно. Так что потом я поместил на страницу новостей Живого Журнала объявление: «Помогите. Нам нужно купить серверы».

Мы собрали, кажется, тысяч 6 или 7 долларов или около того, купили два больших Dell и поставили их у провайдера Speakeasy в деловом центре Сиэтла. Кто-то порекомендовал нам эти Dell, огромные шестиянитовые громадины, килограммов под пятьдесят каждая. Логическое разделение было следующим: сервер базы данных и веб-сервер. Это единственное разделение, которое я знал, поскольку работал с двумя процессами – MySQL и Apache.

Какое-то время все работало как надо. Веб-серверы торчали напрямую во внешний мир, у них было по две сетевые карты и небольшой кабель к серверу базы данных. Потом веб-сервер перестал справляться с нагрузкой, но это не было проблемой, поскольку на тот момент у меня имелось несколько одноюнитовых серверов. Итак, у нас было три веб-сервера и один сервер базы данных. Тогда я попробовал три-четыре программы балансировки нагрузки для протокола HTTP – mod_backhand,

`mod_proxy` и `Squid` – и возненавидел их все. С тех пор не люблю балансировщики нагрузки.

Потому упала база данных. «Вот черт», – сказал я себе. Веб-серверы прекрасно масштабируются, ведь они не сохраняют состояния. Просто добавляешь новые серверы и распределяешь нагрузку. Это был долгий напряженный период. «Так, я могу слегка оптимизировать запросы», но это дает лишь неделю, а потом они опять перестают справляться с нагрузкой. В какой-то момент я задумался, что же нужно каждому конкретному запросу.

Тогда я решил – казалось, мне первому в мире пришла такая мысль, – разбить все это на разделы (`partition`). Я подготовил документ с рисунками, в котором говорилось, как наш код будет работать. «В главной базе данных будут храниться только метаданные каких-то глобальных вещей, которые дают небольшую нагрузку, а все данные, связанные с индивидуальными блогами и комментариями, для каждого пользователя будут выделены в кластер базы данных. Пользователям с такими-то идентификаторами предназначен определенный раздел базы данных». Задним числом я понимаю, что именно так все и поступают. Но тогда потребовалось много усилий, чтобы переделать код на работающей системе.

Сейбел: Был ли назначен день перевода со старой версии на новую?

Фицпатрик: Нет. У каждого пользователя был флаг, определяющий номер кластера: если он был равен нулю, значит данные находились в основной базе, если отличался от нуля, значит данные уже находились в каком-то разделе. Потом была версия «Ваша учетная запись заблокирована». Учетная запись блокировалась и выполнялась попытка переноса данных, программа пыталась переместить данные и снова сделать это, если вы в это время вносили какие-то изменения. Примерно в таком духе: «Ждите, пока мы не переместим данные, и не вносите никаких изменений в данные в основном кластере, скоро мы переместим вас в ваш индивидуальный кластер».

Такой перевод в фоновом режиме длился несколько месяцев. Мы прикинули, что если бы мы просто выгрузили данные, написали что-то для разбивки `SQL`-файлов и залили данные назад, то это потребовало бы около недели. Неделя простоя или два месяца медленного переноса? Но в процессе переноса данных 10% пользователей работоспособность сайта снова становилась приемлемой для других пользователей, так что мы смогли увеличить темпы переноса данных с загруженных кластеров.

Сейбел: Это было еще до `memcached` и `Perlbal`.

Фицпатрик: До Perlbal – это точно. Memcached, пожалуй, тоже была позднее. По-моему, я создал memcached сразу после колледжа, когда переехал. Помню, как ко мне пришла эта идея. Сайт был на грани, я пошел в душ и вдруг понял, что у нас ведь есть вся эта свободная память повсюду! Я набросал прототип тем же вечером, написал на Perl сервер и клиент, но сервер упал, потому что для сервера на Perl было слишком много обращений к процессору. Поэтому мы начали переписывать его на Си.

Сейбел: И вам не понадобилось покупать новые серверы для базы данных.

Фицпатрик: Да, серверы были дорогими, а процесс перехода с одного на другой – очень медленным. Веб-серверы были дешевы, и добавление новых сразу давало эффект. А при покупке новой базы данных где-то неделя уходит только на запуск и проверку: нужно проверить диски, все установить и настроить.

Сейбел: Значит, все элементы созданной вами инфраструктуры, такие как memcached и Perlbal, были разработаны в ответ на реальные потребности, связанные с масштабированием Живого Журнала?

Фицпатрик: Да, конечно. Все, что мы создали, делалось только потому, что наш сайт падал, и мы ночь напролет выдумывали новые штуки. Однажды мы даже решили купить систему хранения данных NetApp. Это выглядело так. Мы спросили: «Сколько она стоит?», а они в ответ: «Расскажите нам о вашем бизнесе». – «У нас платные учетные записи». «Сколько у вас клиентов? Какая нагрузка?» – «Мы знаем только, что их число растет, вот и все». – «Тогда цена такая: весь доход, который вы можете заплатить, чтобы не развалиться». – «Да пошли вы». Но все же нам была нужна эта штука, и мы ее купили. Скорость ввода/вывода нас не слишком впечатлила, цена была слишком высока, и здесь по-прежнему оставалась единственная точка отказа. Они попытались продать нам конфигурацию с высокой скоростью доступа, но мы сказали: «Да пошли вы. Мы эту ерунду больше не купим».

Итак, мы начали работу над файловой системой. Я даже не уверен, что к этому моменту был опубликован документ по GFS, кажется, я просто услышал о ней от кого-то. В то время я всегда использовал хеш-значение ключа для указания на фрагмент памяти. Почему бы не сделать то же самое с файлами? Файлы постоянны, поэтому нам нужно записывать, где они хранятся, поскольку при добавлении новых узлов хранения меняется и конфигурация. И дело не только в вводе/выводе и отслеживании местонахождения файлов, но и в высокой доступности системы. Мы нашли решение, и я пришел к следующей схеме: «Нам нужно хранить все обращения к файлам, чтобы знать, где что лежит».

Я написал схему для MySQL, сначала главного устройства, а потом для устройства отслеживания файлов. И меня осенило: «Черт! Да эту роль может выполнять протокол HTTP! Это же совсем не сложно».

Помню, как пришел на работу, всю ночь обдумывая это. У нас в здании была общая комната для совещаний – большая и мрачная. «Итак, ребята, прекращаем работу. Все идем вниз и будем рисовать». Я говорил это каждый раз, когда нам предстояло заняться проектированием, и мы просто находили доску, на которой можно было бы рисовать.

Я объяснил схему, и кто с кем должен общаться, и кто что будет делать с запросом. Потом мы поднялись наверх, и я первым делом заказал все оборудование, потому что на его доставку уходило недели две. Потом мы занялись кодированием, надеясь завершить его до получения оборудования. Нам всегда что-то угрожало. Что-нибудь постоянно ломалось, так что нам все время приходилось создавать новые компоненты инфраструктуры.

Сейбел: Если бы кто-нибудь в самом начале сказал: «Вам нужно знать А, Б и В», – упростило бы это вашу жизнь?

Фицпатрик: Всегда легче сделать что-то как надо с первого раза, а не переносить с уже работающего сервиса. Это всегда большой геморрой. Все, о чем я говорил, вы можете сделать на одном компьютере. Проектируете систему таким образом, чтобы было с чего начать. И не делаете предположений о возможности объединений вот этих пользовательских данных с этими и так далее. Предположим, вам нужно загрузить 20 объектов, и ваша реализация может загрузить их из одной таблицы, но код более высокого уровня, которому нужны эти 20 объектов, может собирать их с нескольких машин. Если бы я делал так с самого начала, куда меньше было бы головной боли с переносом.

Сейбел: Итак, урок в основном таков: «Имейте план на тот день, когда ваши данные перестанут влезать в одну базу».

Фицпатрик: Думаю, сегодня это уже общеизвестный факт в сообществе веб-разработчиков. Сейчас многие перегибают палку, считая что их сайт разрастется до невероятных размеров. Но в то время все считали, что Apache и MySQL достаточно.

Сейбел: Думаю, вы разрабатывали все эти штуки не только по необходимости, но вам было интересно делать все это.

Фицпатрик: Конечно. Я определенно пытался найти повод применить или изучить что-то новое. Никогда не изучишь что-либо, не написав для этого программу и не начав жить и дышать этим. Одно дело выучить язык ради удовольствия, но нельзя говорить о том, что знаешь его, пока не напишешь на нем большую сложную систему.

Сейбел: Итак, какие языки вы можете назвать своими, какими языками вы жили и дышали?

Фицпатрик: Perl. Си. Когда-то Бейсик, но не уверен, что его стоит учить. Еще я много писал на Лого. В школе у нас были уроки по Лого. Ребята что-то рисовали, а я сумел выйти из графического режима – это можно сделать, когда знаешь, как, – и писал функции. Учитель подошел и сказал: «Что ты делаешь? Ты должен рисовать домики». – «Нет, я пишу на Лого. Посмотрите». – «Ты все делаешь не так». В конце урока у меня уже была библиотека, позволяющая рисовать буквы алфавита любого размера и повернутые под любым углом. Я мог выводить целые сообщения на волнистых баннерах, которые отдалялись, приближались и все такое, и все стали спрашивать: «Какого черта?» Не знаю, считать этот язык или нет.

Но я много писал на Perl и на Си, потом в колледже много писал на C++ по работе и для Windows. Потом я забыл C++, за ненадобностью, но за последний год, работая в Google, я много писал на C++, Python и Java. Я много писал на Java, когда этот язык только появился, но потом он мне осточертел. Сейчас я опять много пишу на Java, и он меня уже снова достал.

Сейбел: Насколько для вас важен язык, на котором вы пишете?

Фицпатрик: Полностью мне все еще не нравится ни один из них. И я не знаю, какой язык понравился бы мне полностью. Мне не нравится, что в текущем проекте приходится перепрыгивать с одного языка на другой. Я хочу статически типизированный язык, который бы проверял все во время компиляции, когда я этого захочу. Perl очень близок к этому – он позволяет мне кодировать так, как я хочу. Он не выполняет достаточное количество проверок во время компиляции, но я могу заставить делать их во время выполнения. Но и он все же недостаточно хорош.

Я хочу необязательную статическую типизацию. В Perlbal нет нужды в высокой производительности половины всех возможностей, за исключением ядра и копирования байтов туда-сюда. Я хочу, чтобы у меня была возможность во время выполнения давать подсказки в определенных частях кода и объявлять типы. Но если мне лень и хочется что-то протестировать, то могу оформить код соответствующим образом.

Сейбел: То есть типы вам нужны в основном, чтобы улучшить оптимизацию кода компилятором?

Фицпатрик: Нет. Я хочу, чтобы компилятор говорил мне что-то вроде: «Ты делаешь глупость». Но иногда мне плевать на такие предупреждения и я хочу заставить код выполняться независимо ни от чего. Не хочу показаться слишком большим оптимистом по поводу Perl 6, но они обе-

щают много вещей, которых мне не хватает. Правда, не думаю, что он вообще когда-либо будет выпущен.

Сейбел: А C++ вам нравится?

Фицпатрик: Даже говорить о нем не хочу. Жуткий синтаксис, совершенно непоследовательный, а сообщения об ошибках – по крайней мере компилятора GCC – просто нелепы. Можно получить 40 страниц сообщения только потому, что забыл поставить точку с запятой. Но, как и во многих других случаях, основные шаблоны быстро запоминаешь. Даже не вчитываешься в слова, а просто смотришь на структуру сообщения и понимаешь: «Ага, кажется, я забыл закрыть пространство имен в заголовочном файле». Думаю, новая версия спецификации C++, хотя и добавляет огромное количество сложностей, содержит много всего, что сделает процесс ввода не таким мучительным, по крайней мере, потребуется меньше стучать по клавишам. Переменные `auto` и нововведения в циклах `for`¹. Очень напоминает Python. И лямбда-выражения. Можно даже подумать, что пишешь на Python, а не на C++.

Сейбел: А C++ вы используете из-за его эффективности.

Фицпатрик: Скорее всего, да. В основном я пишу на нем в Google. Там все, что более или менее требует производительности, пишется на C++. Кроме того, в Google я много пишу на Java.

Сейбел: Насколько я понимаю, в компании Google сложилась «C++-центрическая культура», поскольку они использовали этот язык с самого начала, построив на его основе обширную программную инфраструктуру. Хотя и нельзя просто так взять и забыть свою историю – пожалуй, на C++ в Google написана значительная часть кода, которая не требует такой эффективности.

Фицпатрик: Особенно учитывая, что со временем Java стал быстрее, а JVM – значительно умнее. В Java мне не нравится то, что у всех сложилось стойкое отвращение к JNI². Есть библиотеки на C++. Разработчикам, использующим Python, как внутри компании Google, так и за ее пределами, все равно. Их первая мысль: «Да мы просто обернем все

¹ Имеется в виду новая версия языка C++, которую в настоящее время принято называть C++ 0x. Ключевое слово `auto` позволяет не указывать явно тип, вместо этого компилятор определит его автоматически (строгая типизация сохраняется). Что касается циклов, то здесь речь идет о так называемом `range-based for`, который в значительной степени упрощает итерирование коллекций. – *Прим. науч. ред.*

² JNI (Java Native Interface) – инфраструктура, позволяющая взаимодействовать коду внутри JVM с «неуправляемым» (native) кодом, таким как API операционной системы или любой код на Си или C++. – *Прим. науч. ред.*

это с помощью SWIG¹. У них есть собственный путь, и они счастливы. Разработчики на Python могут сразу же использовать все, что написано на C++, потому что они не относятся так благоговейно к языку источника.

А сторонники Java говорят: «Надо писать только на чистом Java. Мы не можем использовать JNI, потому что если JVM рухнет, то мы не узнаем, почему». Проблема в том, что все приходится писать дважды – один раз для C++, Python и прочих языков, а второй раз для Java. Так что если они найдут хороший способ взаимодействия или избавятся от страха перед JNI, то я не буду иметь ничего против Java.

Сейбел: А как насчет вопроса «ручное управление памятью против автоматического»? Об этом все еще спорят. У вас есть твердое мнение на этот счет?

Фицпатрик: По правде говоря, нет. Занятно смотреть, как люди высказывают твердое, ничем не подкрепленное мнение. Лично мне ручное управление памятью не кажется таким уж раздражающим, по крайней мере, в C++ с умными указателями. Я могу днями писать на C++, ни разу не применив явно оператор `new` или `delete`. Вот так.

Я переписал `memcached`, уже работая в Google, для работы с инфраструктурой Google и для добавления ее к App Engine². Она написана целиком на C++, потому что мне было нужно очень строгое управление памятью для уменьшения фрагментации. И я очень рад возможности ручного управления памятью в C++.

Сейбел: Изначально программа `memcached` была написана на Си. Вы переписали ее на C++, потому что этот язык предпочтительнее в Google или у него есть другие преимущества?

Фицпатрик: Сперва я хотел просто взять существующую реализацию и перенести ее на C++, но работы оказалось слишком много. Оказалось не так много кода, которым я мог бы воспользоваться, поэтому было гораздо быстрее просто переписать его на C++. Объем кода уменьшился вдвое.

Сейбел: Это из-за C++ или из-за того, что вы стали опытнее?

¹ SWIG (Simplified Wrapper and Interface Generator) – инструмент с открытым исходным кодом, предназначенный для взаимодействия языков программирования C/C++ с языками сценариев, такими как Tcl, Perl, Python, Ruby и др. – *Прим. науч. ред.*

² Google App Engine – сервис хостинга сайтов и веб-приложений на серверах Google с помощью различных служб Google. См. http://ru.wikipedia.org/wiki/Google_App_Engine. – *Прим. науч. ред.*

Фицпатрик: Может, и из-за опыта. Лет в 11–12 я путешествовал с родителями по стране и писал игру Mastermind для калькулятора TI-85 – пару сотен строк – на крошечном экранчике, пытаюсь понять, что же я делаю. Я дважды стирал эту проклятую штуковину. Так что я написал ее три раза. Но с каждым разом становилось все легче. Верно подмечено: разрабатывать систему во второй раз намного проще.

Сейбел: Вы много писали на Perl, весьма симпатичном высокоуровневом языке программирования. Как по-вашему, насколько «низко» нужно спускаться программистам? Нужно ли им знать ассемблер и понимать, как работает процессор?

Фицпатрик: Не знаю. Мне знакомы по-настоящему умные люди, я бы сказал, хорошие программисты, но которые знают только Java. Они думают о решении задачи в пределах известной им области. Они не думают о задаче от начала и до конца. По-моему, надо знать всю цепочку, даже если оперируешь только с одним звеном.

Занимаясь Живым Журналом, я думал о разных вещах, начиная от языка JavaScript и заканчивая вопросами взаимодействия с ядром операционной системы. Я читал код системных вызовов `epoll`¹ в ядре ОС Linux и думал: «А что если у нас будут все эти длительные соединения по протоколу TCP, и код на JavaScript будет опрашивать открытые TCP-соединения, ведущие к системе балансировки нагрузки?» Я попытался понять, сколько памяти нужно каждой структуре данных на одно подключение. Все это вопросы достаточно высокого уровня, но потом мы задумываемся, скажем, об огромном количестве прерываний от сетевой карты – не переключиться ли нам на использование NAPI ядра ОС вместо получения прерывания по каждому принятому пакету от сетевой карты, которые она будет соединять со скоростью, эквивалентной 100 мегабитам, даже для гигабитной сетевой карты? Мы собирали данные, чтобы определить, на каком уровне это будет иметь смысл и освободит процессор.

Мы много чего сделали для достаточно низкоуровневых вещей. Недавно кто-то сказал мне по какому-то поводу: «Java сам заботится об этом; нам не нужно думать об этом». Я ответил: «Нет, Java не может позаботиться об этом, потому что я знаю, какую версию ядра вы используете, и это ядро не поддерживает эту возможность. Ваша виртуальная ма-

¹ `epoll` – новый системный вызов, который появился в Linux 2.6. Призван заменить устаревший `select` (а также `poll`). В отличие от старых системных вызовов, длительность работы которых зависела от количества прослушиваемых дескрипторов, `epoll` использует алгоритм, который не зависит от количества дескрипторов, позволяя добиться хорошего масштабирования при увеличении количества прослушиваемых дескрипторов. – *Прим. науч. ред.*

шина может скрывать это от вас, предоставляя какие-то абстракции, которые выглядят эффективными, но они будут эффективными только при запуске на определенном ядре». Я расстраиваюсь, когда люди даже поверхностно не знают, как все устроено.

На практике никогда ничего не работает нормально. За прекрасными абстракциями скрывается всякая дрянь. Библиотеки могут выглядеть прекрасно, но работают отвратительно. И если именно вы отвечаете за покупку серверов или за поддержку, то очень полезно знать, что же на самом деле происходит внутри, не доверяя чужим библиотекам, коду и интерфейсам.

Я даже склоняюсь к мысли, что сегодня вряд ли стал бы программистом. Это совсем неинтересно. Вот почему мне так нравятся вещи вроде App Engine. Кто-то сказал, что Google App Engine – это Бейсик нашего поколения. Потому что для нынешнего поколения все перешло в Сеть. Когда я занимался программированием, был только один язык, и он был установлен на моей собственной машине, а для развертывания системы достаточно было нажать кнопку Run. Сегодняшние дети не хотят заниматься такими глупостями, как «прыгающие мячики» на собственном компьютере. Им нужен веб-сайт.

Мне до сих пор пишут что-нибудь вроде: «Привет, у меня появилась идея: я хочу сайт, который уделал бы Википедию, YouTube,...» Каждый хочет сделать веб-сайт, поскольку четыре его любимых веб-сайта не совсем правильны, и хочет что-то внешне похожее.

То, что App Engine предоставляет всего одну кнопку Put this on the Web (Выложить в Сеть) и можно писать все на одном языке – Python, который кажется довольно простым для изучения, – просто прекрасно. Это отличное введение в программирование – вас избавляют от множества уровней всякой ерунды.

Сейбел: Как же это вяжется с вашим расстройством по поводу поклонников Java, когда они говорят: «Java позаботится об этом за вас». Разве это не одно и то же, когда вы говорите: «App Engine позаботится об этом за вас»?

Фицпатрик: Не знаю. Может, просто мне известно, что происходит на самом деле. В принципе, JVM не так уж плоха. Думаю, проблемы начинаются тогда, когда люди принимают на веру некие абстракции, не понимая, что происходит на самом деле.

Сейбел: У вас был большой опыт программирования к тому времени, как вы поступили в колледж и начали слушать курс по компьютерным наукам. Как это вам помогло учиться?

Фицпатрик: Поначалу я пропускал много занятий по компьютерным наукам – такая была там скука. Я появлялся только на экзаменах.

Дальше, на третьем-четвертом курсе стало поинтереснее. Но тут, как назло, я окончил колледж. А на занятия магистров меня не пускали, ведь я не учился в магистратуре.

Помню, на курсе по компиляторам последнее задание было таким: взять один из существующих языков, с которыми мы возились, и добавить определенный набор возможностей, включая одну функцию по своему выбору, за которую полагались дополнительные баллы. Я решил реализовать проверку выхода за границы массива во время выполнения. Преподаватель взял мой скомпилированный код и запустил на нем свой набор тестов, и выполнение нескольких из них завершилось неудачно. Тогда он сказал: «Извините, но ваш код не прошел мои модульные тесты. Вы получаете тройку». Я посмотрел на код его тестов и сказал: «Так ваш тест содержит ошибку диапазона (off-by-one error)». Он исправил оценку на пятерку. Но дополнительных баллов я так и не получил и разозлился на колледж.

Еще помню курс по базам данных, который читал человек, видимо, без реального опыта работы с базами данных. А я в то время уже работал с Oracle, Microsoft Server и особенно плотно с MySQL. И я задавал вопросы практического характера, на которые хотел получить ответы – тогда они были актуальны для меня, – но мне выдали стандартную фразу из учебника. Я сказал: «Нет-нет, это не работает».

Сейбел: Вы окончили колледж в 2002 году. Теперь вы можете лучше оценить то, чему вас учили, или нет?

Фицпатрик: Половина курсов мне очень нравилась, я или узнавал что-то новое, чего в то время еще не знал, или получал соответствующие базовые знания и изучал правильную терминологию. До того я неплохо знал программирование, но у меня не было достаточного словарного запаса, чтобы объяснить то, что я делаю. Я мог выдумать собственную терминологию, но в результате люди могли бы подумать, что я не знаю, о чем говорю. В этом плане формальное образование помогло мне.

Сейбел: Вы сожалеете о том, что приходилось совмещать работу с учебой? Может быть стоило заниматься либо одним, либо другим?

Фицпатрик: Нет, мне кажется, это был наилучший вариант. У меня были приятели, которые только учились, но я уже знал так много, что мне было бы скучно. Один мой приятель был действительно знающим, но он считал, что пошел в колледж за знаниями, а вовсе не за дипломом, – и параллельно изучал арабский, китайский и японский. И все эти безумные языки программирования. Каждую неделю он говорил: «У меня теперь новый любимый язык. Эту неделю я программирую

только на OSamI». Таким образом, он был постоянно занят. Я тоже был постоянно занят и боролся со скукой, но по-другому.

У меня были приятели, которые бросили колледж в первый же год и стали делать всякую ерунду для Сети. Некоторые занялись порносайтами и всяким таким, типа «мы заработаем кучу денег». И они с головой уходили в работу, но делали только деньги, больше ничего. Колледж – прекрасное место для общения и вечеринок. Если бы я занимался только Живым Журналом, я бы умер от стресса.

Сейбел: Вы довольны тем, что изучали компьютерные науки?

Фицпатрик: Пожалуй, я мог бы обойтись и без этого. Но я много делал и такого, чего сам бы никогда не сделал, так что, думаю, это было полезно. Мне хотелось бы, наверное, заняться чем-нибудь еще, например остаться еще на один год и изучить что-то совершенно постороннее, скажем лингвистику. Мне немного жаль, что я учился в колледже вполсилы из-за того, что много уже знал изначально. На первых курсах я вообще почти не появлялся на занятиях, а когда под конец стало интересно, получилось так: «Поздравляем, вы закончили обучение».

Сейбел: А о магистратуре думали?

Фицпатрик: Да. Было бы интересно, но я был слишком занят.

Сейбел: Вы читаете современную компьютерную литературу?

Фицпатрик: Мы с друзьями посылаем друг другу статьи – хорошие такие статьи. Я, например, недавно читал статью насчет изменения размеров фильтров Блума во время выполнения. Потрясающая статья. Статьи с конференций по системам хранения данных, как из промышленных кругов, так и из академических, о разных прикольных системах – я стараюсь читать все это. Что-то попадалось насчет Reddit¹, то ли приятель прислал мне статью, то ли в чьем-то блоге была ссылка на нее.

Сейбел: Вы упомянули о статьях из научных и промышленных источников. Как по-вашему, сегодня есть место, где эти источники сливаются?

Фицпатрик: Иногда у меня и правда возникает такое впечатление. Но часто интереснее читать статьи, основанные на практическом опыте, ведь они пытаются решать реальные проблемы, и их решения работа-

¹ reddit.com – социальный новостной сайт, на котором зарегистрированные пользователи могут размещать ссылки на какую-либо информацию в Интернете. Как и другие подобные сайты, reddit поддерживает систему голосования за понравившиеся сообщения – наиболее популярные из них оказываются на главной странице сайта. – *Прим. науч. ред.*

ют, в отличие от мыслей вроде: «Мы думаем, будет очень прикольно, если...». Из научного мира исходит много безумных идей, которые на самом деле не работают, так и оставаясь безумными идеями. Может быть, позднее эти идеи превратятся в коммерческие продукты.

Сейбел: Как вы проектируете программное обеспечение?

Фицпатрик: Я начинаю с интерфейсов между некоторыми элементами. Какие методы более свойственны системе – удаленные вызовы или запросы? Если речь идет о хранилищах, я пытаюсь понять, какие запросы будут более частыми. Какие нужны индексы? Как данные будут читаться с диска? Потом я пишу заглушки для различных частей системы, развивая их со временем.

Сейбел: Вы пишете заглушки, чтобы можно было писать тесты до написания остального кода и выполнять их по ходу разработки?

Фицпатрик: Более того. Я всегда проектирую ПО именно таким способом, даже без применения тестов. Сначала я проектирую интерфейсы и хранилища данных, а затем берусь за их реализацию.

Сейбел: В каком виде осуществляется проектирование? Псевдокод? Реальный код? Каракули на белой доске?

Фицпатрик: Обычно я беру редактор и пишу заметки с псевдокодом для схемы базы данных. Доведа я до ума, создаю реальную схему и копирую/вставляю все скрипты, чтобы удостовериться в том, что операторы `create table` работают. После этого приступаю непосредственно к реализации. Я всегда начинаю с файла *spec.txt*.

Сейбел: Бывает ли так, что, написав уже порядочный кусок кода, вы сталкиваетесь с необходимостью пересмотреть свой первоначальный план?

Фицпатрик: Бывает. Но я начинаю с самых сложных кусков или с тех частей, в которых не уверен, и пытаюсь реализовать их в первую очередь. Я стараюсь не оставлять ничего сложного или потенциально неожиданного под конец: я предпочитаю с самого начала решить наиболее трудные вещи. Причина, по которой я так и не завершил ряд своих проектов (друзья говорят, что их целая куча), связана с тем, что я начинал с наиболее сложной части проекта, изучал то, что хотел изучить, и не возвращался к оставшейся неинтересной части.

Сейбел: Можете ли вы дать совет программистам-самоучкам?

Фицпатрик: Всегда старайтесь делать что-нибудь чуть более трудное, чем раньше, то, что вам не по зубам. Читайте чужой код. Я слышал это много раз, но дошло только со временем. Несколько лет я писал много кода, не читая чужой код. Потом я попал в Сеть, а там сплошь и рядом открытый код, в развитии которого каждый может принять участие.

Но я был до смерти перепуган, предполагая, что не смогу в нем разобраться, ведь автором этого кода был не я, и все его устройство не укладывалось у меня в голове.

Потом я начал делать патчи для Gaim, программу мгновенного обмена сообщениями под GTK, начал копаться в коде – и увидел его в целом, я понял это, просто рассматривая отдельные фрагменты кода. После просмотра чужого кода я понял (не могу сказать, что полностью понимаю собственный код), что начал видеть паттерны. «Так, отлично. Я понял структуру, по которой он строится».

И тогда я начал действительно получать удовольствие от чтения кода, потому что, не понимая некоторый паттерн, задавался вопросом: «Какого хрена они сделали это именно так?» – и начинал изучать код внимательнее. Потом говорил себе: «Ух ты, да это же и правда отличный способ решения этой проблемы, мне понятно, как он оправдывает себя». Я бы делал это и раньше, но боялся, поскольку считал, что если код не мой, то я не смогу его понять.

Сейбел: А каким образом вы читаете чужой код? Вы начинаете с того, что читаете код, чтобы понять его в общих чертах, или читаете только тогда, когда нужно внести какие-то исправления?

Фицпатрик: Обычно я хочу что-то исправить. Или просто читаю чужой код, если действительно уважаю его автора. Может, это помогает осознать, что он тоже смертный, и не стоит его боготворить. Или узнать из его кода что-нибудь полезное.

Сейбел: Допустим, вы знаете, какие изменения хотите внести. Как вы поступаете?

Фицпатрик: Прежде всего нужно достать архив исходников или получить последнюю версию из SVN и заставить эту проклятую штуку компилироваться. Преодолеть это препятствие. Для большинства оно оказывается самым сложным из-за дополнительных зависимостей в системе сборки или из-за неверных предположений об уже установленных библиотеках. Иногда мне хочется, чтобы крупные проекты шли с образами виртуальных машин, с полностью настроенным окружением для сборки.

Сейбел: Вы имеете в виду что-то вроде VMware?

Фицпатрик: Да. Если просто хочешь по-быстрому что-то исправить, то вот тебе все зависимости. Связь с людьми устанавливается достаточно быстро. Все отлично работает.

Так или иначе, когда у вас есть чистая работающая сборка, забейте на все и просто сделайте одно долбаное изменение. Измените заголовок

окна на «Брэд говорит „Hello world“». Измените хоть что-нибудь. Пусть там все ужасно, просто начните вносить изменения.

Затем по ходу работы пишите патчи. Думаю, это лучший способ начать диалог. Если участвуешь в списке рассылки и пишешь что-то вроде: «Привет, я хочу добавить возможность X», человек, поддерживающий эту систему, скорее всего, ответит: «Какого хрена, я занят. Отвали. Терпеть не могу возможность X». Если же напишешь что-то вроде: «Я хочу добавить возможность X. Я думал сделать такой вот патч», – а это совершенно неверный путь – но ты говоришь: «Но я думаю, что это неправильно. Думаю, что правильный путь – это реализовать X», более сложный путь, и тебе, скорее всего, ответят что-то вроде: «Черт, он старался и, смотрите, пошел совершенно неверным путем».

Возможно, это заденет того, кто поддерживает этот код, и он решит: «Слушай, не могу поверить, что вот на ЭТО потрачено столько сил. Ведь так просто сделать правильно». Или: «Боже, столько работы – и все впустую. Надеюсь, больше этим путем не пойдут». И тогда тебе ответят.

Это лучший путь завязать диалог. Даже в Google я часто начинаю так разговор с командой разработчиков, с которыми не знаком. Исправив ошибку в их коде, я прежде всего посылаю им патч по электронной почте и говорю: «Ребята, что вы об этом думаете?» Или на внутренней ревизии кода говорю им: «Вот описание. Что вы об этом думаете?» Они могут, конечно, сказать: «Черт, нет, это совершенно некорректное исправление».

Сейбел: Вы читаете код ради собственного удовольствия или только тогда, когда это нужно вам по работе?

Фицпатрик: Иногда читаю. Я беру исходный код Android просто так, без видимой причины. То же самое с Chrome: когда его код стал открытым, я сделал зеркало репозитория и стал изучать код. И тоже самое сделал с Firefox и с Open Office. Пользуешься какой-то программой, а потом получаешь доступ к ее исходному коду и можешь на него взглянуть.

Сейбел: У таких программ объем кода довольно велик. Читая код просто ради интереса, как глубоко вы вникаете?

Фицпатрик: Как правило, я просто делаю конвейер из `find` в `less` («найти среди значений меньше, чем») и стараюсь понять структуру каталогов. Потом что-то привлекает мое внимание или я нахожу что-то, чего не понимаю. Я беру файл наугад, чтобы получить общее впечатление от него. Затем случайным образом прыгаю по коду, пока не надоест, и тогда снова беру произвольный кусок и начинаю разбираться с ним.

Я не раз выполнял сборку проекта, одновременно читая его код, поскольку эти задачи вполне можно выполнять параллельно, особенно

если сборка проекта оказывается сложной. Собрав в конце концов проект, я могу поиграть с кодом, если захочу.

Сейбел: Значит, когда вы читаете хороший код, то он соответствует уже известным вам паттернам или вы открываете для себя новые? Но не всякий код хорош. Каковы первые признаки плохого кода?

Фицпатрик: Ну, я стал достаточно придирчивым, поработав в Google с его очень строгими стандартами оформления кода для всех языков. Для наших шести или семи основных языков есть очень четкие стандарты оформления кода, в которых сказано: «Вот так мы располагаем код. Так называем переменные. Так расставляем пробелы и отступы, используем такие-то паттерны и соглашения, так объявляем статические поля».

Мы стали выкладывать это и в Интернете в качестве справочного руководства для удаленных сотрудников, которые участвуют в наших проектах. Мы хотим иметь строго документированную политику, поэтому не говорим просто: «Нам не нравится, как вы оформляете код».

Сейчас, работая над проектом на Си, первым делом я добавил стандарт оформления кода. Поскольку проект зрелый и в нем участвуют многие, у них уже есть стандарты кодирования. Они даже не всегда оформляются в письменном виде – программист просто должен соблюдать оформление уже написанного кода. Ему, может быть, не по душе стандарт расположения фигурных скобок, но к черту все это, гораздо важнее иметь последовательное оформление кода в файле и в проекте целиком, чем оформлять код любимым для тебя способом.

Сейбел: Вам приходилось заниматься парным программированием?

Фицпатрик: Думаю, это довольно занятно и полезно во многих случаях. Иногда надо просто подумать и побыть одному. Не думаю, что это полезно всегда, но это однозначно весело.

Я запускаю слишком много проектов. И завершаю их, так как иначе чувствую за собой вину, но я определенно слишком часто переключаюсь с одного на другое и слишком распыляю силы. Вот почему мне нужно парное программирование – оно заставляет меня сидеть на одном месте целых три часа, или два, или хотя бы только час, и работать над одной задачей с другим человеком, и при этом я не скучаю. Если приходится заниматься скучным патчем, мне говорят: «Да ладно, нам же нужно это сделать», – и мы доводим это дело до конца.

Я люблю работать один, но в этом случае я постоянно перепрыгиваю между задачами. Я всегда беру в самолет запасные батареи для ноутбука, на котором есть полная среда разработки и локальный веб-сервер. Я запускаю веб-браузер и что-нибудь тестирую. При этом в браузере открываю дополнительные вкладки и набираю «reddig» или «lwn» –

сайты, которые читаю постоянно. Автозаполнение, жму Enter – и получаю сообщение об ошибке. Я делаю так несколько раз в минуту. Черт! Поступаю ли я так же на работе? Посещаю ли я эти сайты постоянно, не задумываясь? Это ужасно. У одного моего приятеля правила для iptables¹ настроены таким образом, что при попытке подключиться к определенным IP-адресам в определенное время дня происходит перенаправление на страницу с надписью «Работать надо!». Я до этого еще не дошел, но, видимо, и мне уже нужно нечто подобное.

Сейбел: Что вы думаете о владении кодом? Важно ли человеку владеть кодом индивидуально или предпочтительнее – командой?

Фицпатрик: Не думаю, что у кода должен быть владелец. И, по-моему, никто на самом деле не думает о пользе владения. Вот как это устроено в Google: есть одно огромное дерево с общим корнем и единая система сборки всего кода. Поэтому каждый может взять и поменять что угодно. Но существуют ревизии кода (code review), и у папок есть владельцы, как минимум двое у каждой, на случай, если один из них уволится или уйдет в отпуск.

Для сохранения кода в репозитории требуется выполнение трех условий. Нужно, чтобы кто-то посмотрел код и сказал, что он выглядит нормально. Нужно иметь специальный сертификат удобочитаемости (readability) по данному языку, который доказывает, что ты знаком с его стандартом. Еще нужно одобрение одного из владельцев этой папки. Так что если ты владелец папки и имеешь такой сертификат по этому языку программирования, остается лишь найти кого-то, кто скажет: «Да, выглядит хорошо». Это достаточно хорошая система, поскольку в этом случае получается как минимум двое владельцев, а может быть, и двадцать или тридцать. Если немного поработать с базой кода, кто-нибудь добавит тебя в число владельцев. Мне кажется, это отличная система.

Сейбел: Давайте вернемся немного назад. Как начинался Живой Журнал?

Фицпатрик: Мы просто возились с друзьями со всякой ерундой, которой нам хотелось бы заниматься и которая казалась нам забавной. Комментарии в Живом Журнале были практически шуткой. Я проверил свою страницу, перед тем как идти на занятия. Мы только что придумали френдленту, и я увидел, что мой приятель написал какую-то глупость. Мне захотелось его подколоть, но я не мог ему ответить! И я пошел на пары, но во время занятий постоянно думал: «Как сделать

¹ iptables – утилита командной строки, стандартный интерфейс управления работой брандмауэра netfilter для ядер Linux версий 2.4 и 2.6. – *Прим. науч. ред.*

систему ответов?» Я думал о существующей схеме и о возможностях ее изменения. Во время двухчасового перерыва между занятиями я добавил систему комментариев, написал что-то хитроумно-ехидное и пошел на урок. Когда я вернулся, парень ответил: «Какого хрена? Теперь можно комментировать!»

Все в Живом Журнале было чем-то вроде игры. Все, что связано с безопасностью – вроде постов только для друзей и закрытых постов, – появилось после того, как приятель описал, как пошел на вечеринку, а на следующий день проснулся пьяный в канаве. Его родители прочли это и подняли шум: «Как? Ты пьешь?» Вот он и предложил: «Брэд, надо сделать возможность блокировать чтение этого дерьма». А я ему: «Будет сделано!» У нас уже была система френдов (друзей), и мы сделали так, чтобы некоторые сообщения были доступны только друзьям, – оставалось не добавлять в друзья своих родителей, и все.

Сейбел: На заре Живого Журнала ваша жизнь, видимо, состояла из бессонных ночей, когда засыпаешь под утро после многочасовой работы. Насколько это обязательный атрибут жизни программиста?

Фицпатрик: По-моему, ночь – просто наименее напряженное время суток. Днем все время что-то происходит: то тебе хочется перекусить, то идешь на занятия, то тебе кто-то звонит. Тебя всегда что-то прерывает. Я не могу расслабиться. Если я занимаюсь работой за два часа до какого-нибудь собрания, то эти два часа не такие продуктивные, какими могли бы быть в тот день, когда собрание назначено на утро или его вообще нет. Если я знаю, что меня не будут отвлекать, то гораздо меньше беспокоюсь.

Ночью я чувствую, что это мое время, потому что все остальные спят. Нет шума, никто не отвлекает, я могу делать что угодно. Я до сих пор засиживаюсь допоздна – вот, например, в эти выходные я немало поработал над разными вещами. Но потом несколько дней хожу сонный. В колледже мне часто приходилось работать по ночам, потому что я занимался одновременно несколькими проектами, включая Живой Журнал, а это можно было делать только ночью. Ну, и обслуживать наш сервер тоже приходилось ночью. А если на дворе лето, то почему бы так не работать? Раз утром не надо идти на занятия или куда-нибудь еще, то можно сидеть по ночам.

Сейбел: Поговорим о рабочем времени и интенсивности работы. Не сомневаюсь, что вы работали по 80, 120, 150 часов в неделю. При каких обстоятельствах это необходимо, а при каких – просто говорит о желании самоутвердиться?

Фицпатрик: Не уверен, что в моем случае это было необходимо или говорило о желании самоутвердиться. Мне было интересно все это, и это

было именно то, что я хотел делать. Периодически что-то ломалось, но даже если ничего не ломалось, я продолжал этим заниматься, потому что постоянно работал над новыми возможностями, которые очень хотел реализовать.

Сейбел: Бывали ситуации, когда вам действительно приходилось оценивать необходимое время для реализации чего-то?

Фицпатрик: Да, когда я попал в компанию Six Apart. Это был мой первый подобный опыт, и произошло это три с половиной года назад. Мы начали заниматься переносом данных по просьбе одного из клиентов. Нужно было добавить поддержку этого в код, протестировать все и отправить заказчику. Мне было ужасно не по себе. Мне и сейчас не по себе, потому что я постоянно забываю о поправке на отвлечения и о том, что не могу избавиться от поддержки десятка уже существующих проектов.

Думаю, постепенно я научился лучше оценивать время, но, слава богу, меня не просят об этом слишком часто. И сейчас, когда на носу дедлайн, я думаю: «Ура! Дедлайн!», — меня это так заводит, что адреналин так и хлещет, и я работаю и заканчиваю эту чертову работу. На самом деле, в Google дедлайнов нет. А бывает примерно так: «Как насчет того, чтобы это запустить? Есть такое ощущение?» Реальные сроки сдачи проектов ставятся редко и главным образом в такой форме: «Хорошо бы запустить это к такому-то числу», — и все прилагают для этого максимум усилий. Но если не заканчиваешь что-то в срок, то подводишь других. Большинство же проектов, над которыми я работаю, реализуются по принципу «Когда сделаем, тогда и сделаем».

Сейбел: Наняв программистов для Живого Журнала, вы руководили ими?

Фицпатрик: Я предполагал, что никому из них руководство не нужно, что все они будут работать самостоятельно, как я. Это был отличный опыт в управлении персоналом. Я понял, что есть те, кто делает только то, что поручено, без какого-либо стремления к совершенству. Они работают по принципу: «Сделано. Следующее задание?» — или ничего не говорят, а просто шарят по Интернету. Так что у меня было несколько таких неприятных случаев. Но, кажется, через год-другой я понял, что все люди разные.

Есть пуристы: они городят абстракцию на абстракции поверх абстракции. Они делают все страшно медленно и очень строги в вопросе оформления кода. Они считают себя мастерами программирования, а я доказываю им, что их код не работает, что он неэффективен и непохож на любой другой код, с которым взаимодействует.

Сейбел: Вы нашли способ извлекать пользу из работы таких людей?

Фицпатрик: С одним парнем я испробовал с десяток различных способов. Он был лет на десять меня старше, точно не знаю, я никогда его об этом не спрашивал – боялся, что задавать подобные вопросы незаконно. Но у меня было ощущение, что ему не хочется работать на какого-то паршивца. Мне тогда было 22. С этим парнем мы так и не сработались, и это был единственный человек, которого я уволил.

К остальным я в конце концов нашел подход и выяснил, что их мотивирует. У одного отлично получалось делать «заплатки» и создавать прототипы. Он писал на Perl как сисадмин. Он мог соединять различные вещи, писать скрипты командной оболочки, писать действительно ужасный код на Perl и на Си, но почему-то все это у него работало. Мы поражались: «Черт возьми, как ты умудрился изучить все это и как тебе удалось увязать все эти компоненты друг с другом?»

Мы устанавливали голосовой сервис для Живого Журнала: записываешь что-то и постишь в Живой Журнал. Было очень много динамичного. Адская работа. А ему нравилось. Он разобрался со всем и заставил это работать. Позднее мы полностью все переписали, но выяснили, как это работает, именно благодаря ему. Он набросал интерфейс, который мы потом доводили до ума. Как только я понял, что в этом его призвание, мы прекрасно сработались.

Сейбел: Вы нанимали людей для своей компании. Предполагаю, что и в Google вам тоже приходится этим заниматься. Как вы распознаете в человеке отличного программиста?

Фицпатрик: Обычно я ищу тех, кто много чего написал по собственной инициативе, а не потому, что ему сказали это сделать. Я не имею в виду учебные задания и проекты от предыдущего работодателя. Я о тех, кто чем-то увлечен, кто выполнял сторонние проекты. Как он их сопровождал и насколько серьезно к ним подходил? Вдруг клепал что-то на скорую руку и забывал о проекте?

Сейбел: У вас есть любимые вопросы на собеседовании?

Фицпатрик: Один из вопросов, который я задавал несколько раз, – это вопрос из моего теста на школьном факультативном курсе по программированию: даны два десятичных числа в виде строки произвольной длины, нужно их перемножить. Есть много способов решения этой задачи. Если человек силен в математике – в отличие от меня, – он может найти какой-нибудь изящный и эффективный способ решения. В худшем случае можно создать класс, который выполняет серию сложений.

Я говорю им с самого начала: «Не волнуйтесь. Вам не нужно сделать это предельно эффективно. Сделайте хотя бы как-нибудь». Некоторые нервничают, не зная с чего начать. Это плохой признак. В худшем слу-

чае можно реализовать алгоритм, который применялся в начальной школе.

Я действительно в начальной школе написал программу, чтобы она выполняла за меня деление и умножение в столбик, показывая свою работу. Включая все шаги и зачеркивание. Потом мы взяли задачи, примерно по десять на страницу, я вбил их все в компьютер, а затем переписал решения от руки. То же самое я проделал на химии для нахождения орбиталей электронов. Я тогда кое-что понял: даже при написании такой жульнической программы учишься, потому что для этого нужно действительно глубоко понять задачу.

Сейбел: Думаете, это годится для всех? То есть вместо того чтобы обучать детей делению в столбик, будем учить их программированию: «А теперь напишите программу для деления в столбик». И написав эту программу, они заодно освоят деление. Или это сработает, только если есть природная склонность к этому?

Фицпатрик: Для меня это сработало. Часто бывает, что кто-то учит тебя чему-то, и ты говоришь: «Да-да, понятно». И обманываешь себя. Но как только придется заняться этим на практике, понять все граничные случаи, придется на самом деле освоить этот предмет. Но я не знаю, подходит ли это всем.

Сейбел: Говорят, в Google, как и в Microsoft, на собеседованиях задаются вопросы в виде головоломок.

Фицпатрик: Кажется, такие вопросы запрещены или, по крайней мере, не приветствуются. Может, кто-то и задает их, но, скорее всего, в общем случае они не приветствуются.

Сейбел: А вас о чем спрашивали на собеседовании?

Фицпатрик: Например, был вопрос: представьте, что у вас есть несколько компьютеров, подключенных через коммутатор, которые занимают целую стойку. Напишите алгоритм, чтобы каждая машина в стойке знала статус любой другой машины – включена та или нет. То есть задача в целом сводилась к определению присутствия. Это было серьезным ограничением. В основном они описали схему работы сети Ethernet: вы можете послать широковещательное сообщение всем или посылать сообщение по конкретному MAC-адресу. Надо было проанализировать множество различных стратегий для минимизации полосы пропускания и времени определения того, что один из компьютеров вырубился. Это была интересная задача.

Сейбел: Какую из найденных вами ошибок вы считаете самой серьезной?

Фицпатрик: Я стараюсь их не запоминать. Ненавижу, когда предположения столь сильно расходятся с реальностью. Недавно (это явно не

пример самой плохой ошибки в моей жизни) я потратил полтора часа на отладку, потому что писал в один файл, а читал другой – с таким же точно именем, только путь к нему был на один элемент короче. Я продолжал перезапускать этот огромный MapReduce, наблюдая за выходными данными, и наконец запустил GDB для пошаговой отладки. «Какого хрена? – говорил я себе. – Ничего не меняется!» В конце концов я глянул на пути и воскликнул: «Бог ты мой!», – не знаю, как я мог потратить на это полтора часа. Я был так одержим, что даже не вернулся, чтобы проверить корректность командной строки.

И так бывает нередко. Мы постоянно сталкиваемся с подобными вещами в Perl, например, когда переменная `$_` не определена в лексической области видимости. Возишься с `$_` в сортировке, а на самом деле используешь значение, определенное где-то очень далеко. Эта ошибка доставала нас постоянно, создавая немало проблем. Когда мы наконец выяснили в чем дело, я провел аудит всего нашего кода, и мы ввели правило «никогда не делай этого».

Сейбел: Какие инструменты вы используете для отладки? Отладчики? Операторы `println`? Еще что-то?

Фицпатрик: Я использую операторы `println`, если среда позволяет это. Если в среде есть хорошие отладчики, использую отладчик. GDB хорошо поддерживается в Google и порой просто незаменим. Стараюсь использовать его пореже. Я в нем не такой уж большой специалист, но могу осмотреться и представить положение вещей в целом. Если приходится забираться в дебри, то я всегда могу как-нибудь выпутаться. Я люблю утилиту `strace`, просто не представляю жизни без нее. Если я не знаю, что делает моя или чья-то программа, то запускаю ее под `strace` и вижу, что конкретно в ней происходит. Если бы мне пришлось выбрать только одну утилиту, я бы выбрал именно ее. Все инструменты, вроде `Valgrind` и `Callgrind`, очень хороши.

Но в последнее время, если происходит что-то странное, я поступаю так: «Хорошо, вот эта функция слишком велика; давайте разобьем ее на части поменьше и напишем модульные тесты, чтобы проверить работоспособность каждой части независимо и найти место, в котором мои предположения оказались ошибочными, а не втыкать операторы `println` где попало».

Бывает позже, в процессе рефакторинга, я начинаю думать о коде больше, и проблема становится очевидной. Тогда я могу вернуться к той огромной уродливой функции и исправить ее, но половину исправлений я уже внес; я могу продолжить, чтобы облегчить работу того, кто будет поддерживать код после меня.

Сейбел: Как вы используете инварианты в своем коде? Одни программисты добавляют специальные утверждения, другие добавляют инварианты на каждом шагу, что позволяет им проверить формальные свойства своей программы. Между этими двумя крайностями много промежуточных вариантов.

Фицпатрик: Я не сторонник чисто формального подхода. Мое основное правило: если некорректные данные могут приходиться от конечного пользователя, то это не ошибка времени выполнения. Но если взаимодействие осуществляется между двумя кусками моего кода, я прерываю выполнение настолько жестко, насколько это возможно, – чем раньше упадет программа, тем лучше.

Я стараюсь мыслить главным образом в терминах предусловий и проверяю данные в конструкторе и в начале функции. При этом, если возможно, я использую условные проверки, чтобы компилятор мог их впоследствии выкинуть. Тут есть много подходов, я недостаточно подкован, чтобы говорить о том, какой из них самый верный. Есть языки, в которых все это является частью самого языка. Но почти во всех языках, которыми я пользуюсь, это оставлено на усмотрение программиста.

Сейбел: Вы писали когда-то, что оптимизация – ваш любимый процесс в программировании. Это все еще так?

Фицпатрик: Я люблю оптимизацию за то, что без нее можно обойтись. Вы делаете это, когда все уже работает, а это самое важное. Дальше вы либо экономите деньги, либо устраиваете соревнования по гольфу на языке Perl¹: как сделать этот код короче или значительно быстрее? Нам надо было обнаружить наиболее часто используемые участки кода в Живом Журнале, и я устроил небольшие состязания: «Вот код. Вот тестовая программа. Сделайте код максимально быстрым». И выслал парсер заголовка балансировщика нагрузки. Мы писали безумные регулярные выражения, без поиска с возвратом, пытаюсь захватить что-то самым эффективным способом. Мы соревновались друг с другом, получая все более эффективные решения. Но на следующий день пришел один парень. Он написал все на C++ с применением XS² и говорит: «Я выиграл».

Сейбел: Сегодня обратная сторона этого в том, что...

¹ Code golf – состязания в программировании: побеждает программа с минимальным количеством знаков, решающая данную задачу. Понятие пошло именно из языка Perl (Perl Golf Context), но потом распространилось и на другие языки программирования. – *Прим. науч. ред.*

² XS – интерфейс, определяющий формат файлов для взаимодействия кода на Си/C++ и кода на Perl. – *Прим. науч. ред.*

Фицпатрик: Время программиста дороже подобной ерунды? Да, это верно, но только если компьютеров мало. Если количество компьютеров возрастает, время программиста начинает стоить меньше, чем компьютеры, на которых установлено ПО. В этом случае стоит писать на Си, и профилировать свой код, и исправлять компилятор, и платить людям за работу на GCC для ускорения компиляции.

Сейбел: Но даже Google использует C++, а не ассемблер, так что есть какая-то точка, в которой попытка выжать максимальную производительность становится невыгодной. Или есть теория, что хороший компилятор C++ генерирует лучший код, чем все программисты на ассемблере, которых еще поискать?

Фицпатрик: Мы все еще пишем кое-что на ассемблере, но крайне редко. Мы профилируем огромное количество кода, и бывают случаи, когда требуется переписать код с Perl на Си, а потом с Си на ассемблер. Но даже для платформы x86 есть различные варианты этой платформы. Вы действительно хотите писать разный ассемблерный код для каждого варианта платформы x86? Этот процессор использует SSE 2, а этот только SSE 3.1. Пусть лучше всем этим занимается компилятор.

Сейбел: Вы учились программированию по руководству программиста еще в детстве. Есть ли книги, которые вы настоятельно рекомендуете начинающим или всем программистам?

Фицпатрик: Если говорить о Perl, то даже программисту, который знает его хорошо, я бы посоветовал книгу Марка Джейсона Доминуса (MJD) «Higher-Order Perl» (Высокоуровневый Perl). Книга действительно отличная, автор начинает с простых вещей, и вы думаете: «Знаю, знаю, что такое замыкания», — но он потихоньку долбает ваш мозг, и к концу книги он просто готов взорваться. И хотя теоретически я знал все это, но увидев столь экстремальный подход, изменил свое мнение. Я рекомендовал эту книгу многим своим друзьям, им она тоже взорвала мозг. В целом, я бы советовал читать книги, которые заставляют думать по-новому. Это просто самый свежий пример, который мне вспомнился.

Сейбел: Вижу, у вас есть книга «The Art of Computer Programming»¹, но она не выглядит слишком потрепанной. Вы много из нее прочли?

Фицпатрик: Я не трогал ее около пяти лет, да, не меньше. Иногда я брался за нее и читал отдельные куски ради удовольствия. Но к тому времени, как я купил эту книгу, я уже знал многое из того, что в ней описано, по университетским занятиям. Так что от нее было бы больше пользы раньше. Но до появления Интернета я ничего о ней не знал.

¹ Дональд Э. Кнут «Искусство программирования». — Вильямс, 2008.

Сейбел: Как вы считаете, в каком объеме программист должен знать математику? Чтобы прочесть и как следует понять книги Кнута, нужно иметь приличную математическую подготовку. Но необходимо ли это сегодня программисту?

Фицпатрик: Ему не нужно так уж много математики. Для большинства программистов в ежедневной работе гораздо важнее статистика. Если рисуешь всякие графики, то математика, само собой, нужна, но большинство программистов в основном разрабатывают разные корпоративные приложения на Java и всякую ерунду для Сети. Очень помогает знание логики, ну и статистика тоже очень полезна.

Сейбел: Вы явно все еще получаете удовольствие от программирования. Но если почитать ваши записи в Живом Журнале времен учебы в колледже, создается впечатление, что вы порой уставали и начинали ненавидеть компьютеры.

Фицпатрик: О да, я всегда ненавидел компьютеры. Похоже, тут уже давно нет никакого прогресса. Кажется, что компьютеры все медленнее работают, все чаще падают и глючат, чем раньше. Но поскольку я оптимист, то верю, что все изменится к лучшему. Наверное, десять лет назад я получал больше удовольствия от работы, чем сегодня. Может быть, десять лет назад мой компьютер работал быстрее; мне кажется, что тогда он работал лучше. Компьютеры стали быстрее, но в то же время программы стали больше глючить и работать медленнее.

Сейбел: Как думаете, почему?

Фицпатрик: Не знаю. Может, понизилась планка? Или, раз компьютеры стали быстрее, можно забыть об эффективности – или совсем не думать о том, что делаешь? У меня нет ответа. Может быть, всё выше-сказанное в совокупности. А может, просто появилось такое множество слоев абстракции, что люди перестали понимать, что же происходит внутри, а компьютеры стали настолько быстрыми, что скрывают их глупость.

Сейбел: То есть, возможно, программы не настолько быстро работают, как должны бы работать при такой скорости компьютеров. Но ведь десять лет назад вообще не было таких возможностей, какие сегодня предоставляет пользователям тот же Google.

Фицпатрик: Да. Некоторые пишут эффективный код и им пользуются. Я не играю в компьютерные игры, но когда порой наблюдаю за чьей-то игрой, думаю: «Черт, как такое возможно?» Я просто в шоке. Очевидно, что некоторые делают все правильно.

Наверное, в основном я недоволен своими настольными приложениями. Похоже, большинство хорошего, интересного происходит на стороне сервера. А работая на своем компьютере, я все больше разочаровы-

ваюсь. Наверняка мой Мак не должен постоянно показывать мне вращающиеся пляжные мячи смерти¹.

Сейбел: Не интересует ли вас написание более качественных настольных приложений?

Фицпатрик: Проблема в том, что они никому не нужны. Хочется разрабатывать приложения, которыми будут пользоваться, а сейчас всем нужны веб-приложения. Однажды я потерял ноутбук, и все спрашивали: «Как, ты потерял информацию?» Но на нем не было никакой информации. Это был просто терминал для работы в Сети. А диск был зашифрован, так что я не беспокоился насчет паролей, cookies и всего такого. Мне кажется, люди уже не хотят скачивать программы.

Сейбел: А что вас сильнее мотивирует: то, что ваш продукт используют другие, или просто удовольствие от программирования?

Фицпатрик: Конечно же, есть программы, которые я пишу для себя, и пишу явно в расчете на то, что ею буду пользоваться я один; в этом случае я меньше забочусь насчет патчей и всего такого. Но часто мне хочется работать с другими. Наличие пользователей – ключевой момент для совместной работы. Чем больше пользователей, тем больше ошибок они находят, тем больше сценариев использования. Работать с другими людьми интереснее, особенно в проектах с открытым исходным кодом.

Всегда приятно, когда тебе пишут: «Привет, мы используем твою программу для того-то». Глядя на то, сколько сайтов используют memcached, балансировщик нагрузки или еще что-нибудь, я думаю: «Как здорово!» А сколько владельцев порносайтов сообщали мне, что пользуются моей файловой системой! Это о чем-то говорит. Я помогаю порноиндустрии. В Крейгслисте каждый запрос направляется через веб-сервер, который использует memcached. Вот так.

Сейбел: Как по-вашему, программисты склонны излишне увлекаться всем новым – языками, инструментами и так далее?

Фицпатрик: Да, возможно. Может быть, тут есть отчаянная надежда, что новая версия вдруг окажется нормальной, что новый язык будет делать все, что нужно. Но так думают и пользователи – они всегда хотят получить новую версию, даже если она хреновее.

Не знаю, отличаются ли статистически программисты от других людей. Новое – значит, лучшее. Далеко не всегда это так, но люди надеются. Они хотят, чтобы было именно так.

¹ Симптом зависания программ, работающих под управлением MAC OS: курсор превращается во «вращающийся пляжный мяч смерти» (Spinning Beach Balls of Death). – *Прим. науч. ред.*

Помню беседовал как-то со своим дантистом – она вспоминала все новые и новые методы в стоматологии, кажется, за последние лет пять, и была от них просто в восторге.

Сейбел: Искусство программиста сейчас во многом состоит в том, чтобы найти нужные компоненты и понять их ровно настолько, чтобы суметь ими воспользоваться. Что вы об этом думаете?

Фицпатрик: В CPAN¹ есть все. Там 14 парсеров ID3. Выберите один.

Сейбел: Так вот проблема современного разработчика – выбрать одно из 14 решений. Как вы делаете выбор?

Фицпатрик: Запускаем поиск в Google и смотрим, какой из них попадает в начало списка. Какой больше всего нравится людям? И нужно знать людей. Я стал значительно больше участвовать в жизни сообщества открытого исходного кода, посещать все эти конференции, поскольку там я встречаюсь с людьми и понимаю, кто пользуется большим уважением, кто действительно крут.

Потом я смотрю их код: а, помню того парня, он просто потрясающий. Он весел, дружелюбен, привлекателен, внимателен, и он действительно заботится о своем коде. Очень серьезно относится ко всем жалобам на его код. Я могу спокойно пользоваться его кодом, потому что знаю: если найду ошибки, он обязательно их исправит. Или наоборот, какой-нибудь ворчун, который, может, и пишет замечательный код, но постоянно ворчит и не хочет тебя слушать, если ты нашел ошибку или у тебя есть вопрос. Вот и выбираешь таких людей, поддерживающих конкретный проект, которым доверяешь, относишься с уважением.

Сейбел: Можно ли быстро выяснить, подходит тебе что-то или нет?

Фицпатрик: Я просто начинаю. Не включаю это сразу же в свой код, а сперва пишу тестовую программу, которая использует несколько нужных мне функций, чтобы убедиться, что все работает. Или пишу модульный тест только для этой библиотеки и только для тех данных, которые предполагаю использовать вместе с ней. У многих библиотек нет собственных тестов. Но даже если есть, может быть так, что читаешь документацию и не доверяешь тому, что в ней написано, или по ней невозможно понять, что делается. Поэтому я пишу собственные тесты для того, что мне нужно. Поскольку для изучения этой библиотеки мне все равно придется написать пробную программу с ее использованием, я вполне могу начать с модульного теста.

Сейбел: А как насчет инструментов, которыми вы сейчас пользуетесь, – по-прежнему применяете Emacs?

¹ CPAN (Comprehensive Perl Archive Network – всеобъемлющая сеть архивов Perl) – архив документации и ПО на языке Perl. – *Прим. науч. ред.*

Фицпатрик: Да, я все еще использую Emacs. Правда, не знаю его так хорошо, как хотел бы. Я знаю все горячие клавиши, но сам не настраиваю его. Пользуюсь чужими настройками и почти могу прочесть их. Иногда мне это надоедает, и я думаю: «Пора бы написать кое-что на Lisp¹ для привязки этой штуки к горячей клавише», – но никогда этого не делаю.

Стив Йегг (Steve Yegge) работает над проектом, который в основном заменил бы язык Lisp на JavaScript. Жду, пока он закончит, чтобы не учить еще один язык. Пишу все эти штуки на JavaScript, но не рассматриваю его как язык. Это язык для браузеров. В Google я много чего писал на JavaScript и потом встраивал это в Java и C++. Я понял, что JavaScript для этого отлично подходит.

Сейбел: Есть инструменты, которыми вы регулярно пользуетесь, при том что терпеть их не можете? Кроме ваших настольных приложений.

Фицпатрик: Да, все настольные приложения. У меня их куча. Эти браузеры все время виснут, падают и занимают кучу памяти. Вся моя операционная система виснет. Коллеги, видя, как я работаю в Emacs, пытаются убедить меня, что Eclipse или IntelliJ все это делают у них автоматически. Раз в полгода я пробую то или другое. Но эти чертовы штуки просто крутятся там, съедая память и падая в процессе ввода (наверно, не справляются с тем, что я набираю). Ладно, фоновая подсветка синтаксиса или компиляция в другом потоке. Но зачем при этом блокировать ввод с клавиатуры? Что ж, снова попробую то или другое через полгода. И я счастлив, что не обязан использовать все это. Мне и с Emacs неплохо.

Моя кривая обучения выглядит так: я быстро изучаю что-нибудь, пока не смогу работать с этим достаточно быстро и качественно. Обычно это где-то 80–90%, тогда я работаю весьма продуктивно, мне не приходится постоянно что-то искать, вот и отлично. После этого я продвигаюсь уже медленнее. И только если мне что-то очень уж приглянулось, говорю себе: «Пора поглубже изучить документацию этого языка – мануалы – и обшарить все потайные углы и щели».

Сейбел: Насколько это разумно сегодня? Ведь столько вещей, которые можно изучить. Можно бесконечно изучать работу со своим редактором, но сколько программ вы сумеете написать при этом?

Фицпатрик: Да, но для меня такие усилия всегда окупаются (по крайней мере, что касается редакторов). Когда я изучаю что-то, это окупается в течение недели, может быть двух. Если я пишу дурацкий скрипт

¹ Lisp, или Emacs Lisp – диалект языка Лисп, используемый в текстовых редакторах GNU Emacs и XEmacs. – *Прим. науч. ред.*

командной оболочки или маленький скрипт на Perl, или еще что-то, что автоматизирует мою жизнь, это всегда окупается.

Сейбел: Значит, вы никогда не попадались в ловушку бесконечного усовершенствования инструментов?

Фицпатрик: Нет. Я все это делаю с какой-нибудь целью. Да, я знаю тех, кто постоянно работает над своими инструментами, никогда ничего не доводя до конца. И я могу двинуться в этом направлении, но недалеко.

Сейбел: Каков, по-вашему, самый важный навык для программиста?

Фицпатрик: Мыслить как ученый; за один раз изменять только что-то одно. Терпение и стремление понять коренную сущность вещей. Это особенно ценно при отладке или проектировании чего-то, что не желает работать. Иногда молодые программисты говорят: «Черт, эта штука не работает», – и переписывают ее целиком заново. Стоп. Попробуйте понять, что происходит. Научитесь разрабатывать программу шаг за шагом, так чтобы была возможность проверять ее на каждом шаге.

Сейбел: Вы делали что-нибудь особое, чтобы улучшить свое мастерство программиста?

Фицпатрик: Иногда я отхожу от привычного пути – пишу что-нибудь на языке, которым обычно не пользуюсь. Я знаю, что это займет больше времени, но также знаю, что это все-таки благо. Например, когда я пришел в Google, мне часто приходилось писать всякие однодневные программы, и я всегда писал их на Perl. А потом сказал себе: «Стоп, а напишу-ка я это на Python». Теперь я пишу массу всего на Python, и это меня не напрягает – даже не приходится что-то искать. Perlbal был изначально написан на C#, просто чтобы выучить этот язык.

Сейбел: Какие навыки, помимо профессиональных, должен развивать программист?

Фицпатрик: Коммуникабельность, хотя я не уверен, что ее действительно можно развить. Больше общайтесь по электронной почте. Умение общаться в письменном виде очень важно. Но оно и в жизни пригодится, правда? Было какое-то исследование насчет того, кто из выпускников более успешен – умные ребята или общительные? Выяснилось, что именно общительные ребята могут заработать любые деньги, а не те, кто хорошо учился. Думаю, это интересно.

Сейбел: Мне кажется, со временем что-то изменилось. Раньше программисты могли прятаться по офисам, как гномы. Сейчас они активно переписываются, общаются друг с другом.

Фицпатрик: Да, везде, где я работал, в проектах с открытым исходным кодом или в компаниях, все зависят друг от друга. Побудительный мо-

тив: «Я напишу это, потому что знаю, что тебе это понадобится через две недели, или через две недели мне понадобится твое». Все это относится к человеческим взаимоотношениям.

Сейбел: Говорят, лучший программист на порядок продуктивнее худшего. Вы с этим сталкивались?

Фицпатрик: Да, но, по-моему, это характерно для любой области. Все зависит от опыта. Не то чтобы я часто встречал двух программистов, которые занимались примерно одним и тем же, и один сделал в десять раз больше другого. Но, похоже, если постоянно не совершенствовать навыки, начинаешь деградировать и выпадаешь из обоймы.

Думаю, есть те, кто занимается этим только ради заработка, но на самом деле без удовольствия. Что нормально. Но зачем сравнивать их с программистами по призванию? Кто продуктивнее – тот, кто постоянно думает о деле, или тот, кто думает о нем только в рабочее время?

Сейбел: Вы говорили о научном подходе к отладке. Считаете ли вы себя ученым, инженером, художником или ремесленником?

Фицпатрик: Ученым или инженером. Пожалуй, скорее инженером. А потом уже ученым – и только из-за принципа изменять за один раз что-то одно и подхода к диагностике проблем. Я считаю себя инженером в смысле подхода к проектированию. Среди моих друзей есть те, кто считает себя художником или ремесленником, но я о себе так никогда не думал.

Сейбел: С другой стороны, инженеры часто завидуют программистам. Помните шутку: «Если бы небоскребы строились так, как мы пишем программы, первый же дятел уничтожил бы цивилизацию». Как вы считаете, создание программного обеспечения – это понятная инженерная дисциплина?

Фицпатрик: Нет. Пока что это не так. Да, чтобы писать код, лицензия не нужна. Я не сторонник кучи предписаний, но не хотел бы, чтобы некоторые из РНР-программистов, допускающие все эти атаки через XSS, занимались системами управления полетами. Я бы предпочел, чтобы здесь была четкая официальная граница.

У меня есть друг, инженер-строитель, так он постоянно учится и получает все эти сертификаты. Как-то легче на душе от мысли, что те, кто строит мосты, по которым я хожу, тщательно изучают это дело, сдают множество тестов и постоянно учатся.

Сейбел: А какой тест вы могли бы дать программисту, чтобы убедиться, может ли он писать работоспособные программы?

Фицпатрик: Понятия не имею. Страшно подумать об этом.

Сейбел: Не говоря о лицензировании, есть ли у программистов этическая ответственность перед обществом? Очевидно, что это профессия, а у каждой профессии есть кодекс деловой этики.

Фицпатрик: Программа не должна никого убивать. Это касается, например, систем управления полетами. Но это достаточно частный случай. Я бы хотел попросить тех, кто создает формы для ввода номеров кредитных карт, предоставить мне возможность вводить эти чертovsky пробелы и дефисы. Компьютеры отлично справляются с удалением всей этой ерунды. Сделайте так, чтобы я мог не заботиться о формате. Но это вопрос не этики, а всего лишь человеческой глупости.

Сейбел: Сейчас вам 28. Не боитесь ли вы, что скоро станете слишком старым для программирования, что это удел молодых?

Фицпатрик: Нет. В худшем случае я всегда могу все бросить и заняться тем, что мне самому нравится. Мне не кажется, что я сейчас с кем-то соперничаю, и мне плевать, что кто-то другой справится лучше меня, потому что я знаю массу людей, которые и так меня превосходят. Я понял, что мы всегда где-то посередине, и я рад там находиться.

Сейбел: Значит, вы продолжите заниматься программированием ради удовольствия, даже если уйдете с работы?

Фицпатрик: Конечно. Я постоянно занимаюсь всякой ерундой. У меня на мобильнике есть простая плоская игра. Как-то раз я так устал, что не мог делать ничего серьезного и написал программку, которая проходит эту игру. Я пробовал разные размеры доски, создал множество случайных досок и построил график количества ходов для досок разного размера. Я послал эту программу автору той игры, потому что в игре очень плохо оценивалась игра партнера. Обычно для успеха в игре нужно превзойти партнера. В моей почтовой рассылке все только и говорили о том, что игра из-за этого оказалась проще, чем ожидалось, поскольку оценка партнера бралась, по-видимому, с потолка. Так что я отправил ему график для каждого размера доски. Думаю, в новой версии автор это исправил. Это был забавный хак, на одну поездку домой. Я могу уволиться и заниматься подобными глупостями целый день.

3

Дуглас Крокфорд

Старший архитектор JavaScript в Yahoo! Дуглас Крокфорд занимается программированием с середины 1970-х – тогда, в колледже, не получив студийного времени по своему основному предмету – телевидению, он прослушал курс Фортрана. Всю карьеру он совмещал программирование и медийную деятельность в таких компаниях, как Atari, Lucasfilm, Electric Communities, а теперь – Yahoo!.

Крокфорд по натуре склонен к простоте и аккуратности. Он изобрел JSON – формат передачи данных, широко применяемый в Ajax-приложениях, поскольку XML, с его точки зрения, слишком сложен. В его недавней книге «JavaScript: The Good Parts» (JavaScript: положительные стороны) утверждается, что JavaScript – весьма неплохой язык, если избегать некоторых его возможностей. В беседе со мной Крокфорд подчеркивает важность иерархии как способа борьбы со сложностью и описывает свой метод чтения кода, который начинается с его чистки.

Ко времени нашего интервью Крокфорд уже был известен как непримиримый критик ECMAScript 4 (ES4) – новой версии стандарта языка ECMAScript (JavaScript), поскольку считает ее слишком сложной. Он высказывался в пользу версии с менее масштабными изменениями – ES3.1. В конце концов, точка зрения Крокфорда и других защитников ES3.1 возобладавала, версия ES3.1 получила название ES5, а ES4 была официально отвергнута.

Мы с Крокфордом беседовали о том, что ему не нравится в ES4, о важности чтения кода как части работы в команде и о том, как совершенствовать Сеть, несмотря на имеющиеся в ней старые системы.

Сейбел: Как вы начали заниматься программированием?

Крокфорд: Я учился в Университете Сан-Франциско. Поступил туда потому, что там была хорошая система подготовки специалистов для телевидения. Но в первый год мне не удалось получить доступ к студии, и по счастливой случайности я решил посещать курс языка Фортран на математическом факультете. Оказалось, у меня прекрасно получается, так что я продолжил изучение в следующем семестре.

Это был 1971–1972 учебный год. В библиотеке хранилось множество перфокарт. Разделение времени только-только начало внедряться. В Университете Сан-Франциско не было сильного инженерного факультета, который бы занимался всеми компьютерами. Вместо этого они были распределены по всему колледжу. Свои лаборатории были на факультете естественных наук, в школе бизнеса, на гуманитарном и педагогическом факультетах. Интересно, что все эти факультеты использовали компьютеры.

Сначала я работал в научной лаборатории, потом в лаборатории факультета гуманитарных наук. Поэтому я встречался с приходившими туда экономистами, психологами, географами – все это были очень интересные люди. Узнавая о задачах, над которыми они работают, я многое понял насчет того, как чувствует себя обычный человек, общаясь с этими жуткими машинами, и стал задумываться над тем, как облегчить им жизнь.

Потом я все же получил доступ к студии и занялся всякими телевизионными делами. Это было весело, но в конце концов я решил посвятить себя компьютерам. При этом постоянно думал, как объединить обе эти вещи. Я предчувствовал многое из того, что сейчас называют мультимедиа и цифровым мультимедиа. Впоследствии я не раз возвращался к работе с медиа, а потом опять к компьютерам.

Сейбел: Итак, вы начали с Фортрана и поняли, что делаете в нем успехи. Чем еще привлекало вас программирование, кроме мысли о том, что у вас это получается?

Крокфорд: Только это. Это был мой первый семестр в колледже, надо было прослушать один курс на математическом факультете, я выбрал его наугад, и это оказался курс Фортрана. Так что я не шел туда специ-

ально для того, чтобы научиться программированию. Просто так вышло.

Сейбел: Помните свою первую действительно интересную программу?

Крокфорд: Давно это было. Кажется это была программа дизассемблирования работающей системы на Фортране для системы с разделением времени, которую я использовал. В процессе ее написания я понял, как работает система, и многое узнал о программировании, основанном на этой модели, включая некоторые вещи из того, что обычно не публиковалось.

Сейбел: По сравнению с тем временем, что больше всего изменилось в вашем подходе к программированию?

Крокфорд: Был период, лет десять, когда важнейшим фактором была производительность. Кажется, это было в самом начале эры микропроцессоров, когда объем памяти был крайне небольшим, а процессоры работали очень медленно. Мы засели за ассемблер для разработки игр или прослушивания музыки, чтобы они влезали в доступную память и работали достаточно быстро. Но этот этап пройден, и сегодня мы создаем большие приложения на JavaScript, работающие в браузере. Это чрезвычайно неэффективная среда по сравнению с теми, в которых мы привыкли работать, но благодаря закону Мура мы вполне можем себе это позволить.

Сейбел: Есть ли сожаления по поводу того пути, который вы выбрали, изучая программирование?

Крокфорд: Есть интересные языки, поработать с которыми мне так и не пришлось. Я много читал об APL и понимаю, почему он погиб, но это был очень логичный язык; жаль, что так и не удалось им позаниматься. Были и другие подобные языки, которыми я интересовался, читал о них кое-что, но так никогда и не получил шанса научиться думать на них.

Сейбел: Получив диплом телеведущего, чем вы занялись?

Крокфорд: Я поступал в магистратуру по технологиям образования. Но скоро понял, что знаю намного больше, чем мне преподают, и что все это пустая трата времени. Я бросил это дело где-то через год и пошел работать в Стэнфордский исследовательский институт в Менло-Парке. Потом я перешел в компанию Basic Four, которая производила миникомпьютеры для бизнеса, и проработал там довольно долго. Я зарабатывал текстовый процессор для компании и начал кое-какие исследования насчет переносных машин и персональных компьютеров. Я пытался обратить внимание компании на персональные компьютеры. Первым в компании купил персональный компьютер и оставил его на своем столе, чтобы инженеры приходили и смотрели на детище IBM.

Но мне так и не удалось изменить корпоративную культуру компании, поскольку они были слишком зациклены на том, что делают.

Однажды на Рождество, кажется в 1981 году, я купил Atari 800. В компьютерном магазине был еще Apple II, но Atari смотрелся шикарнее, и я выбрал его. Я думал написать на нем текстовый процессор или создать для него язык программирования. Но процессор 6502 был просто ни на что не способен. То есть я потратил две тысячи долларов на эту штуку, но что она может? Ну, хотя бы игры. Поэтому я начал создавать компьютерные игры, продал одну из них компании Atari и получил предложение поработать в их исследовательской лаборатории в Сан-нивейле¹. Это была лаборатория, организованная Аланом Кэем (Alan Kay), – первое, что он сделал со времен работы в Xerox PARC. Там было просто здорово. Я проработал там два года, наблюдая, как компания рассыпается. Но мне удалось сделать кое-что интересное и поработать с прекрасными людьми.

Сейбел: До этого вы увлекались компьютерными играми?

Крокфорд: Разве что потратил несколько четвертаков на Space Invaders и Pac-Man². Мне нравились игры, но их фанатиком я никогда не был. Компьютерные игры мне нравились как еще одно место взаимодействия компьютеров и телевидения. Первое такое место, открытое для широкой публики. По-моему, это было действительно интересно.

Сейбел: Что было после краха Atari?

Крокфорд: Я перешел в компанию Lucasfilm и работал там восемь лет.

Сейбел: И разработка игры Habitat началась, когда вы работали там.

Крокфорд: Конечно. Этот проект начал мой друг Чип Морнингстар. Он придумал аватары³ и виртуальный графический мир. Сначала он занимался всем этим. Работало все это на компьютере Commodore 64 и ненагруженных сетях X.25. Проект был невероятно дальновидным, с огромным количеством правильных решений, просто потрясающе. Я наблюдал со стороны, подбадривал их, но в том, что они сделали, моей заслуги нет.

¹ Саннивейл – один из крупных городов Кремниевой долины. – *Прим. науч. ред.*

² Space Invaders (Космические захватчики) – игра для игровых автоматов, Pac-Man (Пакман) – компьютерная игра в жанре аркады. Две эти игры, разработанные в Японии в конце 1970-х, стали культовыми и сильно повлияли на развитие компьютерной индустрии. – *Прим. науч. ред.*

³ Аватар, или юзерпик (от user picture – картинка пользователя) – небольшое изображение, используемое для персонализации пользователя какого-либо сетевого сервиса. – *Прим. науч. ред.*

Сейбел: А потом вы вместе ушли и основали Electric Communities, которая построена на этих идеях?

Крокфорд: Да. Морнингстар и Рэнди Фармер ушли из Lucasfilm и основали компанию American Information Exchange, в которой применили идею социальной сети к онлайн-рынкам. Блестящая идея, но увы, опередившая свое время. Сделав это несколько позже, они могли бы стать чем-то вроде eBay.

Потом у нас появилась идея: давайте попробуем снова и создадим общую платформу, которая бы занималась развлечениями, социальными вещами, бизнесом – да чем угодно, – создадим платформу для всего мира. У нас были кое-какие мысли насчет того, как сделать ее полностью распределенной, чтобы не было единственного сервера, а она была бы распределена по всему Интернету. И мы придумали модели обеспечения безопасности, которые позволили бы ей быть полностью децентрализованной. Эта действительно отличная идея и привела к созданию Electric Communities.

Сейбел: Именно там и была создана первая версия языка E.

Крокфорд: Совершенно верно. Нам нужен был безопасный язык для разработки платформы и приложений под нее. Сначала мы пытались использовать язык Joule компании Agorics. Это был язык на основе модели акторов, и многое в нем делалось необычным способом; язык был блестящий, но слишком уж непривычный.

У нас были сомнения по поводу Joule. Мы собирались научить людей пользоваться этим языком. Но не слишком ли он экстравагантен? Тогда у нас появилась идея насчет языка E, который заимствовал основные концепции акторов языка Joule и был построен поверх языка Java.

Сейбел: Применялся ли язык E кем-нибудь, кроме его создателей?

Крокфорд: В своей первоначальной версии – нет. Старый E был диалектом Java. Из-за этого у нас возникали всякие проблемы с компанией Sun. Потом мы разработали язык сценариев E – более легкий, но с аналогичными возможностями. Именно этот язык сейчас известен как E.

Мы разработали этот язык в компании Electric Communities, но, кажется, так его и не использовали. В какой-то момент мы просто решили, что не будем его использовать. Несмотря на это язык оказался неплох, и мы продолжали его развивать. И я рад, что он все же выжил.

Один из плюсов работы в Electric Communities – там я научился думать в терминах замыканий. А начав заниматься Сетью, посмотрел на JavaScript и сказал: «Кажется, мне это знакомо». Ведь JavaScript многое унаследовал от Scheme, но из документации не понять, что в нем есть замыкания. Так что я открыл это случайно и сказал себе: «Ого,

здорово!» И стал продвигать идею, что на этом дурацком маленьком языке можно делать серьезные вещи.

Сейбел: Это возвращает нас к недавним спорам по поводу ECMAScript 4. Как я понял, вам нравится простота версии ES3 JavaScript.

Крокфорд: Ну, в конечном счете значимость изменений, которые можешь внести в язык, зависит от успеха этого языка. Чем более успешен язык, тем выше цена вносимых изменений. Возрастает цена переобучения людей, кроме того, цена возможной неудачи при подобном росте становится неприемлемой. Если язык действительно успешный, нужно быть очень осторожным в плане изменений. В то же время, если язык пока не выпущен, свободы для внесения изменений гораздо больше.

JavaScript стал самым популярным в мире языком программирования чисто случайно. Сейчас процессоров, на которых могут выполняться программы на JavaScript, гораздо больше, чем для любого другого языка программирования. И несмотря на все проблемы с безопасностью, JavaScript – единственный язык, который даст вашему коду запуститься на каком угодно компьютере.

Но это еще не все: он встроен во множество приложений. Большинство приложений компании Adobe поддерживает JavaScript, так что вы можете использовать его локально. Многие другие приложения также поддерживают эту возможность. То есть он становится невероятно популярным.

Проблема с этим языком в том, что он ворвался на рынок слишком быстро и так же слишком быстро был стандартизирован. Так что большинство его недостатков связаны не с текущей реализацией, а находятся в спецификации. Стандарт указывает *делать это неправильно*. А это просто ужасно. Но ситуация именно такова. В 1999 году развитие этого языка замерло, все должны были его забыть, и он должен был умереть. Но вместо этого совершенно случайно возник Ajax, и теперь JavaScript – самый важный язык программирования в мире.

И теперь мы думаем, что должны его как-то исправлять. Это нужно было сделать еще в 2000-м, но тогда этого никто не сделал, поскольку на него тогда никто не обращал внимания. Теперь же это сделать очень сложно.

Есть еще одна особенность JavaScript в контексте Сети. Обычно, разрабатывая серверное, прикладное или встроенное (embedded) приложение, выбираешь не только язык, но и компилятор, и среду выполнения. Однако в случае JavaScript нет такого выбора: нужно, чтобы код работал на чем угодно.

Поскольку код должен выполняться на чем угодно, ошибки не исправляются. Если разработчик браузера выпустит его с ошибкой, то скажет:

«Ой, мы лажанулись», – и на следующий месяц выпустит следующую версию с другой ошибкой, а мы не можем подстраиваться под все обновления, которые устанавливают их пользователи. Большинство пользователей, установив однажды Internet Explorer, никогда его не обновляют. Ошибки в этом броузере остаются с ними годами.

Сейбел: То есть ситуация сейчас именно такова. Но вы хотите сделать Сеть более совершенной платформой для разработки приложений. Если мы не можем исправить свои ошибки, пока разработчики браузеров не исправят собственные, если даже и это не помогает, значит, мы в тупике. Где же выход?

Крокфорд: Как раз этим я и занимаюсь. Я вижу идеальное решение. Я знаю, каким оно должно быть. Знаю, где мы находимся, и вижу все преграды на нашем пути. И пытаюсь придумать, каким образом нам выйти из этого положения. Мы оказались в ловушке, в том смысле что разработали громоздкие системы – экономические, социальные, технические, которые в значительной степени зависят от этих непродуманных решений.

Несомненно, худшей особенностью JavaScript является зависимость от глобального объекта. В JavaScript нет компоновщиков и информация между элементами компиляции никак не скрывается. Все объединяется в общий глобальный объект. Так что каждый компонент видит все остальное; все компоненты имеют равноправный доступ к DOM; все они имеют одинаковый доступ к сети. Если какой-нибудь скрипт проберется на вашу страницу, он сможет зайти на сервер, представив себя в качестве вашего скрипта, и сервер никак не сможет отличить его от вашего.

У него есть доступ к экрану, он может представиться пользователю вашим скриптом, и пользователь также не сможет определить это. Все новые антифишинговые вещи, добавленные в Chrome, не работают, если страница пришла с вашего сервера, так что все скрипты обладают равными правами независимо от того, откуда они пришли.

Ситуация ухудшается еще и тем, что есть и другие пути попадания скрипта на вашу страницу. Архитектура Сети включает несколько языков: HTTP, HTML, язык URL-адресов, CSS и язык сценариев. Все они существуют и могут встраиваться друг в друга, обладая разными правилами цитирования (quoting), экранирования (escaping) и комментирования. Не во всех браузерах все эти правила реализованы одинаково. А некоторые из этих правил могут быть вообще не определены. Из-за этого злоумышленник легко может встроить какой-либо скрипт в URL, вставить его в кусок CSS, который затем вставить в HTML, а тот – в другой скрипт и так далее.

Сейбел: Классические межсайтовые скриптовые атаки, связанные с ошибками в браузере.

Крокфорд: Именно. Это ужасно. Нам нужно что-то с этим делать, потому что оставлять все это так совершенно не хочется.

В конце концов мы открыли мэшапы¹. Это как раз то, чего мы пытались добиться в области программного обеспечения в течение двадцати лет: компоненты многократного использования, которые можно объединять друг с другом, как детали в конструкторе LEGO, очень быстро создавая новые приложения. Мэшапы действуют аналогичным образом, и получается великолепно: берешь что-то из Yahoo!, что-то из Google, что-то свое, что-то чье-нибудь еще и делаешь из всего этого приложение. Класс! И все это делается в браузере, прямо у тебя перед глазами. Проблема одна: каждый из этих компонентов имеет доступ к тому же, что и другие. Мы сознательно создаем условия для XSS-атак. Модель безопасности браузеров не ожидает от этого ничего хорошего и не предоставляет никаких механизмов взаимодействия при взаимном недоверии. Вся всемирная Сеть построена на сплошных ошибках. Результат – множество неприятностей.

Сейбел: Можно ли из всего этого сделать вывод, что все усилия по принятию стандарта ES4² относятся к альтернативным издержкам и все думают именно об этом, а не о том, как избавиться от имеющихся проблем?

Крокфорд: Именно так. Этот стандарт решает не ту проблему. Он решает проблему, связанную с тем, что люди ненавидят JavaScript. И я могу понять позицию Брендана Айка, поскольку он проделал потрясающую работу, но он торопился, допускал ошибки в руководстве, и в итоге вышла полная ерунда. Его ругали и поносили последние двенадцать лет за его глупость и за то, что он создал глупый язык, но все это не так. Там есть блестящие идеи, и сам Брендан – блестящий парень. Он сейчас пытается оправдаться и доказать, что действительно умен, и показать это всем с помощью языка, который будет содержать все классные возможности, которые он когда-либо видел, объединенные и работающие все вместе.

¹ Мэшап (mash-up) – веб-приложение, объединяющее данные из нескольких источников в один интегрированный инструмент; например использует картографические данные Google Maps для добавления к ним данных о недвижимости с Craigslist, в результате создавая новый уникальный веб-сервис, изначально не предлагаемый ни одним из источников. – *Прим. науч. ред.*

² Имеется в виду четвертая версия стандарта ECMAScript, работа над которой так и не была завершена. – *Прим. науч. ред.*

Не думаю, что именно эту проблему нам нужно сейчас решать. Я думаю, что нужно решить следующее: Всемирная Сеть поломана, и мы должны ее починить. Нам нужно решить, каким путем продвигаться вперед. И мое главное возражение против того, что собирается сделать Брендан, в том, что он всех отвлекает.

Думаю, нужно двигаться шаг за шагом. Если у нас появится модульность или возможность выбирать язык программирования, это уже будет шаг вперед. Это еще не все, что нам нужно, но гораздо больше того, что есть у нас сейчас. Потом есть такие вещи, как Caja и ADsafe, пытающиеся сделать то же самое с помощью сегодняшних технологий. Мы не можем ждать.

ADsafe создает безопасное подмножество языка JavaScript, блокируя доступ ко всему глобальному и опасному. Но даже это подмножество все еще представляет собой полезный язык. У нас все еще остаются лямбда-выражения, а они могут многое. Таким образом, это нестандартный язык. Он не позволяет использовать прототипы так, как мы уже привыкли. Но этот язык остается невероятно мощным, поскольку в нем присутствуют лямбда-выражения.

Сейбел: Возвращаясь к ES4: есть ли в нем хоть что-то, что вам нравится с точки зрения языка?

Крокфорд: В нем исправлены некоторые ошибки языка, чем следует воспользоваться. Но в этом стандарте слишком много неопробованных возможностей. Наш опыт со стандартом ES3 говорит о том, что если ошибка однажды закралась в спецификацию языка, то удалить ее невозможно. Но у нас нет опыта работы с этим языком. Никто пока не написал на нем ни одного крупного приложения.

А он будет стандартизирован и внедрен до того, как мы поймем, что он действительно работает. Думаю, мы слишком торопимся. Я бы чувствовал себя спокойнее, если бы существовали примеры его реализации и полезные приложения, написанные на нем. Вот тогда я бы сказал: «Хорошо, давайте стандартизировать язык, давайте внедрять его по всему миру». А мы все делаем в обратном порядке.

Сейбел: Google Web Toolkit (GWT) позволяет компилировать Java в JavaScript. Многие уже пробовали компилировать другие языки в JavaScript. Это правильный путь?

Крокфорд: Любопытно наблюдать, как JavaScript становится универсальной средой выполнения. Мы никогда не ожидали увидеть его в такой роли.

Сейбел: Но, как вы уже говорили, он везде. Он стал универсальной средой выполнения.

Крокфорд: Что тем более заставляет нас обратить внимание на производительность. Особенно при переходе на мобильные платформы, к которым закон Мура неприменим. Здесь уже имеет большое значение то, сколько времени мы тратим на интерпретацию. Все это дополнительные такты процессора. Так что, думаю, это должно дополнительно улучшить качество среды выполнения.

Я придирчиво наблюдаю за тем, чего достигли GWT и другие средства преобразования. С этими средствами очень трудно работать – если найдете что-нибудь работающее, вам повезло. Лично я опасюсь использовать их, поскольку боюсь дыр в абстракциях¹. Если возникнут проблемы с вашим Java-кодом, или с GWT, или со сгенерированным кодом, то у вас может быть, а может и не быть возможности справиться с этим. Особенно в том случае, если предпочтете ничего не знать о JavaScript, поскольку этот язык скрыт от вас. Тогда, если что-то пошло не так, вы столкнетесь с огромными проблемами. Я пока не слышал о подобных случаях, и это значит, что те, кто этим занимается, делают всё правильно. Но такой риск определенно есть.

Сейбел: Каким бы вы хотели видеть JavaScript?

Крокфорд: Думаю, лучший способ усовершенствовать JavaScript – сделать его более компактным. Если бы можно было оставить только те возможности, которые работают действительно хорошо, убрав малозначительные или ненужные, то мы бы действительно получили заметно улучшенный язык. И я думаю, что этот же подход применим и к HTML, и к HTTP, и к CSS. Мне кажется, что работая со всеми этими стандартами, нужно выяснить, что они делают правильно, а чего в них не хватает, и на этом основании переориентировать их, а не просто добавлять новые функции поверх существующих.

Сейбел: Вечное противоречие: маленькие изящные драгоценности и распolzающиеся, но полезные комья земли. Маленькую изящную драгоценность легко понять, в ней нет недостатков, но чтобы сделать что-либо, нужно построить что-то еще поверх нее. Поэтому все заново реализуют одно и то же снова и снова, что приводит к различного рода разбуханию и уродству.

Крокфорд: На самом деле все не так. Есть разработчики Ajax-библиотек, которые достигли немалой изощренности во владении языком. А есть

¹ Закон дырявых абстракций, сформулированный Джоэлом Спольски, гласит, что использование абстракции любой нетривиальной концепции в любом случае потребует от ее пользователя четкого понимания внутренних аспектов реализации, в противном случае он рано или поздно столкнется с проблемами, с которыми не сможет справиться. См. <http://www.joelonsoftware.com/articles/LeakyAbstractions.html> – Прим. науч. ред.

сообщества разработчиков, которые делают черт знает что с помощью этих библиотек, и это работает. Для создателя приложений совсем не обязательно знать все свойства лямбда-выражений, чтобы извлекать из них пользу. Совсем не обязательно отказываться от языка, пытаясь исправить его ошибки.

На самом деле проблема в том, что Ajax-библиотек стало слишком много. Это следствие того, что JavaScript – очень мощный язык, потребность в подобных библиотеках очень высока, а создавать их относительно легко. Поэтому в течение какого-то времени все их и писали. Я жду, что их станет меньше, но пока напрасно. Из-за этого у нас сейчас другая проблема – библиотек столько, что разработчики не знают, какую выбрать. Думаю, все же в будущем их количество уменьшится.

Сейчас заметна тенденция сближения функциональности Ajax-библиотек. В jQuery можно получить список объектов из DOM с помощью CSS-селекторов и затем управлять группой объектов. Это оказалось действительно хорошей идеей – вот пример того, как JavaScript кое-что делает эффективно. Да, интерфейс взаимодействия с DOM просто ужасен, но он скрыт полностью. Разработчики jQuery очень упростили программную модель и сделали это блестяще.

А теперь каждая библиотека делает то же самое – мы наблюдаем сближение функциональных возможностей. Для пользователей проблема встает еще острее, так как все труднее выбрать нужную библиотеку, поскольку все они становятся очень похожими. Но в конце концов они начнут объединяться, и в результате останется всего несколько библиотек, возможно всего одна. Думаю, одним из победителей будет Microsoft с ее библиотекой Atlas, просто потому что Microsoft всегда оказывается среди победителей. Но, по-моему, у них нет достаточной поддержки. Открытые фреймворки, похоже, намного эффективнее. Думаю, в конце концов победит один или два открытых фреймворка.

Сейбел: Сегодня вы архитектор и евангелист языка JavaScript в Yahoo!, поэтому часть вашей работы, по-видимому, в том, чтобы говорить JavaScript-программистам в Yahoo!: «Делайте так-то». Обязаны ли вы также анализировать код и проекты на соответствие общепринятым практикам кодирования и проектирования?

Крокфорд: Я всегда настаиваю на необходимости чтения кода. Думаю, чтение кода – самое полезное, что программисты могут сделать друг для друга: постоянно уделять часть своего времени чтению кода коллег. При управлении проектами программистов часто предоставляют самим себе, затем всю их работу объединяют, и если результат удается скомпилировать, то полученный продукт выпускают на рынок и забывают о нем.

Это приводит к тому, что если у вас в команде есть слабые или неуверенные в себе программисты, то это обнаруживается слишком поздно. В результате возникает риск того, что в проекте будет множество низкокачественных компонентов, которые приведут к срыву сроков, что неприемлемо. Кроме того, у вас могут быть блестящие программисты, которые оказываются недостаточно хорошими руководителями для других членов команды. Чтение кода решает обе эти проблемы.

Сейбел: Давайте поговорим о том, как читают код у вас.

Крокфорд: На каждом совещании есть ответственные за чтение кода: они разбирают и проверяют код под наблюдением остальных. Это отличная возможность для каждого понять, как его фрагменты кода будут согласовываться с фрагментами других.

Мы садимся за стол, перед каждым лежит стопка бумаг. Мы выводим код на экран и комментируем его по ходу чтения. Кто-то говорит: «Я не понимаю этот комментарий» или «По-моему, этот комментарий не описывает этот код». Это очень важно, поскольку как программист вы перестаете читать свои комментарии, не сознавая, что другому что-то может быть непонятно. Прекрасно, когда коллеги по проекту помогают поддерживать ваш код в чистоте – вы находите ошибки, которые никогда бы не нашли самостоятельно.

Думаю, час чтения кода полезнее двух недель работы тестировщиков. Это действительно эффективный способ устранения ошибок. Если у вас есть человек с большим опытом чтения кода, то новички благодаря ему узнают много такого, чего иначе не узнали бы; если же чтением занимаются новички, то код сможет дать им немало очень ценных советов.

И не нужно оставлять это на заключительный этап работы. Раньше мы откладывали чтение кода до завершающего этапа работы над проектом и никто не делал это постоянно, просто потому что мы опаздывали с выпуском проекта. Теперь я считаю, что чтением кода нужно регулярно заниматься в течение всей работы над проектом. Да, на чтение кода уходит время, но это дает множество преимуществ.

В частности, это облегчает контроль выполнения проекта, поскольку мы видим прогресс каждого участника, что позволяет довольно рано заметить, если мы вдруг начнем сбиваться с намеченного пути.

Я управлял проектами, в которых незадолго до срока сдачи люди говорили: «Все практически готово», – а затем я смотрел их код, но там еще ничего не было или была полная ерунда, или он был очень далек от завершения. Жуткое испытание для руководителя проекта, и я думаю, что чтение кода как раз помогает избежать подобных ловушек.

Сейбел: Допустим, мы читаем мой код. Выводим код на экран – и что дальше? Я буквально читаю код вслух?

Крокфорд: Да, строку за строкой, параллельно его комментируя. Именно так все и должно происходить. Когда есть время, мы читаем код построчно.

Сейбел: Вам приходилось обучать людей чтению кода? Думаю, непросто найти хороший компромисс – сделать достаточно критических замечаний и при этом не задеть самолюбие автора кода.

Крокфорд: Да, здесь требуется взаимное доверие между членами команды, и нужны четкие правила насчет того, что допустимо, а что нет. В неслаженной команде чтение кода приведет к тому, что участники просто разорвут друг друга на части. То есть в процессе чтения кода очень быстро выясняется, слаженная команда или нет. При чтении кода вы можете многое узнать и многому научиться. Поначалу процесс может казаться неестественным, но стоит войти в ритм, как это ощущение пройдет.

Другой момент: нужно писать код так, чтобы другие смогли его прочитать. Здесь важны ясность кода и стиль его написания. Благодаря этому будет постепенно улучшаться качество кода и компетентность команды в целом.

Сейбел: Что для вас делает код читаемым?

Крокфорд: Читаемость складывается из нескольких уровней. Самое простое: нужно быть последовательным в оформлении кода, всегда использовать одинаковые отступы, расставлять пробелы во всех нужных местах. У меня есть дурная привычка, приобретенная еще во времена программирования на Фортране: я использую слишком много однобуквенных переменных, а это нехорошо. Я очень стараюсь избавиться от этой привычки, но это трудно сделать.

Сейбел: Насколько это трудно? Вы пишете код, а затем перечитываете его и думаете: «Боже! Только посмотрите на все эти однобуквенные переменные!»?

Крокфорд: Я мыслю однобуквенными выражениями. Кроме того, в JavaScript есть неоспоримый аргумент, связанный с эффективностью: за скачивание лишних символов вы платите, а более короткие имена переменных помогают сделать программу компактнее.

Сейбел: Для этого есть специальные инструменты?

Крокфорд: Да, можно применить `gzip` – и готово, так что у меня нет оправдания. Если, просматривая свой старый код, я вижу слишком короткие имена, то при наличии времени я их исправляю. Некоторые переменные, вроде счетчиков цикла, я всегда называю `i`. Не думаю, что когда-то стану исправлять и это. Для многих переменных длинные имена просто неоправданны.

Это первый уровень, грамматический. Как в естественных языках, английском или любом другом. Исправляешь пунктуацию, написание заглавных букв, расставляешь запятые в нужных местах. Затем начинаешь обращать внимание на более высокоуровневые вещи, такие как построение предложений, разбивка на абзацы. В языках программирования этому соответствует то, как задача разбита на набор функций или классов.

Сейбел: На какие конкретно вещи должен обращать внимание разработчик, чтобы его код легко читался?

Крокфорд: Действительно важно использовать определенное подмножество языка, особенно в JavaScript, где есть огромное количество неудачных возможностей. Но это актуально для всех языков. Будучи начинающим программистом, я читал руководство по языку программирования и старался понять каждую возможность. И то, как все их использовать. И постоянно использовал все эти возможности. А потом оказывалось, что многие из них были не самыми удачными.

Вспоминается Фортран, но это справедливо для всех языков. Иногда авторы языка допускают ошибки. С моей точки зрения, в языке Си полно ошибок.

Сейбел: Каких, например?

Крокфорд: Например, оператор `switch` изначально является неудачным, не нужно было делать его таким. У оператора `++` огромные проблемы в смысле безопасности, поскольку он провоцирует вас на разные хитрости и попытки сделать слишком многое в одной строке кода. В результате код становится трудным для понимания, что часто приводит к различным ошибкам, таким как переполнение буфера. Так что большинство проблем безопасности, которые мы наблюдаем в операционных системах последние пять лет, связаны с использованием оператора `++`.

Обычно я *вообще* не использую оператор `++`. Бывает, его можно использовать, но чаще нет, и мне трудно различить эти случаи в своем коде.

Сейбел: Но ведь можно возразить, что проблемы безопасности, связанные с оператором `++`, возникают не из-за самого оператора `++`, а из-за отсутствия проверки выхода за границу массива или из-за проблем с обычными указателями. В Java нет подобных проблем с безопасностью, поскольку если есть оператор `++` и происходит выход за границу массива, то получается исключение.

Крокфорд: Да, в Java это менее опасно. А в JavaScript такой опасности совсем нет, поскольку там отсутствуют массивы. Но в любом случае, отказавшись от этого оператора, я заметил улучшение качества моего кода, просто потому что перестал записывать выражения в одну строку.

Другой пример – оператор `continue`. Я еще не встретил ни одного фрагмента кода, который не смог бы улучшить, выкинув оператор `continue`. Да, с его помощью легче создать какую-то сложную конструкцию. Но я заметил, что всегда могу улучшить эту конструкцию, найдя способ выкинуть его. Так что лично я никогда не использую оператор `continue`. Если же я вижу `continue` в своем коде, значит, что-то недодумал.

Сейбел: Как вы читаете чужой код?

Крокфорд: Путем чистки: я переносу его в текстовый редактор и начинаю править. Начинаю с пунктуации и отступов. У меня есть для этого специальные программы, но я предпочитаю делать это вручную, поскольку так ближе знакомишься с кодом. Этому меня научил Морнингстар. Он блестяще проводил рефакторинг чужого кода, всегда применял этот подход и доказал его эффективность.

Сейбел: Вам встречался код, который поначалу выглядел сумбурно, но после чистки вы понимали, что на самом деле он хорош?

Крокфорд: Нет, такого никогда не случалось. Мне кажется, очень сложно небрежно написать хороший код (под хорошим кодом я понимаю читаемый). На этом уровне совершенно неважно, что означает этот код для машины, если я не могу понять, что он должен делать; он может оказаться удивительно эффективным, или компактным, или потрясающим еще в каком-то смысле, но это уже неважно.

Сегодня читаемость кода – мой главный приоритет. Она важнее эффективности и почти так же важна, как корректность, и я думаю, что читаемость кода – важнейший шаг к его корректности. А если код тяжело читать, то, по-видимому, разработчики выбрали неверные компромиссы, и нельзя этот код назвать хорошим.

Сейбел: А как насчет глубоко вложенного цикла, который должен выполняться невероятно быстро? Должен ли весь код быть читаемым – или иногда читаемость можно принести в жертву эффективности?

Крокфорд: Думаю, иногда можно пожертвовать читаемостью, но в таком случае я пишу целый роман, по обе стороны этого блока кода, с объяснением того, почему мы делаем то, что делаем. Обычно это упускают из виду. И я не раз видел, как люди боролись за эффективность, когда это совершенно не требовалось. Они не знали, на что тратит время их собственная программа, и оптимизировали код, не требующий оптимизации, поскольку он никогда не изменится настолько, чтобы его выполнение существенно влияло на общую производительность. От такой оптимизации нет никаких выгод или преимуществ, она лишь вносит дополнительный хаос. Я неоднократно встречал такое.

Сейбел: В языках программирования с фигурными скобками программисты ведут бесконечные холивары («священные войны») по поводу

того, где расставлять эти скобки, доказывают, что тот или иной стиль делает код более читаемым. Занимаясь чисткой кода, вы приводите его в форму, облегчающую восприятие?

Крокфорд: Непременно – ведь я считаю, что использую единственно правильный способ форматирования! Полагаю, Томпсон и Ричи оказали всем плохую услугу, не определив стандарты оформления кода для языка Си. Они сказали: «Мы используем такое оформление, а вы можете оформлять код как-то иначе», – и тем самым сильно навредили человечеству: теперь, возможно, люди всегда будут использовать только их версию.

Сейбел: То есть вы предпочитаете стиль отступов K&R¹?

Крокфорд: Да, думаю, Керниган и Ричи сделали все правильно и их исходный стиль правилен. В особенности это касается JavaScript. JavaScript вставляет точки с запятыми, и во многих местах смысл программы резко меняется в зависимости от того, слева или справа вы поставите скобки. Стиль K&R не страдает этим недостатком, в отличие от стиля без отступов.

Уверен, что для JavaScript это абсолютно правильный способ расстановки скобок. Но не могу сказать то же самое о других Си-подобных языках. Некоторым нравятся скобки без отступа, и я видел, как люди часами спорят о том, какой стиль правильнее. При этом оппоненты не воспринимают доводы друг друга, поскольку каждый защищает стиль, который использовал в школе, или на своей первой работе, или стиль, который применял тот, кто произвел на него впечатление, так что теперь ему нравится именно этот стиль, а все другие кажутся неправильными.

Это примерно то же, что спорить о достоинствах левостороннего и правостороннего движения. Разумных доводов в пользу того или другого нет. Если вы живете на необитаемом острове, то можете ездить как угодно, но общество определенно выиграет, если все будут ездить по одной стороне.

Сейбел: Если вам предложат новую работу, где надо программировать на Си или Java не в том стиле, какой вы предпочитаете, как вы поступите? Скажете: «Хорошо, я перейду на ваш стиль. Уверен, что вскоре буду этому рад»? Или откажетесь от такого предложения?

¹ K&R – стиль оформления кода с помощью отступов, названный так в честь Брайана Кернигана и Денниса Ричи, поскольку все примеры кода в их книге «Язык программирования Си» отформатированы подобным образом. Основной отступ состоит из 8 (реже 4) пробелов (или одной табуляции) на уровень вложенности. – *Прим. науч. ред.*

Крокфорд: Может быть, стоит всегда замечать, какой стиль принят в том или ином месте? Правостороннее или левостороннее там движение? И не соглашаться на работу там, где ездят не по той стороне дороги. Как в сказке Доктора Сьюза, настроение зависит от того, есть ли у тебя на животе звезда. В конце концов приходится перейти на принятый в компании стиль, в надежде, что люди, принявшие этот стиль, знали, что делают. Но если и не знали, неважно. Важнее, чтобы все шагали в ногу.

Сейбел: Итак, чистку кода вы начинаете с оформления. Насколько глубоко или существенно вы перерабатываете код?

Крокфорд: Я переупорядочиваю код так, чтобы объявление и инициализация каждой переменной были размещены непосредственно перед ее использованием. Отдельные языки допускают при этом некоторую гибкость, так что это не всегда необходимо. Но мне такая гибкость не нужна.

Сейбел: Значит, вы против предварительного объявления ?

Крокфорд: Да, против. Или, по крайней мере, предварительное объявление должно быть явным. Я против того, чтобы код располагался в произвольном порядке, кроме случаев литературного программирования (*literate programming*), когда я изменяю порядок кода для его наглядности, отказываясь от привязки к требованиям языка, – и мне это очень нравится. Но без специальных инструментов лучше это не делать.

Сейбел: В одном из интервью вы цитировали Исход, 23:10–11: «Шесть лет засевай землю твою и собирай произведения ее, а в седьмой оставляй ее в покое», – утверждая, что каждый седьмой цикл следует посвятить чистке кода. Какой разумный временной промежуток вы имели в виду?

Крокфорд: Шесть циклов – это циклы между выпусками чего-либо. Если вы выпускаете новую версию ежемесячно, то каждые полгода следует пропускать один цикл выпуска, посвятив это время чистке кода.

Сейбел: То есть, не делая это через каждые шесть циклов выпуска, можно столкнуться с необходимостью серьезного переписывания кода. А как вы определяете, что настало время для серьезного переписывания, – если с вами такое бывало?

Крокфорд: Обычно команда знает, когда пора этим заняться. Руководитель проекта понимает это гораздо позже. Работа замедляется, ошибок становится слишком много, код становится слишком большим и медленным, команда не укладывается в сроки. И все знают, почему. Не потому, что все внезапно поглупели или обленились, а потому, что кодовая база больше не отвечает своим задачам.

Руководителю очень сложно это увидеть, особенно если он не программист. Но даже для руководителя-программиста это очень непростая задача, поскольку к этому времени сделано слишком много вложений. Начать заново – значит вернуться назад и проделать тот же путь. Но это невозможно, поскольку мы не будем двигаться вперед. Поэтому они говорят: «Нет, будем двигаться вперед с тем что есть».

Ошибочно считать, что во второй раз будет затрачено столько же времени, сколько и в первый раз, хотя есть и противоположные примеры. Это так называемая проблема второй системы: когда те, кто уже чего-то достиг, получают задание начать все с чистого листа и делать то, что считают нужным. Обычно это ведет к провалу, поскольку люди становятся слишком амбициозными в своих целях и не видят границ. В результате вы не получаете ничего. Нужна невероятная дисциплина, чтобы можно было сказать: «Нет, мы не начинаем с чистого листа, а заново реализуем то, что уже сделали ранее; давайте делать то, что мы уже знаем».

Одна из главных трудностей программирования в том, что мы обычно занимаемся тем, чем никогда прежде не занимались. Если мы имеем дело с чем-то, что уже делали, то используем это повторно. Но в основном мы делаем то, чего никогда раньше не делали. А создавать что-то, чего никогда не делал, сложно. Заниматься этим очень интересно, но весьма непросто. Особенно когда работаешь по классической методике и должен классифицировать систему, которую до конца не понимаешь. В этом случае очень высока вероятность сделать неправильную классификацию.

Сейбел: Под «классической» методикой вы подразумеваете применение классов?

Крокфорд: Да. Мне кажется, что в мире прототипирования проблем гораздо меньше, поскольку там делается акцент на экземплярах. Найдешь экземпляр, типичный с точки зрения решаемой задачи, и готово. В таком случае решение не приходится адаптировать. Но в классических системах это невозможно – там всегда идешь от абстракций к экземплярам. И очень непросто создать правильную иерархию. Так что зачастую приходится возвращаться и перерабатывать свое решение, когда в конце концов лучше поймешь проблему. Но это может серьезно повлиять на код, особенно если он разросся по сравнению с первоначальной концепцией. Поэтому не делаешь ничего серьезного, пытаешься прикрутить что-то новое поверх уже существующей иерархии, и все еще больше запутывается и ухудшается.

Сейбел: Но ведь вы считаете рефакторинг кода полезным, раз советуете посвящать ему каждый седьмой цикл? И тогда необходимость в серьезном переписывании отпадает.

Крокфорд: Да, я считаю его полезным. Можно подумать о том, чтобы выкинуть все и начать сначала, только в том случае, когда это писал не ты или сделал это плохо, что-то пошло не так, и в результате получилась база кода, с которой невозможно работать. Всегда можно прикинуть, что быстрее – переписать все заново или вносить исправления.

Сейбел: А как быть с проблемой, когда не полностью понимаешь назначение того кода, который собираешься переписывать? Ведь каждый фрагмент кода несет в себе кусочки знаний – маленькие, неприметные кусочки, которые на самом деле являются частицами дорого доставшейся функциональности, о которой не думаешь, решая все переписать.

Крокфорд: Да, это серьезная проблема. Одна из причин, почему в Сети творится такая неразбериха, – отсутствие спецификаций. Спецификации были неполными, при этом еще и трактовались зачастую неверно, и многие из-за неверного понимания стали частью общепринятых правил. Все эти системы значительно сложнее, чем могли бы быть, именно благодаря историческим причинам. Работая на этом уровне, я, конечно, тепло отношусь к этим недокументированным знаниям, которые содержат исходный код.

У Microsoft аналогичная проблема с операционными системами: они годами выпускали лажу, делая ее совместимой с прежней лажей, основанной на всей лаже, созданной раньше. Поэтому ограничения, которые накладывают старые системы при проектировании новых, просто ужасны. В такой обстановке действительно сложно двигаться вперед. В конце концов они могут обнаружить, что двигаться вперед просто невозможно.

Подобные ошибки в спецификации – действительно очень и очень серьезная проблема. У нас есть такие проблемы в мире Ajax. Основные трудности, связанные с Ajax, заключаются главным образом в отличиях на уровне браузеров. Разработка кроссбраузерных приложений значительно сложнее, чем могла бы быть, именно из-за отсутствия полноценной спецификации Сети, а также из-за существенных различий в реализациях.

В последние пять лет ситуация значительно улучшилась, особенно с появлением Ajax-библиотек. Большинство из них делают очень полезные вещи, хотя и не все, что нужно, но достаточно для повышения вашего уровня как программиста. Теперь не нужно копать во внутренностях браузера, для этого у нас есть нечто вроде виртуального слоя, на основе которого можно разрабатывать весьма гибкие и переносимые приложения. Здесь, в Yahoo!, у нас есть команда, основная задача которой – работа с проблемами, связанными с браузерами. Хорошая работа этой команды значительно облегчает жизнь другим нашим разработчикам.

Сейбел: С другой стороны, переписывание системы не всегда срабатывает. Ранее вы упомянули эффект второй системы, а в другом интервью назвали это явление в действии «душераздирающим зрелищем». Когда это случилось?

Крокфорд: Это было в Electric Communities. Мы собрали там команду умнейших программистов, которую я когда-либо видел. У нас был достаточный бюджет, мы собирались заново реализовать кусок, который уже написали Чип и Рэнди, так что мы точно знали, что нам делать. Разница была только в масштабе.

Сейбел: Речь шла о переделке Habitat?

Крокфорд: Да, мы собирались переписать Habitat, только на этот раз с учетом его глобального распределения. Это оказалось крайне нелегко. Хотя нам и удалось его создать, это было мучение. Не хотел бы я вновь пережить нечто подобное.

Сейбел: Как вы думаете, тот совет, который вы дали чуть раньше, – быть достаточно дисциплинированными, чтобы реализовать заново только то, что действительно понимаешь, – мог уберечь от беды?

Крокфорд: Думаю, это помогло бы. Мы не продумали как следует процесс с учетом всех его этапов. У нас не было инкрементального подхода. Если бы он был, я бы направил усилия на две параллельные задачи. Во-первых, разработал безопасную распределенную платформу, которая бы не делала ничего, кроме обмена сообщениями и управления объектами. Во-вторых, переписал бы Habitat с учетом всех наших знаний о современных языках программирования. Просто переписал бы и все.

Второй этап заключался бы в соединении одного и другого. Можем ли мы построить одно поверх другого так, чтобы система продолжала работать? Хорошо, а теперь давайте займемся распределенной частью.

При подобном инкрементальном подходе, думаю, нас бы ждал успех. Но мы попытались сделать все одновременно, и это оказалось слишком сложной задачей.

Сейбел: Думаете, вы решили делать все одним этапом именно потому, что уже знали крупные куски будущей программы?

Крокфорд: Потому что мы были очень умны и опытни. Мы решили, что промашка просто невозможна. Программисты по своей природе оптимисты. И мы были оптимистами, иначе эту работу было бы не сделать. Вот почему мы пали жертвой эффекта второй системы, не сумев спланировать свои проекты, вот почему все оказалось для нас таким трудным.

Сейбел: Легче ли программировать со временем? Станет ли в будущем больше способных к тому, что мы сейчас называем программированием?

Крокфорд: Для меня программирование интересно тем, что я помогаю другим заниматься программированием. Я проектирую язык или инструмент так, чтобы он был доступен множеству людей. Именно из этих соображений была начата работа над языком Smalltalk. Smalltalk пошел другим путем, но первоначальная идея была для меня крайне привлекательна. Как создать язык программирования, предназначенный специально для детей, или как создать язык специально для тех, кто не считает себя программистами?

Сейбел: Видимо, вы полагаете, что научиться программировать должен каждый, хотя бы чуть-чуть?

Крокфорд: По-моему, иного выхода нет. Сегодня компьютеры захватили мир, и для того чтобы защитить себя или чтобы быть полноценным гражданином, вы должны хотя бы немного понимать, как все это работает.

Сейбел: Кое-кто считает, что программирование развивает особый тип мышления – в том смысле, что чтение и решение математических задач – это два разных способа мышления, и оба важны.

Крокфорд: Раньше я тоже так думал. Когда я начал заниматься программированием, у меня были потрясающие озарения: для меня все становилось упорядоченным, я видел структуры и вещи, которых прежде не замечал. И думал: «Поразительно! Каждый должен научиться этому», – потому что внезапно чувствовал себя значительно умнее. Но вскоре, беседуя с другими программистами, я понял, что они этого не понимают. Программист может совершенно неправильно воспринимать окружающий мир, как и любой другой человек. Для меня это оказалось грустным открытием.

Сейбел: Вы все еще получаете такое же удовольствие от программирования, как раньше?

Крокфорд: А как же!

Сейбел: Как по-вашему, становится ли программирование делом молодых?

Крокфорд: Раньше я так и думал. Несколько лет назад у меня произошла остановка дыхания во сне, но я об этом не знал. Я думал, что просто становлюсь старым и усталым, что дошел до той точки, когда сконцентрироваться невозможно. И я решил, что не смогу больше программировать, потому что просто не могу удерживать все нужное в голове. В процессе программирования часто приходится держать в голове множество вещей, пока не сможешь записать их, оформив в виде каких-то конструкций. А как раз этого я больше не мог.

Я утратил эту способность и считал, что причина тому – мой возраст. К счастью, мне полегчало, способность восстановилась, и я снова вер-

нулся к программированию. У меня все прекрасно получается, может быть, даже лучше, чем раньше, потому что я научился не зависеть так сильно от собственной памяти. Я теперь лучше документирую свой код, так как уже не столь уверен в том, что смогу вспомнить через неделю, почему я что-то сделал. На самом деле, иногда я просматриваю то, что написал, и поражаюсь, потому что совершенно не помню этих вещей. Иногда они оказываются ужасными, иногда блестящими. Даже не думал, что я когда-то был на такое способен.

Сейбел: Вы как-то назвали великолепной идеей литературное программирование в стиле Дональда Кнута¹. Вы применяете инструменты для литературного программирования?

Крокфорд: Нет. Я думал об этом и даже создал такие инструменты для нескольких языков, с которыми работаю, но сейчас литературным программированием я не занимаюсь.

Сейбел: Дело в проблеме взаимодействия инструментов между собой? Если ее решить, как думаете, вы бы писали литературные программы?

Крокфорд: Да. Думаю, что было бы значительно легче сопровождать JSLint, например с помощью литературного программирования. В литературном программировании мне нравится то, что проектируешь программу специально для чтения. Мне кажется, это придает ей невероятную ценность.

Сейбел: Каковы, по-вашему, основные возможности инструментов для литературного программирования?

Крокфорд: Основное свойство, которое открыл или создал Кнут, – возможность писать код не по порядку. Если меня интересует конкретная вещь, которая неоднократно встречается в коде, я могу собрать все эти фрагменты вместе, вместе их описать, а потом инструмент расставит их по местам.

Еще одна вещь, от которой освобождает литературное программирование, – размер функции. В идеале функция должна целиком помещаться на экране, чтобы ее можно было прочитать всю сразу. Если это не так, приходится создавать намного больше функций, а если такие функции ничего не добавляют системе, то становятся бесполезным шумом.

Благодаря Кнуту можно взять по отдельности все аспекты этой функции, которые могут быть тесно связаны между собой, но при этом объем слишком велик. Литературное программирование позволяет снабдить каждый элемент чего-то большого четкой описательной меткой и сказать: «Эта функция включает:» – и привести список этих меток. Почти

¹ Дональд Э. Кнут, автор книги «The Art of Computer Programming» (Искусство программирования) – Вильямс, 2008 г.).

то же самое можно сделать с помощью функций, но в этом случае придется вникать во взаимодействие ее элементов и так далее. При этом вводятся новые структуры, которые, собственно говоря, не решают никаких задач.

В идеале я хотел бы увидеть языки программирования, спроектированные специально для литературного программирования. Кнут удачно применил свою идею к Паскалю и Си, но мне хочется видеть новые языки, целиком и полностью разработанные именно для таких задач.

Сейбел: Вы читали программы Кнута, написанные в этом стиле?

Крокфорд: Конечно.

Сейбел: Как вы их читали? Как роман?

Крокфорд: Да, как роман. Я больше читал его книг, чем его программ, но мне действительно нравится, как он располагает код; пишет он действительно здорово, иногда даже вставляет небольшие шутки. Читать его программы действительно приятно.

Сейбел: Что именно вы выносите из них для себя? Вот вы прочли его книгу про TeX. Готовы ли вы добавлять новые возможности в TeX, или просто восхищаетесь тем, какой Кнут классный парень?

Крокфорд: Отличный вопрос. Я ознакомился с книгой про TeX, но читал ее без намерений что-то в нем изменять. Я читал просто с целью посмотреть, что же он такое сделал. Больше всего меня интересовало, как он реализовал разбиение строк, и эту часть я прочел с особым вниманием: я хотел прежде всего разобраться в алгоритме, а не понять, как работает код, чтобы его потом изменять или повторно использовать. Конечно, если бы я намеревался влезть в саму программу, я бы читал книгу по-другому.

Сейбел: Часто ли вы читаете код, написанный в литературном стиле или в каком-то еще, просто для удовольствия?

Крокфорд: Да. Но на самом деле не так-то просто найти код, который можно читать просто для удовольствия. Кнут написал кое-что. Фрейзер и Хэнсон¹ создали компилятор Си в стиле литературного программирования – он очень классный. Но примеров не так много. Очень жаль. Это может говорить о том, что литературное программирование, возможно, потерпело неудачу.

Сейбел: А как насчет главного труда Кнута, «Искусства программирования»? Вы из тех, кто прочел его от корки до корки, или используете

¹ Крис Фрейзер (Chris Fraser) и Дэвид Хэнсон (David Hanson), авторы книги «A Retargetable C Compiler: Design and Implementation» (Перенацеливаемый компилятор Си: проект и реализация). – *Прим. науч. ред.*

его как справочник, или просто поставили на полку и никогда в нее не заглядываете?

Крокфорд: Все что угодно, кроме последнего. Когда я учился в колледже, то пару месяцев не платил за квартиру, чтобы купить его книги. Я читал их и находил там шутки, например в содержании первого тома, но понимал далеко не все. В некоторых местах Кнут гораздо выше моего разума, но книги мне очень понравились, кроме того, я часто использовал их в качестве справочников.

Сейбел: Вы и вправду читали их от корки до корки, пропуская особо трудные математические пассажи?

Крокфорд: Да, ту часть, где много звездочек, я разве что пробежал глазами. Знакомство с книгами Кнута я сделал одним из критериев при найме на работу и был разочарован, осознав, что его читали лишь немногие. Я считаю, что каждый, кто называет себя профессиональным программистом, обязан прочесть Кнута или хотя бы иметь его книги.

Сейбел: Мне кажется, что для чтения книг Кнута нужно уметь читать и понимать математические выражения. Как вы думаете, в какой степени, математическая подготовка необходима программисту?

Крокфорд: Видимо, в небольшой, поскольку большинство программистов такой подготовкой не обладают. В тех приложениях, над которыми я работаю, инструменты, предлагаемые Кнутом, почти не применяются. Если бы мы создавали операционную систему или среду выполнения, то это было бы значительно важнее. Но мы занимаемся валидацией данных форм и пользовательскими интерфейсами. Обычно производительность в них не так важна; большую часть времени приложение ждет данные от пользователя или ответ из сети.

Мне бы хотелось, чтобы все это было совершенно необходимо программистам, но это не так. Может быть, именно поэтому веб-программирование получило столь бурное развитие и стало доступно многим, именно поэтому JavaScript так популярен. Все это не так сложно. А большинство сложностей – лишние. Если мы чуть-чуть почистим платформу, эта работа станет значительно легче.

Сейбел: Итак, Кнут дал вам эти унылые формулы, а потом – раз! – и появилась цельная картина. Даже если почистить платформу, создание и проектирование больших и понятных систем останется нелегким делом. Как вы проектируете свой код?

Крокфорд: Дело не столько в написании программы, сколько в выполнении итераций для ее выживания. Обычно мы создаем программное обеспечение с учетом последующих изменений, а вносить какие-то изменения всегда сложно, поскольку есть риск поломать имеющийся код.

Нельзя предусмотреть все варианты использования системы, но можно попытаться сделать ее достаточно гибкой, чтобы приспособить к чему угодно в будущем, – так я считаю. Как не загнать себя в угол? Как достичь нужной мне гибкости?

Вот за что я люблю JavaScript, так это за возможность рефакторинга кода. Я понял, что делать это очень и очень просто. В то время как рефакторинг посредством глубокой иерархии классов может стать настоящим мучением.

Например, JSLint немало изменился с тех пор, как я начал разрабатывать его в 2000–2001 гг. Да и задачи его существенно изменились: теперь на нем можно делать то, чего я и не представлял, – и во многом благодаря гибкости JavaScript. Я могу вертеть его как хочу, программа может разрастаться, но при этом не становится громоздкой.

Сейбел: Так что же делает его использование настолько простым?

Крокфорд: Я стал горячим поклонником «мягких» объектов. В JavaScript любой объект является тем, чем вы укажете ему быть. Это настораживает людей, которые смотрят на это с традиционной точки зрения, поскольку что вы получите без класса? Оказывается, вы получите именно то, что хотите, и это действительно полезно. Приспосабливайте свои объекты – и они получаются куда более наглядными.

Сейбел: Возможно, проблема языков, основанных на классах, заключается в слишком строгой типизации. Вы получаете большую иерархию классов, и если понадобится внести в эту иерархию изменения, то придется разъединить ее, а потом соединить обратно. В JavaScript другая опасность – язык может быть слишком динамическим. Вы допускаете маленькие ляпы по всей программе, и ее текущая структура начинает зависеть от множества вещей, происходящих во время выполнения. Нет ничего статического, на что можно взглянуть и сказать: «Ага, эта программа структурирована так-то».

Крокфорд: Да, есть от чего встревожиться, но это и хорошо; тревога – это реальность. Это дисциплинирует. В большинстве традиционных языков дисциплину вам навязывает язык. Работая с JavaScript, вы должны дисциплинировать себя сами.

Мой код не разваливается отчасти именно благодаря моему строгому контролю того, как разные вещи взаимодействуют друг с другом, поскольку я знаю, что сам язык не обеспечивает такую строгость. Сегодня я не возьмусь без JSLint за что-то сложное, вроде того же JSLint. JavaScript сам по себе не очень масштабируем, но этот инструмент значительно повышает мою уверенность в том, что система не выйдет из-под контроля.

Сейбел: Итак, «мягкость» объектов в JavaScript может быть опасной. Но если вы никогда не пользовались возможностью расширения объектов, то с тем же успехом могли бы создавать классы в Java. Как вы полагаете, можно ли структурировать программы на JavaScript, так чтобы извлечь максимальную пользу из гибкости языка?

Крокфорд: Мне потребовались годы проб и ошибок. До работы с JavaScript я ничего не читал об этом языке, просто начал работать и все. Я нашел пример программы, довольно жуткий, и всячески вертел его, пока все не заработало примерно так, как мне было нужно. В общем, я начал программировать на языке, не имея представления о том, что это за язык, как он работает и как на нем думать.

Я понимаю, почему люди в нем разочаровываются. Если вы попытаетесь писать программу на JavaScript так же, как на Java, язык станет кусаться. Я прошел через это. Первое, что я сделал, – это попробовал найти способ имитировать что-то вроде классов Java, но не смог. Я так ничего и не добился.

В конце концов я выяснил, что классы мне вообще не нужны, – язык сделает все за меня. Вместо того чтобы бороться с языком, я почувствовал, как сам язык придает мне силы.

Сейбел: При проектировании программ вы предпочитаете идти снизу вверх, сверху вниз или от середины?

Крокфорд: Все сразу. Это способ держать систему целиком в голове. В конце концов, вы должны разделять и властвовать, приведя ее к такому виду, чтобы с ней можно было справиться. Я берусь сразу за все стороны проблемы и пользуюсь одновременно всеми подходами. Я борюсь с системой до тех пор, пока не прояснится ее структура. А остальное придет само.

Сейбел: Как у вас связано проектирование и кодирование? Вы сразу же начинаете писать код и потом постепенно его улучшаете или делаете что-то помимо собственно кодирования?

Крокфорд: Обычно это два независимых процесса. Сейчас, правда, они становятся более похожими друг на друга. Я пользуюсь языком проектирования, или метаязыком, чем-то полуанглийским, слабо структурированным, более описательным, чем язык, предназначенный для кодирования. Но если программа должна быть на JavaScript, то я пишу сразу на JavaScript.

Сейбел: Какими инструментами вы пользуетесь для написания кода?

Крокфорд: Я работаю в небольшом бесплатном текстовом редакторе. Никаких хитрых функций у него нет, да мне ничего и не надо. Здесь не нужно множество формальных инструментов, как для других языков.

Броузеру нужен только файл с исходным кодом, который вы ему посылаете, компилятор встроен в браузер, так что делать-то почти ничего и не надо. Не нужно ничего особенного; весь код запускается в браузере.

Сейбел: И, разумеется, вы используете JSLint.

Крокфорд: Да, причем постоянно. Я стараюсь использовать его каждый раз перед запуском программы, то есть если я что-то изменил, то перед запуском прогоняю код через JSLint.

Сейбел: Значит, вы редактируете код в текстовом редакторе, потом прогоняете его через JSLint и наконец запускаете в браузере. А как насчет отладки?

Крокфорд: Зависит от браузера. Если это Firefox, использую Firebug, если IE – отладчик Visual Studio. Оба эти инструмента очень хороши. У нас в браузерах есть на удивление хорошие отладчики.

Я использовал фреймворки, в которые были встроены специальные модули на основе DOM-элементов, позволяющие анализировать содержимое объектов. Но потом понял, что это не нужно и отладчика вполне достаточно.

Сейбел: Вы когда-либо выполняете свой код пошагово просто так, не пытаясь выловить конкретную ошибку?

Крокфорд: Только если имею дело с чем-то очень сложным. Я выполняю код пошагово в процессе тестирования, но в основном делаю это, когда есть конкретная проблема.

Сейбел: Что вы можете сказать о других методах отладки, таких как операторы утверждений или формальные доказательства корректности? Вы применяли их? Что вы думаете о понятии инварианта?

Крокфорд: Мне нравятся эти методики. Я был разочарован тем, что Eiffel не стал победителем среди объектно-ориентированных языков программирования, уступив первое место C++. По-моему, Eiffel был куда интереснее, мне нравилась его система контрактов на основе предусловий/постусловий. Я хотел бы видеть ее встроенной во все языки, с которыми я работаю. К сожалению, это еще одна из тех идей, которая так и не прижилась.

Сейбел: Какова худшая ошибка из тех, что вам пришлось отлавливать?

Крокфорд: Ошибка в реальном времени, например в видеоигре. Постоянные прерывания, натканные повсюду, полное отсутствие управления памятью – и программа просто внезапно падает, непонятно почему. Справиться с подобными ошибками очень трудно, и на отладчик рассчитывать не приходится.

На Basic Four мы разработали полностраничный терминал для обработки текста, основанный на процессоре Z80 с 64 Кбайт памяти, чего было недостаточно для экрана такого размера. У нас также было подключение к локальной сети, по которой мы посылали страницы на свой сервер.

И у нас возникла проблема – экран то и дело гас. Архитектура была такая: в памяти хранилась строка текста со специальным символом конца строки, за которым следовал адрес следующей строки. Эти ссылки обрабатывались небольшим DMA-процессором. В какой-то момент ссылка пропадала – видимо, возникало что-то вроде гонок.

С нашей точки зрения, если посмотреть логически, все ссылки были правильными. Но мы не учитывали взаимодействия в реальном времени с тем процессором, который мог обращаться к памяти не одновременно с нами. И я таки разобрался в этом. Помню, в тот день я работал дома и по телефону постоянно созванивался со своей командой. Внезапно меня словно озарило. Я понял, в чем дело, объяснил им, как исправить ошибку, и больше она не появлялась.

По-моему, худшие ошибки – это ошибки, происходящие в реальном времени при взаимодействии нескольких потоков. Мой подход к исправлению таких ошибок: постараться их избегать. Поэтому я не люблю многопоточность – я считаю, что это жуткая модель программирования. Можно допустить ее как необходимое зло, но в большинстве случаев без многопоточности можно обойтись.

Одна из вещей, которая мне нравится в модели браузеров, как раз связана с тем, что поток всегда один. Некоторые жалуются на то, что если этот поток будет заблокирован, то заблокируется и весь браузер. Так не допускайте этого! Нас постоянно просят добавить потоки в JavaScript. Пока что нам удается отбиваться, и я очень этому рад.

Событийно-ориентированная модель – та, что применяется в браузере, – работает отлично. Она плохо работает только в том случае, если у вас появляется процесс, который требует много времени. Мне нравится, как Google решил эту проблему в Gears: они используют отдельный процесс, полностью изолированный от остальных, которому можно послать программу для выполнения. По окончании выполнения этот процесс сообщает о результате, и этот результат будет возвращен в качестве события. Просто блестящая модель.

Сейбел: Вас когда-нибудь интересовали формальные доказательства корректности ПО?

Крокфорд: В 1970-е я следил за ними с интересом, ожидая, чем все это закончится. Но особого результата так и не увидел. Программное обеспечение – сложная штука, и что-то может пойти не так по разным причинам.

Программное обеспечение – это прежде всего спецификация того, как программа должна работать. И ничто, кроме полной спецификации, не объяснит вам, каким должно быть поведение в конечном итоге. Именно это делает разработку программного обеспечения настолько сложным.

Сейбел: Как вы тестируете код? Вы заражены тестированием, как сейчас говорят?

Крокфорд: Я скорее стараюсь действовать по обстоятельствам. Это еще один аспект, который я хотел бы изменить в своей работе, но полностью этого еще не сделал.

Сейбел: Речь о JsUnit?

Крокфорд: Да. Тестирование кода пользовательского интерфейса – непростое дело, поскольку он зависит от множества других вещей, так что разбивать его на модульные тесты не слишком эффективно. Кроме того, я заметил, что тот стиль разработки, который я применяю в работе с JavaScript, не разбивает системы на модули, как это делается при создании классов и последующего их тестирования изолированно друг от друга.

В JavaScript тестировать отдельную функцию не слишком разумно, поскольку для ее работы требуется некоторое состояние. Так что я пока не нашел разумный подход к модульному тестированию в JavaScript.

Сейбел: Если в компании есть отдельная группа тестировщиков, то как она должна взаимодействовать с командой разработчиков?

Крокфорд: Я работал в компаниях, где существовало противостояние между командами разработчиков и тестировщиков; по-моему, это нездоровое явление. Там исходили из принципа, что те и другие должны «стучать» друг на друга. Мне кажется, это кошмарная модель.

Гораздо эффективнее, когда две эти команды сидят вместе, и тестировщики помогают разработчикам улучшить программу, а не грызутся с ними. Это влияет на способ отчетности, и работа идет куда результативнее. Кроме того, разработчики участвуют в тестировании, так что нельзя сказать, что вы исключительно разработчик или тестировщик.

Но эффективнее всего, как я считаю, устроить тестирование у клиента. В начале карьеры программиста мне приходилось заниматься этим – бесценный опыт! Вы живете с клиентом целую неделю, помогаете ему установить новую систему и решать с ее помощью задачи.

Это дало мне возможность взглянуть изнутри на то, как используются наши программы, и понять, что можно сделать для облегчения их использования. Разработчики, не имевшие подобного опыта, всегда казались мне непростительно высокомерными. Поразительное неуважение

к людям, которые пользуются нашим продуктом! А все потому, что они этих людей попросту не видели.

Сейбел: Кем вы себя считаете – ученым, инженером, художником, ремесленником или кем-нибудь еще?

Крокфорд: Скорее писателем. Иногда я пишу на английском, иногда на JavaScript.

В конце концов все сводится к коммуникациям и к структуре, призванной их облегчить. Естественные и компьютерные языки функционируют во многом по-разному, но в конечном счете я сужу о компьютерной программе по ее способности взаимодействовать с человеком, читающим эту программу. В этом смысле различий нет.

Сейбел: И если программа хорошо взаимодействует с человеком, то в части ее взаимодействия с компьютером проблемы почти отпадают?

Крокфорд: Можно надеяться. Компьютер капризен и не слишком умен, поэтому нужно приложить дополнительные усилия, чтобы он понял все правильно. Поскольку это так сложно, легко упустить из виду другую часть, не менее важную, с моей точки зрения.

Сейбел: У Дейкстры есть известная статья «On the cruelty of really teaching computing science» (О сложности практического обучения компьютерной науке), где утверждается, что программирование – ветвь прикладной математики. Вы согласны с этим?

Крокфорд: Математика важна для программиста, но это лишь одна из множества других важных вещей. Мне кажется, что преувеличение значения математики может привести к недооценке значения чего-то другого, возможно, более важного, например грамотности.

Как я уже говорил, мне хотелось, чтобы одним из требований при приеме на работу было знакомство с книгами Кнута, и я от него отказался, поскольку не мог найти достаточного количества людей, отвечающих этому требованию. Другое качество, которого я требовал от кандидатов, – нормальная грамотность в том языке, на котором они общаются с другими людьми. Я хотел, чтобы люди умели писать, ведь мы тратим очень много времени на переписку друг с другом: мы пишем электронные письма или документацию, планы, спецификации. Я хотел, чтобы все члены моей команды могли делать это, но оказалось, что это очень редкий навык. Поэтому сейчас я предпочитаю кандидатов с дипломом по английскому языку, а не по математике.

Сейбел: Кажется, у Дейкстры есть примерно такое утверждение: «Если не можешь писать на своем родном языке, лучше не берись за программирование».

Крокфорд: Совершенно согласен.

Сейбел: Сейчас мы все чаще сталкиваемся вот с чем: программирование, освобождаясь от физических ограничений, становится все больше зависимым от исторических случайностей. Многие ваши предложения по выделению некоторого подмножества языка JavaScript и ваша версия HTML5 кажутся попытками исправить эти исторические случайности.

Крокфорд: Да, и в некотором роде это донкихотство. Я знаю, что многое из того, чего я хотел бы добиться, неосуществимо. Но время от времени что-то получается. Так, когда XML предложили в качестве формата обмена данными, моя первая реакция была: «Господи, это же безумно сложно! Все это не нужно для простой передачи данных туда и обратно». Я предложил другой путь, по которому все и пошли. Теперь JSON – самое популярное средство передачи данных в Ажах-приложениях, он активно используется и во многих других приложениях. И он действительно очень прост. Такие случаи возвращают мне веру в человечество, в то, что в конце концов все будет сделано правильно.

Но нельзя, чтобы все разбредались и делали каждый что-то свое. Так не получится, от этого никому не станет лучше. Одни должны создавать продукт, остальные – высказываться «за» или «против». JSON – тоже историческая случайность.

Сейбел: Как вы думаете, индустрия ПО – это система блестящих инноваций или омерзительная свалка?

Крокфорд: Пытаюсь подобрать к выражению «омерзительная свалка» синоним поизящнее. Мне кажется, в целом программное обеспечение становится лучше, хотя и не такими темпами, какими улучшается аппаратная часть, согласно закону Мура. По сравнению с «железом» у нас все происходит очень и очень медленно – чтобы вдвое увеличить эффективность разработки программного обеспечения, нам требуется двадцать лет. Но прогресс все же есть. В основном он связан с тем, что мы теперь не занимаемся подгонкой ПО к конкретному железу, мы больше не заботимся так о производительности. То есть теперь мы больше должны заботиться о качестве. Но увы, как раз этому мы не посвящаем достаточно времени.

Сейбел: Как ни выражайся, суть останется прежней: омерзительная свалка. Как с ней покончить?

Крокфорд: Вот как раз это я и пытаюсь выяснить. Думаю, во многом нынешняя ситуация связана с нашим подходом к созданию стандартов. Если сегодня все работает хорошо, то это потому, что Сеть работает хорошо. Все преимущества, которые принес Интернет, идут от возможности связать все воедино, причем сделать это достаточно надежным способом.

Но стоит немного копнуть, как обнаружатся места, где все работает не так, как надо, и где можно было бы сделать лучше. Вопрос в том, как сделать это прямо на месте. Внесение любого изменения в стандарт – это акт насилия. Это ужасно. Это приведет к неработоспособности множества вещей и нанесет вред людям. Поэтому к изменению стандартов надо подходить очень осторожно, учитывая стоимость внесения изменений. Надо убедиться, что выгоды превосходят нанесенный ущерб. Сейчас, насколько я могу судить, об этом не задумываются. Стандарты меняют, обосновывая это тем, что «мы так хотим», или «так будет понятнее», или еще что-то, совсем необязательно связанное с желанием принести пользу людям. Я борюсь с этим – ведь таким образом мы ничего не улучшим.

Сейбел: Вы склоняетесь к тому, чтобы не создавать лишних спецификаций. Это, конечно, позволяет избежать чрезмерной спецификации и стандартизации того, о чем впоследствии пожалеете. Но чем меньше спецификаций в стандарте, тем больше люди создают самостоятельно, в результате чего появляются неформальные стандарты. Решит ли упрощение стандартов эту проблему или сложность появится с какой-то другой стороны?

Крокфорд: Что нам действительно нужно, так это более точные прогнозы насчет того, что нам может понадобиться в будущем. Может, стоит подождать, пока мы научимся путешествовать во времени, и тогда наконец-то сможем делать все правильно. А пока я смотрю, как люди экспериментируют, придумывают разные подходы к стандартизации. Может быть, взяв лучшие из них, наиболее удобные и приспособленные к дальнейшему совершенствованию, мы получим правильный подход к стандартизации. То есть комитет по стандартизации, вместо того чтобы пытаться предугадать лучший путь, возьмет имеющиеся примеры, которые продемонстрировали лучшие результаты.

Сейбел: Но в целом есть прогресс, как вы считаете?

Крокфорд: Прогресс – это не всегда движение вперед. Иногда мы движемся вперед, иногда отходим назад. Перейдя на персональные компьютеры, мы потеряли кучу всего другого. Тогда рынком правил системы на основе деления времени. Было сообщество, участники которого могли обмениваться электронными сообщениями, файлами, общаться в чате и играть. Все это с приходом ПК было утрачено и появилось снова лишь через двадцать лет.

Мы также сделали большой шаг назад в плане безопасности. Системы с разделением времени уже начали понимать, как защищать систему и пользователей друг от друга. С переходом на ПК все изменилось: у вас был собственный компьютер, и все, что работало на нем, имело одинаковые права доступа и могло делать что угодно, но потом оказалось, что

не все программы, работающие на вашем компьютере, работают в ваших интересах. Мы до сих пор боремся с этим. Да, мы видим множество улучшений в операционных системах персональных компьютеров, но все еще не достигли уровня передовых систем с разделением времени. А сколько лет уже прошло!

Сейбел: О каких конкретно системах вы говорите?

Крокфорд: В MULTICS были интересные возможности по взаимодействию процессов, там было несколько адресных пространств, способных взаимодействовать, но не имевших доступа к содержимому друг друга. Это отправная точка, необходимая для кооперативных вычислений. Сейчас мы думаем, как реализовать это в браузерах. А ведь прошло уже огромное количество времени. Сейчас мы возвращаемся к решениям, внедренным уже тогда.

Сейбел: Я заметил, что нечто похожее происходит и с языками программирования: программы для ПК писались на языке ассемблера, поскольку даже Си был для него слишком высокоуровневым. И только сегодня мы переходим к языкам, по мощности приближающимся к Smalltalk и Лиспу, существовавшим в момент появления ПК. Интересно, программисты задумаются над историческими уроками или так и будем продолжать изобретать велосипед?

Крокфорд: По-моему, мы, к сожалению, уделяем истории слишком мало внимания. И я разочарован, видя, как нынешние программисты совершенно не хотят знать, откуда что взялось. Они просто считают, что все это придумал какой-то комитет по стандартизации, снабдив их набором инструментов и языков, которыми надо лишь правильно пользоваться.

Есть удивительные истории о том, как все это возникло, вследствие чего, кто это сделал, что считается ошибкой сейчас и что обязательно будет считаться ошибкой со временем. Порой я считаю себя археологом программного обеспечения: понемногу у меня скопилась коллекция недооцененных технологий, вещей, которые я считаю просто великолепными и которые намного превосходят то, что мы делаем сегодня. Я по-прежнему надеюсь, что мы заново откроем для себя все это, по достоинству оценим и извлечем из этого пользу, – но если такое и произойдет, то не скоро. Люди заиклены на том, что есть здесь и сейчас, и сдвинуть их с места нелегко.

Сейбел: Назовите некоторые из этих технологий.

Крокфорд: Ну, скажем, уже упоминавшиеся Лисп и Smalltalk. Отличные вещи; некоторые из тех идей переходят в современные языки, и мы, работая с JavaScript, стараемся воскресить те старые идеи. Правда, в этом языке уже есть многое из того. Лексические границы и функции

высшего порядка – это блестяще! А теперь мы пытаемся понять, как внедрить в него больше полезных свойств Smalltalk и Scheme, не нарушая структуру языка. Можно возразить, что лучше бросить все, над чем мы работаем, и просто вернуться к Smalltalk и Scheme; возможно, так действительно было бы лучше, но такой вариант не рассматривается.

Чем больше мы работаем с мэшапами, тем чаще нам требуется импортировать откуда угодно код – который мы никогда не сможем проверить, – и запускать его у себя. Это новый вид программирования. Раньше такого не было. Я считаю, что это – будущее программирования. Мы впервые делаем это в JavaScript, у которого много недостатков, но именно к таким вещам он приспособлен.

Давайте посмотрим на основные этапы истории программирования. Мы начали с машинных кодов, потом совершили скачок к символическому языку ассемблера, потом к высокоуровневым языкам, затем к структурному программированию и наконец к объектно-ориентированному. И каждый такой скачок происходил один раз за поколение.

А вот очередной скачок запаздывает. Мы уже давно застыли на объектно-ориентированном программировании. Можно возразить, что это был Smalltalk-80. Можно пойти и еще дальше вглубь времен, но все равно мы сидим на этих идеях слишком долго.

Думаю, очередной скачок – как бы он ни назывался – будет связан с мэшапами, когда мы сможем брать куски разных программ, соединять их и тут же получать новую программу. Уже десятилетия ведутся разговоры о программной модели, в которой программы собирались бы наподобие конструктора LEGO. Этого пока не произошло, но уже начинает происходить в JavaScript – в наименее ожидаемом месте.

Сейбел: Как вы на собеседовании распознаете хорошего программиста?

Крокфорд: Сейчас я определяю это по чтению кода. Я предлагаю соискателю принести фрагмент его лучшего кода и пройти по нему.

Сейбел: Что конкретно вы в нем ищете?

Крокфорд: Я ищу там качественное представление кода. Хочу понять, чем этот человек гордится. Убедиться в том, что он действительно автор того, что представил. Я понял, что это куда более эффективный способ, чем предлагать головоломки или задавать стандартные вопросы. Помоему, все это бесполезно. А вот коммуникативные способности – это то, что я ищу.

Сейбел: Что вы можете посоветовать программистам-самоучкам?

Крокфорд: Много читать. Есть масса хороших книг – берите и читайте. А если вы веб-разработчик, зайдите на лучшие сайты и посмотрите.

те их код. Правда, я даю этот совет не без опаски. Большинство веб-разработчиков учились своему делу «глядя в исходники», и до недавнего времени эти исходники никуда не годились. Целое поколение программистов выросло на плохом коде, который они считают образцовым. Сейчас положение исправляется, но плохого кода все еще так много, что я боюсь давать подобный совет.

Сейбел: А что вы порекомендуете тому, кто получает диплом по компьютерным наукам и хочет работать программистом?

Крокфорд: На их месте я бы сконцентрировался на коммуникации. Учитесь писать, учитесь читать.

Всем остальным советую то же самое: читайте и пишите. Принимая человека на работу, я не требую от него специальных навыков. До недавнего времени невозможно было найти хорошего JavaScript-программиста – их было крайне мало. Сейчас они появились, но это произошло буквально только что. До того я брал на работу за хорошие способности. Мне неважно, кто вы – хороший Java-программист, хороший Си-программист или хороший программист на чем-нибудь еще. Для меня важно просто знать, что вы можете реализовать алгоритм, понимаете структуры данных и умеете их документировать. Тогда вам будет понятен и JavaScript.

Сейбел: А у вас не было проблем с этим? Тот, кто хорошо освоил один язык, обычно с трудом отказывается от привычных приемов, применяя их даже тогда, когда новый язык для этого плохо подходит.

Крокфорд: У меня было такое, скажем, с Windows-программистами. В Windows есть ряд очень сложных API, их можно осваивать годами. И получается, что вы ничего не можете, кроме как написать обработчик оконных сообщений. Без особой необходимости я никогда не ищу таких узких специалистов. В общем случае я предпочитаю тех, кто разбирается во всем понемногу, кто способен освоить некоторый API, а не тех, кто на нем специализируется.

Сейбел: Вы говорили, что занялись компьютерами, веря, что с их помощью можно улучшить мир.

Крокфорд: У меня именно такое намерение.

Сейбел: И что из этого выходит?

Крокфорд: Большей частью мы справляемся неплохо. Мир улучшается, хотя при этом он не обязательно движется вперед. Возьмите, к примеру, международные отношения за последние десять лет – появление Интернета не компенсировало разлагающий эффект от объединения крупных СМИ. Это большое разочарование.

В результате погибли сотни тысяч людей. Это печально. Я хочу, чтобы Интернет лучше выполнял свои функции и чтобы такого не повторялось. Пока непонятно, какие преобразования потребуются для достижения этой цели. Может, все образуется само собой, но в этом я пессимист. Думаю, нам нужно найти что-то для следующего скачка, чтобы исправить то, что сейчас плохо работает.

Сейбел: Но ведь миллионы блогеров могут сказать: «Мы тут в блогах отслеживаем все, а традиционные СМИ отдыхают».

Крокфорд: Да, это потрясающе. Но мы все еще делаем это неправильно. У нас есть отличная возможность быть друг к другу ближе, обмениваться сообщениями, но это пока не работает. Пока это все не то.

Сейбел: Думаете, решение этой проблемы отчасти зависит от техники? Могут ли программисты и проектировщики систем создать такую архитектуру, которая будет для этого полезна? Или проблема чисто социальная?

Крокфорд: Может быть, новые социальные системы будут строиться на основе новой сетевой инфраструктуры, и, может быть, она все еще не созрела до нужного уровня, и проблема в этом. Может быть, все решится само собой – надеюсь, так и будет. Но мне кажется, что без подобной помощи не обойтись. Сейчас в Сети очень слабы системы идентификации и безопасности, и, на мой взгляд, это необходимые компоненты для построения надежной социальной системы. В этом отношении Сеть все еще несовершенна, возможно, отсюда и столько лишних проблем.

4

Брендан Айк

Создатель языка JavaScript, вероятно, самого популярного у веб-разработчиков и самого проклинаемого, Брендан Айк в настоящее время является главным техническим директором компании Mozilla Corporation – подразделения Mozilla Foundation, отвечающего за совершенствование браузера Firefox.

Поклонник изящных теоретических решений и одновременно практических подходов в программировании, Айк в начале своей карьеры занимался тем, что программировал сетевой код и код ядра в Silicon Graphics и MicroUnity. После MicroUnity работал в Netscape над браузером и в условиях жестокого цейтнота разработал язык JavaScript.

В 1998 году вместе с Джейми Завински он возглавил движение за открытие исходного кода Netscape, что привело к созданию mozilla.org, где Айк стал главным проектировщиком.

В последние годы Айк является одним из руководителей работ по развитию платформы Mozilla и одновременно входит в число разработчиков JIT-виртуальной машины для JavaScript, названной TraceMonkey. В интервью Айк объясняет, что он также пытается «сдвинуть ось исследований» в проекте Mozilla, приведя в проект практически мыслящих ученых, чтобы сблизить академическую теорию и промышленную практику.

Мы затрагивали и другие темы, например: почему JavaScript должен был походить на Java, но не слишком сильно, почему JavaScript должен развиваться, несмотря на неудачу проекта ECMAScript 4, и почему необходимы новые разновидности статического анализа кода.

Сейбел: Где вы учились программировать?

Аик: В конце 1970-х – начале 1980-х я заканчивал Университет Санта-Клары, специализируясь по физике. Мы часто бывали в Стэнфорде и работали на LOTS-A и LOTS-B – больших системах с разделением времени типа DEC TOPS-20, а в Санта-Кларе как раз стояла система TOPS-20: отличный 36-битный процессор от DEC, великолепная операционная система, отличный макроассемблер. Си – это что-то вроде «портируемого ассемблера», но работа с макросами в ассемблере – это просто кошмар. А там были настоящие макросы для ассемблера, и имея навыки структурного программирования, можно было сделать многое. Системы типов не было, но и Си в этом отношении особенно ничем похвалиться не может. Зато был богатый набор системных вызовов, системных служб, ввод/вывод с отображением в память – все то, что первоначально не входило в UNIX.

Моей специальностью была физика, но я все больше занимался программированием, мне нравились занятия по математике и компьютерным наукам, нравилось изучать теорию автоматов и формальные языки. В те времена была настоящая гонка за разработку лучшего восходящего парсера – тогда это делал уасс, затем были созданы и другие системы. Очевидно, что формальная чистота ведет к чистому коду, что всегда было заметно во внешнем интерфейсе компилятора. Внутренний же код был тогда мешаниной из тайных знаний и эвристики, а мне действительно нравились теория формальных языков, теория регулярных языков и все такое.

Сейбел: На каких языках и в каких средах вы тогда программировали? Для целей физики использовался Фортран?

Аик: Это очень занятно. Я был увлечен чистой физикой и не ходил на инженерные занятия, чтобы не иметь дела с пачками перфокарт, которые можно рассыпать – и тогда придется использовать раскладочную машину. Фортран к тому времени меня уже не устраивал. Тогда был очень важен Паскаль, и мы уже начинали осваивать Си. И ассемблер. Я писал низкоуровневый код, делал хеш-таблицы и тому подобное. И это было полезно: так лучше понятны компромиссы, к которым вы-

нуждены прибегать разработчики. Сразу видно, кто работал на уровне битов, а кто всю жизнь работал в защищенной среде.

Меня также интересовали язык Си и UNIX, но с нашим старым железом, то есть DEC, мы только начинали их осваивать. У нас было что-то на основе Portable C Compiler и yacc, мы едва-едва начали генерировать код и занялись портированием программ в UNIX. Физики не получали летней работы, так что я много программировал, был ассистентом в лаборатории, а в последний год сменил специализацию на «математика/компьютерные науки», и именно это значилось в моем дипломе бакалавра.

Сейбел: Помните ли вы свою первую интересную программу?

Айк: Непростой вопрос... Там был жуткий графический терминал от DEC – видимо, усовершенствованный VT100, поскольку он понимал escape-последовательности. Совершенно убогая глубина цвета и разрешение на уровне начала 1980-х. Я начал делать для него копии игр: Pac-Man, Donkey Kong. Эти игры я писал на Паскале, и они использовали escape-последовательности. Это было что-то вроде хобби, которым я занимался все больше и больше. Думаю, это был первый для меня случай нестандартного программирования, когда пришлось задуматься о модульности и защите от самого себя.

Это было тогда, когда я еще специализировался по физике, – наверное, на третьем курсе. На четвертом курсе я сменил специализацию, стал изучать формальные языки и писать генераторы парсеров. Вот так я занимался одновременно вещами серьезными и несерьезными – играми и умными программами. Потом я обратился к компиляторам, стал писать копии макрообработчиков вроде m4 и CPP, все в таком духе. Помню, как мы достали исходник какой-то версии UNIX, как читали по-настоящему заумный код на Си! Препроцессор Си Джона Райзера – возможно, настоящий – был довольно забавной смесью всякой всячины. Он был очень эффективным – использовал глобальный буфер, там было много указателей, и еще он пытался избегать копирования. Я подумал: «Чтобы делать это, должен быть способ лучше».

Так я бросил физику и обратился к компьютерным наукам и программированию. До этого я по-настоящему не занимался программированием – только математикой и компьютерной теорией. Родители не хотели покупать мне Apple II. Я заикнулся об этом однажды. Выпрашивать не стал, но сказал, что с помощью компьютера смогу выучить иностранный язык – это была своего рода маскировка. «Нет, ты будешь тратить время на свои игры», – ответили родители и были правы. Вот так они уберегли меня от игр.

Сейбел: Программисту легче найти летнюю подработку, чем физику. А что еще для вас было в этом привлекательного?

Айк: Связь теории и практики, особенно на пользовательской стороне процесса написания компиляторов. В численные методы я погружался не слишком глубоко. Не так уж интересно ломать голову над тем, как представить действительные числа в виде чисел с плавающей запятой с конечной точностью. Адский кошмар. Пользователи JavaScript до сих пор обжигаются на этом – мы выбрали этот стандарт 1980-х, но он не всегда работает так, как ожидалось.

Сейбел: Тут как с испанской инквизицией – никто не ждет плавающую запятую.¹

Айк: Никто не ожидает встретить получаемые ошибки округления – в пятой степени невообразимые. Округление плохо выполняется в двоичной системе. Поэтому в JavaScript, там, где речь идет о долларах и центах, суммах и разностях, встречаются странные нули с девяткой после них. В одном блоге критиковали Safari и Мак за неправильные математические вычисления. Это стандарт двойной точности IEEE – он встречается везде, и в Java, и в Си.

Кроме того, физика мне казалась чем-то застывшим. Что-то есть в этом нездоровое – люди протирают штаны, открывая что-нибудь вроде темной энергии, то, что принципиально нефальсифицируемо. Меня притягивали более практические вещи, которые тем не менее прочно опираются на математику и логику.

Затем я перешел в Иллинойский университет в Урбана-Шампейне, чтобы наконец получить степень магистра. Я думал, что все уже позади, когда оказался в проекте, для которого IBM набирала специалистов. У них была странная машина 68020, купленная у какой-то компании из Дэнбери (Коннектикут). На этот компьютер они портировали Xenix. Работал он так плохо, что они наняли нашу исследовательскую группу, сделав из нее группу контроля качества. Каждый понедельник появлялся человек в синем костюме и произносил напутственные слова. Преподавателям было, в общем, все равно. Может, я и научился бы чему-нибудь новому, но слышал, о чем говорил в кампусе Джим Кларк, и решил, что хочу работать в Silicon Graphics.

Сейбел: Над чем вы работали в SGI?

Айк: В основном над ядром и сетевым кодом. Я постепенно совершенствовался в языках, так как мы решили создать собственный уровень для управления сетью и анализа пакетов. Я написал язык выражений для сравнения полей и пакетов, и еще транслятор, который уменьшал

¹ «Никто не ждет испанскую инквизицию» (No one expects the Spanish Inquisition) – ставшая крылатой фраза из британского юмористического телешоу «Монти Пайтон». – *Прим. ред.*

и оптимизировал все это до небольшого числа фильтрующих масок для первых 36 байт пакета.

В конце концов я создал другую реализацию языка – компилятор, который генерировал Си-код на основе описания протокола. Кто-то захотел, чтобы наш анализатор пакетов поддерживал AppleTalk. Это было большое и сложное собрание протокольного синтаксиса для последовательностей и полей разных размеров и зависимых типов... в основном массивов, всякие штуки в этом роде. Это было занято – трудная задача, которую нужно решить. В итоге мне пригодились некоторые знания о компиляторах из старой «Книги Дракона» Ахо и Ульмана¹. Думаю, я сделал клон утилиты `unifdef`. Дэйв Йост уже сделал до меня что-то подобное, но его программа не работала с выражениями `#if` и не делала минимизацию выражений в том случае, если одни выражения определялись как `#`, а другие не определялись. Моя программа до сих пор используется и даже, кажется, применяется в Linux.

Я работал в SGI с 1985 по 1992 г. В 1992 г. один мой коллега из SGI перешел в MicroUnity. SGI мне к тому времени уже поднадоела – она все распухала, скупая другие компании, и постепенно переходила под контроль политиков. Так что я оказался в MicroUnity, о которой Джордж Гилдер тогда писал в журнале «Forbes ASAP», что это будет очередной компьютерный монстр. Но все закончилось ничем – компания набрала кредитов на 200 миллионов и обанкротилась. Это было крайне поучительно. Там я занимался компилятором GCC и улучшил свои навыки работы с компиляторами. Еще я писал небольшой язык для редактора видеофайлов MPEG2, на котором можно было писать псевдоспецификации, похожие на стандарты ISO или IEC, и он генерировал тестовые битовые потоки с правильным синтаксисом.

Сейбел: Из MicroUnity вы ушли в Netscape, и дальше все хорошо известно. Скажите, вас устраивает то, как вы учились программированию, или не совсем?

Аик: Я много занимался физикой, прежде чем обратился к математике и компьютерным наукам. Я довольно много занимался математикой, получая через это знания по программированию, а также изучал кое-что самостоятельно и поэтому на занятиях сидел в задних рядах – скучал, ерзал, делал что-то свое. От этого сильно страдала самодисциплина, и я, вероятно, пропустил кое-что важное.

Разговаривая с теми, кто получил PhD, я понимал: кое-что они знают лучше меня. И я думал об упущенных возможностях, которых уже не вернуть. Можно освоить что-нибудь благодаря Интернету, но разве это

¹ А. Ахо, Р. Сети, Д. Ульман «Компиляторы. Принципы, технологии, инструменты». – Вильямс, 2003.

заменит хорошего преподавателя и систематический курс обучения? Правда, я не очень жалею об этом.

Что касается программирования, то я всегда говорил, что занимаюсь низкоуровневым программированием. Объектно-ориентированное программирование, шаблоны проектирования – это не для меня. Я так и не купил книгу Эриха Гаммы. Кое-кто в Netscape потрясал этой книгой как Библией – наши с Джейми Завински враги-коллеги, пришедшие в компанию после ее покупки. Просто невыносимо, учитывая, что это были далеко не лучшие программисты.

Сложись все иначе, я мог бы заниматься и высокоуровневыми вещами. Думаю, работая в Mozilla и имея дело с Firefox, я узнал больше о разработке через тестирование – то был ценный опыт. Было и кое-что еще, например тестирование с использованием случайных данных, которого проводилось много. У нас было много исходных языков, были большие и глубокие конвейеры рендеринга, сильно подверженные ошибкам, связанным с безопасностью доступа к памяти. Вообще, тестирование с использованием случайных данных оказалось самым продуктивным видом тестирования.

Я также стоял за увеличение вложений в статический анализ, и это оказалось правильным, хотя сама по себе штука довольно темная. Но мы наняли тех, кто хорошо в нем разбирался.

Сейбел: Статический анализ какого именно вида?

Аик: Анализ языка C++, выполнить который непросто. Обычно при статическом анализе вы анализируете программу целиком и стремитесь, скажем, доказать корректность состояния памяти. Нужно устранить все неоднозначности, а для этого найти в памяти все альтернативные имена – это проблема экспоненциального характера, обычно нерешаемая ни для одной более-менее крупной программы. Большой прорыв, однако, заключался в том, что больше не надо было волноваться насчет памяти. Если построить полную диаграмму исполнения команд и связать воедино все виртуальные методы с их возможными реализациями, то можно частично оценивать код, не запуская его. Можно найти недостижимый код, можно найти избыточные проверки и пропущенные проверки на NULL.

Можно сделать еще больше на высших уровнях, где работаешь, когда в голове есть система доказательств для программы, которую пишешь. Но в обычных языках нет системы типизации для выражения терминов доказательства. И это серьезная проблема. Согласно Карри–Говарду, существует зависимость между логическими системами и системами типизации, типы являются термами, а программы – доказательствами, и поэтому в принципе можно описать высокоуровневую модель, которую намереваешься создать. Например, такой-то массив на раннем

этапе должен иметь ограничение по длине, а на прочих этапах имеет другие ограничения – или не имеет их вовсе. Хитрость отчасти и состоит в том, что на разных этапах действуют разные правила. Или, например, вы надежно защищены внутри своей абстракции и ради большей эффективности нарушаете собственные инварианты – но при этом знаете, что делаете, и знаете, что с внешней точки зрения вы в порядке. Все это очень трудно реализовать в программах с жесткой проверкой типов.

Когда пишешь программу на языке Haskell, приходится выбрать систему доказательств еще до того, как толком поймешь, что именно делаешь. Динамические языки стали популярными, поскольку человек может быстро создать прототипы и держать в голове потенциальную систему типизации. А уже потом, если язык поддерживает это свойство или при перекодировании в статический язык, можно создавать типы. Это одна из причин того, почему мы были заинтересованы в наличии необязательной типизации в JavaScript. Мы заинтересованы в этом до сих пор, хотя среди руководства есть разногласия. Есть неплохие шансы на то, что в будущей версии JavaScript мы получим какую-нибудь смешанную систему типизации.

Поэтому мы хотим снабдить наш C++ аннотациями, на которые можно опираться при консервативном статическом анализе. Именно при консервативном, когда программа не зависит, до бесконечности пытаюсь решить проблему экспоненциального свойства. Это поможет нам с проверкой безопасности сборщика мусора или с разделением функций на те, которые получают управление из скриптов, и те, которые передают управление скриптам, или с восстановлением стека интерпретатора, чтобы выносить заключение о безопасности. Мы получим параметры безопасности, которые сможем проверять. Многие из них – это высокоуровневые параметры и относятся не только к безопасности памяти. Поэтому мы должны продолжать борьбу на этом поле.

Сейбел: Да, это весьма высокий уровень программирования. Как повашему, насколько близко сегодняшние программисты должны стоять к «железу»? Нужно ли тому, кто в основном пишет приложения на JavaScript, разбираться в языке ассемблера?

Айк: Многие знакомые мне JavaScript-программисты очень умны, и лучшие из них хорошо знакомы с экономическими расчетами. По мере написания кода они измеряют производительность и тестируют, делая все на высоком уровне. Им необязательно знать, как пишутся инструкции для машины.

Многие из них начинают интересоваться этим, когда слышат о компиляции «на лету», видят создаваемые нами виртуальные машины. И все больше народа работает с графикой. При высокой мощности языка и хороших графических параметрах, думаю, немало JavaScript-

программистов начнут пользоваться языком на более низком уровне. И потом, экономические расчеты для физических и виртуальных машин – что важнее? Возможно, расчеты для виртуальных машин.

Абстракция – великая сила. Что у меня вызывает аллергию еще с 1990-х, так это CORBA, COM, DCOM, всякая объектно-ориентированная чушь. Каждый новый проект в то время обязательно имел какую-нибудь примочку, которой было нужно 200 000 вызовов только для запуска и приветствия. Это профанация. Настоящие программисты таким не занимаются. В SGI ядро делали настоящие крутые программисты, и там никто не занимался ерундой. Выделение динамической памяти ядра через `malloc` было тогда совсем новым делом. Мы использовали таблицы с фиксированным размером и паниковали, заполнив их целиком.

Стоять близко к «железу» для меня лично значило оставаться честным, избегая всякого дерьма. Но со временем, как известно, оборудование стало лучше, мы научились отделять хорошие абстракции от плохих. Наверное, сейчас можно не знать язык ассемблера и при этом быть хорошим программистом, писать грамотный код.

Сейбел: А если взглянуть с обратной стороны: может ли сегодня тот, кто раньше писал замысловатый код на языке ассемблера, успешно заниматься высокоуровневым программированием? Или здесь нужны разные навыки?

Айк: В некоторых областях эти навыки смыкаются. Есть разница между миром простых указателей и солнечным, радостным миром Java-Script. Здесь проходит граница между истинными программистами и всеми прочими.

Важно держать все в голове. Конечно, у всех разная память. Особо памятьливые могут держать в уме набор высокоуровневых инвариантов при архитектуре с безопасным доступом к памяти, не заботясь при этом об указателях. Но иногда меня беспокоит, что мы постепенно утрачиваем способность писать для «железа». Это делают другие; компилятор генерирует код. Видимо, спрос на таких людей будет расти.

Сейбел: Итак, этот вид программирования никуда не денется. А есть ли те, кто может быть успешным программистом сегодня, однако был бы беспомощен в мире низкоуровневого кода? Или программирование требует особого склада ума, и те, кто им обладает, просто разделились на низкоуровневых и высокоуровневых программистов?

Айк: Я давно не занимался кодом ядра, и если нужно будет заняться, это потребует от меня усилий. Теперь нужно писать больше кода. Кроме того, удачные абстракции позволяют сейчас справляться с задачами, не решаемыми в прошлом.

Сейбел: Вернемся к тем десяти дням, когда вы реализовали изначальную версию JavaScript. Я слышал, кто-то обратил ваше внимание на книгу Абельсона и Сассмана¹, и первоначально вы собирались встроить в браузер Scheme.

Аик: Чем прежде всего были озабочены в Netscape? Все должно быть как в Java! Другие создавали алголоподобный синтаксис для Лиспа, но у меня не было времени взять Scheme за основу. Пришлось делать все напрямую, а значит, я мог совершать те же ошибки, что и другие.

Я не стал делать тотальный динамический контекст, на котором настаивал Стеллмен для Emacs и которым он наводнил Elisp. В JavaScript контекст в основном лексический, но есть отступления в виде динамических элементов: глобальный объект, оператор `with`, функция `eval`. Но ничего похожего на `$`-переменные в Perl до введения `my` или на команды `upvar` и `uplevel` в Tcl. В 1990-е все это было очень модно.

Однако я не остановился на Scheme – из-за спешки. Было очень мало времени, чтобы задуматься над последствиями своих действий. Я экономил усилия на многих объектах, которые должны были реализовываться в браузере. Поэтому я сделал `window` глобальным объектом, который является источником связывания необъявленных новых имен и делает невозможными статические суждения о свободных переменных. А жаль. Дуг Крокфорд и прочие приверженцы объектной модели были недовольны тем, что вы получаете нежелательную власть над глобальным объектом. Это другой способ сказать то же самое. В JavaScript есть безопасные ссылки на объекты в памяти, и мы уже близки к цели, но остаются большие недочеты – те самые отступления.

Эти переменные, привнесенные на высокий уровень, теперь становятся изменяемыми свойствами объекта, которые можно тасовать как угодно за спиной у кого-то – это плохо. Должно быть лексическое связывание. Тогда, если спускаться вниз, к функциям и вложенным функциям, будет больше похоже на Scheme. У вас не будет богатых форм связывания, макросов вроде `fluid-let` – скорее, что-то вроде `set!`. Но изначальное связывание, создаваемое при помощи локальной переменной, – это лексическая переменная.

Сейбел: Выходит, сегодня для получения пространств имен создаются высокоуровневые функции?

Аик: Да. Функция создается и тут же вызывается. Это дает безопасную среду для связывания, приватные переменные. Дуг – ярый пропагандист всего этого. Тем, кто работал со Scheme и Лиспом, это уже

¹ Х. Абельсон, Д. Д. Сассман «Структура и интерпретация компьютерных программ». – Добросвет, 2006.

было отчасти знакомо, но многим JavaScript-программистам пришлось осваивать все с нуля. Дуг и его коллеги провели большую работу по их обучению. Увы, сделать грамотных Scheme-программистов из всех не получилось, но, по крайней мере, люди стали понимать функциональные идиомы, пусть неглубоко, но хотя бы на уровне шаблонов.

Сейбел: Итак, JavaScript примерно десять лет был в тени. Сейчас наблюдается его бурное возрождение благодаря Ajax. Все говорят: «Нам нужно взглянуть на это по-другому». Вы недавно оказались в центре драматической истории, связанной с соперничеством между ECMAScript 4 и ECMAScript 3.1. В конце концов был предложен план «Гармония», предусматривавший объединение двух версий в одну. Что стояло за ES4 – желание показать, что вы действительно классный программист, а JavaScript – хороший язык?

Айк: Не думаю. Может, Дуг и думает так. Вряд ли он знает меня настолько хорошо. Нет, я и вправду не ищу признания ни среди приверженцев Java, ни среди рядовых разработчиков.

Сейбел: Был ли ES4 вашим детищем? Как вы оцениваете его с сегодняшних позиций – как идеальный вариант JavaScript?

Айк: Нет. То был плод коллективных усилий и в какой-то мере компромисс, поскольку мы работали с компанией Adobe, создавшей производный язык ActionScript. Третья версия этого языка повлияла на нашу работу. А ее основой стали наработки Вальдемара Хорвата в отношении изначальной версии JavaScript-2 и предложений по четвертой версии ECMAScript конца 1990-х. Их положили под сукно в 2003 г., когда компания Netscape закончилась и была основана Mozilla.

Вальдемар сделал все как надо – я дал ему ключи от королевства в конце 1997 г., когда уходил создавать mozilla.org вместе с Джейми. Вальдемар – это могучий ум: кажется, он выиграл Путнамовскую олимпиаду в 1987 г. PhD MIT (Massachusetts Institute of Technology). Он сохранил за языком его динамическую окраску, но при этом вел борьбу за включение некоторых элементов, свойственных «программированию побольшему», например пространств имен.

Есть противоположный подход, более педантичный: «У нас будет лишь несколько примитивов, мы удалим из спецификации весь синтаксический сахар¹. Мы все переведем на лямбда-выражения. Так должен писать каждый, потому что я так думаю». Это упрощенчество, которое подходит не для каждого. Конечно, один из способов выстроить в уме

¹ Синтаксический сахар (syntactic sugar) – дополнения синтаксиса, которые не добавляют новые возможности, но делают язык программирования более удобным в использовании. – *Прим. науч. ред.*

собственную систему доказательств – это упрощать все, делить языки на подмножества. Мощный метод. Но ведь не каждый сможет программировать в крохотном подмножестве.

Сейбел: В одной из дискуссий по поводу ES4 вы цитировали статью Гая Стила «Growing a Language» (Выращивание языка). Как старый Лисп-программист, я сделал из нее прежде всего такой вывод: введите в язык макросы, и весь сахар исчезнет.

Аик: Разумеется, есть две крупные проблемы. Си создает куда больше забот, чем s-выражения, поэтому надо определить абстрактные синтаксические деревья, потом стандартизировать их – это мучительный процесс. Есть и проблема гигиены, которую пока еще плохо понимают. Дэйв Херман, который работает вместе с нами, пишет – по крайней мере, писал – диссертацию насчет разновидности логики, которая сможет обеспечить гигиену. И это здорово, ибо нас ждет переход на макросы.

Я говорил об этом Дугу Крокфорду несколько лет назад, когда он пригласил меня выступить в Yahoo! Я начал говорить о сахаре, горячим сторонником которого был. «А не разработать ли нам сперва систему макросов?» – спросил он, и я ответил: «Нет, это займет девять лет». Тогда был реальный риск, что Microsoft откажется от сотрудничества. После нескольких лет летаргии они снова заинтересовались ЕСМА. Их новый парень – он был из Хайдарабада – отнесся к этому с энтузиазмом, сказал, что они включают CLR в IE8, а JScript.net будет их новой реализацией JavaScript для Сети. Но, по-моему, наверху погасили его энтузиазм, и он отказался от своих слов. В нашем руководстве тогда произошел раскол.

Мы беспокоились, что переход на макросы потребует исследований, а это означало, что у нас не будет обязательств перед Microsoft и мы не сможем оказывать на них конкурентное давление. Макросам пришлось подождать. Будем создавать хорошие средства автоматической проверки грамматики, сделаем так, чтобы весь сахар превратился в макросы, когда настанет время макросов. А пока – зачем лишать пользователей сахара? Зубы не испортятся, а ошибок будет меньше.

Сейбел: Вернемся в 1995 год. Какие еще языки повлияли на первоначальный проект JavaScript?

Аик: Self был популярен, в основном благодаря статьям Дэйва Унгара. Я никогда не работал с кодом Self, но вдохновлялся теми статьями. Я люблю Smalltalk. Кое-кто у нас взял из Smalltalk идею делегирования на основе прототипов – с несколькими прототипами, в отличие от JavaScript, и очень плотно стал работать в этом направлении. Smalltalk мне нравился – там был хороший компилятор, инженерные подходы на уровне виртуальных машин и, как я считаю, хороший дизайн языка.

Подобно Кроку и некоторым другим, я сторонник простоты. Мне симпатичны те проектировщики языка, которые берут немного примитивов и смотрят, как далеко можно с этим пойти. С разработчиками JavaScript, по-моему, случилось нечто вроде «стокгольмского синдрома»: «Он делает то, что делает, только потому, что Microsoft не дает это совершенствовать. Зачем нам улучшать синтаксис – теперь все идет через лямбда-программирование». Если оставить в стороне «стокгольмский синдром» и тот факт, что Microsoft тормозит развитие Сети, проектировщику языка стоит взять одну-две идеи для ядра и настойчиво воплощать их в жизнь.

Сейбел: А с NewtonScript вы знакомы?

Айтк: Только после того, как кто-то ткнул пальцем, и я понял: «Да у них же что-то похожее на нашу цепочку областей видимости по ссылке на родителя и наш одиночный прототип!» Думаю, то была конвергентная эволюция на базе Self. А что касается обработки событий DOM, здесь повлияли HyperTalk и аткинсоновский HyperCard. Так что я учитывал не только Self и Scheme, но и обработчики событий onFoo в HyperTalk: я реализовал это в DOM в виде onClick и не только.

Стыдно признаться, но положительное влияние на меня оказал также awk. Да, я был старым UNIX-хакером, уже вышел Perl, но для рутинной работы я часто использовал awk. Я назвал функции первого класса «функциями» прежде всего из-за awk. Слово «function» из восьми букв выглядит тяжелым, но тем не менее.

Сейбел: По крайней мере, это не «lambda» – иначе JavaScript был бы заранее обречен. Скажите, а на JavaScript что-нибудь повлияло отрицательно – то есть в смысле «я так ни за что не сделаю»?

Айтк: Работа была очень спешной, и я не сильно задумывался о том, чтобы отмежеваться от Ada или Common Lisp. Java в какой-то мере оказал такое воздействие. Мне надо было, чтобы язык напоминал Java, но не содержал безумных вещей, вроде различия между примитивными типами и объектами. И я не хотел создавать классы. Поэтому я стал смотреть в сторону Self и делать первоначальные наброски.

Сейбел: Вы думали о том, чтобы сделать язык, стоящий ближе к Java, – взять Java и упростить, избавиться от примитивных типов и прочих ненужных сложностей?

Айтк: Со стороны руководства ощущалось давление: они хотели, чтобы синтаксис был похож на тот, что в Java. Еще они хотели, чтобы язык не становился слишком объемным, – для сложного программирования был Java, а наш язык замышлялся как его младший брат.

Сейбел: Чтобы напоминать Java, но не слишком.

Айк: Не слишком. Если бы я ввел классы, то столкнулся бы с большими проблемами. У меня и времени, правда, на это не было, но я бы их все равно не ввел.

Сейбел: Вернемся в наши дни. ES4 была официально отвергнута, и сейчас идет работа над ES-Harmony, сочетающей черты ES3.1 и ES4. Как вы полагаете, это удачное решение?

Айк: Дуг, пожалуй, слишком уж торжествовал в своем блоге: «Мы победили. Мы одолели дьявола». Год назад я подарил ему в Лондоне шуточный слайд: Дуг в виде Гэндальфа на мосту в Казад-Думе, глядящий вниз на ниспровергнутого ES4рога. Ему очень понравилось. Я впервые подшутил так над ним – его порой оставляет чувство юмора, когда он касается этих проблем. Но ему понравилось. Может, он и герой, но ES4 совсем не был чудовищем.

ES4, как мне кажется сейчас, был слишком объемным. Но нам нужно было прагматично подойти к стандартам. Мы не могли сказать: «Все, что вам нужно, – это лямбда-выражения». Алонзо Чёрч доказал, что это не так. И мы не могли вставлять еще больше таких выражений. Такой подход, предполагающий высокую квалификацию каждого, многого не учитывает, например того, что многих программистов в Java-вузах научили плохому. JavaScript однажды *умрет*, но можно совершенствовать его, поддерживать его конкурентоспособность как в теоретическом, так и практическом плане. При условии, конечно, что мы не откажемся от сахара ради чистоты.

Язык должен совершенствоваться, чтобы можно было решить стоящие перед программистами задачи. С некоторыми можно справиться, если писать собственные библиотечные абстракции. Но возможность написания абстракций для языка очень ограничена, если нет расширений – вы не можете написать геттеры и сеттеры¹. Объекты будут смотреться чужеродно, свойства не смогут стать кодом и так далее. Кроме того, вопросы безопасности нельзя будет разрешать неявно или автоматически.

Сейбел: Как вы полагаете, языки в целом со временем улучшаются?

Айк: Думаю, да. Может быть, сейчас наступает второй «золотой век» – растет интерес к языкам и их созданию. Мы говорим о программировании: нужно совершенствовать свои навыки, как в писательском ремесле или музыке. Но используемый язык – «тональная система» – тоже важен. Надо поэтому совершенствовать языки, а не довольствоваться

¹ Геттер, сеттер – специальные методы, используемые в объектно-ориентированном программировании и позволяющие реализовать гибкий механизм инкапсуляции. – *Прим. науч. ред.*

тем, что есть. Сеть требует совместимости, и JavaScript, возможно, склонен оставаться таким, каков он есть. Но не стоит заикливаться на этом. Мы обязаны улучшать JavaScript, даже если он не заменит тот, что применяется для веб-разработок. Другой вариант – создать нечто совсем новое.

Есть, например, Ruby, испытавший влияние языков Ada и Smalltalk. Это прекрасно. Я не имею в виду эклектичность. Но Ruby перехваливают. Нет, ничего плохого сказать не могу, просто некоторые мальчишки-фанаты трубят, что он решит все проблемы, а это не так. Новые языки нужны, но хвалить их надо в меру, не как C++ – «шаблоны проектирования нас спасут». Конечно, может быть, это реакция на консерватизм приверженцев Си-мира UNIX 1980-х.

Но в какой-то момент нам требуются более совершенные языки. Ведь необходимо иметь инструменты для доказательства, производящие автоматическую верификацию некоторых постулатов, сделанных в вашем коде. Вам ведь не нужны ответы на все, правильно? Динамические инструменты, такие как Valgrind и его детекторы состояний гонки¹, тоже очень полезны. Простых решений нет, но есть более совершенные языки, на которые надо переходить.

Сейбел: В какой степени языки должны предотвращать ошибки программистов?

Айт: Язык для «синих воротничков», такой как Java, должен иметь четкую и логичную базовую систему, так как «синим воротничкам» трудно понять, что это за вывороченный синтаксис с ковариантными и контравариантными ограничениями типов. Я немало пострадал из-за того, что порой выкидывают Си и C++. Программирование отчасти состоит в конструировании, а конструирование отчасти состоит в решении проблем безопасности. Они важны при разработке броузера и еще больше – если вы делаете программу для аппарата лучевой терапии. Так или иначе, речь идет о более совершенных языках для написания параллельных программ или эффективного использования аппаратного параллелизма. Мы не всегда будем использовать синхронизированные блоки и уж точно не будем использовать мьютексы или взаимные блокировки. Поэтому языки могут служить для установления разумного компромисса между безопасностью и выразительностью.

Мне кажется, с JavaScript мы стремились именно к этому, в противоположность диким, неуправляемым французам, которые хотели видеть JavaScript чем-то вроде x86 с лямбда-выражениями. Нам совершенно

¹ Race conditions – ошибки в многопоточном коде из-за одновременного изменения объекта в памяти несколькими потоками, оставляющими объект в некорректном состоянии. – *Прим. науч. ред.*

незачем вводить оператор `call/cc`. Предположим на минуту, что обойдется без проблем с реализацией: народ будет сбит с толку. Не обязательно большинство, но многие из тех, кто считает себя крутыми программистами. Есть нечто вроде программистской пирамиды, на вершине которой располагается Правильная вещь. Люди карабкаются к вершине, даже если некоторые срываются, получая травмы.

В JavaScript можно нарваться на неприятности разными способами. Есть функции первого класса. Есть малопонятные для людей прототипы – они не отвечают классическим стандартам объектно-ориентированного программирования.

Многовато. Я не минималист и не утверждаю, что мы должны прекратить развитие языка. Это удобная отговорка в Microsoft, которая меня бесит, потому что люди тратят время, а ошибки не устраняются. Даже при наличии лямбда-выражений вы все равно не избавитесь от кое-каких трудноустраняемых ошибок.

Дуг стал учить народ применению различных шаблонов, но я согласен с Питером Норвигом: эти шаблоны выявили определенный дефект языка. Они не бесплатны. Бесплатного ланча не бывает. Мы должны добиваться эволюции языка в верном направлении. Не исключено, что появится необязательная типизация, и она может напоминать систему контрактов PLT.

Сейбел: Многое из того, что вы делаете, от статического анализа вашего C++ до работ в области компиляции «на лету» и добавления новых свойств в JavaScript, говорит о том, что вы стараетесь держаться на переднем крае компьютерных исследований.

Аик: Мы ведем справедливую борьбу, но надо делать это с умом. Надо также изменить направление исследований, так как – для меня это было очевидным еще в школе – академическая наука выглядит чем-то ненормальным. Она сильно оторвана от практики.

Надо это исправлять. Мы успешно работали с практически мыслящими учеными. Денег у нас немного, поэтому отчасти мы занимаемся тем, что попросту разговариваем с людьми, служим для них объединяющим центром.

Если ученые охотятся за правительственными грантами и никак не причастны к вашей деятельности, это в какой-то степени потеря для вас. Далее, вы наблюдаете бурный взлет динамических языков, слушаете безумные заявления насчет того, как они убьют Java, и что статические языки – просто смешно. Но ученые убеждены, что статическая типизация – это венец эволюции, они исследуют особые ее виды – язык ML, вывод типов Хиндли–Милнера. Это полный отрыв от практики.

Сейбел: Почему так происходит? Они не решают реальных проблем – или решение было бы половинчатым?

Айк: Мы делали кое-что с SML/NJ для размещения на резидентном сервере эталонной реализации четвертой версии JavaScript, ныне несуществующей. Мы пытались создать дефиниционный интерпретатор и даже не использовали модель Хиндли–Милнера. Мы собирались аннотировать типы и аргументы так, чтобы избежать этих безумных сообщений об ошибке, когда типы не унифицируются и подбирается наудачу фрагмент исходного кода – обычно не тот, что нужно. Это вопрос качества реализации. Возможно, это также теоретическая проблема, связанная с типами, – когда не получается унификация, трудно найти, какой фрагмент кода тому причиной.

Итак, вы можете проводить дополнительные исследования и разрабатывать высокоуровневую модель когнитивных ошибок, допускаемых программистами, чтобы лучше определить этот фрагмент кода. Может быть, я касаюсь мелкой проблемы, но, скорее всего, она значительна.

Академическая наука не способна предложить нам более совершенную модель. Мне кажется, сейчас это негодная вещь – но, возможно, не по вине ученых. Экономические основы этой науки прогнили. Мы все знали, что придем к массовому параллелизму, – никто не предложил решения. Сейчас все только и говорят, что о транзакционной памяти. Но это не решение. Вы не хотите иметь дело с тем, как вложенные транзакции отменяются и распространяются на один процессор за другим. Это неэффективно, это не будет работать в отдельных случаях. Вы не сможете переложить на это все свои конкурентные или параллельные алгоритмы, даже пытаться не будете.

Некоторые, например Джо Армстронг, проделали действительно большую работу с подходом без разделяемых ресурсов. Таких встречается много в создаваемых индивидуально системах в реализациях браузеров. В этом смысле выделяется Chrome. Мы сделали это по-своему в нашей реализации JavaScript. Но этот подход, по-моему, совсем не интересует ученых. Транзакционная память вызывает больше интереса, особенно там, где это касается компьютерной архитектуры, поскольку они способны разработать хорошие наборы инструкций и оборудование под них. Но это не решит всех наших сегодняшних проблем.

Мне кажется, что прогресс неизбежен, что он затронет языки программирования. И поэтому разговоры о «втором золотом веке» я воспринимаю как должное. Просто пока не установлена связь между пользователями языков и потенциальными разработчиками из числа ученых, способными создать действительно революционный язык.

Сейбел: Вы получили степень магистра, но диссертацию так и не защитили. Вы бы порекомендовали будущим программистам защищать диссертацию по компьютерным наукам? Или это нужно далеко не всем?

Айк: По-моему, далеко не всем. Для этого нужны особые навыки, и в конце концов вы спрашиваете себя, а может, степень досталась мне просто в качестве компенсации за перенесенные страдания? Зато после этого мы можете прибавлять после фамилии «доктор компьютерных наук». Это открывает перед вами кое-какие двери. Но мой опыт работы в Кремниевой долине на протяжении 20 лет инфляционного бума – который, возможно, подходит к концу – говорит о том, что это дело невыгодное. И я ни о чем не жалею.

Это соблазнительно – поизучать что-нибудь систематически, и может быть, даже не особо напрягаясь. Выход на рынок, зависимость от закона Мура, конкуренция, короткий жизненный цикл продукта, одноразовые программы – все это тягостно, если в этом варятся все. Поэтому тем, кто хочет получить степень, кто имеет для этого нужные навыки, всегда будет чем заняться. Есть интересные области для исследования. Мы в Mozilla занимаемся вещами, промежуточными между тем, что ценится в академических кругах, и тем, что воплощается на практике. Это компиляторы, виртуальные машины, даже отладчики, профилировщики – вроде тех, что делает Valgrind. Для исследователей это слишком пресно и малоденежно, не очень ново, слишком много чисто инженерных задач – но здесь возможны крупные прорывы. Мы сотрудничаем с Андреасом Галом (Andreas Gal), и его работы по этим темам были отвергнуты как чрезмерно прикладные.

Конечно, нам нужны исследователи, которые склоняются к прикладным задачам, но также и программисты, которые занимаются исследованиями. Мы должны четко понимать, что делаем, а не быть «синими воротничками», на которых смотрят свысока мудрецы, засевшие в башнях из слоновой кости.

Сейбел: А что насчет доказательства правильности программ?

Айк: Это трудно. А люди по большей части ленивы. Ларри Уолл прав: лень должна считаться достоинством. Вот почему я предпочитаю автоматизировать этот процесс. Ученые его любят, большинство программистов ненавидят. Писать предикаты утверждений – это может принести пользу. У нас в Mozilla было несколько плохих утверждений – скорее они должны были быть предупреждениями, – но со временем хороших становится все больше. Благодаря этому нас наконец озарило, что есть инварианты – те, которые вы хотели бы реализовать в некоей идеальной системе типов.

Думаю, полезно считать утверждения точками доказательства правильности программы. Но не нужно стремиться к полному доказательству. Сколько дыр в серьезных доказательствах, напечатанных в научных журналах!

Сейбел: Давайте сменим тему. Можете ли вы припомнить худшую из ошибок, которую вам довелось отлавливать?

Айк: Худшие ошибки связаны с многопоточностью. В Silicon Graphics я делал работу, связанную с ядром UNIX. Как и все тогдашние UNIX-ядра, оно представляло собой монолитный монитор, который завершался после вхождения в ядро через системный вызов. Исключая прерывания, оно гарантированно работало вплоть до завершения, и блокировка вашей структуры данных никогда не наступала. Прекрасно и просто.

Но вот в SGI пришли блестящие молодые умы из HP. Настала эпоха симметричной мультипроцессорной обработки данных. Старая группа, которая занималась ядром, распалась. Теперь ядро делали новые ребята. Темп работы сильно ускорился, но какие у них были инструменты? Си, семафоры, блокировки, возможно, также мониторы, условные переменные. Все коды написаны от руки. Тысячи ошибок. Полный кошмар.

Мне предложили тогда съездить в Австралию и Новую Зеландию – я описал все это в своем блоге. Мы тогда как раз исправляли ошибку в полевых условиях. Это было страшно тяжело – найти ее и исправить, потому что ошибка была такого свойства: код для однопроцессорного ядра помещался в ядро, созданное для симметричной мультипроцессорной обработки данных, и мы совсем не беспокоились насчет определенных условий гонки. Поэтому для исправления пришлось создавать контрольный пример, что само по себе было не просто. И все это при нехватке времени – клиент хотел исправления в полевых условиях.

Диагностировать ее было трудно, так как она была связана с синхронизацией по времени. Машины использовались не по назначению, как концентраторы терминалов. Люди подвешивали псевдотерминалы к реальным терминалам. Это делали студенты в лаборатории или сотрудники брисбенской компании, производившей ПО для горной промышленности: множество отсеков и в конце стеклянная стена, а за ней компьютеры, в том числе двухпроцессорная машина от SGI. Было нелегко, и я рад, что мы все же нашли ошибку.

Обычно такие ошибки не сидят годами, но отыскать их крайне трудно. Нужно как бы приостановить все, думать о них постоянно, видеть их во сне... А заканчивается все тем, что вы делаете элементарные вещи. Так бывает со многими ошибками. Все заканчивается бисекцией, по методу

волка и забора¹. Вы постоянно следите за выполнением, за состоянием памяти, пытаетесь прикинуть размер ошибки, течение исполнения программы, понять, к каким данным можно обратиться. Если это куча голых указателей, дело плохо: следует обратиться к более современным инструментам, которые появились вместе с гигагерцными процессорами, вроде Valgrind и Purify.

Инструментирование и наличие контролируемой модели всей иерархии памяти – это большое дело. Роберт О’Каллагэн, могучий новозеландский ум, создал собственный отладчик на базе Valgrind: он записывает каждую инструкцию, и можно в любой момент восстановить состояние программы целиком. Это не только отладчик, путешествующий во времени. Это целая база данных: вы видите структуру данных, замечаете поле с безумными значениями, выясняете, кто делал там последнюю запись. Вы идете от следствий к причинам – в отладке это занимает очень много времени. Это в тысячу раз медленнее, чем все происходит в реальном времени, но у вас есть надежда.

Можно также использовать записывающие виртуальные машины – они записывают состояние только при системных вызовах и на границах ввода/вывода. Они могут воссоздать состояние поврежденной программы на каждой границе – правда, со всем, что между границами, намного сложнее. Зато все можно закончить быстро, практически в реальном времени, потом перенести программу в Chronomancer, запустить ее в медленном темпе, воссоздать все состояния и найти ошибку.

К сожалению, технология отладки мало исследована. Вот еще пример пропасти между учеными и практиками. Ученые создают доказательства правильности, часто вручную, – правда, эта работа все больше автоматизируется благодаря POPLmark и подобным инструментам. Но в реальной жизни везде есть только отладчики, встречаются даже развалюхи родом из 1970-х, вроде GDB.

Сейбел: В реальной жизни есть еще разрыв между приверженцами символических отладчиков и операторов вывода?

Айк: Да. Поэтому я использую GDB и доволен тем, что в нем, по крайней мере на Make, есть возможность поставить точку прерывания, и это обычно работает. Я могу наблюдать за адресом, могу засечь момент, когда правильные биты сменяются неправильными. Это довольно полезно. Я также использую команду `printf` для бисекции. Когда я уже близок

¹ «Wolf fence» – алгоритм поиска, аналогичный методу поимки льва в пустыне (здесь – волка на Аляске). Реализуется размещением в коде инструкций `print`, позволяющих определить местоположение ошибки. – *Прим. ред.*

к цели, то обычно пытаюсь сделать что-то внутри GDB или пользуюсь командными сценариями (скриптами), хотя они очень слабы. Язык сценариев сам по себе очень слаб. По-моему, Ван Якобсон добавил циклы, но не знаю, использовались ли они в настоящем GDB, после семинаров, организованных Фондом свободного программного обеспечения.

Однако отладчики могут сделать бесконечно больше, чем делают сейчас, и в этом смысле Chronomancer и Replay – шаг вперед. Они изменили для меня весь процесс. Но вот насчет многопоточности не знаю. Есть Helgrind и другие динамические детекторы гонок, которыми мы пользуемся. Они дают ложные срабатывания, которые надо отсеивать. С ними пока еще не все до конца понятно.

Многопоточные процессы страшат меня – до того как я женился и завел детей, они съели немалую часть моей жизни. Не все задумываются насчет параллелизма и всех возможных комбинаций команд, происходящих даже в небольших сценариях. Если ваш код соединяется с чужим, он выходит из-под контроля. Вы не можете мысленно смоделировать происходящее. Большинство людей не могут. Я мог бы стать одним из этих всезнаек с сайта Slashdot: когда я писал в своем блоге, что против многопоточности, кое-кто говорил: «Да он ничего не знает, это несерьезно». Давай, болтай. Я слетал в Австралию и Новую Зеландию, мне достались кое-какие бонусы. Но все это было так мучительно и продолжалось слишком долго. Как сказал Уайльд о социализме, «Он отнимает слишком много вечеров».

Сейбел: Как вы проектируете код?

Айк: Я много прототипирую. Создаю нечто вроде высокоуровневого псевдокода и затем постепенно заполняю его снизу вверх. Этот псевдокод я обычно не пишу, а держу в голове, и иду снизу вверх, пока оба конца не сойдутся. Часто я работаю с готовыми фрагментами кода, добавляя новую подсистему или что-то постороннее, и почти все делаю снизу вверх. Если в середине я сталкиваюсь с трудностями, то опять пишу псевдокод и начинаю работать все в том же направлении – снизу вверх, пока не закончу его. Я стараюсь не затягивать с этим, так как проверить псевдокод невозможно – надо смотреть, как он работает, выполнять его шаг за шагом, убеждаясь, что он делает именно то, чего от него ждешь.

Еще до этого этапа я могу установить кое-какие связи между объектами, вчерне набросать модули. Обычно есть два-три алгоритма, и можно прикинуть их сложность – линейная она или константная. Всякий раз, когда я выполнял поиск с линейной сложностью, который затем складывался в квадратичный, для веб-разработчиков это была проблема. Поэтому мы предпочитаем делать много структур данных с константным временем доступа. Но даже и тогда эта константа может не быть

единицей – она может быть достаточно большой, чтобы о ней побеспокоиться.

Поэтому мы создаем множество прототипов, работаем над разными кусками с обоих концов, которые сводим в середине. Я считаю, что сейчас мы в Mozilla недостаточно переписываем код. Мы слишком консервативны. У нас открытый исходный код, поэтому вокруг нас создаются сообщества, мы заинтересованы в новых людях. Мы работаем в интересах пользователей и поэтому не можем позволить себе трехлетний перерыв на переписывание – хотя если очень постараться, то смогли бы.

Но если вам нужно избрать другой путь, и вы не знаете в точности какой, – переписывайте. Если хотите понять, что, черт возьми, вы тут делаете, потребуется несколько попыток. Код становится лучше по своему проектному решению, вы останавливаетесь на этой версии, начинаете ее латать, пока не дойдете до предела. Это нечто вроде эволюционного тупика для кода. Возможно, при приемлемых невозместимых издержках этот код останется на годы. А может, потребует замены. Вдруг в мире открытого исходного кода появится более совершенная стандартная библиотека?

Это возвращает нас к ремеслу программирования. Вы не просто пишете код в соответствии со старым проектным решением. Вы хотите постоянно практиковаться, а это значит думать о проектном решении кода и применять свой опыт в написании кода к этому решению.

У меня страшная аллергия на всякого рода эзотерические решения, шаблоны проектирования, доступные немногим. Питер Норвиг, работая в Harlequin, сказал о том, что шаблоны проектирования – всего лишь дефекты в вашем языке. Возьмите язык получше! И он был абсолютно прав. Что это такое – молиться на шаблоны, постоянно думая, какой бы из них применить!

Сейбел: Итак, постоянное обогащение опыта позволяет лучше выбирать направление. Но что если, создавая код, вы видите крупные дефекты в проектном решении?

Аик: Так бывает и нередко. Иногда очень сложно отказаться от кода и вернуться, чтобы начать все заново, – попадаешь в ловушку обязательств. У меня было такое с JavaScript. Я написал интерпретатор байт-кода в страшной спешке, и делая это, я уже понимал, что кое о чем пожалею. Но то было решение, понятное для других, и я надеялся, что другие помогут мне с этой программой. Поэтому о проектном решении я думаю всегда – не каждый раз мы можем позволить себе роскошь пересматривать фундаментальные основы кода. А именно это случается при масштабном переписывании.

Сейбел: Каким образом вы решаете, что необходимо масштабное переписывание? Благодаря Джоэлу Спольски Netscape в некотором смысле стала примером того, как опасно масштабное переписывание.

Айтк: В Netscape хотели, чтобы приобретенная ими компания, потрясавшая известной книгой о шаблонах проектирования, вывела их на первое место благодаря новому движку рендеринга, который стал бы для всех ориентиром. Сверху это смотрелось хорошо: там использовались C++ и шаблоны проектирования. Но было множество проблем.

А вот вторая причина того, что мы взялись за переписывание: я трудился в mozilla.org и был сильно недоволен Netscape, как и Джейми, – тот вообще собирался уходить. Я считал, что нужно пустить на наше поле новых работников, а со старым запутанным кодом, сделанным на коленке в 1994 году, сделать это было нельзя. И с моим прекрасным кодом интерпретатора в стиле ядра UNIX.

Нам нужна была большая перезагрузка. Четыре года с момента выпуска программы! Не говоря ничего такого топ-менеджерам, поскольку они и слушать об этом не желали, мы стали искать оптимальное решение за них. В итоге полетело несколько топ-менеджерских голов. Правда, менеджеры, в отличие от меня, все равно сказочно заработали на опционах. Но для Mozilla это было выгодно.

Сегодня можно назвать это удачей, поскольку развитие Сети ускорилось. Видимо, Microsoft – некоторые утверждают, что это было связано скорее с антимонополистскими расследованиями, чем с желанием его руководителей, – хотела плотно оседлать Сеть и не допустить ее эволюции. Это дало нам возможность заняться переписыванием, размахивая знаменем стандартов (довольно сомнительный ход, особенно с учетом качества стандартов). Как и Джоэл, я скептически смотрю на переписывание. Трудно примирить все интересы, найти деньги и при этом правильно отреагировать на требования рынка. Исключений единицы.

Те случаи переписывания, о которых я говорил раньше, связаны с прототипами. Это крайне важно, а объем работы намного меньше. Можно порезать кучу кода, не очень много по числу строк, но с большими последствиями, и удовлетворить всем нужным инвариантам. Или это компиляция «на лету», или еще что-то, что позволяет решить задачу.

Сейбел: Вы занимались литературным программированием в духе Кнута?

Айтк: Я делал кое-что наподобие его первоначальных программ – просто классно, мне очень нравилось. Это было извлечение слов. Там было некое подобие древовидного хеша, программирование было целиком литературным. Потом Дуг Макилрой сделал все то же самое, только с конвейером.

Наши программы подробно откомментированы, но нет средства изъять из них прозу и проверить ее – хотя бы вручную – относительно кода. Разработчики Python сделали в этом смысле кое-что интересное. Все что я сделал – не более чем подробное комментирование. Я периодически обновляю старые комментарии, но это тяжело, и иногда мне это не удается, и я жалею, что кто-то из-за меня получил неверную информацию.

Сейчас мне нравятся возражения Макилроя. Это не то что полное опровержение литературного программирования, но близко к тому. Не хочется писать много – неважно, прозы или кода. В каком-то смысле, на низшем уровне код должен говорить сам за себя. На более высоких уровнях – гигантские функции, границы модулей – уже нужна документация. Документирующие комментарии, или документарные строки. Встраивание тестов в комментарии. Думаю, разработчики Python сделали тем самым большое дело.

Кое-что, как видно, пришло из литературного программирования – документарные строки, встроенные тесты. Хотелось бы, чтобы языки поддерживали больше таких вещей. Мы пытались включить встроенную документацию в ES4 через поддержку метаданных первого класса или интроспекций, но не смогли договориться между собой.

Сейбел: Вы читаете чужой код?

Аик: Это часть моей работы. Ревизия кода – обязательный шаг перед коммитом, который был когда-то необходим в основном из-за плохой кадровой политики Netscape, но мы до сих пор пользуемся им, а также делаем интеграционные ревизии. Мы устраиваем также особые «суперревизии», когда изменяется много модулей и вы не знаете всех скрытых инвариантов, которые Джо Шмо¹, который больше не работает в Mozilla, держал в своей голове. В принципе, есть люди, способные охватить взглядом целостную картину. Иногда мы обходимся без этого, когда все хорошо знают, что делают, и понимают друг друга без слов, как джедаи. Но лучше поступать так не слишком часто.

У нас нет предварительных ревизий проектных решений. Поэтому иногда такие ревизии случаются потом сами. «У тебя слишком много кода. Вернись-ка назад и сделай по-другому». Но так бывает редко. Мы не навязываем «модель водопада», жестко последовательную схему разработки. Когда я занялся программированием в начале 1980-х, она была как раз в моде, просто кошмар. Вы пишете документацию, потом код, потом понимаете, что код не годится, вы его полностью меняете – и вся документация насмарку.

¹ Джо Шмо – аналог Васи Пупкина. – *Прим. ред.*

Сейбел: Вы говорите о коде, который пишется для Mozilla. А вы читали когда-нибудь код не по работе, а ради самообразования?

Айк: В этом смысле открытый исходный код – потрясающая вещь. Я люблю смотреть код, который пишут люди с разных концов света, хотя посвящаю этому не так много времени. Больше всего меня привлекают серверные фреймворки, а еще такие языки, как Python и Ruby.

Сейбел: То есть их реализации?

Айк: Реализации и код библиотек. Возьмем библиотеки Ajax: приятно видеть, насколько умны могут быть люди и как с небольшим набором инструментов – замыкания, прототипы, объекты – можно создавать удобные, порой очень удобные абстракции. Нельзя сказать, что они всегда крепко скроены или безопасны, – но до чего удобны!

Сейбел: Как вы поступаете, если надо прочесть большой кусок кода?

Айк: Я начинаю «сверху». Если фрагмент достаточно велик, у вас есть указатели функций, а течение исполнения программы не слишком ясно. Иногда я пропускаю его через отладчик и терзаю таким вот способом. Я также смотрю на шаблоны низкого уровня, которые могу распознать. Если это языковой процессор или в нем есть понятные мне системные вызовы, я пытаюсь выяснить, как используются эти примитивы. Как они используются на более высоких уровнях? Это позволяет мне немного освоиться с кодом. Но по-настоящему его понимаешь, когда создается целостный образ, когда смотришь на код сверху, снизу, с разных сторон, возясь с отладчиком, пропуская через него код шаг за шагом, как бы это ни раздражало.

Если удастся посмотреть, что происходит в куче – проследить указатели, пройти по списочным ячейкам, что-то еще, – осознаешь, что это стоило труда, как бы тошно ни казалось. Для меня это так же важно, как чтение исходника. Исходник можно читать долго, можно застрять в нем, смертельно скучая и понапрасну уверяя себя, что понимаешь вон то темное место.

Создавая регулярные выражения в JavaScript, я обращался к Perl 4. Я пропускал его через отладчик и читал код. Это давало мне идеи: реализация в обоих языках оказалась схожей. Их рекурсивный характер с поиском с возвратом был немного необычен, так что пришлось поднапрячься. Удалось только отладить некоторые регулярные выражения и проследить за исполнением. Другие программисты, как я слышал, тоже говорили об этом: нелегко пробираться сквозь код, понимать, как динамическое состояние системы выглядит при беглом взгляде или при проверке базовой функциональности. Я согласен с ними.

Сейбел: Вы делаете то же самое с собственным кодом, даже если не ищите ошибку?

Айк: Разумеется, я делаю проверку на вменяемость. И делаю много операторов-утверждений, и если они срабатывают, я тут же оказываюсь в отладчике. Но иногда пишешь код, применяя разные хитрые вспомогательные схемы. Тестируешь его, он работает... пока не пропустишь его через отладчик. Особенно если в нем есть особо заумный кусок, срабатывающий, только если звезды сойдутся определенным образом. Используешь условные точки прерывания, точки прерывания по изменению данных, наконец, ловишь его, когда этот код срабатывает, – ага, звезды сошлись! И понимаешь, что живешь не на планете сказок. Если, пребывая в исходном коде, вы чувствуете себя обитателем этой планеты, беритесь за отладчик. Это важно, я так всегда поступаю.

Сейбел: Вы обнаруживаете проблему, когда пробираетесь сквозь исходник, держа в уме, что должно произойти вот это и вот это, но ничего не происходит?

Айк: Да, или – это моя личная проблема – я живу на планете сказок. Сейчас я постарел, стал скептичнее, работаю лучше, но все еще настроен слишком оптимистично. В каком-то уголке моего сознания Говорящий сверчок шепчет: «Да у тебя там, должно быть, ошибка, ведь ты наверняка о чем-то забыл». Со мной так всегда.

Иногда я точно знаю это – знаю, что где-то ошибся, – и покрываюсь потом. У меня свербит не то в заднем мозгу (где это, кстати?), не то в микротрубочках. Так или иначе, я чувствую, что должен быть начеку, и отладчик помогает мне быть начеку. Он помогает мне принять решение или понять, что вот этот тест-вектор не покрывает всех комбинаций кода, так как передо мной обширное гиперпространство.

Сейбел: Кроме чтения кода многие программисты читают еще и книги по специальности. Есть ли книги, которые вы можете порекомендовать?

Айк: Мне надо было бы читать побольше. Но это как в музыке: главное – практика. Можно узнать очень многое, читая чужой код. Мне очень нравились книги Брайана Кернигана, написанные предельно логично – написать небольшой фрагмент кода, использовать его заново по мере продвижения, строить модули. Еще «Искусство программирования» Кнута, тома 1–3, особенно та часть, где говорится о полученных алгоритмах. Двойное хеширование – я люблю эти главы. Лемма о золотом сечении, которую требуется доказать в качестве упражнения.

Но, по-моему, по книгам не слишком-то научишься программировать. Программирование отчасти близко инженерному искусству: в один прекрасный день может понадобиться математика. И есть куча практических вещей, которые даже не поднимаются на уровень инженерного

искусства, как оно понимается в строительстве и технике. Возможно, со временем все это чуть больше формализуется.

Нет, конечно, есть определенный объем знаний: компьютерная наука – все же наука. Помню, лет двадцать назад был спор в Usenet, и кто-то заметил: «Недонаука, показатель строгости – одна треть». Есть куча мимолетных брошюрок, которые явно не выдержат проверку временем. Журнальные статьи лучше, так как на них обязательно требуется отзыв, да и контроль там строже. Области автоматизированных доказательств – это впечатляет. Но эти статьи все еще не доходят до программистов. Поэтому в компьютерных науках кое-чего, пожалуй, не хватает, и я из-за этого недоверчиво отношусь к чтению книг. Видимо, все же не надо настолько уподобляться луддитам.

Есть наука, есть важные вещи, которые надо усвоить. На это можно потратить много времени. Я знаю чистых теоретиков в отделе развития JavaScript. Многие из них хорошие программисты. Некоторые не программируют вообще. Они мало знакомы с практикой. У них случаются любопытные и порой полезные озарения, но писать программы, выпускать их, делать работоспособными, чтобы они завоевали свою долю рынка, – все это далеко от теории. Но я интересуюсь теорией, она делает нашу жизнь лучше.

Сейбел: Но ведь есть и другие книги – те, которые учат практике программирования, не нагружая вас теорией.

Аик: Такие книги мне нравятся. Мы говорили о труде Кнута, посвященном литературному программированию. Там был раздел о практике программирования – он мне пришелся по душе. Люблю книги по Smalltalk. Сейчас мне кажется, что они оказали большое влияние – книга Адели Голдберг, а до нее журнал «Byte».

Сейбел: С воздушным шаром на обложке?

Аик: Да. Она сильно перевернула мои взгляды. 1981 год или около того. Тогда я мало занимался программированием. Я думал о нем, читал о нем, терзал старое железо, учась на последнем курсе. Чистота среды в Smalltalk, то, в какой степени она поддерживала сама себя, – все это заставило меня обратиться к программированию, к языкам и виртуальным машинам. Работая с UNIX, я имел дело с физическими машинами и операционными системами, но не переставал читать. Была такая книга издательства Springer-Verlag – сборник статей, в котором народ в основном фантазировал насчет универсального формата объектных файлов и байт-кода Java еще до его появления. Но Smalltalk был мощной штукой. Я освоил Smalltalk позже, в Иллинойском университете, когда его приспособили для работы на компьютерах Sun того времени. Работало все медленно.

Сейбел: Сменим тему. Как вы распознаете талантливого программиста?

Айк: Недавно мы наняли такого – он был другом одного из самых могучих умов в нашей компании. То ли студент последнего курса, то ли бакалавр – кажется, еще не закончил колледж. Он познакомился с парнем, который работал у нас, – оба они ОСaml-программисты. Он размышлял о проблемах, которые были поставлены в моем статическом анализе. По его ответам на вопросы не скажешь, что он еще мальчик. Кое-кто из наших говорил: «Да что он такого великого сделал? Мы берем только звезд, зачем тратить на него время?»

Я отвечал: «Нет, ребята, вы не правы. Надо брать специалистов, пока они молоды. Он сделал много чего для себя на ОСaml; он знает не только исходный язык, но и рабочую среду, он разобрался с системными методами и написал на ОСaml операционную систему – пусть пока игрушечную. Но это стоящий парень». Я даже не дал ему никакого теста – просто слушал его рассказ о том, что и почему он сделал. Он не переживал эту вечную жвачку насчет шаблонов C++. У нас, к сожалению, есть и такие мальчики. Милые люди, в чем-то неплохие программисты, могут работать на Java. Но нам нужен был кто-то необычный, а этот парень был необычным.

Поэтому главной проблемой стало убедить людей, что его возраст и отсутствие большой практики ничего не значат. Но мы взяли его, и он стал суперзвездой. Он придумал инструменты для статического анализа; сперва создавал их на платформе Berkeley Oink с открытым кодом, а потом на GCC в качестве плагинов вместе с разработчиками GCC. Сейчас он активно взялся за наши программы для мобильных: делает элементарное профилирование, выводя отметки времени через `printf`, ищет, где можно снизить издержки.

Разговаривая с ним, я понял, что он талантлив. Хорошо было и то, что его рекомендовал блестящий программист, – такие люди тянутся друг к другу и обычно не рекомендуют посредственностей. Они хотят работать с такими же блестящими программистами. Может показаться, что я выдумываю, но для меня это и правда один из способов распознать талант. Наверное, именно поэтому мы берем к себе суперпрограммистов. Думаю, все люди из Valgrind оказываются у нас. Некоторые из них могут все и совсем не выделяются.

Сейбел: Итак, это один из способов проводить собеседование: дать человеку поговорить о собственных проектах.

Айк: Да. Я никому не даю головоломок. Некоторые из наших, правда, дают. И это меня тревожит, поскольку влияет на отсев кандидатов.

Сейбел: Можно ли сразу выявить нужного человека?

Айк: Сомневаюсь. В Google дают поиграть в пики, так что у них есть сколько-то народа, отлично решающего головоломки. Но они не всегда обладают здравым смыслом и способны на зрелые суждения. Так что я сомневаюсь. Пожалуй, в какой-то мере это необходимо, ведь человек с хорошо подвешенным языком необязательно силен в программировании. Надо посмотреть, как он принимает решения прямо на месте, без обдумывания. Мы даем людям чисто практические задачи. Ни головоломки, ни продвинутой математики – задачи по программированию.

Проверяйте знание C++, поскольку C++ – сложная штука. Это что-то вроде базовой проверки, вовсе не решающий аргумент. Прошел человек такую проверку – хорошо, нет – тревожный сигнал. Для приема на работу нужно что-то еще. Нужны подробности: что человек сделал, каков его подход к программированию, с какими языками он работал.

Кроме того, мне, видимо, нравятся необычные люди. Я не имею в виду всякие странности. Я не хочу нанимать человека, с которым трудно работать, – нам нужны таланты. Нужны те, кто мыслит нестандартно.

На последнем курсе меня сильно впечатлила книга Пирсига «Дзен и искусство ухода за мотоциклом». Еще я много читал Платона и других древних философов. В философском плане я склонялся к идеализму. Полагал, что обратный порядок байтов лучше прямого, так как байты меньшего порядка располагаются по меньшим адресам – в этом есть какая-то гармония, геометрическая правильность. Но попробуйте прочесть шестнадцатеричный дамп! Важны практические вещи, важны подробности. Есть известная фреска «Афинская школа», на которой Аристотель указывает вниз, а Платон вверх. Так вот теперь я на стороне Аристотеля. С возрастом я становлюсь все скептичнее и все больше интересуюсь тем, что реально работает.

Когда я опрашиваю потенциальных кандидатов, то мне очень сложно не увязнуть в мелочах, в практических вопросах. Этот парень знает ОСам? Хорошо. Но стоит ли брать его на работу? Только из-за этого – нет. Но он еще делал что-то для себя, умеет решать задачи с ходу, думает о проблемах компиляции или анализа. Значит, возьмем. Но, возможно, главным доводом здесь была рекомендация его друга, блестящего программиста.

Сейбел: Программирование все еще доставляет вам удовольствие?

Айк: Да. Это как привычка. Тут есть некая загадка. Меня привлекает не задача создать работающий код, а скорее поиск верной идеи в духе соотношения 90/10, как в нью-джерсической философии. Вы создаете хорошее теоретическое ядро, которое не решит всех ваших проблем, но если вы попадаете на оставшиеся 10%, то ничего страшного. На этом пути можно добиться успеха: код остается сравнительно небольшим

и несложным, и есть некая игра в переходе от теории к реализации. Вот это мне по-прежнему нравится, все так же волнует меня. Я обдумываю такие проблемы по ночам и не могу заснуть.

Сейбел: А есть то, что со временем вам стало нравиться меньше?

Айк: Ну, не знаю... Наверное, C++. Мы научились пользоваться всеми его свойствами, которых слишком много. Система типизации в нем, пожалуй, лучше, чем в Java. Но мы все еще применяем отладчики и компиляторы 1970-х – полный идиотизм! Не понимаю, как мы до сих пор их терпим.

Нетерпение и неприязнь к примитивным инструментам – вот из-за чего я старался совершенствоваться в программировании. Наш код сегодня испещрен проверками утверждений, и они фатальны. Но именно это помогает мне, особенно когда я применяю к коду тот самый принцип 90/10, не удовлетворяющий всем инвариантам. Я что-то упустил, утверждение срабатывает, и вдруг – раз! – я понимаю, как его исправить.

Даже сейчас я часто убеждаюсь в собственных недостатках, когда слишком усердно оптимизирую что-нибудь. Рисуя себе радужную картину, я забываю о какой-нибудь важной проблеме. Это всегда испытание для меня, ведь программист должен быть оптимистом. Считается, что мы параноики и невротики, вечно озабоченные чем-то персонажи Вуди Аллена, но на самом деле параноику нечего делать в программировании.

Сейбел: Как вам кажется, программирование – это удел молодых?

Айк: Думаю, у молодых огромные преимущества, прежде всего психологические. Им не хватает лишь мудрости! Становишься старше, работаешь медленнее – но ты усвоил горькие уроки и хочешь передать свой опыт следующему поколению. Я вижу, как они отворачиваются от меня и сами усваивают эти уроки, – и сжимаю кулаки.

Но человек знающий, который старается быть в курсе всего, не обязательно должен выдавать много кода. Нет, конечно, объем продукции тоже важен. Но что первостепенно для меня – и об этом мы много беседовали в Netscape, когда там искали главного инженера, – это найти человека, не менеджера, а того, кто своим авторитетом заставит других программистов писать код в нужном духе. Ведь один программист просто не справится со всеми задачами.

Вот эта возможность воздействовать на людей, когда они перенимают твой подход и твой опыт, так что в результате получается код, непосильный для одного, так же важна для меня, как возможность сидеть ночами и писать в одиночку собственный длинный код.

Я по-прежнему работаю слишком много, а теперь у меня еще и маленькие дети. Моя жена – славная девушка, но ей не очень нравится, когда

я засиживаюсь за работой. Ведь я занимаюсь не только программированием, но и вот этими, более важными вещами. В случае с JavaScript нам надо было развивать язык. Это требовало не то что проповеднического пыла, но умения заставить людей думать о последствиях эволюции языка, о том, каким они его видят. И надо было разбираться с кучей разных ответов.

Не все программисты способны на это – многие склонны работать, забившись в свой угол. Но работая в Netscape, я понял, что мне нравится взаимодействовать с людьми, которые пользуются моим кодом. Если я забьюсь в свой угол, мне будет не хватать этого. А я хочу постоянно решать такие задачи. Да, я могу выстроить для себя прекрасный воздушный замок, но будет ли он удобен для других? Вряд ли. В чем же тогда смысл? Как говорил Гиллель-старший, «кто я без других»?

Я не связываю себя только с JavaScript. Поначалу мы страшно спешили, делали массу ошибок, и как-то Джейми переслал мне пост из Usenet со словами: «Говорят, ваш детище – урод». Теперь у меня настоящие дети, и те дела больше меня не волнуют.

5

Джошуа Блох

Сегодня Джошуа Блох – главный Java-архитектор в Google, до этого работал в Sun Microsystems, где был удостоен звания Заслуженный инженер и руководил созданием и реализацией Java Collections Framework, появившегося в Java 2, а также внес некоторые дополнения в язык для версии Java 5. Получил степень бакалавра в Колумбийском университете, PhD – в Университете Карнеги–Меллона, где работал над Camelot – распределенной системой обработки транзакций, позднее получившей название Encina; она была выпущена компанией Transarc, где Блох являлся старшим системным проектировщиком. Написал книгу «Effective Java»¹, удостоенную Премии Джолта за 2001 год, и еще две в соавторстве – «Java Puzzlers» (Java: головоломки) и «Java Concurrency in Practice» (Java: параллельность на практике).

Как и следовало ожидать от того, чья работа состоит в пропаганде использования Java в Google, Блох – активный сторонник этого языка. Несмотря на сегодняшний всплеск интереса к таким инструментам параллельной обработки данных, как Software Transactional Memory или механизм обмена сообщениями в языке Erlang, Блох считает, что в Java реализован «наилучший, сравнительно с другими языками», подход к параллельным процессам, и предсказывает возрождение интереса к Java по мере того, как программисты будут переходить на многоядерные процессоры.

¹ Джошуа Блох «Java. Эффективное программирование». – Лори, 2002.

Блох также защищает отношение к программированию как к проектированию API. Мы беседовали о том, как это помогает ему самому конструировать программы, о том, что Java становится слишком громоздким языком, и о том, что выбирать язык программирования – почти то же, что выбирать бар.

Сейбел: Как получилось, что вы занялись программированием?

Блох: Так и хочется сказать, что это у меня в крови. Мой отец работал химиком в Брукхейвенской национальной лаборатории. Когда я учился в четвертом классе, он пошел на курсы программирования. Компьютеры, понятно, тогда были большими ЭВМ за стеклом, и надо было отдавать пачку перфокарт оператору. Пощупать их было нельзя, но сама мысль о том, что вот есть такие вычислительные машины, делающие для тебя разные штуки, меня поразила. Поэтому я чуть-чуть поднабрался от отца Фортрана, пока он учился на курсах.

Сейбел: Это какой же год?

Блох: Кажется, 1971. Но по-настоящему я загорелся этим лишь через несколько лет – благодаря системе разделения времени. На Лонг-Айленде был компьютер DECsystem-10, которым пользовались все школы округа Саффолк. Другой такой же предназначался для округа Нассау. Сколько известных теперь людей начинали на одной из тех двух машин!

Как только появляется интерактивность, человек сразу загорается. Я программировал на Бейсике, как и все в 1973–1976 годах. Вот тогда уже все было серьезно. Любопытно, что у меня от того времени сохранились программы на телетайпной бумаге Teletype – телетайпы все еще живы! – и глядя на них, я понимаю, что мой стиль был отчасти заложен уже тогда.

Сейбел: Можете ли вы вспомнить свою первую по-настоящему интересную программу?

Блох: 4 июля 1977 года я написал свой вариант известной игры 20Q (Двадцать вопросов) и назвал его «Животные». Там было бинарное дерево с вопросами типа «да/нет» на внутренних узлах и животными на листьях. Когда программа встречала незнакомое животное, она «заучивала» его название, задавая пользователю вопросы, предполагающие ответ «да» или «нет». Так она училась отличать новое животное от того, название которого определила неверно. Бинарное дерево хранилось на диске, так что программа со временем становилась «умнее».

Помнится, подумалось: «Черт, да она же учится!» Это было что-то вроде прозрения. Еще помню, как классе в десятом работал на той самой

DECsystem-10. Нам не позволялось писать то, что сейчас называют мгновенными сообщениями, – слишком много системных ресурсов они отнимали.

Сейбел: Как и сейчас.

Блох: Давайте не будем об этом. Мгновенные сообщения – моя погибель. Хотя нет, моя погибель – электронная почта, сообщения – пустяки. Так или иначе, я был непослушным мальчиком, поэтому включился в проект для Лонг-Айлендской математической ярмарки, который я называл «программами межпрофессиональной коммуникации».

Сейбел: Вы писали эти программы?

Блох: Да, кроме одной, которую писал мой приятель Томас Де Беллис. Интересно, что программа Тома была написана целиком на Бейсике. Она была строчно-ориентированной и использовала для связи файлы. Она не была особенно быстрой или эффективной – но работала! Я написал две – одну строчно-ориентированную, другую с посимвольной записью – на MACRO-10, языке ассемблера для PDP-10. Для связи там использовалось подобие разделяемой памяти, названное «старшим сегментом» (high segment).

Тогда я ничего не знал о параллельном программировании, почти не понимал мьютексы. Но там были буферы сообщений и независимые агенты, которые пытались общаться друг с другом параллельно. Поэтому там имелись состояния гонки, и порой программа теряла символ другой. Старшекласснику постичь все это было почти невозможно.

Сейбел: Вы сказали, что некоторые элементы вашего стиля проявились уже в первых программах. Какие?

Блох: Стремление сделать программу читаемой. Как говорит Кнут, программа – это прежде всего литературное произведение. Я уже тогда каким-то образом понял, что программа должна быть читаемой. И я сохранил этот подход.

Сейбел: А что изменилось?

Блох: Трудно сделать программу читаемой, когда можешь давать переменным имена длиной только в один символ. Так что сейчас я больше забочусь об именах переменных. Да и вообще, когда берешься за новый язык с новыми свойствами, многое меняется. То, что смутно понимал, укладывается в голове.

Например, правило «не повторяться». Раньше я куда свободнее копировал-вставлял. А сейчас вообще стараюсь не применять этот прием. Превеличиваю, конечно, но самую малость. А вообще, как только обнаруживаю, что занимаюсь копированием-вставкой, сразу думаю: «Что не так в моей архитектуре? Как это исправить?» Исправление требует

некоторого времени. Я стал строже к себе, и это помогает мне писать качественные программы. Сам себя плохому не научишь.

Сейбел: Если бы вам предложили вернуться в прошлое и начать все сначала, хотели бы вы что-нибудь всерьез изменить? У вас в голове ничего не повредилось от Бейсика? Или от чего-то еще?

Блох: На самом деле, это довольно занятная вещь. Дейкстра, царство ему небесное, по-моему, был здесь в корне неправ. Многие действительно классные программисты начинали с Бейсика, потому что им был доступен только он.

И все же, как я считаю, полезно пользоваться разными языками. В колледже я писал программы сразу на нескольких. На разных занятиях применялись разные языки. На занятиях по математике и естественным наукам – Фортран, на занятиях по программированию – Паскаль, SAIL, Симула или что-то в этом духе. А на занятиях по искусственному интеллекту мы работали на Лиспе.

Может быть, стоило освоить еще больше языков. Интересно, что всерьез заниматься объектно-ориентированным программированием (ООП) я стал довольно поздно. Java был первым объектно-ориентированным языком, на котором я работал по-настоящему, отчасти потому, что я так и не смог заставить себя работать на C++.

Сейбел: Когда это было?

Блох: Это началось в 1996 году, когда я пришел в Sun. Думаю, мне стоило познакомиться с этими концепциями чуть раньше. Однако я не считаю, что все они хороши. ООП – занятная штука. Это прежде всего две вещи. Первое – модульное построение, что очень удобно. Но оно было еще до ООП. Возьмите старую литературу, например работы Парнаса по скрытию данных, и вы увидите, что понятие о том или ином типе класса как об абстракции существовало до ООП. Второе – наследование свойств, но у меня, как и у многих сегодня, отношение к нему смешанное.

Кроме того, мне надо было попробовать себя в разных областях помимо компьютерных наук. Чем больше различных вещей вы узнаете в молодости, тем лучше для вас. Я очень мало занимался графическими интерфейсами пользователя – надо было заставить себя засесть за них. Мне отчего-то интереснее всего было заниматься библиотеками, готовить модули для других. Так что я десятилетиями занимался структурами данных и алгоритмами.

Сейбел: Есть ли книги, которые должен прочесть каждый программист?

Блох: Разумеется, «Design Patterns» (Шаблоны проектирования), хотя я отношусь к ней не совсем однозначно. Она дает общий для всех словарь. И в ней масса хороших идей. С другой стороны, мешанина стилей

и языков; кроме того, этот труд частично уже устарел. Но прочесть его стоит, я твердо убежден.

Затем «Elements of Style» (Элементы стиля). Хотя она, собственно, не о программировании, прочесть ее нужно по двум причинам. Во-первых, потому, что разработчик ПО тратит значительную часть времени на написание прозы. Если вы неспособны создавать логичные, стройные, хорошо читаемые спецификации, вашими программами никто не сможет пользоваться. Поэтому все, что способствует улучшению стиля изложения, есть благо. И во-вторых, большинство высказанных там мыслей применимы и к программам.

Если же говорить о том, что бы я взял на необитаемый остров, то это будет несколько странный выбор. Например, для меня страшно важна «Hacker's Delight»¹.

Сейбел: Это же книга про перестановку битов?

Блок: Да. Люблю это дело, и оно напрямую связано с моей работой. Для тех, кто пишет библиотеки, компиляторы, криптомодули, создает низкоуровневую графику, она обязательна к прочтению. Уоррен собрал всю, так сказать, устную традицию и дал ей достойную строгую математическую обработку. Я был сам не свой, когда эта книга вышла.

Еще, конечно, «The Art of Computer Programming»² Кнута. На самом деле я не читал ее полностью – даже близко к этому не подходил. Но когда я работаю над тем или иным алгоритмом, то смотрю, что сказал по этому поводу Кнут. И часто нахожу то, что мне нужно, – там есть все.

Но у меня нет ни способностей, ни времени, чтобы прочесть ее от и до, так что на этот счет врать не буду. Очень важна, по-моему, старая книга «The Elements of Programming Style»³. Все примеры даны на Фортране IV и PL/1, так что она несколько устарела. Но учитывая возраст этой книги, удивительно, что высказанные там идеи все еще в силе.

Из старого назову также «Мифический человек-месяц»⁴ Фредерика Брукса. Ей уже сорок лет, но она актуальна, как будто вышла вчера. А читать ее – одно удовольствие. Это должен сделать каждый. «Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше», – вот ее главная идея, и она ничуть не устарела. Но

¹ Генри Уоррен мл. «Алгоритмические трюки для программистов». – Вильямс, 2007.

² Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008.

³ Керниган Б., Плджер Ф. «Элементы стиля программирования». – М.: Радио и связь, 1984.

⁴ Ф. Брукс «Мифический человек-месяц или Как создаются программные системы». – СПб.: Символ-Плюс, 2000.

там есть и масса других важных вещей. Кое-что уже устаревает, но все равно – читать обязательно.

А сегодня непременно нужно читать литературу о параллельном программировании. Поэтому стоит взять в руки «Java Concurrency in Practice» (Java: параллельность на практике). Несмотря на заголовок, многое в ней применимо не только к Java.

Сейбел: Вы ее написали вместе с Брайаном Гетцем?

Блох: Мое имя стоит на обложке, но я так спокойно ее рекомендую, потому что на самом деле я тут ни при чем. В основном ее писали Брайан с Тимом Пайрлзом и все прочие – программисты, работающие над JSR-166, спецификацией Java, касающейся параллельных процессов. Правда, эти остальные упомянуты скорее из вежливости – мы поставляли материал, но ничего не писали.

Да, и еще 11-е издание словаря английского языка «Merriam-Webster». Без него никуда. Его не то чтобы нужно *читать*, но при написании программ вы должны давать переменным правильные имена. И, конечно, стиль должен быть хорошим. Без приличного словаря я просто теряюсь.

Сейбел: Про имена переменных и про уменьшение копирования-вставки все понятно. А что еще изменилось в вашем подходе к программированию с накоплением опыта?

Блох: С возрастом я понял, что надо не просто сделать работающую вещь – программа должна хорошо читаться, быть легкой в обслуживании и эффективной. В противоположность распространенному мнению, я считаю, что чем яснее и изящнее программа, тем быстрее она работает. А если нет, то заставить ее работать быстрее легко. Как говорится, легче оптимизировать правильный код, чем исправлять оптимизированный код.

Отчасти смена подхода касается конкретных свойств разных языков. Каждый язык снабжен набором инструментов. Для каждой конкретной работы нужно брать правильный инструмент – но правильный инструмент в одном языке может оказаться неправильным в другом. Банальный пример: если вы пишете на Java 5, использование `enum` вместо `int` или булевых выражений сильно упростит вашу программу, сделает ее безопаснее и прочнее.

Сейбел: С учетом этого, что вы можете сказать о приобретении беглости в пользовании новым языком?

Блох: Думаю, это во многом как с обычными языками. Один подход состоит в том, чтобы знать много языков: если вы освоили итальянский и испанский, а теперь желаете заняться португальским, вам будет не

слишком трудно. Чем больше вы знаете, тем больше опираетесь на ваши знания.

Взявшись за новый язык, используйте все накопленные знания, но держите ум незашоренным. Я знаю тех, кто раз и навсегда решил, что все программы должны писаться таким-то способом. Не буду называть конкретные языки, но некоторые языки по тем или иным причинам способны толкнуть на этот путь. И если такой программист сталкивается с новым языком, то критикует его и может сказать, что, мол, в раю им не пользовались, – что бы это ни значило. И взявшись писать на новом языке, старается писать на первоначальном райском языке – насколько это возможно с новым языком. И тогда особенности языка теряются.

Представьте, что из всех инструментов вы владеете только молотком, и вдруг кто-то дает вам отвертку. И вы говорите: «Ну, молоток так себе, но попробую взять за жало и забивать гвозди ручкой». И вот у вас никудышный молоток, который на самом деле отличная отвертка. То есть нужна непредубежденность и готовность пользоваться всеми своими знаниями. И, конечно, писать код! И еще раз: писать код! Чем больше вы применяете язык на практике, тем скорее его выучите.

Сейбел: Откуда этот фанатизм у приверженцев того или иного компьютерного языка?

Блок: Не знаю. Но выбирая язык, человек выбирает не только определенные параметры – он выбирает сообщество пользователей. Это все равно что выбирать бар. Вы хотите пойти туда, где наливают что-нибудь приличное, но это не главное. Главное – кто там собирается и о чем они говорят. Так же и с компьютерными языками. Со временем вокруг каждого образуется сообщество – оно включает не только людей, но и ПО: инструменты, библиотеки и так далее. Вот почему зачастую языки, которые на бумаге выглядят лучше остальных, проигрывают – вокруг них не сложилось правильное сообщество.

Сейбел: Java поражает меня в том смысле, что Java-сообществ целых два. Есть реализаторы и системные программисты, которые работают в Java-soft, Weblogic и тому подобных местах. И есть те, кто использует Java, серверы приложений, готовые фреймворки для бизнес-приложений. Это два очень разных бара.

Блок: Вокруг Java, как и вокруг других языков, сложилось множество сообществ. Если же сообщества нет, это обычно означает, что перед нами нишевый или сырой язык. По мере развития языка и роста числа пользователей его сообщество становится более разнообразным по составу. И, кроме того, чем больше вкладывают денег в язык, тем ценнее он становится.

Это как закон Меткалфа: полезность сети пропорциональна квадрату численности пользователей этой сети. То же и с языками: люди пользовались каким-то языком, и вдруг появляются Eclipse, FindBugs, Guice. Даже если Java не идеальный язык для вас, пользуясь им, вы имеете такие вот попутные выгоды. Можно создать собственное сообщество по численному программированию на Java, по какому угодно виду программирования.

Сейбел: Программирование приносит вам такое же удовольствие, как в школьные годы?

Блох: Да, хотя и другого рода. Думаю, как и для многих школьников, программирование было для меня убежищем от неподвластных мне сторон жизни. И, кроме того, в молодости энергии полно, можно ковыряться в программах часами напролет.

С возрастом появляются семья, дети, все такое, появляются новые обязанности, надо заниматься новыми важными вещами. Но остается этот необычайный подъем, когда пишешь программу, видишь, как все встает на свое место, и наконец получается несколько прекрасных строчек кода, читаемого, быстродействующего, делающего то, что ты хочешь.

Сейбел: А случалось так, что по мене накопления опыта вы понимали: просто заставить программу работать недостаточно, есть и другие соображения? Вас это не обескураживало?

Блох: Конечно, такое бывает. И с книгами то же самое – трудно сесть за них. Мне вообще трудно начать работать, я стараюсь от этого уклониться. Начать – самое сложное, будь то программа, книга или что-то еще. Правда, иногда я себя подбадриваю: «Ну давай, Джош, ты занимаешься этим тридцать лет и не хуже других знаешь, как это делается. Так что вперед». И еще я напоминаю себе, что в прошлый раз, когда я садился за это, все получилось – значит, должно получиться и теперь.

Сейбел: Итак, имея за плечами некоторый опыт, порой бывает сложнее взяться за работу. Скажите, а есть ли что-нибудь вне программирования, какой-то жизненный опыт, который помог улучшить ваши программистские навыки?

Блох: Конечно. Думаю, здесь помогает все, что делаешь, если делать это хорошо. Идеи приходят откуда угодно. Вот пример: в диссертации я делал анализ одной распределенной структуры данных – реплицируемой разреженной памяти. Основную идею я взял из курса химии, который прослушал. То было уравнение между состоянием равновесия и скоростью реакции: если в системе имеется динамическое равновесие, то можно составить уравнение вида «Элементы приходят в определенное состояние с такой же скоростью, с какой выходят из него». Я вывел сразу три уравнения для трех переменных, решил их и получил результа-

ты, которые в точности отражали наблюдаемое поведение той довольно сложной распределенной структуры данных. Идея взята из химии и применена в компьютерной науке.

Многое из того, что мы наблюдаем в жизни – методы постройки зданий, языковые явления, – может быть применено в других областях. И конечно, математика. Математика и программирование чертовски близки. Поэтому держите глаза открытыми и будьте готовы применять найденные идеи в других местах.

Сейбел: Знакомы ли вам выдающиеся программисты, которые не очень любят математику или плохо ее знают? Нужны ли сейчас программисту математический анализ, дискретная математика и тому подобное? Или все зависит от склада ума, которым можно обладать даже без соответствующего образования?

Блох: Полагаю, так думают те, кто этого всего не изучал. Но знание математики, само собой, помогает в работе. Я работал с парнем по имени Майк Макклоски. Он мыслил математически, не зная теорию чисел. Он переписывал BigInteger. Раньше она была надстройкой над Си-библиотекой, но Майк переписал ее на Java так, что потери в скорости не было. Недавно он завершил свою работу, в процессе освоив теорию чисел. Без математического склада ума он этого не сделал бы, но выучи он эту теорию до того, ему не пришлось бы осваивать ее в ходе работы.

Сейбел: Но он решал сугубо математическую задачу.

Блох: Да, пожалуй, не слишком удачный пример. Но даже при решении задач, прямо не имеющих отношения к математике, математическое мышление все равно нужно программисту. Например, доказательство по индукции так тесно связано с рекурсивным программированием, что одно без другого не понять. Вы можете не знать такие термины, как «база индукции» и «индукционный переход», но должны понимать их суть для написания хороших рекурсивных программ. Так что даже если задача программиста не связана с математикой, без знания математических понятий ему придется туго.

Матанализ, мне кажется, не так важен. С годами произошло кое-что любопытное. Обычно считалось, что если вы образованный человек и закончили колледж, то должны знать матанализ. И он содержит массу прекрасных идей – хорошо, когда понимаешь, как обращаться с понятием бесконечности.

Но есть дискретный и непрерывный способ осознать понятие бесконечности. И я считаю, что для программиста важнее овладеть дискретным. Я только что упоминал индуктивное доказательство. Можно доказать то, что будет верным для всех целых чисел. Просто волшебство! Доказываешь что-то для одного числа, потом доказываешь, что одно

число влечет за собой другое, – и вот доказательство верно для всех целых чисел. Думаю, это важнее для программиста, чем, скажем, иметь понятие о пределах.

К счастью, нам не нужно выбирать. Можно освоить и то и другое. Даже если вы не собираетесь использовать матанализ так активно, как дискретную математику, все равно знать их нужно. Но дискретный подход все равно полезнее непрерывного.

Сейбел: Вы говорили о том, что у программирования много общего с написанием прозы. Обычно с компьютерами и программированием всегда была тесно связана именно математика. Но если говорить о веб-фреймворках или веб-приложениях на их основе, требуют ли они скорее писательских навыков?

Блох: Да. Вы говорили о двух несхожих между собой сообществах Java-программистов. Для тех, кто создает библиотеки, компиляторы, фреймворки, намного важнее математика. А для создания веб-приложений на базе фреймворков требуются навыки словесного и визуального общения. Я прихожу в бешенство, когда веб-сайт заставляет меня что-то делать неправильно. Ясно, что человек не подумал о том, как с его сайтом будут взаимодействовать пользователи. Истина в том, что программирование находится в точке пересечения многих дисциплин. Смотря по тому, что вам знакомо лучше, вы достигнете успеха в создании тех или иных приложений. Но библиотеки, компиляторы и фреймворки также должны быть читаемыми и легкими в поддержке. И если у вас неважно с написанием текстов, вам будет нелегко добиться этого.

Сейбел: Каков ваш подход к проектированию программ? Что вы делаете – запускаете Emacs, начинаете писать код, вертите его по-всякому, пока он не примет нужный вид? Или садитесь на диван со стопкой бумаги?

Блох: Несколько лет назад на конференции по объектно-ориентированному программированию я делал доклад «Как создать качественный API и почему это важно». Несколько его вариантов есть в Сети. Там я подробно объясняю свой подход.

Главное – понимать, что именно вы строите, какую проблему решаете. Важность анализа требований трудно переоценить. Некоторые думают, что это просто – идешь к клиенту, тот говорит, что ему нужно, и готово.

Ничто не может быть дальше от истины. Это не только переговоры – это также процесс понимания. Некоторые клиенты излагают вам не задачу, а свое решение. Например, клиент говорит: «Мне нужна поддержка для 17 атрибутов этой системы». И вы начинаете расспрашивать, что он собирается делать с системой, какой видит ее и так далее. Вы какое-

то время мечетесь туда-сюда, пока, наконец, не осознаете реальную потребность клиента. Это сценарии использования.

На этом этапе самое важное – иметь обширный набор сценариев использования. А от них уже можно отталкиваться в своих поисках решения. Нужно тщательно, не жалея времени, обдумать его, потому что если решение неправильное, все ваши дальнейшие усилия пойдут прахом.

Хуже всего – а я сталкивался с такими случаями – это когда вы сажаете в офисе команду смысленных парней, которые через полгода выдают вам 247-страничную спецификацию, толком не понимая, что они разрабатывают. Через полгода это будет программа с подробной спецификацией – программа, которая может оказаться бесполезной. Часто можно услышать: «Мы столько потратили на эту спецификацию, что должны ее реализовать». В итоге получается бесполезная система, которая никому не нужна. Вот что ужасно. Если нет сценариев использования, вы создаете нечто, а потом пытаетесь написать что-то очень простое и тут говорите себе: «Черт, ведь чтобы распечатать XML-документ, нужно много страниц стандартного кода». Это ужасно.

Поэтому берите сценарии использования и создавайте набросок API – совсем небольшой. Обычно он помещается на одной странице. Предельной точности тут не требуется. Нужны описания пакетов, классов и методов, а если не совсем понятно, что они должны делать, напишите об этом – по одной строке для каждого вида элементов. Но это не та вылизанная документация, которую вы потом будете распространять.

На этой стадии нужна гибкость: придайте интерфейсу форму ровно настолько, чтобы вы могли реализовать сценарии использования на этом новорожденном API и понять, соответствует ли он задаче. Любопытно: по прошествии времени все кажется простым, но при создании API обычно ошибаешься, даже держа в уме сценарии использования. При написании кода для сценариев вы замечаете, что у вас слишком много классов, какие-то нужно объединить, какие-то выкинуть. К счастью, ваш интерфейс умещается на странице, и его легко поправить.

Чем больше вы полагаетесь на свой API, тем больше вы к нему добавляете. Но основное правило вот какое: сначала напишите код, использующий API, а потом уже код его реализации. Иначе вы можете потратить впустую время, написав код, который не будет использоваться. На самом деле код, использующий API, надо писать даже до подробной разработки спецификации – иначе можно потратить время на детальную спецификацию чего-то в корне неработоспособного. Вот такой у меня подход к проектированию.

Сейбел: Какие здесь особенности у Java-коллекций, представляющих собой особый вид автономных API?

Блох: Могу сказать, что это не настолько специфично, как можно подумать. Программирование любой сложности требует проектирования API, поскольку большие программы строятся по модульному принципу, и надо конструировать межмодульные интерфейсы.

Хорошие программисты стараются делать вещи, работающие автономно, по нескольким причинам. Первая состоит в том, что вы, возможно неосознанно, создаете модули, пригодные для повторного использования. Если строить монолитную систему, а затем, когда она разрастется, разбивать ее на части, то у вас не будет четких границ, и вы получите кучу мусора, который невозможно поддерживать. Поэтому то, о чем я говорю, есть просто самый разумный подход к программированию, и неважно, ощущаете вы себя проектировщиком API или нет.

Надо, правда, учитывать, что программирование – обширная сфера. Если вы пишете на HTML и только, то это не лучший образ действий, но для многих видов программирования – действительно лучший.

Сейбел: Итак, вы стоите за модули, которые сцепляются между собой не слишком тесно. Сегодня на этот счет есть две точки зрения. Сторонники первой призывают, как и вы, проектировать межмодульные API в самом начале работы. Защитники другой заявляют: «Делайте простейшую вещь из всех возможных» и «Беспоощадный рефакторинг!».

Блох: Мне кажется, они не исключают друг друга. В каком-то смысле я говорю о разработке через тестирование и рефакторинг применительно к API. Как тестировать API? До начала реализации пишутся сценарии использования. Я не могу запустить их, но это разработка через тестирование. Я тестирую качество API, реализуя в коде сценарии использования, чтобы понять, насколько мой API отвечает поставленной задаче.

Сейбел: Значит, вы пишете клиентский код, использующий API, потом смотрите на него и спрашиваете себя: «Это в самом деле тот код, который мне нужен?»

Блох: Именно так. Иногда даже не доходит до того, чтобы взглянуть на клиентский код. При попытке начать его писать происходит одно из двух: или ты не можешь его написать, так как чего-то не хватает в API, или можешь его написать, но понимаешь, что ошибся в подходе.

Неважно, насколько хороший вы программист, – вам не создать нормальный API, пока вы не начали писать код для него. Вы проектируете что-то, пытаетесь это использовать и замечаете: что-то здесь совсем не так. Если же сделать это вначале, вы не потратите время впустую на создание всех лежащих ниже слоев – это большой плюс. Я говорю, как видите, о разработке через тестирование и о рефакторинге API, а не о рефакторинге кода реализации ниже слоя API.

Насчет простейшей вещи из всех возможных – я «за» обеими руками. Основная аксиома проектирования API такова: «Сомневаешься – выкидывай». Простейшая вещь должна быть достаточно велика, чтобы соответствовать всем намеченным сценариям использования. Это вовсе не означает, что надо склеивать воедино сырые фрагменты кода. На этот счет есть куча афоризмов. Мой любимый – тот, что ошибочно приписывают Телониусу Монку: «Сделать просто – непросто».

Сырые программы никому не нужны. «Делайте простейшую вещь из всех возможных» и «Беспощадный рефакторинг!» – эти два призыва совсем не означают, что надо писать сырой код, отказавшись от предварительного проектирования. Я беседовал об этом с Мартином Фаулером. Он твердо стоит за продумывание целей и задач, чтобы придать программе разумные размеры и структуру. «Не пишите 247-страничную спецификацию до того, как начнете писать код», – говорит он, и я согласен.

Мы с ним расходимся по одному вопросу: я не считаю, что тесты могут хоть в какой-то мере заменить документацию. Если вы работаете над тем, для чего другие должны писать код, вам нужны четкие спецификации, и тесты должны подтвердить, что код им соответствует.

Итак, между двумя лагерями есть разногласия. Но, по-моему, непреодолимой пропасти, как считает кое-кто, нет.

Сейбел: Раз уж вы упомянули Фаулера, написавшего несколько книг по UML, скажите: вы используете UML в качестве средства проектирования?

Блох: Нет. Думаю, это здорово – создавать понятные для других графические схемы. Но, если честно, не могу припомнить, какие там компоненты круглые, а какие квадратные.

Сейбел: Вы занимались всерьез литературным программированием в духе Кнута?

Блох: Нет. Я не против самой идеи, просто не представлялось случая. И потом, как бы поаккуратнее выразиться, я стараюсь не подпадать под влияние ни одной религии, будь то объектно-ориентированное или функциональное программирование, христианство или иудаизм. Я заимствую из них хорошие идеи, но не практикую их. В литературном программировании есть много отличных идей, но это не тот бар, что мне нужен: в нем зависает не так много программистов.

Зато я могу часами спокойно возиться с именами идентификаторов, переменных, методов и так далее, чтобы мой код был читаемым. Если выражение, содержащее эти идентификаторы, похоже на обычное английское предложение, ваш код, скорее всего, будет правильным и более легким в поддержке. Думаю, у тех, кто заявляет: «Не стоит труда,

это же всего-навсего имя переменной», – ничего не получится. С таким подходом не написать удобной в сопровождении программы.

Сейбел: Одно из отличий программы от литературного произведения – если не говорить об экспериментальной литературе – состоит в том, что не существует одного-единственного порядка чтения программы. Как вы читаете большие чужие программы?

Блох: Хороший вопрос. На самом деле я люблю хорошо написанные программы. Я знаю людей, способных взять большую неважно написанную программу и зарыться в код, пока не станет вырисовываться общая картина. Завидую такой способности – у меня ее никогда не было.

Я хочу иметь возможность брать небольшие модули, читать их, понимать по отдельности. Если же части программы тесно связаны между собой и надо читать ее целиком, чтобы понять отдельные элементы, – это просто кошмар. В этом случае мне надо заставить себя даже просто *попытаться* сделать это, и надо иметь доступ одновременно ко всему коду. Я распечатываю все, сажусь на пол, раскладываю вокруг листы распечатки и делаю на них пометки.

Если я читаю хорошо написанный код, то стараюсь взглянуть на него с высоты птичьего полета: кто-нибудь где-нибудь должен был оставить описание программы в целом. Если я нахожу такое описание, то знаю, где искать важнейшие модули. Я знакомлюсь сначала с ними, при необходимости погружаясь в более низкоуровневые модули для лучшего понимания.

Еще одно: хотя сам код линейный, его исполнение может быть нелинейным. Если мне повезло и фрагмент кода может быть прочитан насквозь – здорово. Если нет, мне нужно иметь доступ к инструментам, позволяющим быстро найти методы, которые вызываются, классы, которые расширяются, и так далее. Это позволяет мне проследить основные пути выполнения кода.

Сейбел: Вы применяли пошаговое исполнение кода, чтобы его понять?

Блох: Конечно! Это до сих пор мой любимый способ отладки, особенно для параллельного кода: система может находиться одновременно в стольких состояниях, что их невозможно перечислить. Я просто смотрю на код, мысленно прохожу его, думаю, какие инварианты в какое время должны соблюдаться. В нашем распоряжении есть много затейливых отладочных инструментов, но ни один не сравнится по своей силе с простым прогоном кода – при помощи отладчика или чтения с исполнением кода в уме. Я обнаружил таким способом множество ошибок и делаю это и при написании кода.

Когда я пишу программу, то спрашиваю себя: что вот здесь должно быть истинным? Очень важно перенести эти утверждения в код, чтобы

сохранить их на будущее. Если язык позволяет сделать это при помощи конструкций утверждения, воспользуйтесь ими, если нет – поместите утверждения в комментарии. В любом случае это ценная информация и утрачивать ее нельзя. Это позволит вам оценить программу в полугодовой перспективе, а вашим коллегам – оценить ее в принципе.

Сейбел: Вы чувствуете, как люди понимают инварианты и как использовать утверждения, когда это нужно?

Блох: Нет. Вы, вероятно, знаете, что утверждения (assertions) – первый элемент, помещенный мною в Java, и я сознаю, что они так и не стали частью Java-культуры. Лишь немногие Java-программисты пользуются ими – даже не знаю, почему. Кстати, о математике: инварианты являются в высшей степени математической идеей.

Сейбел: Но для их понимания не нужно знать математику на «отлично».

Блох: Не нужно. Но позвольте мне побыть адвокатом дьявола. Математика дает определенную четкость мышления. Я готовил к математической олимпиаде школьников четвертого и пятого классов. В этом возрасте некоторые дети уже понимают суть доказательства, что предположение должно быть явно и безапелляционно истинным, а не думают: «По-моему, это верно, раз есть примеры того, как оно работает».

Чтобы воспринять понятие инварианта, нужно сначала воспринять понятие доказательства. К сожалению, оно недоступно даже многим взрослым. Этот тип мышления обычно прививается в математических классах.

Сейбел: Чувствую, вы готовы сказать, что заняться программированием – лучший способ воспитать у себя этот стиль мышления. Вы бы преподавали программирование как науку об инвариантах...

Блох: В какой-то мере я согласен, но так можно зайти слишком далеко. Вернемся к Дейкстре. Уверен, вы читали его книгу «On the Cruelty of Really Teaching Computing Science» (О жестокости реального преподавания компьютерных наук), и полагаю, что в ней он абсолютно неправ. Дейкстра говорит, что студентов нужно подпускать к компьютеру лишь после того, как те в течение семестра научатся обращаться с символами и понимать их подлинный смысл. Но это же бред! Ведь это удовольствие – приказать компьютеру сделать что-то и наблюдать, как он это делает. Я не в силах лишить студентов такого удовольствия. Да и не в состоянии – ведь компьютеры повсюду. Десятилетние дети пишут программы.

Сейбел: Как человек, пропагандирующий Java в Google, не находите ли вы, что этот язык мог бы использоваться более широко? Оставим в стороне поступь истории и уже сделанный людьми выбор и представим,

что у вас есть волшебная палочка. Если бы могли заменить весь C++ на Java, это бы сработало?

Блох: До известной степени. Крупные программные блоки могут быть изменены таким образом, и все движется в этом направлении. Но если взять само ядро системы, к примеру внутренние циклы индексных серверов, то в нем даже крошечное улучшение показателей значит страшно много. Когда множество машин используют один фрагмент кода, увеличение скорости даже на пару процентов принесет серьезные выгоды для ваших финансов и для окружающей среды. Поэтому часть кода пишется на ассемблере, а что такое Си, как не ассемблер под другим названием?

Я по характеру не фанатик. Если это работает – прекрасно. Я двадцать лет писал код на Си. Но с точки зрения сбережения времени программистов эффективнее использовать более современный язык, который будет безопаснее, удобнее и выразительнее. Обычно время программиста куда ценнее компьютерного времени. Но это не обязательно так, если одна и та же программа запускается на тысячах машин. И есть программы, где целесообразно использовать менее безопасные языки, дающие нам большую скорость. Сегодня, как мне кажется, в смысле экономии компьютерного времени нет особой разницы, на каком языке писать ту или иную программу. Если кто-то говорит, что его язык эффективнее в десять раз, то это, скорее всего, неправда.

Но в смысле экономии времени программистов разница есть. Прежде всего, более современные языки свободны от многих типов ошибок. Во-вторых, в них есть прекрасные наборы инструментов, позволяющие сделать работу программиста более эффективной. Все это отчасти обусловлено тем, что именно эти языки люди учат в школе, – но также и их базовыми инженерными характеристиками. Например, если в языке есть макрогенератор, писать для него хорошие утилиты намного сложнее. Парсинг C++ куда более непростое дело, чем парсинг Java.

В Google сейчас немалая часть кода пишется на Java – гораздо больше, чем раньше. Точных данных назвать не могу, но, думаю, мы уже едва ли не перегнули палку. И есть большой разрыв между тем, сколько строк кода у нас написано на различных языках, и тем, сколько процессорных циклов выполняется на том или ином языке. Было бы большой глупостью, по-моему, писать внутренние циклы индексирующих серверов на Java. Если вы, скажем, создаете компанию, то можете писать коды на Java или на другом современном языке с хорошими параметрами безопасности, а потом отказаться от него, если нужно. Но что касается Java, тут есть вся необходимая инфраструктура – библиотеки, средства контроля и так далее. В случае их применения Java будет если

не идеальным, то вполне надежным партнером. Когда я только пришел в Google, это было не так.

Компании очень рано выстраивают свою ДНК. Это может принести им громадный успех, но потом очень трудно отказаться от той архитектуры, когда она перестает отвечать потребностям. Помню, когда я был начинающим специалистом в исследовательском центре IBM, в Йорктаун-Хайтсе. Это было году в 1982-м, и там всю применялась пакетная обработка данных. Даже применяя разделение времени, они мыслили в терминах виртуальных считывателей карт и виртуальных перфаторов. Везде записи в 80 колонок! В DEC так и не смогли мыслить иначе как в терминах разделения времени. Ну, а что касается Microsoft, то еще вопрос, смогут ли они держать в уме что-то иное, помимо настольного персонального компьютера.

Сейбел: А через двадцать лет скажут, что компания Google навсегда ушиблена онлайн-рекламой.

Блох: Конечно. В Google так или иначе господствует стереотип, что Java – язык медленный и ненадежный. И понятно, почему: версия Blackdown Java, созданная для Linux около 1999 года, была медленной и ненадежной. Старые предрассудки очень живучи. Но по правде говоря, Google использует Java в очень важных с деловой точки зрения случаях, к примеру для рекламы.

Так что в определенной степени там не считают этот язык медленным и ненадежным. Но основной поиск в Google, занимающий наибольшее число машинных циклов, основан на C++, и ясно, что это имеет историческое обоснование. Это будет продолжаться еще какое-то время.

Сейбел: Какими инструментами вы сегодня пользуетесь для программирования?

Блох: Ждал этого вопроса. Да, я старпер, и гордиться тут нечем. Команды Emacs навсегда врезались в мой мозг. Я стараюсь писать небольшие программы, библиотеки и так далее. Как видите, я в основном обхожусь без современных инструментов, хотя и знаю, что с ними работа идет быстрее.

Для больших программ я использую IntelliJ, так как вся моя группа работает с ней, но у меня выходит неважно. Да, она производит впечатление – мне нравится то, как эти утилиты делают за вас статический анализ. Кое-кого из поклонников таких инструментов, как IntelliJ, Eclipse, NetBeans и FindBugs, я привлекал для выверки текста книги «Java Puzzlers». Многие ляпы были найдены автоматически с помощью этих программ. Просто здорово.

Сейбел: Стали бы вы работать продуктивнее, если бы потратили месяц для подробного изучения IntelliJ?

Блох: Да. Современные интегрированные среды разработки отлично справляются с масштабным рефакторингом. Брайан Гетц заметил, что сегодня программисты пишут более качественный код, потому что раньше не могли делать такой рефакторинг, как сейчас. Они полагаются на эти инструменты, чтобы вносить сквозные изменения, не затрагивающие работу кода.

Сейбел: Как насчет прочих инструментов?

Блох: С утилитами для программирования у меня не очень хорошо – а жаль. Инструменты сборки и системы управления версиями изменяются сильнее, чем хотелось бы, и мне трудно следить за ними. Поэтому, сталкиваясь с новой средой, я всегда пристаю к коллегам, более привычным к таким инструментам. Я вечно спрашиваю: «А как сегодня это делается?» Коллеги закатывают глаза и помогают мне, а я пользуюсь средой, пока она окончательно не откажет.

Гордиться тут нечем. Разработчики программ могут быть искусны в одном и малоискусны в другом. Кое-кто утверждает, что это не так, что разработчики взаимозаменяемы, что каждый может и должен уметь все. Но на практике это не так. И если заставлять каждого разработчика делать все, результат будет никудышным.

Я говорю прежде всего о тех, кто, по выражению Кевина Бурильона, «лишен гена эмпатии». Нельзя создавать хорошие API или языки, не представив себя в шкуре рядового программиста, который пользуется ими. Однако есть люди, создающие хорошие API и языки. И есть знатоки технической стороны проектирования языка, которые говорят: «Это делает все несовместимым с LALR(1), надо сделать по-другому». Это крайне полезное знание. Но оно не заменяет гена эмпатии – такой знаток может создать кошмарный язык, не пригодный для использования.

Есть и другие – способные выжать из языка все, что возможно, ради большей эффективности. Надо найти им нужное применение – они будут счастливы и принесут пользу вашей компании. Вообще, необходимо знать сильные места ваших разработчиков и пользоваться этим. Это я так оправдываюсь за свое плохое знание инструментов. Слабое оправдание, понятно.

Сейбел: Поговорим об отладке. Можете ли вы назвать худшую ошибку из тех, что вам встречались?

Блох: Мне сразу приходит в голову один кошмарный и в то же время любопытный случай. Это было в начале 1990-х, я тогда работал в питтсбургской компании Transarc. Мне пришлось заниматься реализацией транзакционной разделяемой памяти при очень плотном графике. Проектирование и реализацию я закончил в срок и даже успел написать

несколько библиотечных компонентов. Но я нервничал из-за того, что произвел много нового кода в спешке.

Для тестирования кода я написал чудовищного «убийцу». Он запускал множество транзакций, каждая из которых содержала рекурсивно вложенные транзакции – вплоть до определенной глубины вложения. Каждая из вложенных транзакций могла блокировать и читать некоторые элементы разделяемого массива в восходящем порядке и что-то прибавлять к каждому из них, сохраняя инвариант, так что сумма всех элементов массива равнялась нулю. Каждая субтранзакция либо фиксировалась, либо прерывалась – соотношение случаев было 90:10, как-то так. Множество потоков запускали эти транзакции параллельно и воздействовали на массив в течение долгого времени. Поскольку я тестировал разделяемую память, то запускал несколько многопоточных «убийц», каждый в своем собственном процессе.

При разумном уровне многопоточности «убийца» работал вполне надежно. Но когда этот уровень повысился, я обнаружил, что иногда – именно иногда – «убийца» не проходил проверку внутренней целостности. Я не понимал, что делается, и, естественно, думал, что это моя ошибка – ведь я написал столько нового кода.

С неделю я потратил на модульные тесты для каждого компонента – все было в порядке. Потом я написал программу проверки целостности для каждой внутренней структуры данных и мог делать проверку после каждого изменения – пока не случилось, что элемент не проходил проверку. Наконец я уловил непрохождение проверки на низком уровне – такое было не каждый раз, но теперь я мог проанализировать происхождение. И пришел к неизбежному выводу: мои блокировки не работали. У меня были параллельные последовательности операций типа «прочитать–изменить–записать», так что две транзакции блокировали, читали и записывали одно и то же значение. И последняя запись затирала первую.

Я написал собственный диспетчер блокировок, поэтому стал подозревать его. Но ведь он без проблем прошел модульные тесты! Наконец я определил, что виноват был не он, а реализация мьютексов в нижележащем слое. Тогда операционные системы еще не поддерживали многопоточность, и пакет для ее поддержки нам пришлось писать самим. Вышло так, что разработчик, отвечавший за код мьютексов, случайно перепутал метки подпрограмм «заблокировать» и «попробовать заблокировать» в ассемблерной реализации потоков в Solaris. Так что каждый раз, когда вы думали, что вызываете безусловную блокировку, на самом деле она только пыталась произойти, и наоборот. И когда случался конфликт – в то время редкость, – второй поток оказывался

в критической секции, как если бы в первом потоке не было блокировки. Самое забавное, что вся компания на несколько недель оказалась без мьютексов, и никто не заметил.

В своей превосходной статье «Engineering a Sort Function» (Разработка функции Sort) Бентли и Макилрой цитируют чудесное высказывание Кнута насчет приведения себя в самое поганое настроение, на которое только вы способны. Как раз это я и сделал для той серии тестов. Но это сделало ошибку крайне трудно обнаружимой. Прежде всего, из-за многопоточности каждый случай оказывался почти невозпроизводимым. Далее, оказались ложными мои представления не о чем-нибудь, а о ядре системы. Обычно начинающие программисты легко приходят к выводу, что язык или система не в порядке. Но тут базовая конструкция, на которую я опирался, – мьютекс – действительно оказалась сложенной.

Сейбел: Итак, ошибка содержалась не в вашем коде, но вы тем временем написали столь подробные тесты для кода, что ошибку волей-неволей пришлось искать вне его. Как по-вашему, мог ли – или должен ли был – автор мьютексов написать тесты для нахождения этой ошибки, которые избавили бы вас от полутора недель отладки?

Блох: Мне кажется, хорошая автоматическая программа проверки мьютексов спасла бы меня от мучений, но не забудем, что это было в начале 1990-х. Мне и в голову не приходило винить разработчика за то, что он не создал достаточно хороших модульных тестов. Даже сегодня писать модульные тесты для многопоточных программ – подлинное искусство.

Сейбел: Мы говорили о пошаговом прохождении кода. А какими средствами отладки вы пользуетесь сейчас?

Блох: Наверное, я кажусь неандертальцем, но важнейшие инструменты для меня, как и раньше, – мои глаза и мозг. Я распечатываю все необходимые фрагменты кода и очень внимательно их изучаю.

Отладчики – хорошее средство, и порой мне хочется пользоваться оператором `print`, но вместо этого я прибегаю к точке останова. Время от времени я применяю отладчики, но и без них чувствую себя вполне уверенно. Имея возможность использовать операторы `print` и внимательно читать код, я вполне могу находить ошибки.

Я уже говорил, что пользуюсь операторами утверждения для проверки сохранности сложных инвариантов. Если инварианты ломаются, я хочу знать, когда это случилось, какие действия привели к этому.

Кстати, я вспомнил еще одну труднонаходимую ошибку. Правда, не могу сказать точно, было это в Transarc или на последнем курсе Уни-

верситета Карнеги–Меллона, когда я работал над системой распределенных транзакций Camelot. Не я нашел эту ошибку, но сам случай меня глубоко поразил.

У нас был трассировочный пакет, позволявший коду выводить отладочную информацию. Каждое отслеженное событие снабжалось меткой с указанием идентификатора потока, где оно произошло. Иногда идентификаторы оказывались неверными, и мы не понимали, почему. Наконец, мы решили, что с этой ошибкой можно еще пожить сколько-то времени, – она казалась безобидной.

Но выяснилось, что ошибка не в трассировочном пакете – все было гораздо серьезнее. Чтобы найти идентификатор потока, трассировочный пакет вызывал код из потоковой библиотеки. А тот делал штуку, очень в то время распространенную: смотрел старшие биты адреса стековой переменной. То есть он брал значение указателя стековой переменной и сдвигал его вправо на фиксированное число позиций, получая таким образом идентификатор потока. Дело в том, что у каждого потока был стек определенного размера, который выражался заранее известной степенью двойки.

Выглядит логично, так? Но, к сожалению, те, кто создавал объекты в стеке, делали их слишком большими по тогдашним меркам. Массив из 100 элементов, по 4 Кбайт каждый, – всего 400 Кбайт в стеке одного потока. Получался перескок через красную зону стека в стек соседнего потока. И мы получали неверный идентификатор потока. Хуже того: когда поток обращался к локальным для потока переменным, он считывал переменные другого потока, поскольку его идентификатор использовался как ключ для доступа к этим переменным.

Итак, то, что мы приняли за безобидный недочет трассировочного пакета, оказалось признаком действительно серьезной ошибки. Событие приписывалось потоку 43 вместо потока 42, так как один поток невольно подменял собой другой, и это могло иметь катастрофические последствия.

Вот почему нам нужны языки с хорошими параметрами безопасности. Лучше обойтись без таких случаев. Недавно у меня был разговор в одном университете: там хотели обучать программистов сначала языкам Си и C++, а потом Java, так как они хотели, чтобы программисты овладели системой «на всю глубину». Меня спросили, что я думаю об этом.

Думаю, посыл здесь правильный, но выводы ошибочные. Да, студентам нужно изучать низкоуровневые языки, и даже язык ассемблера, и даже устройство чипов. Правда, чипы сейчас превратились в невероятно сложных чудовищ и теряют в производительности именно из-за

своей сложности. Но знание того, что происходит на низших уровнях системы, сильно облегчает высокоуровневое программирование.

И я считаю, что все это важно изучать. Но это не значит, что надо начинать с такого низкоуровневого языка, как Си! Зачем студентам, только-только приступающим к программированию, сталкиваться с переполнением буфера, ручным выделением памяти и тому подобным?

Мы с Джеймсом Гослингом однажды обсуждали появление Java, и он сказал: «Время от времени нужно нажимать кнопку перезагрузки. Это едва ли не самое прекрасное, что может случиться». Обычно вам приходится поддерживать совместимость со старыми программами, но иногда – нет, и это здорово. Но к сожалению, как это случилось с Java, проходит десятилетие – и ваша система сама становится проблемой для других.

Сейбел: Значит ли это, что язык Java уже немного устарел и что он быстро усложняется, но при этом совершенствуется куда медленнее?

Блох: Очень непростой вопрос. Например, Java 5 вышел намного более сложным, чем мы хотели. Я даже не представлял, насколько обобщенные типы и особенно символы подстановки¹ усложняют язык. Надо отдать должное Грэму Гамильтону – он понял все это в свое время, а я нет.

Интересно, что он годами боролся за невключение обобщенных типов в язык. Но понятие вариативности – которая и лежит в основе символов подстановки – вошло в моду в то время, когда мы старались не снабжать Java обобщенными типами. Если бы они появились раньше и без всякой вариативности, мы бы теперь имели более простой и легкий в работе язык.

При всем том от символов подстановки есть реальная польза. Есть глубокая несовместимость между методом выделения подтипов и обобщенными типами, и символы подстановки позволяют во многом ее нивелировать. Но это достигается ценой переусложнения. Некоторые считают наилучшим решением вариативность на стороне объявления, в противоположность таковой на стороне использования, но я не сильно уверен.

Нельзя твердо судить о чем-то, если это не было использовано многими программистами в реальной рабочей обстановке. Есть языки, хорошо работающие в своей узкой области, и некоторые говорят о них: «Отличный язык, жаль, что им пользуется так мало народа». Иногда, однако, для этого есть веские причины. Надеюсь, какой-нибудь язык, где ис-

¹ Обобщенные типы и символы подстановки часто упоминаются как «generics» и «wildcards». – *Прим. науч. ред.*

пользуется вариативность при объявлении, к примеру Scala или C# 4.0, ответит на этот вопрос раз и навсегда.

Сейбел: Что же дало импульс к появлению обобщенных типов?

Блок: Как часто бывает с идеями, которые на практике оказываются хуже, чем в теории, мы верили собственным заявлениям для прессы. Я представлял себе это так: почти все коллекции у нас однородны – список строк, хеш строк на целые числа и так далее. Но по умолчанию они создаются разнородными – все это коллекции объектов, которые надо приводить к нужным типам при выборке, – абсурд! Не лучше ли указать системе, что вот это, например, хеш строк на целые числа? Пусть она сделает приведение типов за меня, а во время компиляции укажет мне, если я допущу ошибку. Больше ошибок будет отслежено, система будет иметь больше высокоуровневой информации, а это хорошо.

Обобщенные типы, как и многое из того, что мы добавили в Java 5, казались мне средством автоматизации того, что раньше делалось вручную: пусть этим займется язык! Кое-где я попал в точку: цикл `for-each` – отличная штука. Он скрывает от вас сложное устройство итератора или индексных переменных. Код становится короче, но площадь концептуальной поверхности при этом не увеличивается. Даже скорее уменьшается: мы ввели ложный полиморфизм массивов и других коллекций, и можно выполнять итерацию над `ArrayList` или над массивом, совершенно не интересуясь, над чем именно она выполняется.

Но главная причина того, почему эта идея не сработала для обобщенных типов, – они стали крупным прибавлением к системе типизации, и без того сложной. С системами типизации нужно обращаться осторожно, поскольку это может повлечь далеко идущие и непредсказуемые последствия для языка.

А урок таков: если вы совершенствуете зрелый язык, нужно больше чем когда-либо задумываться над балансом возможностей и сложности. Сложность во многих разделах языка растет квадратично: прибавив всего одно свойство, вы получаете куда более сложную структуру. Если язык близок к тому, чтобы превысить уровень понимания программистов, усложнять дальше просто нельзя – все пойдет прахом.

Если же все-таки усложнять, исчезнет язык или нет? Нет, не исчезнет. Мне кажется, C++ давно превысил этот уровень, а сколько народу им пользуется! Но тем самым вы побуждаете людей разбивать его на части. И почти в каждой известной мне лавочке, где используют C++, говорят: «Да, мы используем C++, но не применяем ни множественное наследование, ни перегрузку операторов». Есть свойства, которые вы не используете, потому что код тогда получается слишком сложным. Думаю, не стоит и пытаться. Каждый программист должен иметь возможность

читать код любого из своих коллег, а в нашем случае эту возможность легко утратить.

Сейбел: Не кажется ли вам, что Java без обобщенных типов был бы сегодня лучше?

Блох: Не знаю. Обобщенные типы по-прежнему мне нравятся – они находят за меня ошибки в моем коде. Эти средства помогают найти мне вещи, которые обычно включаются в комментарии, и перенести их в код, где компилятор может обеспечить их корректность. С другой стороны, когда я вижу сообщения об ошибке, связанные с параметризованными типами, а потом нахожу сделанные для этих типов обобщенные объявления, вроде моего `Enum – class Enum<E extends Enum<E>>`, то понимаю, что обобщенные типы не были достаточно хорошо проработаны, чтобы их включить.

Программист должен или быть оптимистом, или застрелиться. И мы говорим: «Конечно, мы это умеем. Мы знаем все об обобщенных типах еще с тех пор, как познакомились с языком CLU. Это технология 25-летней давности. То же самое сегодня можно слышать про замыкания, правда, о них говорят, что им уже 50 лет. «Это легко и не усложняет язык».

Черт возьми, конечно усложняет! Но, думаю, обобщенные типы послужили для нас хорошим уроком. Нельзя добавлять что-то к языку, пока не поймешь, как поведет себя концептуальная поверхность, пока не будет веских доводов в пользу того, что программисты смогут эффективно пользоваться новым свойством и оно облегчит им жизнь.

Если бы мы знали, как простые люди отреагируют на обобщенные типы, то, конечно, придумали бы что-нибудь другое. Значит ли это, что эти средства вообще не надо было изобретать? Наверное, все-таки не значит. Думаю, они полезны. Главный аргумент в их пользу – раз большинство коллекций однородны, а не разнородны, работать с однородными коллекциями должно быть легче. Кроме того, приведение типов вообще не очень хорошая штука. Оно не всегда срабатывает и не делает вашу программу красивой. Поэтому, полагаю, должна быть возможность задавать тип коллекции, и он должен проверяться автоматически. Но нужны ли для этого страдания из-за переусложненности средств? Нет. Видимо, нам все же стоило сделать их попроще.

Сейбел: Скажите, а пользователи требовали обобщенных типов? Кто-нибудь жаловался, что их отсутствие мешает писать программы?

Блох: Ну, что касается разработчиков, ответ, увы, отрицательный. Пожалуй, виноват здесь я – эта штука казалась мне красивой, и я думал, что стою на правильном пути.

Но при разработке программ мы часто чуем какие-то вещи нутром. Кто-нибудь просил меня о `foreach`? Опять же нет. Но я знал, что стою на правильном пути, и это оказалось так – многие пользуются этим. Но большой грех для разработчика – создавать программы, которые просто отлично смотрятся, хорошо сделаны и так далее. Если вы не решаете реальные проблемы реальных пользователей – в нашем случае Java-программистов, – то не надо ничего добавлять.

Есть чудесное выступление Гослинга «The Feel of Java» (Почувствовать Java); в нем он говорит, что нужно трижды ощутить необходимость чего-то, прежде чем внедрять это. Нельзя добавлять программу только из-за ее красоты.

Но люди все равно добавляют. Что делают разработчики? Пишут код. И, работая над библиотекой или языком, они хотят добавить туда что-то свое. Некий внутренний голос должен подсказывать, какое сочетание свойств будет работать хорошо, что нужно добавлять, а чего не нужно. Ведь чаще всего вы можете добавить к языку больше, чем должны. Это означает не то, что ваши программы плохи, а лишь то, что надо правильно выбирать, не валя все в кучу.

Сейбел: Я читал книги «Java Puzzlers» и «Java Concurrency in Practice». Меня удивило, что в языке, который изначально был очень простым, столько секретов.

Блок: Секреты есть, но это неизбежно, они есть во всех языках. Можно было бы написать книгу «Си: головоломки».

Сейбел: Ну, этот язык – *сплошная трудность*.

Блок: Да, тут понадобилась бы целая книжная полка. В Java такие случаи особенно нужно отмечать, ведь его считают простым языком. В каждом языке свои проблемы, в Java их не так много, и они по большей части довольно забавны и интересны.

Сейбел: Говоря о программировании, есть ли что-то, чему научила вас работа над Java и обдумывание его структуры?

Блок: Очень многому. Об одном я упоминал в своем посте «Nearly All Binary Searches and Mergesorts Are Broken» (Почти все двоичные поиски и сортировки слиянием сломаны): невероятно трудно правильно написать даже небольшую программу. Мы обманываем сами себя, считая, что наши программы более-менее свободны от ошибок. Это не так. Большей частью наши программы не содержат ошибок лишь настолько, чтобы справляться с возложенной на них задачей.

Я усвоил, что учитывая, насколько трудно писать корректные программы, надо принимать помощь, откуда только возможно. Все, что удаляет потенциальные ошибки, – хорошо. Вот почему я убежденный сторон-

ник статической типизации и статического анализа – они позволяют устранить ошибки определенного типа. Все, что облегчает программисту его задачу, – нужно и полезно.

Я укрепился в своем мнении насчет того, что нужна качественная документация API. Javadoc во многом способствовал успеху платформы, хотя не все это замечают. Качественная документация API всегда была частью Java-культуры, как я считаю, потому что Javadoc присутствовал с самого начала.

Я также утвердился в своем мнении, что чем проще – тем лучше. Я наблюдаю все больше сложных добавлений, которые оказываются только вредными в долгосрочном плане, а иногда и в краткосрочном. Создавая программу, я включаю свой собственный «измеритель сложности»: если стрелка уходит в красную зону, пора переписывать.

Порой кто-нибудь говорит мне: «Джош, глупец, ты просто не улавливаешь, что тут происходит. Тут все именно так, как должно быть, и жаль, что ты этого не понимаешь!» Но я не покупаюсь на такие разговоры. Я считаю, что если программа становится слишком сложной, то с ней что-то не так и надо искать более простые пути.

Тони Хоар как-то на вручении премии Тьюринга блестяще сказал о том, что есть два способа проектировать систему: «Один – сделать ее настолько простой, что в ней *совершенно очевидно* не будет недостатков, второй – сделать ее настолько сложной, что в ней не будет *очевидных* недостатков».

И дальше тоже прекрасно: «Первый способ намного труднее. Он требует умения, увлеченности, озарений не меньше, чем открытие простых физических законов, управляющих сложными природными явлениями. Он также требует настойчивости в достижении целей, когда приходится учитывать физические, логические и технологические ограничения, и идти на компромиссы, когда нельзя удовлетворить взаимно противоречащие требования. Никакой комитет не сможет сделать этого – или сделает слишком поздно».

Сейбел: Как по-вашему, вы будете заниматься Java до пенсии или же перейдете к другому языку?

Блох: Не знаю. Как-то так получилось, что я моментально перешел с Си на Java. После окончания школы и до 1996 года я программировал почти только на Си, а потом – почти только на Java. Конечно, при определенных обстоятельствах я могу перейти на другой язык – но на какой? Может быть, такого языка еще нет в природе. По-моему, мир созрел для нового языка программирования, но инерция платформ сегодня куда сильнее, чем раньше. Современная платформа – это не только язык и несколько библиотек. Это множество инструментов, виртуальная

машина, то есть гигантский комплекс. И перспектива создания новой платформы выглядит сейчас намного более пугающей.

Что будет дальше, я не знаю. Но если действительно потребуется, я все еще могу сменить язык. Я хочу быть открытым для разных возможностей, возиться с другими языками. У меня сейчас для этого нет времени, но я хотел бы его иметь.

Сейбел: Назовите языки, с которыми вам хочется повозиться больше всего.

Блох: Например, Scala, хотя у меня есть сомнения насчет его будущей популярности. Я очень уважаю Мартина Одерски – он реализовал в своем языке немало красивых идей. Но, возможно, этот язык сложноват и слишком академичен, чтобы иметь широкий успех. Честно говоря, я еще не изучил его толком, так что могу быть неправ.

Затем Python. Из старых – Scheme. Будет неплохо несколько месяцев поизучать «Structure and Interpretation of Computer Programs» вместе с сыном. Говорят, это отличная книга. В качестве первого шага я купил ее. Но для освоения нужно время.

Сейбел: Сегодня многие озабочены созданием программ, которые бы в полной мере использовали возможности многоядерных процессоров. Java стал первым крупным языком, в котором появились встроенные механизмы для многопоточной работы. Как вы считаете, приспособлен ли он к многоядерному миру?

Блох: Скажу больше: думаю, у него лучшие средства, чем у любого другого языка. Интересно, что сейчас часто слышатся разговоры о смерти Java. Мне кажется, все это несерьезно. Между прочим, лучшие блоки для реализации многопоточности сейчас есть именно в Java. И язык готов пережить небольшое возрождение. Не знаю, куда мы зайдем в ближайшие двадцать лет и лучшее ли это средство для работы с многоядерностью. Но из того, что есть сегодня, Java на голову выше своих конкурентов.

Сейбел: Что это за конкуренты?

Блох: Как я считаю, C++ и C#.

Сейбел: А как насчет Erlang или транзакционной памяти?

Блох: Насколько я знаю, транзакционной памяти в рабочем виде пока еще нет ни в одном из крупных языков. Если окажется, что это того стоит, то, наверное, она появится и в Java не позже, чем в прочих языках.

У Erlang свой подход к параллелизму – акторы: если они окажутся успешной находкой, то могут быть реализованы во многих языках. Как вы знаете, Одерски с компанией уже реализовали их в Scala. Пока

я не уверен, что акторы – лучшее из придуманного для многоядерного параллелизма, но если все же это так, то и в Java вы скоро увидите их.

Сейбел: Итак, Java, по вашим словам, имеет блоки, позволяющие получить портируемый доступ к параллельным потокам, предоставляемым операционной системой, а также конструкции более высокого уровня в рамках API `java.util.concurrent`. Но все равно, это ведь средства довольно низкого уровня в сравнении с Erlang или транзакционной памятью?

Блох: Не уверен. Некоторые конструктивные блоки в Java действительно низкоуровневые, например `AtomicInteger`; есть среднеуровневые, например `CyclicBarrier`, и наконец высокоуровневые – `ConcurrentHashMap` и `ThreadPoolExecutor`. Уверен, что транзакционная память и акторы найдут свое место в наших конструктивных блоках для многопоточных задач, как только народ убедится, что эти новинки работают хорошо. Если, конечно, они будут работать хорошо.

Некоторые виды транзакционной памяти могут получить значение в будущем, например конструктивные блоки для разработчиков параллельных библиотек. Но мне кажется, транзакционная память не избавит создателей приложений от заботы о блокировках. Она не введет нас в счастливый мир, где потоки не мешают друг другу.

Тому есть несколько причин. Первая из них состоит в том, что если вы пытаетесь делать автоматическую блокировку или оптимистичное управление параллелизмом, основываясь только на чтении и записи на уровне байтов, то между потоками происходит «мнимый конфликт»: физические конфликты не соответствуют логическим. Если вам нужны блокировки, то убедитесь, что захвачены лишь те, которые помогают решить логические конфликты.

Так, например, если у вас есть два потока и оба прибавляют значения для счетчика, они должны выполняться параллельно. Они могут обращаться к одному и тому же участку памяти, но при этом не конфликтуют в логическом плане. Если один поток считывает значение счетчика, а другой увеличивает его, то есть конфликт. Но ведь у вас может быть множество потоков, которые считывают значения, и других, увеличивающих их. Я пока не видел систем, которые могли бы справляться с такими вещами самостоятельно. Может быть, мой пример несколько искусственный, но часто физические ограничения намного суровее логических.

Вторая проблема с транзакционной памятью в том, что внутри нее осуществляются не все операции, например операция ввода/вывода. А вот третья проблема: некоторые виды транзакционной памяти позволяют «обреченным транзакциям» видеть память в неустойчивых состояни-

ях – потенциально это крайне опасно. С этими проблемами мы уже сражались, когда строили транзакционные системы общего назначения. Решения есть, но все они усложняют систему или снижают скорость работы.

Так или иначе, насколько мне известно, транзакционная память еще не вышла из стадии исследований. Прекрасно, что этим занимаются. Но по-моему, она не решит разом все проблемы параллельности – по крайней мере, в обозримом будущем.

Сейбел: Сменим тему. Каков ваш стиль работы в команде?

Блох: Я довольно уживчив, предпочитаю «приятельское программирование» – когда работаешь вместе с кем-то, но не за одной клавиатурой. Вы пишете разные части программы, обмениваетесь кодом. Можно вообще пребывать в разных полушариях. Мы с Дугом Ли таким образом плотно работали несколько лет. Один писал интерфейс, другой говорил: «Все отлично, но я поправил там кое-что, вот погляди».

Наконец получался интерфейс, который нас устраивал. Я реализовывал однопоточную версию, Дуг – многопоточную, во время работы мы обнаруживали разные просчеты и снова поправляли интерфейс. Мы читали код друг друга, Дуг обычно говорил: «Ты можешь сделать вот так – все заработает гораздо быстрее», – а я отвечал: «Дуг, это ты можешь». Он был очень силен во всем, что ускоряло работу системы, – виртуальные машины были для него как друзья. Этот вид программирования я очень люблю, он как бы сам подталкивает к удаленному сотрудничеству.

Мне нравится сидеть с кем-нибудь за одним терминалом и работать над кодом, но таким образом я сделал не много программ с нуля. Обычно это случается, когда я делаю ревизию кода; если вижу, что код надо сильно править, то предлагаю человеку посидеть вместе. Это полезно во многих отношениях, например как средство обучения, способ передать знания старшего поколения младшему.

А работать совсем в одиночку мне не нравится. Когда я пишу программу и обдумываю какое-нибудь сложное проектное решение, мне нужно с кем-то советоваться. Везде, где я работал, рядом был кто-нибудь, с кем я мог поделиться. Для меня это очень важно – обратная связь.

Сейбел: Так что же важнее – обратная связь или просто шанс проговорить задачу вместе?

Блох: И то и другое. Мы делаем очень хитрые вещи – часто есть не одно правильное решение или одно, но которого никто до тебя не нашел. Надо полагаться на свой инстинкт, но иногда полезно выслушать того, кто смотрит на вещи по-другому.

Я знаю тех, кто любит программировать в вакууме, но это, по-моему, им вредит. Надо замечать ошибки как можно раньше, выявлять недостатки структуры, пока они не отразились на коде. И если вы экспериментируете с разными подходами или просто с разными свойствами, то нужно обсуждать их с другими. При этом не стоит слепо верить никому – мнения могут быть разными, а за свою работу отвечаете только вы.

Сейбел: Еще один вечный вопрос – кажется, его еще в 1970-е поднимал Вайнберг в «The Psychology of Computer Programming», – дискутируется и сейчас: должен ли кодом владеть один человек, и только он и должен с ним работать, – или им должны владеть все, кто работал над проектом, и всем должно быть разрешено вмешиваться в него?

Блох: По-моему, владение кодом отрицать нельзя. Это похоже на рождение ребенка: вы даете жизнь коду, и он *ваш*, особенно если он большой, или сложный, или оригинальный. Прежде чем начать работать с чужим кодом, спросите разрешения, особенно если считаете, что нашли ошибки, – ведь вы можете быть неправы. Портить чужой код некрасиво.

Конечно, для компании плохо, если кодом владеет один человек. А если он покинет компанию? Поэтому важно, чтобы сразу несколько программистов знали каждый фрагмент кода и могли с ним работать. Но мечтать о том, чтобы все владели всем кодом, по-моему, не стоит.

Это возвращает нас к теме сфер компетенции. Писать код, перемалывающий биты, способны немногие, и если вы оказываетесь внутри кода, работающего с битами, поговорите с тем, кто умеет обращаться с ним, если сами не умеете. Умелые программисты это любят. Они могут днями работать над тем, чтобы сократить последовательность инструкций на одну или доказать какую-нибудь идентичность, чтобы ускорить вычисления. Но испортить код очень легко. И очень легко написать что-нибудь, что станет хорошо работать, скажем, при $2^{32} - 1$ из 2^{32} потенциальных вариантов ввода. Модульный тест может выявить, что ваше решение не работает с этим одним значением, а может и не выявить. И тогда вы будете виноваты в том, что испортили код.

Сейбел: Если говорить о запутанном коде, я заметил вот что: у слишком умных – в определенном смысле, по крайней мере – программистов получается самый плохой код. Они держат все у себя в голове, и в итоге получается не код, а тарелка спагетти.

Блох: Согласен. Те, кто способен делать сложные вещи и лишен нужной эмпатии в отношении других, часто становятся жертвой такого подхода. «Я понимаю это и могу этим пользоваться, значит, это годится», – вот их логика.

Сейбел: А есть ли в программировании нечто, привлекающее людей именно с таким внутренним складом?

Блох: Конечно! Всяческие головоломки – наша страсть. Но эта страсть должна сдерживаться пониманием того, что мы решаем реальные проблемы реальных людей. Если же это не так, то мы занимаемся самоудовлетворением и все. Думаю, первая компания, в которой я работал, разорилась именно из-за этого. Надо было понять, что наша цель – не разработка программ сама по себе.

Мы не думали о реальных клиентах с их проблемами. Если вы теряете из виду своих клиентов – вам конец. Думаю, это нелегко осознать любителям головоломок, идущим в программисты. Но ведь можно и самому получать удовольствие от своей работы! Пробудите в себе ген эмпатии, проектируя свои API, а потом сколько угодно придумывайте всякие затейливые штуковины для ускорения их работы.

Оптимизируйте алгоритмы и структуры данных – особенно параллельные. Вот настоящие головоломки! Надо думать с математической точностью над сложными вещами, уметь по-новому сочетать примитивы, чтобы достичь своей цели. Но всегда нужно понимать, когда это уместно, а когда даст программу, трудную в использовании или поддержке.

Сейбел: Разве сейчас возможности для такого программирования не сокращаются? Многие из таких вот низкоуровневых программ уже реализованы в вашей виртуальной машине или параллельных библиотеках. И для многих программирование теперь означает склеивание блоков воедино.

Блох: Полностью согласен. Да, в относительных цифрах процент креативных программистов уменьшается. Когда-то вы покупали машину, для которой не было даже операционной системы, не говоря о языке программирования или готовых приложениях. Каждому приходилось что-то выдумывать.

Тот мир ушел или уже уходит. Но в целом потребность в креативных программистах так же велика, как и всегда. Мы хотим получать удовольствие от нашей работы. Мы хотим преимуществ от безопасных языков и одновременно – скорости вручную отлаженного ассемблерного кода. Кто-то ведь должен создавать виртуальные машины и сборщики мусора, придумывать чипы – хотя это «железо», но в то же время программные произведения.

Думаю, для любителей решать сложные задачи есть масса возможностей, но их энергию надо направлять в нужное русло. Над ними должны стоять менеджеры, использующие их способности в интересах компании.

Тут есть одна проблема: часто такие люди умнее всех прочих в компании и поэтому считают, что это они должны принимать все решения. Но только то, что они самые умные, не означает, что им можно доверять принятие решений. Ум – качество не скалярное, а векторное. Без эмпатии или чувственного разума вы не создадите ни API, ни графический интерфейс, ни новый язык.

Мы ставим перед собой также и эстетические цели. В нашей сфере требуются умение искусно работать, знание математики, навыки общения и написания прозы. Обычно при словах «разработчик программ» люди не думают обо всем этом, но без всего этого не выйдет хорошего разработчика. Надо постоянно об этом помнить. И все же наша профессия – одна из самых увлекательных в мире. Думаю, нам повезло, что мы выросли в те времена, когда все эти качества толкают человека на путь программирования. Что бы мы делали несколькими поколениями раньше?

6

Джо Армстронг

Джо Армстронг известен прежде всего как создатель языка программирования Erlang и Open Telecom Platform (OTP) – платформы для создания приложений на Erlang.

Среди современных компьютерных языков Erlang выглядит чудачком. Он старше и одновременно моложе многих популярных языков: Армстронг начал работу над ним в 1986 году – за год до появления Perl, – но язык был доступен лишь в качестве коммерческого продукта и использовался только в изделиях компании Ericsson, пока не был выпущен с открытым исходным кодом в 1998 году – через три года после появления Java и Ruby. Истоки Erlang кроются скорее в логическом языке Пролог, чем в каком-либо из языков семейства Алгол. Кроме того, он был разработан для специфического вида ПО с высокой доступностью и надежностью, как например АТС. Свойства, благодаря которым этот язык применялся на АТС, почти неожиданно сделали его пригодным также для написания программ, предназначенных к использованию в параллельных системах. Это привлекло к Erlang внимание программистов, когда стало понятно, что будущее за многоядерными процессорами.

Армстронг в каком-то смысле тоже чудак. Будучи физиком, он стал осваивать информатику, когда во время написания диссертации по физике, испытывая финансовые затруднения, нанялся исследователем в команду Дональда Мичи – одного из британских первопроходцев в области иску-

ственного интеллекта. В лаборатории Мичи Армстронг познакомился со всеми достижениями в этой сфере, стал членом-основателем Британской робототехнической ассоциации и написал несколько статей о зрении роботов.

Когда финансирование работ в области искусственного интеллекта иссякло благодаря известному «Отчету Лайтхилла», Армстронг на несколько лет вернулся к программированию для целей физики, работая в научной ассоциации EISCAT и затем в Шведской космической корпорации, пока наконец не оказался в Лаборатории информатики компании Ericsson, где и создал Erlang.

Сидя на кухне стокгольмской квартиры Армстронга, мы несколько дней обсуждали подход к параллелизму, реализованный в Erlang, необходимость поиска более простых и эффективных способов связи между программами, а также важность вскрытия «черных ящиков».

Сейбел: Как вы научились программированию? Когда все это началось?

Армстронг: В школе. Я родился в 1950 году – тогда компьютеров было мало. В последнем классе, когда мне было примерно 17, местный совет приобрел мейнфрейм – вероятно, производства IBM. На нем можно было писать программы на Фортране. Тогда все писали программы на специальных бланках и отсылали их. Через неделю программные бланки и перфокарты возвращались к вам, и надо было их одобрить. Но те, кто набивал перфокарты, часто ошибались, так что они путешествовали туда-сюда, иногда по два раза. В конце концов все это поступало в компьютерный центр.

Из компьютерного центра их опять присылали к вам, и компилятор Фортрана, споткнувшись о первую же синтаксическую ошибку, с остальной частью программы вообще не работал. Чтобы запустить свою первую программу, требовалось месяца три. Тогда я научился вместо одной программы писать несколько параллельных подпрограмм, и отсылал уже их. Я написал программку, которая выводила на печать изображение шахматной доски. Но все подпрограммы были решены как параллельные задачи, поскольку время обработки задания было вероятно большим.

Сейбел: Значит, приходилось писать подпрограммы, по сути, с модульными тестами для их проверки?

Армстронг: Да, и потом соединять их. Не знаю, можно ли это назвать обучением программированию. Потом я поступил на физическое отде-

ление Лондонского университетского колледжа. Кажется, программирование там начиналось уже на первом курсе. Время обработки заданий составляло часа три. Но при запуске одновременно четырех-пяти программ все шло быстрее.

Сейбел: А в старших классах были уроки информатики?

Армстронг: Нет, был факультатив – что-то вроде компьютерного клуба. Помню, как мы ходили смотреть на компьютер. Множество серьезных, взрослых людей в белых халатах, с ручками в кармане, двигались торжественно, как в церкви. Компьютер был очень дорогой.

Сейбел: Вы изучали физику. А как вы перешли к программированию?

Армстронг: Когда я был на последнем курсе, на некоторых занятиях требовалось писать программы. Мне это нравилось. Я очень хорошо освоил отладку и делал ее для других, если ничто больше не помогало. Стандартной таксой была бутылка пива. Дальше все зависело от сложности – двухбутылочная задача, трехбутылочная задача и так далее.

Сейбел: То есть сколько человек должен был вам за отладку его программы – две, три бутылки?

Армстронг: Да. Исправляя ошибки, я обычно читал программу и говорил: «Тут можно бы сделать и попроще», – и переписывал ее. Меня поражало, что народ пишет такие сложные программы. Я понимал, как обойтись пятью строчками, а другие писали их десятками! И я удивлялся: как можно не видеть простого решения.

Но по-настоящему я занялся программированием после того, как стал бакалавром и решил, что буду писать диссертацию. И вот я начал трудиться над диссертацией по физике высоких энергий и присоединился к группе, которая работала с пузырьковой камерой. У них был компьютер Honeywell DDP-516. Я мог работать с ним сам! Да, там были перфокарты, но я мог запускать программы – вставляешь карту, нажимаешь кнопку – трррр! – и машина тут же выдает ответ. Просто восторг! Я написал для этого компьютера шахматную программку.

Тогда применялась память на магнитных сердечниках, которые соединяли вместе пожилые тетеньки. Если вы заглядывали внутрь, то видели магнетики и кучу проводов. Страшно дорогая штука – двадцать дисков весом под пятнадцать кило, на которых умещалось мегабайт десять. И телетекстовый интерфейс, чтобы настраивать программы.

А потом появился один из первых дисплеев – можно было набирать программы и редактировать их. Фантастика! Никаких больше перфокарт. Помню, мы разговорились с сотрудником, который обслуживал компьютер, и я сказал: «Когда-нибудь такой будет у каждого». А он

мне в ответ: «Ты с ума сошел, Джо!» – «Почему?» – «Да потому, что это слишком дорого».

Вот тогда я и научился программировать всерьез. Мой научный руководитель говорил: «Тебе надо бросить диссертацию по физике и заняться компьютерами, раз ты так их любишь». Я возражал, что хочу закончить ее, если уж начал. Но он был прав, как видим.

Сейбел: Так вы написали диссертацию или нет?

Армстронг: Нет. У меня закончились деньги, и я поехал в Эдинбург. Когда я изучал физику, то, как и другие, занимался в физической библиотеке. В уголке там прятался стеллаж с компьютерными книгами. Среди них были четыре коричневых тома под названием «Machine Intelligence» (Машинный интеллект), изданные отделением машинного интеллекта Эдинбургского университета. Я в принципе изучал физику, но с жадностью поглощал эти книги и думал: «До чего же занятно!» Поэтому я написал Дональду Мичи, который возглавлял это отделение, о том, что все это мне очень интересно, и спросил, нет ли у него какой-нибудь работы. Мне пришел ответ. Мичи писал, что работы пока нет, но он был бы рад познакомиться, посмотреть, кто я такой.

Через несколько месяцев Мичи не то позвонил, не то написал мне, что в ближайший вторник будет в Лондоне и хотел бы встретиться со мной. Так как он пересаживался на эдинбургский поезд, то предлагал увидеться на вокзале. Мы встретились, и Мичи сказал: «Хм, здесь не очень-то поговоришь – пойдемте в паб». Мы поболтали в пабе, и чуть позже он написал мне: «У меня в Эдинбурге есть место в лаборатории, не хотите ли поработать?» Я переехал в Эдинбург и стал помощником Мичи по исследовательской части. Вот так я бросил физику ради компьютерной науки.

Во время Второй мировой Мичи работал с Тьюрингом в Блетчли-парке, и ему достались все бумаги Тьюринга. У меня был стол в библиотеке Тьюринга, так что я сидел среди тьюринговских бумаг. Год я проработал в Эдинбурге. Потом все там рухнуло. Дело в том, что правительство поручило математику Джеймсу Лайтхиллу выяснить, как в Эдинбурге обстоят дела с искусственным интеллектом. Лайтхилл выяснил и заявил, что ничего коммерчески ценного из этого не вырастет.

Вообще, все это напоминало гигантский детский манеж. Я был членом-основателем Британской робототехнической ассоциации, и нам казалось, что все это – дело чрезвычайной важности. Но те, кто финансировал нашу работу, не хотели и слышать о роботах! Где-то около 1972 года финансирование прекратилось. Народ стал говорить: «Что ж, занятная была работа, теперь надо искать что-то другое».

Итак, надо было возвращаться к физике. Я приехал в Швецию и стал заниматься физическими компьютерными программами в научной ассоциации EISCAT. Мой начальник, старше меня, пришел из IBM; я должен был разрабатывать спецификацию, а он – реализовывать ее. Мы спорили по этому поводу. Он говорил: «Плохо то, что у нас нет описания задания и подробной спецификации». А я отвечал: «Так это же прекрасно, что нет описания, – можно самому определять задание для себя». Через год он ушел, а я занял его место – место ведущего разработчика.

Я спроектировал то, что сейчас бы назвали прикладной операционной системой, то, что работает как надстройка над обычной ОС. К тому времени компьютеры уже заметно подешевели. У нас были норвежские NORD-10 – кажется, они задумывались для конкуренции с PDP-11.

Там я провел почти четыре года, а потом получил работу в Шведской космической корпорации, разработал еще одну прикладную операционную систему для управления «Викингом», первым шведским спутником. Проект был интересный. Я работал на машине, названия которой не помню, но это был клон компьютера Amdahl. Только строковые редакторы, никаких полноэкранных. Все программы хранились в одном каталоге. Не больше десяти букв в имени файла, не больше трех в расширении. Плюс компилятор Фортрана или ассемблер – вот и вся система.

При этом я думаю, что все эти современные штуки с наворотами вряд ли делают вас продуктивней. Как работа может улучшиться при иерархической файловой системе? Как бы то ни было, программы создаются главным образом в голове. Мне кажется, что работа с такой довольно простой системой дисциплинирует мышление. Если все файлы надо складывать в один каталог, нужна строжайшая дисциплина. Если нет системы управления версиями, нужна строжайшая дисциплина. И если работать дисциплинированно, то, по-моему, нет особой нужды ни в иерархической файловой системе, ни в системе управления версиями. Они не решают главную проблему. Пожалуй, с ними удобнее работать вместе нескольким людям. Если же работать одному, то разницы я не вижу.

Кроме того, сейчас у нас что-то вроде избытка выбора. У меня имелся только Фортран – помнится, не было даже консольных скриптов. Командные файлы для запуска программ, компилятор Фортрана – и все! Ну и ассемблер, наверное, если кому-то он был нужен. Никакого мучительного выбора, как сейчас. Думаю, быть сегодня молодым программистом просто жутко: двадцать языков, десятки фреймворков и операционных систем – можно погибнуть, выбирая. А в то время – ничего подобного. Просто начинаешь работать, потому что язык и программы выбирать не приходится. Просто берешь и работаешь.

Сейбел: Разница еще и в том, что сегодня не понять, как устроена система сверху донизу. Проблема не только в выборе, но и в том, что не во всех используемых «черных ящиках» хочется разбираться.

Армстронг: Ну да. Если «черные ящик» работает неважно и надо что-то в нем подкручивать, то легче взять и написать все самому. А вот что точно не работает, так это повторное использование кода. С этим совсем плохо.

Сейбел: Вы создали не только Erlang, но и Open Telecom Platform – платформу для разработки приложений. Что вы скажете о ее многократном использовании?

Армстронг: В какой-то мере это возможно. Но возникает все та же проблема. Если эта платформа полностью решает вашу задачу, если какой-нибудь программист, ничего не знающий о структуре ОТР, посмотрит на нее через несколько лет и скажет: «Вот именно то, что мне нужно», – отлично, вы уложились в эту меру повторного использования. Если же этого *не* случится, проблема останется.

Не так давно мне говорили: «Все это как-то искусственно, мы все время стараемся впихнуть код в этот ОТР». Я отвечал: «Ну что ж, переделайте ОТР». Но ведь этого не сделаешь. При этом фреймворк – всего лишь программа, довольно простая по устройству. Действительно простая. Я залезаю в нее, и она делает то, что нужно этим людям, после чего они соглашаются: «И правда, просто». Но при этом заявляют: «Наши менеджеры не хотят, чтобы мы возились с фреймворком». Ну так назовите это по-другому, и все.

Сейбел: А как по-вашему, реально ли на самом деле вскрыть все эти «черные ящики», заглянуть внутрь, понять, как они работают, и приспособить их для своих нужд?

Армстронг: С годами я, кажется, все чаще допускаю вот эту типичную ошибку: боюсь вскрыть «черный ящик». Он кажется мне таким непроницаемым, таким сложным, что я не хочу открывать его. Один-два я все же открыл; я хотел сделать оконную графическую программу для Erlang и подумал: «А не запустить ли ее на X Windows». Что такое X Windows? Это сокет с протоколом поверх. Открываешь сокет и транслируешь через него сообщения. Зачем библиотеки? Erlang основан на сообщениях. Идея в чем? Вы посылаете сообщения по какому-то адресу, и там что-то делается. То же и в X Windows – у вас есть окно, вы посылаете сообщение, и начинается выполнение. Если же вы делаете что-то в окне, сообщение вам возвращается. Очень похоже на Erlang. Однако *программирование* в X Windows происходит при помощи библиотек с обратными вызовами. Это не философия Erlang. Послать сообщение, чтобы начала выполняться команда, – вот философия Erlang. Итак,

оставайтесь подключенными и обращайтесь непосредственно к сокету, без библиотек.

И знаете, это очень легко. X-протокол принимает 80–100 сообщений, а вам нужно, скажем, только 20. Вы отображаете их в Erlang, стоит чуть поколдовать – и вы можете отправлять сообщения прямо в окна, а те уже займутся всем. К тому же это быстро работает. Правда, выглядит не очень – я мало заботился о графике, о внешнем виде, над этим надо еще поработать. Но главное – это легко.

Кроме того, я создал программу верстки, где границей абстракции, которую я перешел, был PostScript. Подходя к этой границе, боишься ее пересекать, поскольку думаешь, что за ней что-то невероятно сложное. Но и тут все оказывается легко. Это язык программирования. Хороший язык программирования. Границу абстракции пересечь легко, и это очень полезно.

Когда издавалась моя книга по Erlang, издатель сказал: «У нас есть софт для рисования схем». Но все такие программы обычно ставят стрелку не совсем туда, куда надо. Да и рука после них болит. Я подумал, что это должно занять всего несколько часов – программа, которая бы выдавала PostScript-файлы и ставила стрелку точно в нужное место. Времени тратится на создание схем с помощью программ примерно столько же, сколько в системах WYSIWYG. Но у последних есть два преимущества. Во-первых, рука не болит и, во-вторых, при увеличении даже в 10 000 раз стрелка показывает туда, куда надо.

Конечно, начинающий программист не обязан осваивать все эти абстракции. Но надо держать в уме такую возможность, не отвергать ее, понимая, что прямой путь может быть эффективнее пакетного. Вообще, думаю, приобретая написанную кем-то программу, обычно тратишь много времени, чтобы приспособить ее к своим нуждам: программа делает не совсем так, как надо, а чуть по-другому.

Сейбел: Вы сказали, что с повторным использованием кода дело обстоит «совсем плохо». Но ведь открытие каждого «черного ящика» и возня с его содержимым едва ли исправит эту ситуацию.

Армстронг: Думаю, проблемы с повторным использованием кода есть в объектно-ориентированных, а не в функциональных языках. Ведь объектно-ориентированные языки тянут за собой всю неявно окружающую их среду. Вы хотели только банан, а получили еще и гориллу, которая держит этот банан, и все джунгли впридачу.

Если у вас образцово прозрачный код, если у вас чистые функции – все данные вводятся через их аргументы, все возвращается и не оставляет за собой никакого состояния, – программа более чем пригодна для повторного использования. Можно использовать ее где угодно. Если надо

применить ее в другом проекте, вы просто вырезаете код и вставляете его в другой проект.

Программисты научились пользоваться разными языками, но не научились пользоваться легкими способами склейки программ. Конвейеры UNIX – «А переходит в Б переходит в В» – банально простой способ склейки. Пользуются ли ими программисты? Нет, они берут несколько API и связывают их в одной и той же области памяти, что очень сложно и не позволяет поддерживать разные языки. Если языки из одного семейства, это проще – с императивными языками, например, все прекрасно. А если, допустим, у нас есть Пролог и Си? У них совершенно разное представление об управлении памятью. Их невозможно скомпоновать. То есть повторное использование кода тоже невозможно. Вероятно, тут замешаны крупные коммерческие интересы: кому-то очень не хочется, чтобы все это работало вместе. Появляются тысячи рабочих мест для консультантов, тысячи утилит для решения проблем, которых нет, – они решены много лет назад.

Удивительно, но очень мало языков программирования описывают взаимодействие между программами. Вернемся к склейке программ и способам описания протоколов. У нас нет способов описания межпрограммных протоколов: если я шлю вам вот это, вы присылаете мне вон то. У нас есть способы описания пакетов и типов пакетов, но что касается протоколов – таких способов очень мало.

Программирование полностью отличается от того, как мы конструируем вещи в реальном мире. Допустим, вы автопроизводитель. Вы покупаете компоненты у субподрядчиков – батареи у Lucas, генераторы еще у кого-то – и соединяете их вместе, последовательно. Когда вы строите дом, то кладете кирпичи один на другой и помещаете дверь *там-то*. Именно так делаются чипы: есть готовая плата, обеспечивающая эти соединения. Покупая чипы, вы соединяете их ножки проводами. Хорошо бы и программы делать так, но мы не делаем.

Причина, по которой мы этого не делаем, связана с параллелизмом. Чипы, соединенные последовательно, выполняют операции параллельно. И они посылают сообщения. Они основаны на парадигме обмена сообщениями – той парадигме программирования, в которую я верю. Но сегодня она не применяется для создания программ. Erlang мог пойти по этому пути – во всяком случае, я бы этого хотел: эволюционировать в сторону компонентного строения. Пока я еще этим не занялся, но хотел бы разработать графический внешний интерфейс для создания компонентов и начать писать программы для связывания их воедино. Потокоевое программирование крайне декларативно. В нем нет такого понятия, как последовательные состояния. В нем нет программного

счетчика, который в них переходит. Что-то просто есть, и все. Декларативная модель легка для понимания. Но в большинстве языков она не реализована.

Я не хочу этим сказать, что содержимое «черного ящика» не может быть сложным. Возьмите, например, такую утилиту, как `grep`. Если глядеть извне, это что-то вроде квадрата. На входе – поток данных, файл. Можно сказать `cat foo | grep`, и `grep` получает некие аргументы, регулярное выражение, которому нужно найти соответствие. Хорошо. И `grep` выдает все строки, соответствующие этому регулярному выражению. На уровне восприятия то, что делает `grep`, до крайности просто. На входе – файл. Регулярное выражение. На выходе – набор (поток) строк, соответствующих этому выражению. Но это не означает, что действующий внутри «черного ящика» алгоритм прост – он может быть предельно сложным.

Происходящее внутри «черного ящика» может быть предельно сложным. А процесс склеивания сложных компонентов необязательно сложен. Использовать `grep` совсем несложно. Вот чего я не вижу в архитектуре систем: четкого различия между склеиванием составных частей и устройством, порой очень сложным, самих частей.

Соединяя программы при помощи API языка программирования, мы не получаем абстракцию «черного ящика». Мы отправляем их в одну и ту же область памяти. Если `grep` есть модуль, который предоставляет вызовы в своем API, и вы снабжаете его указателем `char*`, и вам надо выделять для этого память, и вы занимаетесь глубоким копированием этой строки – можно ли создать параллельный процесс, делающий все это? В таком случае это становится трудным для понимания. Не знаю, почему люди склеивают программы таким замысловатым образом. Надо выбирать методы попроще.

Сейбел: Если сравнить то, что вы думаете о программировании сейчас, и то, что думали в начале карьеры, в чем главное отличие?

Армстронг: Главные отличия в том, как я думаю о программировании, не имеют ничего общего с аппаратным обеспечением. Да, оно становится быстрее и мощнее, но наш мозг в миллион раз мощнее лучших программных инструментов. Я могу писать программу и вдруг, через много времени, обнаружить, что в ней ошибка: если произойдет вот это, и вот это, и вот это, случится сбой. Смотрю в код – и правда, ошибка! Хотя ничто в работе программы не говорит об этом. А теперь скажите, какая среда разработки на такое способна? Поэтому перемены, которые со мной произошли, – это перемены в моей голове.

Есть две перемены, связанные с опытом программирования. Вот первая: когда я был моложе, то обычно, закончив писать программу, пере-

ставал работать над ней. Ведь все закончено! Потом наступало озарение: «Что за бред?! Я идиот! Надо переписать». И потом снова то же самое.

Помню, как размышлял: не лучше ли сначала все продумать, а потом записать? Не лучше ли испытать озарение, еще не написав программу? Думаю, теперь у меня получается. Поэтому я считаю, что в течение двадцати лет учился программированию. Сейчас я знаю, как это делается. Я ставил эксперименты, чтобы научиться. Теперь я умею программировать и не нуждаюсь в экспериментах.

Иногда, правда, я ставлю совсем небольшие эксперименты – пишу крошечные программы, только чтобы ответить на какой-то вопрос. Я все продумываю, и программа более-менее работает по моему плану, потому что я продумал ее. Это, конечно, занимает много времени. Пишешь программу, потом наступает озарение, все переписываешь – написание программы может занять год. Поэтому теперь я предпочитаю целый год думать над ней. Просто я ничего не набираю на клавиатуре.

Это первое. Второе – это интуиция. Когда был моложе, я мог программировать ночь напролет, до четырех утра, и страшно устал. Как настоящий крутой программист, я писал код, час за часом. Выходило неважно, но я упорно продолжал работать, когда интуиция уже покинула меня.

И я усвоил вот что: если устал, то выходит дерьмово, и на другой день это выбрасываешь. Двадцать лет назад я продолжал работу, даже чувствуя, что с кодом не все в порядке. С годами я заметил, что по-настоящему хорошо получается, когда я полностью в потоке – не забочусь о времени, мало думаю о самой программе, просто сижу расслабленно, что-то набираю и гляжу, что там на экране. Вот тогда код выходит хорошим. Раньше я не понимал, что это такое, когда что-то подсказывает тебе: «Нет-нет, все не так». Сейчас, если что-то говорит «нет», я останавливаюсь. Я знаю по опыту, что надо на время завязать с кодом – оставить эту задачу, подумать о чем-то другом.

Я хорошо успевал по математике в школе и считал, что у меня логический склад ума. Но потом психологические тесты показали, что у меня прекрасно развита интуиция, а вот с логикой хуже. Не то чтобы плохо – я могу заниматься математикой и программированием вполне на уровне. Но поскольку я хорошо успевал по математике, то думал, что наука вся построена на логике и математике. Теперь не скажу этого, потому что знаю, как много значит интуиция, уверенность в правильности.

Сейбел: Итак, теперь вы намного больше размышляете о коде. Что именно вы делаете на этом этапе?

Армстронг: Ну, я не просто сижу и думаю – я делаю пометки на бумаге. Пожалуй, это слабо связано с кодом – если взглянуть со стороны,

я о чем-то думаю, что-то черчу. Еще один очень важный момент – я спрашиваю своих коллег: «А как бы вы решили эту задачу?» Нередко бывает такое: приходишь к кому-то и говоришь: «Прямо не знаю, как сделать вот это – так или так, выбрать А или Б», – рассказываешь про А и Б и вдруг на полуслове хлопаешь себя ладонью по лбу: «Конечно, Б! Большое, большое тебе спасибо».

Если просто написать все это на доске, ничего не выйдет – нужна обратная связь. Нужен человек, выступающий в роли доски. Вы объясняете что-то, рисуете альтернативные решения, человек вступает в разговор и подсказывает оригинальный ход. И вы внезапно видите решение. Для меня это не распространяется на само написание кода, но диалог с коллегами, решающими сходные проблемы, может быть очень полезен.

Сейбел: Что тут главное – реплики других, вопросы, сам процесс объяснения?

Армстронг: Думаю, вот что: вы переводите задачу из той части мозга, которая ее решает, в ту часть, которая вербализует. Это две разные части. Вы форсируете решение. Я, правда, никогда не пробовал говорить вслух в пустой комнате.

Сейбел: Я слышал, что на одном факультете компьютерных наук в кабинете преподавателя был плюшевый медведь, которому следовало изложить вопрос, прежде чем побеспокоить преподавателя. «Значит так, мистер Медведь, я работаю над такой-то задачей и думаю, как лучше сделать... ага, понял!»

Армстронг: Да? Надо попробовать.

Сейбел: Поговорите со своими кошками.

Армстронг: О, само собой! Я работал с одним парнем чуть постарше меня, очень умным. Всякий раз, когда я приходил к нему в офис с каким-то вопросом, он говорил: «Программа – это черный ящик. Есть вход, выход и функциональная связь между ними. Какой у тебя вход, какой выход, какая функциональная связь?» В ходе беседы я вдруг выкрикивал: «Да ты гений!» – и выбегал из комнаты, а тот удивленно качал головой: «Он даже не объяснил мне, в чем проблема». Так что тот парень был вроде медведя.

Сейбел: А что вы чертите – куски кода или просто линии, фигуры?

Армстронг: Чаще всего кружки со стрелками. Когда рисуешь что-то на доске, объясняя людям, то это кружки со стрелками, уравнения, разные обозначения, но не код. Только иногда это куски кода, потому что так экономнее всего что-то выразить. Это в период обдумывания. Код пишу лишь изредка, чтобы понять, сколько времени что займет. Написав строк десять кода, я оцениваю время.

Сейбел: То есть сколько времени займет выполнение программы на компьютере?

Армстронг: Да. Я не знаю, будет это микросекунда или миллисекунда. Я могу что-то предполагать, но предположение должно подтвердиться. Поэтому я смотрю только на те части, про которые ничего не понимаю. Но у меня большой опыт программирования, связанный с Erlang, и я примерно понимаю, что будет делать программа. За сколько-то лет пути решения задач не изменились: определить самые трудные части, написать небольшие прототипы, выявить области, в которых не уверен, написав совсем небольшие кусочки кода. В принципе, сейчас я делаю то же самое, но уже не так нуждаюсь в этих небольших экспериментах, если пишу на Erlang. Если же пишу на Ruby или Java, то вспоминаю прошлое и провожу эксперименты, так как не знаю, что может быть дальше.

Сейбел: И посреди обдумывания вы вдруг понимаете, каким должен быть код?

Армстронг: Да, все части складываются воедино. Но, вероятно, объяснить другим я не смогу – просто приходит сильнейшее чувство, что если начать писать программу сейчас, она заработает. Я не знаю в точности, что это за решение. Это как с яйцом. Цыпленок готов вылупиться из яйца. Я тоже готов.

Сейбел: Теперь вы в потоке, и вас нельзя прерывать.

Армстронг: Да-да.

Сейбел: Как я понимаю, остается множество вещей, которые нужно разобрать на уровне кода. Вам нужно сосредоточиться.

Армстронг: Именно так. Но есть два типа сосредоточения. Вещи, которые требуют реального сосредоточения, не те, что можно сделать автоматически, – это вещи, которые надо обдумывать. Это своего рода хитрая сборка мусора, нужно понять, что именно и где разметить. Нужно крепко подумать. Ты знаешь, что решение найдется, потому что ты его уже ограничил со всех сторон. И знаешь, что оно в этом маленьком черном ящике.

Когда Микеланджело расписывал, к примеру, потолок Сикстинской капеллы, у него была команда помощников. Он сперва рисовал общий вид фрески – вот здесь все закрасить голубым, а здесь зеленым. Это напоминает создание программ. Сперва общий набросок, где все расставлено по местам. Некоторые области можно закрасить одним цветом и довольно быстро – это не требует размышления.

А вот рисовать глаза – хитрое дело. Знаешь, что можешь сделать это и что глаза на верном месте, потому что набросок правилен. Начинаешь

прорисовывать глаза в деталях. Это не так-то просто, надо всерьез сосредоточиться. А если взять лоб или щеки, особого сосредоточения не нужно – это однородные области. Допустим, на щеках пробиваются волосы, но все равно можно сосредоточиться наполовину.

Затем все набираешь, исправляешь синтаксические ошибки, запускаешь несколько небольших тестов, чтобы проверить, как работает программа. Тут уже можно расслабиться. Увидел мелкую ошибку компиляции, исправил. Поднаторев в языке, можно даже не читать диагностику. Там все равно только номера строк. Ага, такая-то строка неправильная, нужно ее переписать.

Когда я читал курс лекций по Erlang в Чикаго, то бродил по классу и понимал: так, здесь что-то не то. Здесь пропущена запятая или программа даст сбой еще до такого-то события, а у вас нет связей. Моя жена – отличный корректор, сразу видит ошибки. Пропущенная запятая, орфографическая ошибка буквально бросаются ей в глаза. Мне бросаются в глаза ошибки в чужом коде. Это почти бессознательно: смотришь на экран – и бац! – видишь ошибку. Так что речь идет об исправлении несущественных погрешностей.

Сложнее найти небольшие ошибки в написании имен переменных. Чтобы избежать их, я специально подбираю для разных переменных очень непохожие имена. Например, легко спутать переменную `personName` и список имен `personNames`, поэтому переменную я называю `personName`, а список – `listOfPeople`. А вот пунктуацию я вижу хорошо – все, что касается запятых, скобок и так далее. Плюс к тому Emacs сам все раскрашивает, сам делает абзацные отступы, разные скобки выходят разного цвета. Работа сильно облегчается.

Сейбел: Когда вы переходите к написанию кода, то откуда начинаете – сверху, снизу, с середины?

Армстронг: Снизу. Я пишу небольшой фрагмент и тестирую его, потом еще один и так далее. Сейчас пишу сначала модульные тесты, а потом уже код. И твердо уверен в своем подходе.

Сейбел: Вернемся в прошлое. После Шведской космической корпорации вы работали в исследовательской лаборатории компании Ericsson?

Армстронг: Да. Это было счастливое время, на редкость счастливое – кажется, 1984 год. Я пришел в лабораторию года через два после ее создания, там царил оптимизм. Мы считали, что решим все задачи, убедим руководство запустить новые проекты, производительность компании взлетит. Эти взгляды не имели ни малейшего отношения к реальности. Мы полагали, что придумаем новые полезные программы, и мир встретит нас с распростертыми объятиями. Позже мы поняли, что откры-

вать новые вещи не так-то легко. И *невероятно* трудно убедить людей пользоваться новыми, более совершенными вещами.

Сейбел: Erlang относился к этим новым, более совершенным вещам?

Армстронг: Да, разумеется. Вначале был Пролог. Я создал нечто вроде небольшого языка, люди стали им пользоваться. Роберт Вирдинг поглядел и сказал: «Прикольно», – а потом спросил, можно ли слегка изменить мой Пролог. Вообще, это довольно опасно – Роберт пишет в комментариях к программе: «Джо придумал, я кое-что изменил», – но на самом деле меняется все. Так что мы переписали код от и до и крепко спорили: «Не могу прочесть твой код, сплошные пробелы после запятых», – в таком духе.

Наконец мы обнаружили кое-кого в Ericsson, кому нужен был новый язык программирования – или, скорее, им нужен был более лучший подход в программировании телефонии. Мы встречались с ними раз в неделю, так продолжалось полгода, может быть, месяцев девять. Общая идея была такова, что они узнают от нас о программировании, а мы от них – о телефонии, о том, что тут за проблема. Это было очень трудно, но при этом стимулировало нас. Язык изменился, поскольку перед нами были живые люди, которые пользовались им, проводили исследования. По результатам они говорили: «Хорошо, но слишком медленно», – и указывали, что все должно работать в 70 раз быстрее. Итак, мы взялись выполнить их требования, заставить язык работать в 70 раз быстрее за два года или около того.

У нас было несколько фальстартов, были и тяжелые моменты. И была одна крупная ошибка. Никогда не обещайте людям, что все будет работать с такой-то скоростью, не осуществив реализации. Но наконец мы поняли, как это сделать. Я написал компилятор на Прологе, Роб занимался библиотеками и разными программами. Примерно после двух лет работы я подумал, что смогу реализовать эту абстрактную машину на Си, – так мне впервые довелось писать на Си. Майк Уильямс, случившийся рядом, посмотрел мой Си-код и сказал: «Это самый плохой Си-код, который я видел в жизни. Никуда не годится». Не думаю, что было *настолько* плохо, но Майку не нравилось. Поэтому Майк создал виртуальную машину на Си, а я – компилятор на Прологе. Затем компилятор скомпилировал сам себя, произвел байт-код, мы вставили его в машину, поменяли грамматику и синтаксис, скомпилировали компилятор в нем самом и получили нечто такое, что могло самозагружаться. Дело пошло. Это был уже не Пролог, а новый язык.

Сейбел: Было что-нибудь, что оказалось трудно встроить в Erlang?

Армстронг: Да. Мы полностью абстрагировались от памяти. Если вы переводите картинку из JPEG в двоичное изображение, а это очень

сильно зависит от точного места размещения данных, получается так себе. Неважно получаются алгоритмы, работающие через деструктивное изменение состояния.

Сейбел: Если бы вы создавали большую программу для последовательной обработки изображений, то писали бы код для преобразования картинок на другом языке?

Армстронг: Да, на Си, или на ассемблере, или на чем-то еще. Или на диалекте Erlang с последующей кросс-компиляцией между Erlang и Си. Создать диалект – своего рода предметно-ориентированный язык¹. Можно было бы также написать программы на Erlang, которые бы генерировали программы на Си, а не писать последние от руки. Но целевой язык – Си, или ассемблер, или что-то еще. Писать от руки или генерировать? Интересный вопрос. Я склоняюсь к генерированию – это проще.

Но я бы использовал структуру Erlang. У меня есть кое-что для создания семейного альбома и всего такого. Там я использую ImageMagik и несколько консольных скриптов, но управляю всем из Erlang. Я просто пишу обертки вокруг них, запускаю `os:command` и потом команду в ImageMagik. Обертки – это удобно. Я бы не хотел обрабатывать картинки в Erlang, писать это на Erlang глупо. Си для этого подходит куда больше.

Сейбел: И, кроме того, уже написан ImageMagik.

Армстронг: Вот это меня совершенно не волнует. Если бы я писал это на OCaml, я бы взял и написал это, потому что в OCaml можно достичь эффективности такого рода, но Erlang делать этого не может. Будь я OCaml-программистом, что бы я сделал? Переписать ImageMagik? Поехали!

Сейбел: Просто потому, что написать свое увлекательнее?

Армстронг: Я люблю программировать, так почему бы и нет? Я всегда говорил, что Erlang не годится для обработки изображений, – я и не пробовал делать в нем это. Наверное, ничего не выйдет, хотя кто знает? Надо попробовать. Хм, занятно. Не искушайте меня.

По-настоящему хороший программист много программирует. Исключений я не встречал. Если я не программирую два-три дня, то ощущаю зуд. И потом, чем больше делаешь, тем быстрее работаешь. Побочный эффект написания всяких дополнительных программ в том, что рутинные вещи начинают выполняться намного быстрее.

¹ Предметно-ориентированные языки также известны как DSL (Domain Specific Language). – *Прим. науч. ред.*

Сейбел: Вы делали что-нибудь специально для улучшения своих программистских навыков?

Армстронг: Да нет. Я изучал незнакомые мне языки, но вовсе не с целью совершенствоваться в программировании. Может быть, с целью улучшить навыки проектирования языков.

Мне нравится разбираться в том, как все работает. Хороший способ проверки – реализовывать все самому. Для меня программирование – это не ввод кода в машину, это процесс понимания. Программирование есть понимание. Я люблю понимать, как все устроено. Почему бы мне не реализовать программу для JPEG, о которой мы говорили? Ведь мне нравится разбираться в вейвлет-преобразованиях. А программирование как раз позволяет в них разобраться. Зачем я создаю интерфейс для X Windows? Чтобы разобраться, как работает X-протокол.

Реализация чего-нибудь – сильный мотивирующий фактор. Рекомендую всем. Хотите понять Си – напишите для него компилятор. Хотите понять Лисп – напишите для него компилятор или интерпретатор. Некоторые говорят: «Компилятор – это же так трудно». Совсем нет. Это легко. Есть масса мелочей, которые не трудны и которые нужно освоить. Надо разбираться в структурах данных. Надо разбираться в хеш-таблицах и в парсинге. В генерировании кода. В техниках интерпретации. Каждый из этих предметов не особенно сложен. Начинающие думают, что это все большие и сложные темы, и поэтому к ним не подступают. Все, что вы не делаете, трудно, все, что вы уже сделали, легко. Люди даже не пытаются ничего делать, и я думаю, это ошибка.

Сейбел: Кое-кто из моих собеседников советовал учить разные языки, потому что в них применяются разные подходы к решению задач.

Армстронг: Да, но только если эти языки делают разные вещи. Нет смысла учить несколько языков, делающих одно и то же. Я немало писал на JavaScript, и на Tcl, и на Си, и на Прологе – в действительности на Прологе, на Фортране и на Erlang даже очень много. Чуть-чуть на Ruby и на Haskell. Я знаком со всеми языками, но не на всех могу программировать. Хотя я могу писать программы на довольно многих языках.

Сейбел: Но не на C++?

Армстронг: Нет. На C++ я едва могу читать и писать. Не люблю C++; плохо его чувствую, он слишком переусложнен. Мне по душе компактные небольшие языки, а этот большой и громоздкий.

Сейбел: Какие языки повлияли на структуру Erlang?

Армстронг: Пролог. Разумеется, он вырос из Пролога.

Сейбел: Сегодня в нем уже непросто распознать то, что идет от Пролога.

Армстронг: Унификация, сопоставление образцов – все это напрямую идет от Пролога. Как и структуры данных. Синтаксис кортежей и списков слегка различается, но тем не менее очень похож. Еще теория взаимодействующих последовательных процессов Тони Хоара. Я также читал об охраняемых командах Дейкстры. Вот почему я требую, чтобы сопоставление с образцом всегда было явным, не должно быть случая по умолчанию, всегда должна быть ветка с совпадением. Вот основные влияния.

Сейбел: А функциональный облик языка – он откуда?

Армстронг: После внедрения параллелизма в Пролог надо было сделать так, чтобы после какого-нибудь действия не происходил возврат назад. В Прологе можно вызвать что-нибудь и вернуться назад, полностью уничтожая эффект от вызова. Поэтому идея была та, чтобы после команды, скажем «Пуск ракет», они вылетают – ррраз! – и отменить уже нельзя. Чистые Пролог-программы обратимы. Но при взаимодействии с реальным миром все действия совершаются только в одном направлении. Подали команду «Пуск» – ракеты пущены. Подали команду «Зеленый сигнал светофора» – загорается зеленый, и нельзя потом пересмотреть решение, сочтя его плохим.

Итак, у нас есть язык с поддержкой параллельных процессов, и уже внутри них мы имеем самый настоящий Пролог с возвратами и всем прочим. Пролог, в котором там и сям сделаны срезы, чтобы избежать возврата, стал весьма детерминированным.

Сейбел: Где необратимые программы могут посылать сообщения другим процессам?

Армстронг: Да. Но это просто вызов функции, хотя, возможно, и не функции, запускающей ракеты. Это вызов функции, которая вызывает другую, а та – функцию, запускающую ракеты. По этой причине код, который вы создаете, становится все более функциональным, превращаясь в диалект Пролога – функциональное подмножество. А функциональное подмножество можно сделать полноценным функциональным языком.

Сейбел: Erlang сильно отличается от прочих функциональных языков, поскольку в нем есть динамическая типизация. Вы ощущаете свою принадлежность к сообществу функциональных программистов?

Армстронг: Безусловно. На конференциях по функциональному программированию мы спорим, выпячивая расхождения. Мы спорим о «жадных» и «ленивых» вычислениях, о динамической и статической типизации. Но несмотря ни на что, остается центральная идея функционального программирования: x не обозначает место в памяти, это неизменяемое значение. Мы задаем x равным 3, и дальше с этим ничего

не сделаешь. Различные функциональные сообщества сходятся на том, что это крайне полезно для понимания программ, для параллелизма, для отладки. Однако есть функциональные языки с динамической типизацией, такие как Erlang, и функциональные языки со статической типизацией. У тех и других свои сильные и слабые стороны.

Было бы здорово, если бы Erlang пользовался преимуществами статической типизации. Возможно, в отдельных местах мы сумеем аннотировать программы так, чтобы типы стали более явными, тогда компилятор сможет порождать типы и генерировать более качественный код.

Сторонники статической типизации говорят: «Мы оцениваем сполна плюсы динамического подхода при маршалинге структур данных». Мы не можем послать любую программу по проводу и воссоздать ее на другом конце – нам нужно знать тип. И вот мы имеем то, что Карделли назвал перманентно неконсистентной системой. Система, которая постоянно растет и меняется, при этом ее части могут быть временно неконсистентными. Если я меняю код, он не ведет себя как неделимый объект. Часть узлов меняются, остальные нет. Они общаются друг с другом – ведь в какие-то периоды времени они консистентны. А когда мы переходим через коммуникационную границу, как определить, что она проведена правильно? Тут надо кое-что проверять.

Сейбел: Когда-то вы отлаживали чужие программы за пиво. Почему вы считали себя большим специалистом по отладке?

Армстронг: Мне страшно нравилось заниматься отладкой. В какой-то точке программы распечатываешь переменные, что-то еще и смотришь, совпадает ли результат с ожидаемым. В одной точке программа может быть правильной, а в другой – нет. Надо посмотреть посередине, потом еще раз посередине и так далее. При условии, что можно воспроизвести ошибку. Невоспроизводимые ошибки с трудом поддаются отладке. Но я сталкивался только с воспроизводимыми. Делишь участок кода пополам каждый раз, пока не найдешь. В конце концов ошибка обнаружится.

Сейбел: Считаете ли вы, что у вас был более систематический подход?

Армстронг: Да, другие попросту сдавались, неясно почему. Я не понимал, почему они не могут отладить. Как вы полагаете, отладка – трудное дело? По-моему, нет. Просто останавливаешь программу и прокручиваешь в замедленном режиме. Я сейчас говорю о пакетном Фортране.

Конечно, если говорить об отладке систем реального времени или программ чистки памяти, то я помню, как однажды «упал» Erlang. Это было в самом начале, сразу после запуска, я сидел и что-то настукивал – и Erlang «упал». У него в оболочку было встроено что-то вроде команд Emacs. Набираешь `erl`, чтобы запустить его, и оказываешься

в REPL¹. Я набрал четыре-пять символов и сделал орфографическую ошибку. Потом я несколько раз сдвинул курсор назад, исправил, и он «упал» с ошибкой сборки мусора. Я знал, что это была очень, очень серьезная ошибка. Я попытался в точности вспомнить, что я настукивал, – там было около 12 символов – начал сначала, набил их, и все пошло без сбоя. Я сидел часа полтора, пробуя сотню разных штук. И вот снова сбой! Тут я все записал и смог приступить к отладке.

Сейбел: А чем вы пользуетесь? Операторами печати?

Армстронг: Да. Великие боги программирования говорят: «Вставь оператор печати в программу там, где, по-твоему, допущена ошибка, рекомпилируй и запусти».

Еще есть Закон отладки Джо – не помню, вычитал я его где-то или сам придумал. Звучит он так: «Все ошибки будут не дальше трех операторов в ту или другую сторону от места последнего изменения программы». Когда я работал в Шведской космической корпорации, мой начальник там раньше занимался «железом». Мы были вместе с ним в Эсранге на севере Швеции – там площадка для запуска ракет и станция слежения за спутниками. Как-то раз он ломал голову, отлавливая ошибку в оборудовании, подсоединяя осциллографы, что-то меняя. Я спросил: «Может, я могу чем-нибудь помочь?» От ответил: «Нет, Джо, это ведь железо». Тогда я сказал: «Это должно быть как в программах – ошибка недалеко от места последнего изменения». Он подумал и сказал: «Ты гений! Я же поменял конденсатор». Дело в том, что он заменил конденсатор на другой, более крупный. Мой шеф отпаял его, поставил тот, что был вначале, и все заработало. И это верно для всего. Ремонтируешь машину, что-то не так – причина в последнем действии. Всегда вспоминайте, что вы меняли.

Сейбел: Вы доказывали когда-нибудь корректность своих программ? Импонирует ли вам такой формализм?

Армстронг: И да и нет. Я преобразовывал программы алгебраически, чтобы показать, что они эквивалентны, но никогда не применял доказательство через теоремы как таковое. Помню, когда я читал курс денотационной семантики, то отказался от этой идеи. Было дано упражнение: пусть $x = 3$ и $y = 4$ в $x + y$; докажите, что схема жадного вычисления, заданная уравнением таким-то, и схема ленивого вычисления, заданная уравнением таким-то, обе приводятся к 7.

Четырнадцать страниц лемм и прочего. Потом я подумал: «Ну как же – $x = 3$, $y = 4$, $x + y = 7$ ». В то время я писал компилятор для Erlang. Если

¹ REPL (read-eval-print loop) – собирательное название интерактивных сред разработки языков семейства Лисп, Python, Ruby и др. – *Прим. науч. ред.*

бы пришлось на десятках страниц доказывать, что $3 + 4 = 7$, то доказательство правильности компилятора заняло бы не одну тысячу страниц.

Сейбел: Вы предпочитаете работать один или в команде?

Армстронг: Я люблю рабочую атмосферу в компании программистов: в целом я человек общительный. Но работать предпочитаю один. Нет, конечно, мне нравится сотрудничать с другими в смысле обсуждения проблем. Я всегда считал, что мысли, которые приходят в голову во время кофе-брейков, особенно по дороге туда и обратно, очень ценны. Бывает масса озарений. Отличная возможность объяснить свои идеи другим. Для меня важно переместить их из одной области мозга в другую. Часто, объясняя что-то, начинаешь лучше понимать задачу.

Сейбел: Вы занимались парным программированием, когда садишься за компьютер и начинаешь писать код вместе с коллегой?

Армстронг: Да, с Робертом. Робертом Вирдингом. Мы делали это, когда оба продирались через дебри, не понимая толком, что делаем. А если не знать толком, что делаешь, очень полезно работать с человеком, который и сам в таком же положении. Если один программист сильнее другого, то для более слабого это плюс – он наблюдает за работой более опытного. Можно многому научиться. Но если разница между ними слишком велика, чтобы чему-то научиться, то попросту чувствуешь себя дураком. Ну, а заниматься парным программированием с программистом твоего уровня, когда оба не знают, что делают, довольно прикольно.

Есть еще то, что я назову особыми задачами. Я не возьмусь за них, если у меня простуда или если я в плохой физической форме. Я знаю, что написание программы займет три дня, что мне надо спланировать свой день, не читать почту, работать, не отвлекаясь, четыре часа подряд. Я делаю это дома, зная, что меня не потревожат. Мне требуется полное сосредоточение. И не думаю, что парное программирование тут поможет. Слишком много помех.

Сейбел: А что это за особые задачи?

Армстронг: Представьте, что у вас есть сборщик мусора – это императивное кодирование, – где надо не забыть пометить все регистры. Или, допустим, вы делаете лямбда-поднятие в компиляторе, где черт ногу сломит. Вы даете новые ярлыки всем переменным, у вас есть четыре-пять слоев абстрактных типов данных, перемешанных между собой, фреймы стека с разными вещами – и вы понимаете, что вам надо очень серьезно поразмыслить. Нужна концентрация.

Я работаю над задачами по настроению. Иногда совсем нет вдохновения – тогда думаю, кого бы пойти отвлечь, или читаю почту. А иногда

чувствую, что вот сейчас время писать какой-нибудь трудный код. Чтобы работать над кодом, надо быть в нужном состоянии. А разве это возможно, если работаешь в паре? Из двоих всегда один не в настроении, хочет читать почту и все такое.

Сейбел: Скажите, а с Робертом Вирдингом вы занимались последовательным парным программированием – когда посылают друг другу куски кода и переписывают их?

Армстронг: Да, когда работают поочередно. Иногда у меня была программа, которая требовала двух-трех недель. Я говорил Роберту: «Ну, с меня достаточно, забирай». И он забирал. Каждый раз, когда это происходило, обратно возвращалось нечто неузнаваемое. Роберт безжалостно кромсал код, потом пересылал мне, и я его так же безжалостно кромсал.

Сейбел: Но это приносило пользу?

Армстронг: О, да. Я был в восторге, когда ему удавалось что-то улучшить. Мы продвигались очень хорошими темпами. Роберт был склонен к обобщению. Однажды я нашел переменную – отслеживал ее в 45 подпрограммах, и в конце концов выяснилось, что она ни разу не использовалась, хотя присутствовала в 45 различных функциях. Я спросил: «Зачем эта штука – она ни разу не используется?» И Роберт ответил: «Зарезервирована под будущее расширение». Я ее убрал.

Я решил написать специализированный алгоритм, убирающий все, что не нужно в данной конкретной программе. Чем более специализированными они были, тем короче становились. А когда Роберт брал мою программу, она удлинялась, делаясь более универсальной. Думаю, это философия UNIX: программа должна делать то, что от нее требуется, *и ничего больше*. У Роберта была другая философия: программа должна быть частным случаем некоей более общей программы. Поэтому мы попеременно добавляли то общего, то специального.

Сейбел: Похоже на глубокое философское расхождение. Была ли польза от того, что программа металась между двумя крайностями?

Армстронг: Была. Каждый цикл оказывался улучшен. Думаю, программа в целом очень выиграла – больше, чем если бы ее писал один из нас.

Сейбел: Расскажите, как вы проектируете программы. Может быть, приведете пример, связанный с ОТП?

Армстронг: ОТП проектировали я, Мартин Бьёрклунд и Магнус Фрёберг. Над первоначальным проектом работали только мы трое. Мы собирались каждое утро за чашкой кофе, долго беседовали – час, два часа – и за это время исписывали всю доску. Я сразу же писал всю доку-

ментацию, а они – весь код. Правда, иногда я тоже выдавал фрагмент другой кода. И когда я писал документацию и понимал, что не могу что-то описать, мы это меняли. Или они приходили ко мне со словами: «Программа не будет работать, мы поняли сегодня утром, потому что вот это не так, и это, и это». К концу дня у нас была вся нужная документация и весь нужный код – или достаточно того и другого, чтобы работать. И на этом мы успокаивались.

Иногда что-то не срабатывало, и мы откладывали все на завтра. На второй заход в тот день уже не было времени, но один заход в день – то, что нужно. Около двух часов уходило на дискуссии, столько же – на документацию и код. Четыре часа плотной умственной работы, нормальный трудовой день. Поэтому все шло просто отлично. Не помню, как долго это длилось – может, недель десять-двенадцать. После этого у нас уже была базовая конструкция, появилось больше народу. Мы определились с архитектурой, можно было наращивать программу. Привлекли к проекту еще троих-четверых.

Сейбел: Как распределялась работа между новоприбывшими?

Армстронг: Мы знали, какими будут прототипы и какими – окончательные версии. Я всегда занимал позицию проектировщиков систем: в первую очередь делай трудное. Выявляй сложные проблемы и решай их. Ну, а легкие уладятся сами. Конечно, нужен опыт, чтобы разделить проблемы на простые и сложные. Например, что-нибудь вроде программы переключения на резервный IP – это сложно, а разбор файла конфигурации – это легко. В прототипе должен быть файл конфигурации, который просто считывается. Не надо проверять его синтаксис, так как еще нет грамматики. В конечном продукте этот файл, наверное, будет в XML, со всей грамматикой, прошедшей проверку. Но это делается на автомате. У компетентного программиста на это может уйти несколько недель. Но это все выполнимо, предсказуемо, тут нет неприятных сюрпризов. А вот правильно настроить коммуникационные протоколы, чтобы они работали как надо при отказе программы, – здесь нужна небольшая группа программистов.

Сейбел: В этом случае вы писали документацию еще до кода или, по крайней мере, одновременно с ним. Вы так делаете всегда?

Армстронг: Зависит от сложности задачи. Если она очень сложная, я часто начинаю с документации. Чем сложнее, тем больше я склонен начинать с документации.

Мне нравится делать документацию. По-моему, до появления документации в нормальном виде программу нельзя считать готовой. Писать спецификации мне тоже нравится. Некоторые говорят: «Хотите знать, что делает программа? Читайте код». Думаю, это непрофессиональный

подход. Код показывает мне, что программа делает, а не то, что она должна по идее делать. Код – это решение задачи. Если нет спецификации или какой-либо документации, приходится догадываться о задаче по решению. Догадка может быть неверной. Я хочу иметь объяснение – в чем состоит задача.

Сейбел: На этой стадии вы пишете внутреннюю документацию для других программистов или документацию для пользователя?

Армстронг: Для пользователя. Мышление при этом переключается в другой режим. Для этого я создаю каталог с таким-то именем, сохраняю в нем такой-то файл, переименовываю его таким-то образом – я описываю структуру. Я как бы обдумываю вопрос. Кнут бы наверняка сказал: «Любое программирование по сути литературно». То есть вы не пишете код, а потом документацию, – вы пишете их одновременно: это литературное программирование. Но я так не считаю. Не знаю, насколько его взгляды сформировались благодаря тому, что он публикует свои программы.

Может, это переключение между полушариями мозга, а может, еще что, но при написании документации думаешь о программе по-другому, чем при написании кода. Пожалуй, литературное программирование способствует такому переключению и поэтому может быть очень продуктивным. Я ввел кое-какие литературные элементы в Erlang, хотя использовал их мало. Вообще, это любопытная мысль – написать что-нибудь, пользуясь литературными элементами Erlang. Я не против самой идеи, просто я нетерпелив, рвусь писать код, а не документацию. Но если хотите что-то хорошо понять, создание документации – необходимый шаг.

Если бы я программировал на Haskell, то с самого начала задумался бы о типах, написал для них документацию. Работая с Лиспом или Erlang, можно начать с кода, не особенно заботясь о типах. В каком-то смысле написание документации – тоже забота о типах. Начинается со связи «есть». «Мелодия есть последовательность нот». Хорошо. Мелодия есть последовательность аккордов, каждый из которых является сочетанием нот равной длины. Простыми определениями терминов в документации – такое-то есть то-то – вы делаете своего рода анализ типов и декларативно размышляете о структурах данных.

Сейбел: Как вы полагаете, языки программирования в целом становятся лучше? Мы уже встали на путь, когда можем вооружиться новыми идеями, усвоив уроки прошлого?

Армстронг: Да. Новые языки вполне хороши. Haskell и ему подобные, Erlang. Есть интересные языки, которые надо использовать активнее.

Например, Пролог – прекрасный язык, но малоиспользуемый. Ковальски назвал его решением в поисках задачи.

Сейбел: Дэн Ингаллс говорит, что мы должны пересмотреть свои взгляды на Пролог, после того как уже несколько десятилетий испытываем действие Закона Мура.

Армстронг: Пролог сильно отличается от других языков. Там реализован любопытный способ мышления, и он подходит не для всех задач, хотя и для очень многих. Он мало распространен, к нашему стыду, – ведь программы на нем выходят очень короткими. Когда я написал первую свою программу на Прологе, то испытал нечто вроде шока. Поразительное было ощущение. Смотри – где программа, которую ты написал? Ты всего лишь рассказал немного фактов о системе, о своей задаче, а он сообразил, что делать. Просто чудесно! Надо бы бросить Erlang и вернуться к Прологу.

Сейбел: Есть ли навыки, не связанные прямо с программированием, которые тем не менее помогли вам лучше делать вашу работу? Или такие навыки, которыми должен обладать программист?

Армстронг: Умение писать. Кто-то из компьютерных теоретиков сказал: «Если у вас плохо с английским, вы никогда не станете хорошим программистом».

Сейбел: Кажется, Дейкстра.

Армстронг: Со мной время от времени советуются университеты насчет учебных планов по компьютерным наукам. Я ведь работаю в компании – вот они и хотят знать, что нужно на практике. Я говорю: «Учите их писать и подбирать убедительные доводы». Большинство выпускников, имеющих степень по компьютерным наукам, не сильны в писании текстов.

Думаю, учить этому очень тяжело, потому что это очень индивидуально. Кто-нибудь перечеркивает твой текст красной ручкой и объясняет, в чем ты неправ. Это отнимает очень много времени. Вы знакомы с советом Хэмминга молодым исследователям?

Сейбел: Из доклада «You and Your Research» (Вы и ваше исследование)?

Армстронг: Хэмминг говорит примерно так: «Делайте правильные вещи. Если вы не делаете правильные вещи в правильных областях, то неважно, что именно вы делаете». Еще он говорит: «Один день в неделю я обязательно осваиваю что-то новое. То есть трачу на освоение нового на 20% больше, чем мои коллеги. Выходит, через 4,5 года я буду знать вдвое больше них, а с учетом сложных процентов получается, что через 5 лет я буду знать втрое больше». Не помню точно, какие там были цифры. По-моему, это верно. Поскольку я занимаюсь исследованиями, то на

освоение чего-то нового трачу не 20% времени, а 40%. Я занимался ими 30 лет. И понял, что знаю очень много. Вы спрашивали, как улучшить навыки программирования? Тратьте 20% своего времени на узнавание чего-нибудь нового. Прочтите Хэмминга, он очень хорошо пишет.

Сейбел: Бывало так, что вы находили какой-то код просто красивым?

Армстронг: Да, но почему – не знаю. Любопытно, что если двум программистам дать одну и ту же задачу – конечно, смотря какую, я говорю о задачах скорее математического свойства, – то они с большой вероятностью напишут одинаковый код. Если произвести форматирование, переименовать переменные и функции, получится одно и то же – одинаковые алгоритмы. Мы создаем что-то – или просто снимаем покрывало? Это как со статуей. Мы снимаем покрывало и обнаруживаем, что алгоритм всегда был там. Изобретаем ли мы новый алгоритм или уже существующую структуру? С некоторыми алгоритмами, в основном математическими, такое случается. Но, скажем, если я занимаюсь реализацией протокола для телефонии, такого чувства нет. Нет ощущения, что я снимаю покрывало со статуи.

Сейбел: Похожий случай с красотой математических задач – это часть природы. Но есть и другие уровни, на которых код доставляет эстетическое наслаждение.

Армстронг: Да. Это вроде фэн-шуй. Я люблю компактный код, хорошо сбалансированный и структурированный. Удаляешь одно, другое, и настает момент, когда удалить больше ничего нельзя – программа не будет работать. Вот в этот момент она прекрасна. Любое возможное изменение может ухудшить алгоритм, но сейчас он прекрасен.

Сейбел: Вы говорили о том, как с Робертом Вирдингом пересылали друг другу куски кода. Как каждый из вас изменял низкоуровневые детали форматирования, по поводу которых программисты без конца спорят?

Армстронг: Они не затрагивают красоту алгоритма.

Сейбел: Но это часть эстетики, это вкус.

Армстронг: Да, но я не назову код уродливым, потому что в нем есть пробел после запятой. Уродливое – это линейный поиск, когда можно применить деление пополам, или логарифмический метод. Делайте линейный код, если у вас список из десяти элементов, – пожалуйста! Но при больших структурах данных применяйте бинарный поиск. В линейном виде это выходит некрасиво. Математические алгоритмы – как платоновские идеальные сущности. Что-то близкое к архитектуре. Вы любуетесь красивым зданием, которое не есть математический объект. Это не шар, не сфера, не призма – это небоскреб. Но выглядит он великолепно.

Сейбел: Что делает человека хорошим программистом? Нанимая на работу программиста, на что вы смотрите?

Армстронг: Как человек выбирает задачи. Что для него важнее – задача или решение? Мне симпатичнее человек, который говорит: «У меня была вот такая занятная задача». Вы спрашиваете, в каком самом интересном проекте он участвовал, просите показать соответствующий код, спрашиваете, как он решал задачу. Для меня не столь важно, хорошо ли соискатель знает язык А или язык Б. Обычно программисты всеми языками владеют одинаково хорошо – или плохо. Хороший Си-программист будет успешно работать и с Erlang, это очень надежный показатель. Есть, конечно, исключения, но умственные навыки, необходимые для работы с одним языком, можно применить и к другим.

Сейбел: В некоторых компаниях соискателям дают решать логические задачи. Вы так делаете?

Армстронг: Нет. Есть очень хорошие программисты, которые в них собираются туго. Помню одного из парней, работавших с Erlang, – у него была степень по математике, и он напоминал алмазный бур, прогрызающий гранит. Как-то раз он загрипповал и взял домой листинги из Erlang. Потом он пришел, добавил в код атомарное выражение и сказал: «Это поставит эмулятор на замкнутый цикл». Он обнаружил, что изначальное хеш-значение этого выражения равно нулю, и взял какой-то модуль для перехода к следующему выражению – оно тоже оказалось равным нулю. И он декомпилировал хеш-алгоритм для одного патологического случая. Он даже не запускал программы – посмотреть, как они работают: он *читал* их. Но медленно. Довольно медленно. Не знаю, как бы он решил по-быстрому логическую задачу.

Сейбел: Есть ли еще навыки, необходимые хорошему программисту?

Армстронг: Где-то я читал, что нужно иметь хорошую память. Думаю, это так.

Сейбел: Билл Гейтс однажды сказал, что до сих пор мог бы выйти к доске и написать большие фрагменты кода на Бейсике, который писал для Altair, хотя прошло уже лет десять. А вы можете вот так вспомнить свой старый код?

Армстронг: Да, кое-что восстановить по памяти могу. Иногда я забываю какой-нибудь старый код и совершенно не беспокоюсь – просто пишу его заново, и получается примерно то же самое. Могут измениться имена переменных, расстановка функций. Но код выйдет изоморфным. Или даже лучше, чем раньше, потому что я дополнительно его обдумал.

Возьмем, к примеру, сопоставление образцов в моем компиляторе десятилетней давности – я могу сесть и набрать его. Он будет отличаться

от старой версии, но в лучшую сторону. Код как бы самоулучшается со временем. Но структура остается прежней.

Я не беспокоюсь о потере кода – если шаблоны в голове, я все вспомню. Даже не столько вспомню, сколько сделаю заново. Не думаю, что это припоминание, скорее воссоздание. Если Билл может вспомнить именно сам текст, то я – нет. Но, конечно, я долго помню структуру кода.

Сейбел: Скажите, обмен сообщениями в духе Erlang поможет раз и навсегда решить проблему параллельного программирования?

Армстронг: Нет, конечно. Это всего лишь усовершенствование, пусть и большое, по сравнению с разделяемой памятью. Думаю, Erlang выполнил, по крайней мере, одну задачу – продемонстрировал это. Работая над Erlang, мы всегда говорили на конференциях: «Надо будет копировать все данные». И слушатели, кажется, воспринимали наши доводы насчет безотказной работы системы – копирование всех данных вводилось для этого. Нам возражали: «Но ведь тогда все будет работать страшно медленно», – а мы отвечали: «Зато безотказно».

Удивительно, но наш язык при определенных обстоятельствах оказывается еще и более быстрым. И то, что мы делали ради безотказной работы, во многих случаях было так же эффективно, а то и лучше, чем с разделяемой памятью. Мы спросили себя: «Почему так происходит?» Потому что улучшается распараллеливаемость. В системе с общей памятью нужно блокировать данные при обращении к ним. А блокировка обходится дорого. И потом, возможно, копировать придется не так уж много данных. Если данных немного, то копирование лучше, чем бесчисленные обновления, обращения, блокировки. А на многоядерниках, если принять старую разделяемую модель, блокировка может остановить работу всех ядер. У вас тысячеядерный процессор, одна программа делает глобальную блокировку – и тысяча ядер прекращают работу.

Я также очень сомневаюсь в неявном параллелизме. В языке могут быть параллельные конструкции, но если они не отображаются на параллельном процессоре, а только эмулируются системой, – преимущества никакого. Вообще, есть три вида процессорного параллелизма.

Есть конвейерный параллелизм, когда в чипе создается более длинный конвейер, чтобы делать вещи параллельно. Это предопределено при создании чипа. Обычный программист не может сделать ничего с параллелизмом на уровне инструкций.

Есть параллелизм данных; это не настоящий параллелизм, скорее он связан с процессами в кэше. Если вы хотите, чтобы программа на Си выполнялась эффективно, и если $*p$ находится на 16-байтовой границе, то при доступе к $*p$ доступ к $*(p+1)$ практически бесплатен, поскольку

строка кэша вытягивает его. Вам надо знать, насколько широки строки кэша, сколько байтов вы сможете утянуть при одном перемещении в кэш? Этот вид параллелизма можно успешно использовать, если быть очень осторожными со структурами данных и точно знать, где что находится в памяти. Все запутанно, и разбираться с этим не очень охота.

И наконец, многоядерные процессоры. К концу десятилетия они будут 32-ядерные, а к 2019 году ядер будут миллионы. Поэтому вам нужно взять параллельные участки своей программы и отобразить на ядра компьютера. Согласен, это довольно громоздкая операция. Вычисление начинается в другом ядре, из него потом приходит ответ – это требует времени. Если надо просто сложить два числа, это не стоит усилий – больше времени потратится на перенос данных между ядрами, чем на то, чтобы сделать все на месте.

Erlang неплохо приспособлен для этого. Программист говорит: «Мне нужен процесс, и еще, и еще». Процессы распределяются между ядрами. Может быть, стоит подумать над их физическим распределением между ядрами. Не исключено, что процесс, порождающий другой процесс, обменивается с ним данными. Имеет смысл поместить его в то ядро, которое ближе. Если же обмен данными не очень интенсивен, то можно поместить и подальше. Процессы, занимающиеся вводом/выводом, стоило бы расположить у края чипа, с другими процессами, занимающимися вводом/выводом. Чипы становятся все больше, и надо задуматься над тем, что размещение данных в середине чипа обходится дороже, чем на краю. Если есть три сервера и база данных, то ее мы поместим в середину, а серверы, которые общаются с клиентами, на край. Но все это подлежит исследованию.

Сейбел: Вам дорога мысль о том, что Erlang – это средство, позволяющее организовать параллельные процессы. Что вам дороже – идея параллелизма на основе обмена сообщениями без разделения данных или Erlang как язык?

Армстронг: Конечно, идея. Меня спрашивают: «Что будет с Erlang? Станет ли он популярным языком?» Этого я не знаю. Думаю, он уже стал важным языком. Он может повторить судьбу Smalltalk – очень важного языка, который имел горячих сторонников, но широко не применялся. С Erlang может произойти то же самое. Возможно, заложенные в нем идеи придут к миллионам пользователей, если их реализует Microsoft в Common Language Runtime, добавив там-сям фигурных скобок.

7

Саймон Пейтон-Джонс

В 1987 году он стал одним из инициаторов проекта, в результате которого появился язык программирования Haskell. Сегодня Саймон Пейтон-Джонс – ведущий исследователь лаборатории Microsoft Research, находящейся в британском Кембридже. В 1998 году он выпустил переработанное описание языка Haskell, являющееся текущим стабильным описанием. Кроме того, Пейтон-Джонс – ведущий разработчик Glasgow Haskell Compiler (GHC), «стандартного компилятора де-факто», согласно сайту haskell.org. Он же снабдил язык часто приводимым официальным девизом «Избегать успеха любой ценой».

Влиятельный исследователь и бывший преподаватель, так и не получивший кандидатской степени, Пейтон-Джонс ценит как практическую, так и теоретическую красоту. Он учился программировать на машине без постоянного места хранения информации и всего со 100 ячейками памяти, а будучи студентом, писал высокоуровневые компиляторы для большого компьютера колледжа и одновременно собирал собственные простейшие машины за счет своих скудных средств. Пейтон-Джонс занялся функциональным программированием после того, как на занятии преподаватель показал способ создания цепных списков без использования мутации и раскрыл красоту «ленивых» вычислений. В функциональном программировании Пейтон Джонс увидел элегантный и дерзкий вызов всей индустрии создания программ, возможность не добавлять еще один кирпич

к стене, а строить новую стену. В 2004 году был избран в члены Ассоциации вычислительной техники за «вклад в функциональные языки программирования».

Мы говорили с ним о том, почему, по его мнению, функциональное программирование обещает изменить методы создания ПО, почему транзакционная память лучше приспособлена для параллельных программ, чем блокировка и переменные условия, и почему даже в Microsoft Research трудно исследовать вопрос о влиянии языка на эффективность работы программиста.

Сейбел: Когда вы научились программировать?

Пейтон-Джонс: В школе. Intel только что выпустила 4004 – первый в мире микропроцессор. Но у нас ни 4004, ни чего-нибудь похожего не было – любители в то время могли достать его лишь с большим трудом. Был только школьный компьютер IBM – странная машина, собранная из каких-то запчастей. Системы постоянного хранения данных не было, и программу всякий раз приходилось набирать с нуля.

Всего было 100 ячеек памяти, в каждой из которых, кажется, хранились восьмизначные десятичные числа. Там располагались и программа, и ваши данные. Искусство заключалось в том, чтобы уместить программу в эти 100 ячеек. Не помню, как именно я написал свою первую программу; мы с одним парнем все время торчали за этим компьютером. Мне тогда было пятнадцать – 1973 или 1974 год.

Освоив немного эту машину, мы выяснили, что в Суиндонском техническом колледже есть компьютер. Мы стали ездить туда в один из выходов на невероятно медленном автобусе – поездка занимала около часа. То был Elliot 803; он помещался в особой комнате в полудюжине белых шкафов размером с холодильник, и управляла им женщина-оператор в белом халате.

Скоро та женщина поняла, что мы умеем обращаться с компьютером, и стала оставлять нас работать одних. Там использовались бумажные перфоленты и телетайп, в машину вводилась перфолента с программой. Мы писали на Алголе, который стал моим первым высокоуровневым языком. Писалась программа на перфоленте и правилась на ней же. Если надо было что-то изменить, мы прогоняли ленту через телетайп, распечатывали новую, останавливали ее в нужном месте и печатали, что требовалось, – очень трудоемкий способ. Что-то вроде строкового редактора с физическим носителем информации. Вот такими были мои первые опыты программирования. Я загорелся этим делом.

Сейбел: В школе ведь не было никаких уроков.

Пейтон-Джонс: Конечно, нет! Совсем ничего, никаких компьютеров не было в программе.

Сейбел: То есть просто – «Эй, ребята, вот вам компьютер, действуйте».

Пейтон-Джонс: Именно так. Он помещался в большом шкафу, обычно закрытом. Но мы брали ключ, включали его – и вот вам дисплей, показывающий только то, что есть в регистрах, и десятичные номера – содержимое ячеек памяти. Вводишь программу, нажимаешь Go. И можно было пройти ее пошагово. Не было даже языка ассемблера за отсутствием ASCII-символов. В буквальном смысле слова машинный код, отображаемый при помощи десятичных чисел, даже не шестнадцатеричных.

Сейбел: Но экран-то был?

Пейтон-Джонс: Телевизионный экран. Единственное средство вывода информации.

Сейбел: А для ввода?

Пейтон-Джонс: Нечто вроде сенсорной клавиатуры, никаких механических кнопок – довольно сложное устройство. Надо было только коснуться кнопки. Всего их было десятка два.

Сейбел: Только для чисел?

Пейтон-Джонс: Да, и еще кнопки Go и Step. И еще – «показать данную ячейку памяти». Все крайне примитивно, но оттого раздражало еще больше.

Сейбел: Раз так, выходит, программу надо было продумывать заранее, в мелочах, до того как вводить ее в машину.

Пейтон-Джонс: Сначала мы рисовали блок-схему, потом разбивали ее на отдельные инструкции, переводили инструкции в этот странный цифровой формат и вводили цифры. Программа состояла обычно из 800 цифр. Потом мы нажимали Go. Если везло, мы не ошибались ни в одной из этих 800 цифр, и все работало. Поэтому у нас уходило много времени: один смотрел на экран и проверял, другой говорил: «Переходи к следующей ячейке».

Затем я поступил в Кембриджский университет; эра микропроцессоров только начиналась. В университете был компьютерный клуб. Там стояла большая машина под названием Phoenix с исключительно замысловатой системой учета ресурсов.

Было очень важно, в какое именно время ты пользовался компьютером. Тебе давали некоторое количество единиц машинной «валюты», и чем больше памяти или времени у тебя уходило, тем больше их рас-

ходовалось. Соответственно, чем меньше ресурсов отнимала твоя программа, тем меньше ты тратил. Мы, зеленые студенты, которым этих единиц давали мало, просиживали там ночи напролет – после девяти вечера все стоило дешевле.

И мы с девяти вечера до трех ночи торчали там, сочиняя программы. На чем мы писали? На BCPL. Опять сплошная самодеятельность, как видите. В то время я занимался математикой и формально совсем не учился компьютерным наукам.

Тогда, в 1976–1979 годах, не было степени бакалавра как таковой. В последний год можно было выбрать специализацию, например компьютерные науки. Но нельзя было заниматься одним предметом все три года – надо было еще изучать, скажем, математику, естественные науки. Я занимался математикой и закончил со специальностью «Науки об электричестве». Тогда я считал, что компьютеры – так, игрушка, а специализация должна быть серьезной.

Но математика оказалась нелегким делом, ведь в Кембридже отличная математическая школа. Так что я переключился на электричество.

Сейбел: «Науки об электричестве» – это то, что в Америке называется «Электротехника»?

Пейтон-Джонс: Совершенно верно. Мой школьный приятель Томас Кларк тоже учился в Кембридже. Мы с ним собирали компьютеры. Покупаешь микропроцессор, много транзисторов серии 7400 и соединяешь их проводами. Большой проблемой оставались принтеры и экраны. С ними было трудно.

Сейбел: Стоили очень дорого.

Пейтон-Джонс: Да, очень дорого. Части электрической схемы студент еще мог купить, но принтеры... Это были большие линейные принтеры размером с холодильник. Внутри было много механических частей, так что эти устройства оказывались нам не по карману. И еще устройства для хранения данных, любые. Поэтому мы старались обходиться клавиатурой, экраном, ну, еще простенький лентопротяжный механизм.

Сейбел: Значит, в 1976–1979 годах вы собирали компьютеры из подручных средств. Но ведь как раз в это время выпустили Altair.

Пейтон-Джонс: Да. Самodelки уже выходили из моды. Но нам просто нравилось все это.

С этими нашими машинами была еще одна сложность – программы. Самое продвинутое, что можно было загрузить, – конвеевская игра «Life» (Жизнь). Она работала прекрасно. Но что-то серьезное, вроде языка программирования, требовало слишком много работы – у нас

были крохотные постоянные хранилища данных. Ну и, кроме того, все писалось в шестнадцатеричном коде, никакого ассемблера.

Сейбел: Машинный код в чистом виде.

Пейтон-Джонс: Конечно, большой компьютер Кембриджа понимал BCPL, и мы писали много программ на BCPL. Еще мы начали писать компилятор для языка, который сами изобрели, но забросили – слишком сложно выходило. То были два мира, которые не сообщались между собой. С одной стороны, мы писали компиляторы для большого компьютера на высокоуровневом языке, с другой – возились с железом.

Сейбел: Помните вашу первую интересную программу?

Пейтон-Джонс: Программа для извлечения 24-значных квадратных корней и помещения их в 99 ячеек памяти – это еще для школьного компьютера.

Сейбел: И одна оставалась в запасе!

Пейтон-Джонс: Правильно. Что-то вроде метода Ньютона–Рафсона для квадратных корней. Я страшно гордился этим. Что же было потом? Потом, наверное, тот самый компилятор, который мы забросили, на BCPL. Мы многое с ним связывали и разрабатывали его детально. Системы типизации не было, так что у нас имелись только громадные листы распечаток с картинками, схемами и стрелками.

Сейбел: В BCPL не было системы типизации?

Пейтон-Джонс: Нет. Поэтому мы рисовали типы на больших листах бумаги и ставили стрелки. Это и была наша система типизации. Программа оказалась слишком большой для наших возможностей, мы ее так и не закончили.

Сейбел: Вы извлекли уроки из этой неудачи?

Пейтон-Джонс: Я впервые понял, что если пишешь очень большую программу, то она просто не умещается у тебя в голове. До того все, что я писал, целиком помещалось в голове без особых проблем. И вот тогда я сделал первую попытку создать долговременную документацию.

Сейбел: Но даже ее оказалось недостаточно?

Пейтон-Джонс: Ну, у нас были и другие заботы – надо было получать степень бакалавра. А компьютерами мы занимались по вечерам и ночью.

Сейбел: Что вам не нравится в том, как вы учились программированию?

Пейтон-Джонс: Никто меня этому не учил. Но не уверен, что это большой пробел. Самый большой пробел – то, что я на глубинном уровне так и не приобрел познаний в объектно-ориентированном программи-

ровании. Нет, я знаю, конечно, как писать объектно-ориентированные программы. Но делать это в большом масштабе – дело совсем другое. Если писать большие программы, использовать сложным образом разные иерархии классов, создавать фреймворки – вот тогда приходит глубинное понимание. Это не то же самое, что прочесть книгу.

Я считаю это пробелом, так как не могу авторитетно высказываться насчет того, что можно и чего нельзя делать в объектно-ориентированном программировании. Я всегда очень осторожен в своих высказываниях, особенно стараюсь не говорить отрицательно об императивном программировании – это невероятно сложная и богатая парадигма программирования. Но жизнь сложилась так, что мне ни разу не пришлось в течение нескольких лет писать большие программы на C++. Так приобретаются глубинные познания на уровне рефлексов, и у меня их нет.

Сейбел: Что было после Кембриджа?

Пейтон-Джонс: Я подумал: «Надо бы поработать с компьютерами». И я год занимался послеуниверситетской подготовкой по компьютерным наукам; это мое единственное официальное образование в компьютерной области.

Сейбел: Что-то вроде магистратуры?

Пейтон-Джонс: Что-то вроде магистратуры. То был очень полезный для меня год. Думаю, это примерно соответствовало сегодняшней степени бакалавра по компьютерным наукам, но было рассчитано на студентов, которые начинают с нуля.

Сейбел: Вы несколько лет провели в индустрии, прежде чем вернуться в науку. Чем вы занимались?

Пейтон-Джонс: Это была крошечная компания. Мы производили оборудование и программы для компьютеров, которые устанавливались в приборы измерения веса на ленточных транспортерах. Помню, я спроектировал штуку, наблюдавшую за ячейкой загрузки углепогрузочного транспортера. Она контролировала скорость ленты в зависимости от наполнения ячейки, чтобы соблюдать норму перемещения угля за единицу времени. Небольшая система, но оперировавшая в реальном времени; я использовал для нее PL/Z, слегка похожий на Алгол. Я писал программу на компьютере Z80 с операционной системой Chromix – урезанный UNIX, так сказать.

В компании обычно работало человек шесть, временами больше – до пятнадцати. Из-за ее крошечных размеров все было довольно ненадежно. Денег то было много, то совсем не было. После двух лет я решил, что предпринимательство не для меня. Это был мой главный опыт, если говорить о мелких компаниях; чтобы стать успешным предпринимателем, надо извлекать энергию из стрессовых финансовых ситуаций,

а у меня в таких случаях энергия, наоборот, расходуется. Мой начальник был управляющим директором компании. Чем хуже шли дела, тем энергичнее он выглядел. Он бегал вокруг, порождал новые идеи насчет программ, трудился как пчела и был счастлив. Именно так и нужно себя вести, а если стресс высасывает из тебя энергию, то постоянно ходишь как вареный.

Я решил, что столько напряженной работы не для меня, стал искать работу и пошел преподавателем в Лондонский университетский колледж. У меня не было ни степени, ни опыта исследовательской работы. И декан факультета облегчил мне нагрузку, чтобы я мог заниматься исследовательской работой. Но у меня не было ни малейшего понятия насчет того, что делать. Я сидел в своем кабинете перед чистым листом бумаги и ждал Великих Идей. В полной тишине я обводил комнату взглядом в поисках Великих Идей. И все.

Джон Уошбрук, старший научный специалист факультета, взял меня под свое крыло и сказал кое-что очень важное. «Начни хоть с чего-нибудь, пусть даже мелкого», – сказал он. Это относилось не к программированию, а к исследованиям. Пусть тема будет мелкой, неоригинальной, маловажной – надо взять и написать статью. Я так и сделал. Совет Джона очень много значил для меня.

Я повторял его потом каждому аспиранту. Так и нужно начинать. Как только все завертелось, компьютерные науки превращаются в некий калейдоскоп – все интересно, потому что предмет развивается быстрее тебя. Это не какая-то неподвижная вещь, которую ты изучаешь, а то, что постоянно расширяется и приближается.

Сейбел: Итак, вы вернулись в науку, но степень так и не получили. Почему?

Пейтон-Джонс: Сейчас занять должность на факультете без степени было бы крайне трудно. Тогда – это был 1982, 1983 год – я подал заявку в Лондонский университетский колледж. Моя сестра изучала там компьютерные науки и сказала мне: «Там читают лекции по этой теме, почему бы тебе не пойти туда?» И меня, к моему удивлению, приняли. Видимо, тогда преподавателей не хватало, поэтому принимали каждого, кто хоть как-то зарекомендовал себя в этой области. Иначе как бы они наняли человека без степени?

После семи лет преподавания в колледже я начал задумываться о степени. Однако писать диссертацию – это так мучительно! Выяснилось, что в Кембридже можно получить степень особым образом – представить опубликованную работу и, если повезет, станешь доктором. Я начал хлопотать об этом, но тут получил место профессора в Глазго. Теперь уже никому не было дела, есть у меня степень или нет, и я оставил эту

мысль. У Робина Милнера ее тоже нет, так что я оказался в неплохой компании. На этом все и закончилось.

Сейбел: А в наши дни степень имеет значение? Кто-то говорил мне, что это зависит от призвания – если хочешь быть преподавателем, то она нужна, но если нет, то не имеет смысла. Применимо ли это к компьютерным наукам?

Пейтон-Джонс: Совершенно верно. Это необходимое, хотя и недостаточное условие, если делать исследовательскую карьеру в академических учреждениях или, например, в лабораториях Microsoft Research либо Google, то есть в научных центрах крупных компаний. Без степени вы застрянете на стартовой отметке.

Если же вы не намерены заниматься исследованиями, то прислушайтесь к внутреннему голосу. Если работаешь над чем-то с воодушевлением, продуктивность возрастает пятикратно. Если вам нравится что-то и вы хотите всерьез в этом копаться, написание диссертации – фантастическая возможность провести в Британии три года, а в Штатах еще больше, занимаясь исследованиями. Невероятная свобода, потому что в некотором роде паразитируешь на обществе. Если нет желания полностью посвящать себя науке, можно сесть за диссертацию просто потому, что пытлив и полон энтузиазма. Но, вообще говоря, странная это вещь – работать для себя и писать пухлую работу, которую не прочтет почти никто, – люди будут читать только ваши статьи. Это нестандартный способ заниматься исследованиями.

Получив степень, вы начинаете работать в сотрудничестве с другими людьми над более мелкими проблемами. По-моему, написание диссертации – странный способ подготовиться к дальнейшей карьере, даже исследовательской. Совсем странный в Британии, поскольку у вас очень мало времени. В Штатах, мне кажется, иначе – там можно плодотворно поработать с другими, пока не сосредоточишься на собственных исследованиях.

Сейбел: Раз уж речь зашла о науке, надо сказать, что функциональное программирование популярно среди исследователей, но остальные часто считают его слишком близким к математике и далеким от типичных программистских задач. Это верно?

Пейтон-Джонс: Наполовину. Для меня функциональное программирование – чисто функциональное программирование, где побочные эффекты выделены в свой собственный мир; это элегантный и дерзкий вызов всей индустрии создания программ. Дерзкий, поскольку знаменует собой разрыв с тем, что есть.

А что у нас есть сейчас? Есть крупные компании, которые тратят огромные деньги на экосистемы, редакторы, профилирование, инструмен-

ты, программистов, повышение квалификации и так далее. Те, кто делает погоду, мыслят сугубо практично. Ну, а элегантное и дерзкое функциональное программирование намного менее обеспечено такой инфраструктурной поддержкой. Но погоня за этой поддержкой не всегда оправдана. Ведь только пока кто-то занимается чем-нибудь элегантным и дерзким, вы можете изменять господствующую тенденцию в лучшую сторону. Но вы никогда не достигнете тех же высот, что они.

Поэтому, на мой взгляд, прелесть академических исследований в том, что ученые могут с головой уйти в свои безумные вещи без необходимости отвечать на вопросы о том, что это дает на практике. Что-то окажется потом фантастически важным, что-то не очень, но ничего предсказать заранее невозможно. Главное оправдание для людей вроде меня, которые занимаются чисто функциональным программированием, состоит в том, что с ним связаны большие надежды. Программисты не обязательно пойдут именно по этой дороге, но с функциональным программированием связаны большие надежды. Песчаник императивного программирования выветривается, и под ним обнаруживается гранит функционального программирования.

При этом чисто функциональное программирование первоначально было слишком эксцентричным и академичным, слишком тесно связанным с математикой. За последние 20 лет – те, что я с ним работаю, – оно становится все более ориентированным на практику, не обращаясь к абстрактным идеям, а пытаясь убрать одно за другим препятствия, мешающие реальным программистам использовать функциональные языки для создания приложений. Развитие Haskell может служить примером.

Хорошо, что есть люди, может быть, слегка непрактичные, которые идут впереди общей массы. Перспективы чисто функционального мира могут стать для этой общей массы путеводной звездой. Собственно, так и происходит. Многие в системах типизации и обобщенных типах изначально было разработано в контексте языков функционального программирования – они послужили своего рода лабораторией. Еще один пример: генераторы и «ленивые» потоки. В Python есть списковые выражения на синтаксическом уровне. Есть много индивидуальных находок. Если они переходят на текущий уровень программирования, то получают новые названия, слегка видоизменяются. Не хочу показаться великим поставщиком идей, но здесь действительно есть кое-что, нигде до того не реализованное. Так что это сослужило свою службу.

Сейбел: Какова, по-вашему, связь между исследованиями и собственно программированием?

Пейтон-Джонс: Они активно взаимодействуют. Моя область исследований – языки программирования. Для чего, в конце концов, они создаются? Чтобы легче было программировать. Язык – не что иное, как

пользовательский интерфейс программы. Поэтому программирование и исследования в области языков тесно между собой связаны. У нас пока плохо получается наблюдать за программистами. Нужно проводить формальное изучение того, как программисты пишут программы, чтобы понимать, что они делают. Это очень затратно, и к тому же деятельность у программистов довольно нечеткая – результат не всегда выходит однозначный.

Культура сообществ вокруг языков программирования ориентирована на доказательство качества и полноты систем типов. Мы же стараемся ответить на более важный, но и более сложный вопрос – делают ли они людей более продуктивными? На него, однако, трудно дать убедительный ответ. Что будет эффективнее для выполнения одной и той же операции – функциональная или объектно-ориентированная программа? Даже если вы потратите кучу денег на серьезные эксперименты, сомневаюсь, что люди будут заинтересованы в их результатах.

Сейбел: А вы делали опыты хотя бы в небольшом масштабе? Вы работаете на Microsoft, у которой полно денег. Почему не посадить за одну и ту же задачу команду опытных Haskell-программистов и команду опытных C#-программистов? Ведь вам именно это нужно?

Пейтон-Джонс: Да, вы правы. Отчасти это вопрос денег. Но не только: это также вопрос времени и внимания. Для такого эксперимента нужна новая методология. Нужно повернуть по-другому свое сознание. Кроме того, со стороны кажется, будто у нас много денег, но стандартная картина для Microsoft – это одинокий исследователь за своим компьютером. Мы не можем просто так взять и потратить деньги на какой-нибудь проект. Если бы могли, было бы здорово. Ближе к переднему краю стоят лаборатории в Редмонде – там исследуются прототипы продуктов. Новые версии Visual Studio проходят там широкую проверку на потребительские свойства.

Сейбел: Вероятно, там больше занимаются взаимодействием с пользователями, чем языками программирования.

Пейтон-Джонс: Они также делают кое-что интересное в плане тестирования API. Стивен Кларк и его редмондские коллеги стараются регулярно наблюдать за программистами, получившими новый API, – о чем те говорят, что пытаются сделать. Проектировщики данного API сидят за стеклянной перегородкой и наблюдают.

Иногда эти проектировщики говорят программистам: «Нет-нет, не делайте так! Это неправильно!» Часто такие ситуации бывают очень поучительными. Потом API меняют там, где нужно. Честно говоря, в этом отношении исследования языков не столь продвинулись. Отчасти потому, что там нужно решать более сложные проблемы. Кроме того, в куль-

турном смысле мы еще мало к этому адаптированы. Я считаю это нашей слабостью. И я не чувствую лично себя в состоянии предложить для этого решение.

Сейбел: Если исследователи выдвигают интересные идеи насчет улучшения процесса программирования, достаточно ли быстро они внедряются на практике?

Пейтон-Джонс: В кратчайшее время... Даже не знаю. Я часто разговариваю с людьми, которые занимаются производством продуктов, нужных покупателю, – покупатель готов платить за них. И многое из того, что заботит меня, лежит вообще вне поля их зрения.

Им надо сделать то, что потребитель готов оценить сегодня. У них просто нет времени заморачиваться с тем, что может работать в принципе или даже с тем, что может работать прямо сейчас, но пока еще не совсем доделано.

Тут есть небольшой разрыв – снова старая задача про курицу и яйцо. Порой идеи, развитые исследователями, требуют дополнительных инженерных усилий – не фундаментальных исследований, а приспособления к практическим нуждам.

Я не хотел бы утверждать, что компьютерные «практики» зомбированы и не желают внедрять хорошие идеи, которые могут облегчить им жизнь. То, что они делают, имеет под собой основания. Расстояние между прототипами и практически применимыми вещами часто бывает большим. Думаю, Microsoft здесь неплохо справляется. Microsoft Research позволяет сократить это расстояние и располагает кое-какими механизмами – инкубационными группами и так далее, – которые сближают исследователей и разработчиков ПО, позволяют пересечь разделяющую их границу. Поэтому я считаю, что MSR полезен настолько, насколько дает возможность преодолеть эту границу.

Если копнуть глубже, встают новые вопросы. Для разработчика-практика, имеющего дело с Java, дело не только в том, что функциональное программирование – принципиально иной подход. Дело еще и в способности программ к взаимодействию. Достаточно ли книг и библиотек? Способ программирования формирует вокруг себя целую экосистему – люди, навыки, библиотеки, операционные среды, инструменты и так далее.

Если таких камней преткновения довольно много, застреваешь. Мне кажется, различные элементы исследовательских подходов к языку живут в разных точках спектра. Некоторые довольно близки к тому, что уже есть. Вы можете сказать, что эта штука подключается прямо в фреймворк, не требует модификации Java, это инструмент статического анализа, позволяющий найти ошибки в коде. Ура! Такие вещи

воспринимаются куда легче, чем «вот вам полностью новый способ программирования».

Если же говорить конкретно о функциональном программировании, то отношение к нему сильно изменилось в последнее время. Гораздо больше народа знают теперь, что это такое. Уже не приходится всякий раз объяснять, что такое Haskell. Некоторые говорят: «Я читал про него недавно на Slashdot, по-моему, это круто». Еще несколько лет назад такого не было.

Но что за этим реально кроется? Случайный всплеск интереса? Или просто все больше бывших студентов, узнавших в университете о функциональном программировании, занимают теперь менеджерские и руководящие должности? Возможно, и так. Но, может быть, причина в том, что с усложнением программных систем становится все важнее справляться с последствиями неконтролируемых побочных эффектов, иметь больше гарантий корректности и лучше использовать параллелизм. Думаю, стрелка на шкале «цена–качество» постепенно смещается.

Сейбел: А как вы сами начали заниматься функциональным программированием?

Пейтон-Джонс: Я ничего о нем не знал до последнего года в Кембридже, когда прослушал небольшой курс Артура Нормана. Норман был блестящим, слегка эксцентричным лектором. Он интересовался символической алгеброй, так что хорошо разбирался в Лиспе. На лекциях он объяснял нам, как составлять двунаправленные списки без каких-либо побочных эффектов. Отлично помню это, потому что впервые столкнулся с такой удивительной вещью – составляешь список, выделяешь ячейки и заполняешь их так, чтобы они указывали друг на друга. Кажется, будто *нужно* как-то использовать побочные эффекты.

Но он показал, как в чисто функциональном языке делать это без побочных эффектов. Я понял, что функциональное программирование, о котором я тогда знал очень мало, – это средство создания интересных программ, а не только безделушек.

Сейбел: Думаю, многие после такой лекции сказали бы: «Занятно!» – и вернулись к своему VCPL. Почему же вы пошли дальше, занялись исследованиями, стали объяснять людям, как пользоваться этими программами?

Пейтон-Джонс: Еще сыграли роль статьи Дэвида Тернера о комбинаторах S и K, которые помогают преобразовывать и затем выполнять лямбда-вычисления. О лямбда-вычислениях я знал немного больше – тогда эта идея набирала популярность. Тернер показывал, как преобразовывать лямбда-вычисления в три комбинатора: S, K и I, которые представляют собой закрытые лямбда-термы. Фактически речь шла

о том, чтобы преобразовать сколь угодно сложные лямбда-термы в эти три комбинатора. Можно даже обойтись без I , так как он равен SKK .

Довольно странное преобразование – берешь лямбда-терм, который хоть как-то понимаешь, и получаешь набор из S и K , в котором не понимаешь ничего. Но вот применяешь их к аргументу, и случается чудо – ответ выходит тот же, что и с применением изначального терма. Что-то очень умное – для меня по тем временам невероятное. Но это всегда работает!

Я обратился к функциональным программам по вдохновению. Отчасти, думаю, потому, что был связан с «железом», а это выглядело способом *реализации* лямбда-вычислений. Ведь сперва кажется, что в них вообще нет механизма реализации, что это чистая математика, далекая от компьютеров. А S - K комбинаторы, как мне показалось, позволяют применять их на практике – и так оно и было.

Сейбел: Значит, вы решили, что достаточно заложить эти комбинаторы в машину, а потом уже на их основе выполнять операции?

Пейтон-Джонс: На самом деле именно этим занимались мои друзья. Уильям Стой, Томас Кларк и еще несколько человек создали компьютер $SKIM$, или SKI Machine, который напрямую выполнял S и K . Я не участвовал в их проекте, но тогда все развивалось в этом направлении. Статья Джона Бэкуса «Can programming be liberated from the von Neumann style» (Может ли программирование освободиться от стиля фон Неймана) была невероятно популярна. Он получил Премию Тьюринга за эту статью, и он – изобретатель Фортрана – фактически заявил: «Будущее за функциональным программированием».

Он заявил и другое: «Возможно, для этих программ придется развивать новую компьютерную архитектуру». Как видите, за функциональное программирование высказывались очень влиятельные люди, и мы как сумасшедшие цитировали статью Бэкуса. $SKIM$ тоже вписывался в этот процесс. Мы полагали, что нестандартная реализация – или, по крайней мере, нестандартный подход к программам – даст нам совершенно новые виды компьютерной архитектуры. Это увлечение – «радикально новая архитектура для функционального программирования» – длилось все 1980-е. Пожалуй, мы пошли немного не той дорогой, но все это было крайне захватывающим.

Ленивые вычисления тоже раззадоривали нас. Сейчас я думаю, что ленивые вычисления – это здорово, но тогда они казались основой всего. Ленивые вычисления основаны на той идее, что функция не вычисляет свои аргументы. И снова движущей силой было наше желание сделать что-то элегантное, необычное, принципиально новое.

Иногда хорошо дать волю воображению – получаешь совершенно иной подход к программированию. Не добавляешь еще один кирпич к сте-

не, а строишь новую стену. Это так увлекательно! Во всяком случае, именно это подталкивало меня. Может, потому что это было ловким трюком? Но и ловкие трюки, я считаю, имеют свое значение. Ленивые вычисления оказались очень ловким трюком – мы делали то, что вообще не считали возможным.

Сейбел: Например?

Пейтон-Джонс: Помню, однажды мой приятель Джон Хьюз написал для меня программу. Я тогда работал над двумя реализациями лямбда-вычислений и сравнивал их. Джон написал кое-какие тестовые программы. Одна из них позволяла вычислить число e с какой угодно точностью. То была ленивая программа – и прекрасная, потому что давала *все* цифры в значении e .

Сейбел: Со временем.

Пейтон-Джонс: Со временем, да. Но об этом должен был думать клиент. Не надо было заранее определять, сколько знаков после запятой там будет. У вас был список, из которого вы выбирали сколько угодно элементов, и программа не выдавала следующую цифру, пока не прошло достаточно времени. Такие вещи не совсем очевидны, если пишешь программу на Си. Это можно сделать с достаточной долей смекалки, но для Си это не является естественной парадигмой программирования. Это практически невозможно сделать, если вы не видели ленивой функциональной программы. Программа Джона умещалась на четырех или на пяти строчках. Удивительно.

Сейбел: Потом этот вид вычислений был встроен в языки – взять, например, генераторы в Python или любую программу, позволяющую получать значения. Что вы подумаете? Что есть множество вещей, представимых как серии бесконечных вычислений, из которых можно получать результаты, пока не устанешь? Или что это интересный способ решения некоторых проблем, но далеко не основа всего?

Пейтон-Джонс: Думаю, я тогда не забирался так глубоко; это было просто прикольно. И заниматься этим мне нравилось. По-моему, надо лишь определить, что вам нравится, и следовать этим путем. Функциональное программирование вдохновляло меня, но каких-то глубинных причин заняться им не было. Просто мне казалось, что это прекрасный способ создавать программы. Я люблю лыжи – значит, я буду ходить на лыжах. Не потому, что это изменит мир, а потому, что мне нравится.

Сейчас мне кажется, что ленивые вычисления важны, так как позволяют сохранить чистоту. Я уже говорил об этом по другим поводам. Я люблю эти вычисления и, если есть выбор, всегда возьму ленивый язык. По-моему, они полезны в какой угодно области программирования. Вы, конечно же, читали статью Джона Хьюза «Why functional programming

matters» (О важности функционального программирования). Это, вероятно, первое открытое выступление на тему того, что ленивые вычисления – не просто игрушка. Главное – они помогают создавать модульные программы.

Ленивые вычисления позволяют писать генераторы. Хьюз пишет: давайте сгенерируем все возможные ходы в шахматной игре, независимо от потребителя, который лазает по дереву и делает альфа-бета-отсечение с помощью минимакса. Если же вы генерируете всю последовательность приближений к ответу, потребитель говорит, когда нужно остановиться. Поэтому, отделив генераторы от потребителя, мы получаем модульное строение программы. А если вы занимаетесь генерированием вместе с потребителем, говорящим, когда остановиться, показатель модульности существенно понижается. Модульность означает, что в разных местах идут разные процессы, которые можно комбинировать. Джон в своей статье приводит прекрасные примеры того, как потребитель или генератор могут быть заменены независимо друг от друга; это позволяет соединять вместе различные программы, что было бы намного труднее, если бы это была одна тесно переплетенная программа.

Вот почему ленивые вычисления – хорошая вещь. Они также полезны и на локальных уровнях программы. Так, если Haskell-программист будет писать определение функции вместе с локальными определениями, то сделает это так: « f от x равно тому-то и тому-то *там, где...*», и после инструкции «там, где» – сколько-то определений, нужных далеко не во всех случаях. Но все равно их следует перечислить. Те, которые нужны, будут вычислены, те, которые не нужны, – не будут. И программист думает: «О черт, все эти подвыражения надо вычислить, но вычислить нельзя, потому что все полетит из-за деления на ноль. Поставлю-ка я определение в правую ветвь условия».

А в нашем случае – ничего подобного. Надо просто написать те вспомогательные определения, которые могут понадобиться, и те, что понадобятся, будут вычислены. Это очень, очень удобный инструмент.

Но вернемся к общим положениям. С ленивым механизмом вычисления сложнее предсказать, когда понадобится вычислить выражение. И если вы хотите вывести что-нибудь на экран, то язык с вызовом по значению, где порядок вычисления явно определен, делает это при помощи «функции» с побочным эффектом – я специально ставлю кавычки, так как это вовсе не функция, – с типом, скажем, `string -> unit`. При вызове функции она печатает что-то на экране – в виде побочного эффекта. Это есть в Лисп, в ML, в любом языке с вызовом по значению.

А в чистом языке, если есть функция `string -> unit`, ее никогда не надо вызывать: вы ведь знаете, что она выдаст всего лишь ответ типа `unit`. Больше она не делает ничего. А каков ответ, вы знаете. Но поскольку

функция с побочными эффектами, очень важно вызвать ее. В ленивом языке проблема вот такая: вы говорите «*f* применяется к `print 'привет'`», а вычисляет ли *f* свой первый аргумент, для вас неясно. Это все происходит внутри функции. Если же аргументов будет два – `print 'привет'` и `print 'пока'`, – она может выполнить один, или оба в любом порядке, или ни одного. Поэтому при ленивых вычислениях делать ввод/вывод при помощи побочных эффектов невозможно. Вы не можете написать таким способом рациональную, надежную, предсказуемую программу. Сначала это было непривычно – нет, собственно, никакого ввода/вывода. И потому долгое время в программах использовалась только функция `string -> string`. Целая программа делала только это. Строка на входе была вводом, а строка результата – выводом. Вот и все.

Можно было слегка усложнить схему, закодировав в строке вывода команды вывода, которые выполнялись внешним интерпретатором. Строка вывода могла скомандовать: «Вывести вот это на экран, а вон то сохранить на диске». Интерпретатор уже умел это делать. Итак, мы имели замечательную, чисто функциональную программу, а нехороший интерпретатор интерпретировал командную строку. Но если считываешь файл, как вернуть ввод в программу? Просто: сделать строку с командами вывода, которые интерпретируются нехорошим интерпретатором, и произвести ленивое вычисление – результат поступает обратно на вход. Итак, теперь программа преобразует поток ответов в поток запросов. Поток запросов направляется на нехороший интерпретатор, каждый запрос генерирует ответ, который идет затем на вход. Поскольку вычисление ленивое, программа выдает ответ с таким расчетом, чтобы он успел пройти цикл и прийти на вход. Правда, это работало не без сбоев – если ответ поглощался слишком агрессивно, программа зависала, ведь нужен был ответ на вопрос, которого еще не было на выходе.

Смысл в том, что ленивость загнала нас в угол, и пришлось решать вопрос с вводом/выводом. Это была самая важная проблема ленивого программирования. Но начиналось-то все с другого – с того, как это приколно, как здорово.

Сейбел: За все то время, что вы занимаетесь программированием, как изменились ваши представления о нем как таковом?

Пейтон-Джонс: Думаю, больше всего это связано с монадами и системами типизации. В начале 1980-х я думал лишь о чисто функциональном программировании с относительно простой системой типизации. Теперь же я думаю о нем как о сочетании функционального, императивного и параллельного программирования, причем связь между ними идет через монады. Типы стали гораздо сложнее, позволяя создавать намного более широкий спектр программ, чем я предусматривал когда-то. Можно считать это эволюцией моих взглядов на программирование.

Сейбел: После первой неудачной попытки вы написали не один компилятор. Наверняка вы поняли, в чем секрет создания успешного компилятора.

Пейтон-Джонс: Да, я много чего понял с тех пор. То был компилятор для императивного языка на императивном языке. Теперь я пишу компиляторы для функционального языка на функциональном языке. Но в GHC, нашем компиляторе для Haskell, важно то, что в нем применен промежуточный язык с типизацией.

Сейбел: А в этом языке применена типизация исходного языка?

Пейтон-Джонс: Да, но в гораздо более явном виде. В оригинале широко применяется вывод типов – исходный язык подогнан под эту возможность. В промежуточном языке применена более общая система типизации, более явная и потому более выразительная: у каждой аргумента функции есть свой тип. Нет *вывода* типов, есть *контроль* типов. Это язык с явной типизацией, а исходный – с неявной.

Вывод типов основывается на тщательно составленном наборе правил, которые удостоверяют, что вы находитесь в рамках того, что устройство вывода типов может понять. Если же программу преобразовывать из одного вида в другой на уровне исходного кода, то, возможно, вы выйдете за эти границы. Вывод типов не позволяет сделать это и потому не годится для оптимизации. Оптимизатор не должен беспокоиться о том, не вышли ли вы за границы вывода типов.

Сейбел: Получается, что есть программы, которые корректны, потому что были получены корректным преобразованием исходного кода, но в то же время, если бы были написаны от руки, компилятор бы сказал, что не может вывести типы.

Пейтон-Джонс: Правильно. Такова природа систем со статической типизацией – и вот почему динамические языки по-прежнему важны и интересны. Можно написать программы, для которых не установить конкретную систему типизации, но которые не дают сбоев при выполнении. Это и есть золотой стандарт – нет аварийных сбоев, не добавляются целые числа к символам. Это будут отличные программы.

Сейбел: Когда сторонники динамической и статической типизации начинают препираться, первые говорят: «Очень много программ, где статическая типизация мешает мне написать то, что я хочу». А вторые отвечают: «Да, такие программы есть, но на практике это не составляет проблемы». Что вы думаете по этому поводу?

Пейтон-Джонс: Это отчасти зависит от привычки. Я, например, говорю, что так и не привык писать на C++. С другой стороны, вы не будете страдать от отсутствия ленивых вычислений, если вообще ими не пользовались, а я буду, потому что пользуюсь ими постоянно. Возможно, ди-

намическая типизация – похожий случай. Мое ощущение – насколько оно может быть ценным с моим специфическим опытом – таково, что крупные программные блоки вполне могут иметь статическую типизацию, особенно в таких богатых системах типизации. И там, где такая типизация возможна, она очень полезна по неоднократно приводившимся причинам.

Реже приводится такой довод, как поддержка программ. Если вы хотите поменять кусок своего кода трехлетней давности – не подретушировать одну процедуру, а переписать коренным образом, – то системы типизации будут крайне полезны.

Такое происходит с нашим собственным компилятором. Я могу взять ГНС и что-то переписать, например систему представления данных, которая меняет его полностью, и быть уверенным, что найду все места, где она используется. Будь это более динамический язык, я бы начал беспокоиться, что пропустил то или это, что кто-то полагается на данные, которых там на самом деле нет, – и это в тех местах, до которых я почти не дотрагивался.

Еще одно: статическая типизация для меня отчасти объясняет, что именно делает программа. Это такой не очень развитый язык, на котором я могу сказать что-то по поводу действий программы. Меня часто спрашивают: «Где в функциональном языке аналог UML-схем?» Думаю, лучший ответ – «система типизации». Там, где объектно-ориентированный программист будет рисовать картинки, я стану создавать сигнатуры типов. Они, конечно, не схемы, но поскольку мы в формальном языке, то они есть неотъемлемая часть текста программы и статически проверяются в коде, который я пишу. Поэтому у них масса достоинств. Это почти архитектурное представление того, что делает программа.

Сейбел: А вам приходилось писать такие программы, про которые известно, что они корректны, но по каким-то причинам выходят за границы проверки типов?

Пейтон-Джонс: Такое случается при программировании с обобщенными типами, например когда вы пишете функции, принимающие данные любого типа и сериализующие их. Для этого лучше подходит не типизированный язык.

Сейчас есть целая мини-индустрия, которая исследует возможности применения типизации в программах с обобщенными типами. Это очень увлекательно. Но проще взять язык динамического типа. Я пытаюсь убедить Джона Хьюза написать статью для «Journal of Functional Programming» о том, чем плоха статическая типизация. Думаю, это будет интересная статья, если Джон – сторонник строгой типизации,

изохронный прикладной функциональный программист, который сейчас пишет много на нетипизированном Erlang, – объяснит, чем плоха статическая типизация. Полагаю, статья будет полна интересных размышлений. Не знаю, чем все это закончится.

Пожалуй, я и сейчас сказал бы: «Применяйте статическую типизацию где только можно – это громадный выигрыш в поддержке». Она помогает вам продумать программу, помогает писать ее. Но появление все более изохронных систем типизации говорит о том, что разработчики стараются распространить их на возможно большее число программ. История, таким образом, не закончена.

Сторонники зависимых типов сказали бы: «В конце концов, система типизации сможет выражать абсолютно все». Но система типизации – это забавная штука, нечто вроде компактного языка спецификации. Она говорит кое-что о функции, но не настолько много, чтобы это не поместилось в вашей голове. Поэтому важна четкость. Система типизации на двух страницах перестает передавать нужную информацию.

Мне бы хотелось иметь четкую и компактную систему типов, пусть немного слабых, но именно для того, чтобы быть четкими, вместе с инвариантами, которые, может быть, выражались бы более развернутым, чем вывод типов, способом, но при этом поддающимся статической проверке. Я сейчас работаю над проектом статической верификации входных и выходных условий, а также инвариантов типов данных.

Сейбел: Что-то вроде контрактного программирования в Eiffel?

Пейтон-Джонс: Верно. Чтобы я мог написать контракт для функции, например такой: при задании аргументов со значением выше нуля функция дает результат меньше нуля.

Сейбел: Как вы подходите к проектированию программ?

Пейтон-Джонс: Наверное, я могу сказать, что главная проблема – скажем, если я пишу что-то новое для GHC, – не в том, чтобы облечь идею в код, а скорее в том, чтобы сформулировать идею.

К примеру, мы сейчас в середине большого рефакторинга части компилятора, отвечающей за генерацию кода. Сейчас в компиляторе есть этап, когда он фактически берет функциональный язык и преобразует его в C--, язык императивный. Это важный шаг. C-- называется так, потому что он похож на подмножество Си, но на самом деле он задумывался как портируемый язык ассемблера. Он не записывается ASCII-символами, это просто внутренний тип данных. Поэтому на этом этапе компилятор выполняет функцию преобразования структуры данных, представляющей функциональный язык, в структуру данных, представляющую императивный язык. Как совершается этот шаг?

Сейчас за это отвечает довольно сложный фрагмент кода. Но недавно я понял, что задача раскладывается на две части: 1) преобразование в диалект C--, позволяющий делать вызовы процедур, когда внутри одной процедуры можно вызвать другую, и 2) преобразование *этого* в подязык без всяких вызовов, кроме хвостовых.

Затем главное понять, что здесь является типом данных. Что у нас в C--? Структура данных, представляющая императивную программу. Вторым шагом – пройти по программе, смотря на каждый кусочек в отдельности. Ваше внимание идет по пути управляющей логики программы или, наоборот, откатывается назад через нее. Это удобно представлять через структуру данных под названием *Zipper* – полезная, чисто функциональная структура данных для того, чтобы окинуть взглядом чисто функциональную структуру данных.

Норман Рэмзи из Гарварда нашел способ использовать ее для перемещения по структурам данных, представляющим императивные управляющие графы. Мы с ним и Джоном Диасом с этой целью перепроектировали выходную часть GHC с применением этой технологии. И теперь мы можем использовать тот же самый бэкенд для других языков.

Многие споры шли на уровне типов. Норман говорил: «Вот API», – показывая сигнатуру типов, а я в ответ: «Зачем так сложно?» Он объяснял зачем, а я говорил: «Может быть, вот так будет проще». Так что мы довольно долго бились над описанием типов.

Но много времени уходило не на собственно программирование, а на определение самой идеи. Что мы хотим сделать с анализом потока данных? Надо было дать четкий ответ: что подразумевает такой-то шаг программы. Так что немало времени мы потратили на уточнение того, что у нас на входе и что на выходе, и какие у них типы данных. И всего лишь определив эти типы данных, мы довольно подробно описали работу программы. Даже удивительно, насколько подробно.

Сейбел: Как размышления над типом данных соотносятся с кодированием? Набросав типы, вы можете приниматься за код? Или, наоборот, написание кода помогает в понимании типов?

Пейтон-Джонс: Скорее последнее. Я сразу начинаю писать сигнатуры типов в файл. Даже скорее я начинаю писать код, работающий со значениями этих типов. Потом возвращаюсь к типам и изменяю их. Этот процесс не делится четко на два этапа, когда определил типы и садишься за код.

Пожалуй, в этом смысле мне не хватает дисциплины, так как я ни разу не работал в большой команде. Работая в одиночку, можно позволить себе делать вещи в расчете на то, что они помещаются в твоей голове, что, возможно, не получится в большой команде.

Сейбел: Вы говорили о том, что при последней перетряске кодов в GHC его компоненты стали намного более универсальными. GHC – большая программа, которая эволюционировала со временем, так что вы смогли воспользоваться универсальностью, но и заплатили за ее избыток. Что вы узнали о том, как балансировать между избытком и недостатком универсальности?

Пейтон-Джонс: Я предпочитаю вообще не писать чего-то очень универсального. Стараюсь сделать свои программы как можно более *красивыми*, но не обязательно *универсальными*. Это разные вещи. Я стараюсь, чтобы код выполнял свою задачу максимально ясным и четким способом. И лишь когда обнаруживаю, что уже писал этот код, то спохватываюсь: зачем делать это снова, достаточно сделать это в одном месте, добавив аргументы, чтобы параметризовать несовпадающие участки.

Сейбел: Какие среды и инструменты вы сейчас используете?

Пейтон-Джонс: О, ужасно примитивные. Работаю в Emacs, компилирую при помощи GHC. Вот и все. Есть профилирующие инструменты, поставляемые с нашим компилятором; люди часто пользуются ими, чтобы профилировать программы на Haskell. Мы применяем их для профилирования самого компилятора. GHC производит много промежуточной выходной информации, и мне видно, что там происходит.

Отладка для меня часто связана с тем, что компилятор порождает плохой код, и я изучаю состояние его внутренностей. Или вот: взять небольшую исходную программу, скомпилировать до такого-то места, посмотреть – вот что такое для меня отладка. Я редко прохожу программу пошагово, чаще всего гляжу на значения разных частей скомпилированного кода.

Я даже нечасто пользуюсь всеми хитростями Emacs, хотя некоторые любят этим заниматься. Также масса народу пользуется интегрированными средами разработки – Visual Studio, Eclipse. Мне кажется, неприятие языков функционального программирования отчасти связано с тем, что мы не выпустили свою интегрированную среду разработки. Опять проблема курицы и яйца. Сейчас напирают на курицу – идет всплеск интереса к функциональному программированию. Надеюсь, что и за яйцо тоже возьмутся. Интегрированная среда для Haskell потребует серьезной разработки. Даже при таких оболочках, как Visual Studio или Eclipse, предстоит большая работа над красивым плагином, который бы делал все как надо.

Сейбел: В GHC есть цикл REPL, GHCi. Вы предпочитаете работать с Haskell интерактивно?

Пейтон-Джонс: Ну, сам я сейчас в основном редактирую и компилирую. Но другие просто живут в GHCi.

Сейбел: Когда дело доходит до тестирования, у функциональных языков есть один плюс: если требуется протестировать небольшую функцию, сидящую в глубине программы, надо просто выяснить, что она принимает на входе.

Пейтон-Джонс: Думаю, если входные данные достаточно просты, проблем с моей программой быть не должно. Проблемы возникают, когда GHC пытается скомпилировать какую-нибудь непомерную входную программу и получает неверный ответ.

Тестирование необычайно важно для создания свойств. Очень полезна QuickCheck – библиотека Haskell, генерирующая случайные тесты для функции в зависимости от ее типа. И я старался понять, почему использую QuickCheck – очень приятный инструмент – меньше, чем мог бы. Видимо, потому, что меня беспокоят ситуации, когда трудно сгенерировать тестовые данные. Так или иначе, куча народу создает программы, от которых GHC просто воротит. Для этого у GHC и есть свой багтрекер.

Я обычно начинаю с чего-нибудь, что работает не так. Возможно, компилятор зависнет в каком-то месте, или откажется выполнять программу, которую должен выполнять, или станет генерировать неоптимальный код. Если он просто генерирует плохой код, я гляжу на него на разных стадиях компиляции и соображаю: «Здесь все в порядке, здесь тоже, а здесь нет – в чем дело?»

Сейбел: Как именно вы это делаете?

Пейтон-Джонс: В GHC есть флажки, которые позволяют выводить что-либо на печать.

Сейбел: Встроенные операторы печати для отладки?

Пейтон-Джонс: Да. Плюс к тому структура такая же, как у большинства компиляторов: на верхнем уровне есть конвейерная структура преобразований. Если что-то не так внутри какого-то из шагов, задача усложняется. Но я предпочитаю несложные методы отладки. Покажите мне программу до и после данного шага. Ага, я вижу, в чем ошибка! Если же не вижу, то могу использовать какой-нибудь из небезопасных операторов `printf`, чтобы понять, что происходит.

Есть разные отладчики для Haskell. Один из них, и просто отличный, написал в этом году студент летней школы, Пепе Иборра: это интерактивный отладчик, который теперь поставляется вместе с GHC. Я, правда, его мало использовал – он появился недавно, и, кроме того, не очень понятно, как пошагово проходить функциональную программу.

Были любопытные исследования насчет отладки функциональных программ. Жаль, что у нас нет простого и очевидного решения, но зато это интересная исследовательская проблема.

Я все это говорю, чтобы показать, что использую крайне примитивные техники отладки, например через небезопасные операторы `printf`. Тут нечем гордиться. Но долгое время ничего больше не было – по крайней мере, если брать GHC. Я выработал способы, которые делают для меня этот путь самым коротким.

Сейбел: Это как всегда. Зачем создавать новые отладчики, если люди довольствуются операторами печати?

Пейтон-Джонс: Это скорее культурное явление. Если перейти на отладчики платформы .NET, на которые потрачены десятки и сотни тысяч человеко-часов, думаю, результат будет качественно иным. Вероятно, для хорошей работы отладчики требуют еще больше циклов разработки. Но зато в итоге получается образцово полезная вещь.

Возможно, вы разговаривали в основном с людьми академического склада и с теми, кто в силу возраста не привык к сложным отладчикам. Я бы не стал делать никаких общих выводов. И, конечно, не хочу принизить значение качественных отладчиков – особенно для сложных систем с множеством программных слоев. GHC очень прост сравнительно со средой .NET, где есть слои DOM и UML, и не знаю, что еще. Теперь вокруг столько примочек, что программная поддержка становится действительно важной.

Сейбел: Еще один способ создавать правильные программы – формальные доказательства. Что вы думаете об их полезности?

Пейтон-Джонс: Представьте, что ваша цель – иметь для всего автоматическую проверку правильности. Что это будет означать? Проверка по сравнению с чем? По сравнению с некоей спецификацией. Что за спецификация? Она должна описывать *все*, что делает программа, иначе проверка невозможна. Итак, должна быть формальная спецификация для каждого действия программы. Как написать такую спецификацию? Допустим, вы пользуетесь функциональным языком. Тогда, может быть, ваша спецификация – это и есть ваша программа?

Я тут немного хитрю, ведь в спецификации вы можете сказать то, чего не скажете в программе. Например, «результатом функции является такое y , что при возведении в квадрат дает x ». Это хорошая спецификация для функции квадратного уравнения, но ее особенно не выполнишь. Но все равно, думаю, при попытке написать спецификацию на *все* действия программы она выходит чрезмерно сложной, и вы больше не уверены, что в ней сказаны все нужные вам вещи.

Более продуктивным для практических целей будет описание некоторых *свойств*, которыми должна обладать программа. К примеру, вы пишете: «Клапан 1 никогда не должен закрываться одновременно с клапаном 2. Это дерево всегда должно быть сбалансировано. Эта

функция всегда должна иметь результат больше нуля». Это небольшие частичные спецификации, не полные. Это просто утверждения, в которых вы хотите быть уверены.

Как их написать? Функциональные языки неплохо приспособлены для этого. Именно это и происходит, если писать спецификацию в QuickCheck – свойства получаются функциями языка Haskell. Допустим, мы хотим проверить, что функция `reverse` является своей противоположностью, тогда мы напишем `checkreverse` с типом список из `A` → булевское значение. Итак, `checkreverse` от `xs` будет: `reverse` от `reverse xs` равно `xs`. Это функция, которая всегда оказывается верной. Функция-свойство. Но она написана на том же самом языке, и это здорово.

Теперь можно думать о статических проверках этого. Это может быть трудно, а может и легко. Но все равно, если свойство записано по всем правилам, это большое облегчение. Можно проверить его путем генерирования тестовых данных – именно это делает QuickCheck.

По-моему, писать частичные спецификации гораздо плодотворнее, чем одну спецификацию на *все*, что делает программа. Возможно, их придется писать много и потом подвергать статической либо динамической проверке. Вы не докажете, что ваша программа правильна, но ваша уверенность в этом возрастет. Думаю, только это мы и можем сделать.

Сейбел: Вы определяете много свойств для всех важных, по-вашему, вещей. Потом по возможности проводится статическая либо динамическая проверка. Ведь мы не сможем устроить статическую проверку для них всех?

Пейтон-Джонс: Правильно. Но в функциональном мире у вас больше шансов. Однако мы все равно слишком долго раскачиваемся, чтобы продемонстрировать это. Так или иначе, первое – это записать свойства.

Мне кажется очень важным уйти от этого глобального подхода, «все или ничего», в сторону очень полезного статического и динамического тестирования частичных спецификаций. Это повысит вашу уверенность в правильности программы, а на большее рассчитывать нельзя. Даже так называемые полные спецификации не учитывают вещи вроде того, что программа должна работать за 0,1 с или занимать не более 10 Кбайт памяти. Часто в них ничего не говорится о ресурсах, о времени. Даже если программа формально отвечает спецификации, из-за этих мелочей она может работать не так, как нужно. Думаю, мы обманываем себя, когда говорим, что проверили программу и все в порядке. Лучше честно сказать, что мы повышаем свою уверенность в ней. Тут все может начинаться скромно – вы повышаете уверенность на 75%, прикладывая всего 5% от общего количества усилий. Это хороший показатель.

Сейбел: Поговорим о параллелизме. Гай Стил поручил мне задать вам вопрос о транзакционной памяти – спасет она мир или все же нет?

Пейтон-Джонс: Нет, конечно. Сама по себе – нет. Параллелизм – многоголовый зверь, которого не убьешь одной пулей. Если речь заходит о параллелизме, то я за различные подходы.

Соблазнительно думать, что можно использовать одну программную парадигму для написания параллельных программ и затем реализовывать ее. Тогда для всех программ применялась бы одна парадигма. Но я в это не верю. Я считаю, что для одних стилей программирования больше подходит обмен сообщениями, для других – транзакционная память, для третьих – параллелизм данных. Программисту может потребоваться не один подход, а несколько.

Но если вы меня спросите, лучше ли транзакционная память, чем блокировка и переменные условия? Вот это уже сравнение подобного. Мой ответ – да. Мне кажется, транзакционная память заставит забыть и о том, и о другом. Для всяческих счетчиков, многопоточности с разделяемой памятью на многоядерном процессоре – транзакционная память. Но это, разумеется, не единственный способ справляться с параллельными программами.

Сейбел: Я слышал в ее адрес критику такого рода: оптимистический параллелизм не обеспечивает того уровня параллелизма, на который можно рассчитывать. Утверждается, что легко можно оказаться в ситуации, когда выполнение перестает двигаться вперед.

Пейтон-Джонс: Да, нужно заботиться о зависаниях. Вот мой любимый пример: большая транзакция, которая не фиксируется, потому что в этом месте первой совершается другая, маленькая. Аналогией может быть библиотекарь, который наводит порядок в своей библиотеке. Начинается оптимистическая реорганизация. Две трети работы сделано, тут приходит студент и берет книгу. Он успешно фиксирует свою транзакцию, ведь реорганизация библиотеки еще не зафиксирована. Библиотекарь доходит до конца, обнаруживает отсутствие книги: библиотека изменилась за время реорганизации, структура данных неверна, значит, надо начинать все сначала.

Сейбел: Если есть блокировка и переменные условия, все по-другому – библиотекарь запирает библиотеку, и никто не может взять книгу до полной реорганизации. Поглядев на эту схему, вы немедленно сказали бы: «Мы не можем запереть библиотеку, пока не закончим», – запретив выдачу книг, так что пришлось бы изобретать более сложную схему блокировки.

Пейтон-Джонс: Верно. Надо создать маленькую подбиблиотеку или что-нибудь в этом духе, куда поместить самые ходовые книги, чтобы

студенты могли брать их во время реорганизации основной библиотеки. Надо подумать о стратегии решения конкретной задачи и о том, в каком виде ее выразить. Проблема одна и та же в обоих случаях: как реорганизовать библиотеку, не прекращая полностью выдачу книг. После трудной части – придумывания того, как это сделать, – вы думаете о том, как это выразить. И здесь транзакционная память – абсолютный чемпион. Она превосходит и блокировку, и переменные условия для выполнения параллельных программ.

Сейбел: А если я не хочу допускать, чтобы кто-то пришел ко мне за двадцать первым экземпляром самой ходовой книги и оказался запертым? В физическом мире можно представить, что если кто-то приходит за книгой, мы заменяем ее некой заглушкой, которую библиотекарь использует в реорганизации, и когда книга приходит назад, мы возвращаем ее на место заглушки. Но если реорганизовывать библиотеку в мире с транзакционной памятью, придется повторять транзакцию.

Пейтон-Джонс: Но кое-что остается неизменным – шифр книги, верно? Есть несколько способов решить задачу. Например, вы можете сказать, что при работе с заглушкой сама библиотека не меняется, меняется только сама книга. Вы не изменяете ее ключевое поле – только значение, где книга в данный момент находится. И теперь каталог может меняться, где бы книга ни была. Это прекрасно и поддается выражению естественным способом.

В случае транзакционной памяти библиотекарь просматривает все места в памяти, которые считывал, и проверяет, содержат ли они те самые значения, что и при последнем заходе. Поэтому посещенные им ячейки памяти должны содержать ключевое поле книги, определяющее, куда ее положили. Но библиотекарь не читает содержание книги. Он всего лишь проверяет, содержит ли ключевое поле, скажем, число 73.

Но не буду преуменьшать проблему зависания – она довольно коварна. Нужны хорошие профилирующие инструменты, которые указывают, что транзакция не фиксируется, поскольку сталкивается с другой транзакцией. Нужно, чтобы программа не просто втихомолку подвисала, – нужна обратная связь с ней. То же верно и для системы блокировки. Ненавижу эти часики на экране.

Сейбел: Мне кажется, что в программах с блокировкой мы научились снимать ее так быстро, как только возможно, чтобы минимизировать потери от простоев.

Пейтон-Джонс: Да. Но программировать в этом случае сложнее – мелко модульную блокировку сложно настроить. Мне кажется, одно из больших преимуществ транзакционной памяти в том, что она работает с точностью чрезвычайно мелко модульной блокировки на основе очень простых принципов.

Вот один из них – в системах с блокировкой этого нет. Я определяю высокоуровневые инварианты: у меня несколько банковских счетов, общая сумма денег на них равна N . Деньги перемещаются со счета на счет. Вот мой инвариант. Любая транзакция предполагает этот инвариант в начале и восстанавливает его в конце. Как вы определяете, что она это делает? Мы смотрим на любую транзакцию вида «Возьмите три из этого места и переместите их вон в то». Инвариант сохранен. Каково мое умозаключение в данном случае? Чисто последовательное. Определив высокоуровневые инварианты, я могу делать последовательные умозаключения о каждой транзакции отдельно.

Сейбел: Поскольку транзакции изолированы друг от друга.

Пейтон-Джонс: Да. Это действительно очень мощный принцип. Можно делать последовательные умозаключения относительно императивного кода, несмотря на параллелизм. Вы обязаны определить высокоуровневые инварианты, но это также полезно для душевного спокойствия: вы знаете, что именно пытаетесь сохранять. Если посреди транзакции встречается исключение, это тоже здорово – оно не может уничтожить инварианты, поскольку транзакция тогда завершается ничем. Просто сказка! И совершенно по-другому теперь можно рассуждать о скорости выполнения – вы удостоверились, что все минимально правильно, теперь надо убедиться, что программа нигде не подтормаживает. Это уже труднее: на сегодня есть только профилирующие инструменты и инструменты точечной обратной связи.

Сейбел: Меня поражает вот что: оптимистический параллелизм время от времени используется в персистентных базах данных, но намного реже по сравнению с параллелизмом на основе блокировки.

Пейтон-Джонс: Ну, транзакционную память можно реализовать несколькими способами, и оптимистический параллелизм – лишь один из них. Можно устраивать блокировку по мере продвижения – это уже больше похоже на пессимистический параллелизм.

Сейбел: Тут есть еще и тот момент, что менеджеры блокировок – самая сложная часть баз данных.

Пейтон-Джонс: Верно. Что касается транзакционной памяти, то нужна уверенность в том, что один человек – или команда – может ее реализовать, а остальные могут ею пользоваться. Чтобы иметь шанс убедиться в качестве работы, можно хорошо заплатить исполнителям и на год запереть их в темной комнатухе.

Но после этого у каждого должна быть возможность воспользоваться результатом его/их труда через простейший интерфейс. И вот это, по-моему, здорово. Мне бы не хотелось, чтобы каждому требовался подобный уровень знаний. Вот пример, который когда-то приводил Мо-

рис Херлихи, я только вчера на него ссылался: очередь с двухсторонним доступом, и вам нужно вставлять и удалять элементы.

Последовательная реализация очереди с двухсторонним доступом – это задача, которую проходят в объеме курса бакалавра. Для параллельной реализации с блокировкой по каждому узлу – это тема для научной статьи. Это слишком нелегко, прямо до абсурда. А для транзакционной памяти такую проблему решает студент. Вы просто «заворачиваете» операции «вставить» и «стереть» в одну атомарную операцию – и дело сделано. По-моему, это занятно. Тут есть количественная разница. Сегодня те, кто реализует транзакционную память, должны убедиться, что несколько изменений вносятся в память как единая атомарная операция. Это не так просто – у вас есть только атомарная инструкция «сравнение с обменом». Надо быть внимательным.

Есть проблемы и с зависанием; здесь надо кое-что придумать на уровне логики приложения, чтобы избежать этого. Но потом, реализуя специфичное для приложения решение, вы снова используете транзакционную память. Для такого вида программ это, безусловно, шаг вперед.

Хочу сказать еще кое о чем – мы опять возвращаемся к функциональному программированию. Транзакционная память, конечно, не имеет к нему прямого отношения. Она касается изменения разделяемого состояния – не самой функциональной вещи.

Дело в том, что когда-то я пошел на лекцию Тима Харриса о транзакционной памяти в Java. До этого о транзакционной памяти я вообще не слышал. Тим описывал «атомарные транзакции» и, в общем, больше ничего такого.

Я сказал: «Здорово, но тогда придется протоколировать все побочные эффекты в памяти, все инструкции по загрузке и хранению. А сколько их в Java!» Но в Haskell, благодаря монадам, их почти нет. Загрузка и сохранение в Haskell выражены явно, и программисты считают их важными операциями.

И я подумал, что надо ввести в Haskell все эти атомарные операции, – будет классно. Потом, после лекции, я стал объяснять Тиму, как это можно сделать. Вскоре, поскольку у нас была чистая, элегантная инфраструктура, мы придумали `retry` и `orElse`. Механизм `retry` позволяет осуществлять блокировку внутри транзакции, а `orElse` – выбор внутри транзакции. Тиму и его коллегам при разработке транзакционной памяти такое не пришло в голову, поскольку они имели дело с довольно сложной средой.

И потому они мало задумывались о блокировке – или предполагали, что блокировка будет атомарной операцией вида «запускать эту транзакцию только при наличии такого-то предиката». Но это очень неком-

позиционно. Предположим, вы хотите переложить деньги с одного счета на другой: каковы условия проведения транзакции? Ответ: если на первом счете достаточно денег, а на другом достаточно места, – то есть имеются ограничения на обеих сторонах. Довольно сложное условие. Если задействован и третий счет, все еще сложнее. Все очень некомпозиционно – надо вытаскивать наружу все предусловия методов.

Это то, что у них было. Это работало для небольших программ, но в целом было неудовлетворительно. Поэтому в Haskell мы ввели `retry` и `orElse`, а затем внедрили их в контекст распространенного сейчас императивного программирования, и они широко используются. Отличная работа.

Сейбел: То есть эта концепция на самом деле не завязана на Haskell, вы просто смогли ее придумать?

Пейтон-Джонс: Правильно. При той изначальной простоте легче было оценить хорошую идею. Нас все больше раздражало, что нельзя устроить блокировку, не теряя абстракции. Так и получились `retry` и `orElse`. Видимо, функциональное программирование лучше всего подходит для этого: изучить какого-нибудь неведомого монстра и затем выпустить в обычный мир. Транзакционная память – прекрасный пример; там было взаимодействие в обоих направлениях. Цикл теперь замкнулся, и, по-моему, это чудесно.

Сейбел: Какие книги по программированию вы бы взяли с собой на необитаемый остров?

Пейтон-Джонс: Обязательно «Programming Pearls» Джона Бентли. По поводу жемчужин: в чудной книге «Beautiful Code»¹ есть глава Брайана Хэйеса «Создание программ для „Книги“». Думаю, под книгой Брайан понимает вечно прекрасную программу. Даны три точки, надо найти, с какой стороны линии между двумя точками окажется третья. Есть решения, которые работают неважно, а потом находится идеальное простое решение.

Еще, конечно же, «The Art of Computer Programming»² Дона Кнута. Это не та книга, чтобы читать ее от и до, но одно время я активно ею пользовался. «Purely Functional Data Structures» (Чисто функциональные структуры данных) Криса Окасаки – просто фантастика: нечто вроде курса Артура Нормана, из которого сделана целая книга. Про то, как делать очереди, поисковые таблицы и кучи без всяких побочных эффектов и с хорошими ограничениями алгоритмической сложности. Великолепная книга, читать каждому. К тому же невелика по объему и до-

¹ Под ред. Энди Орама и Грега Уилсона «Идеальный код». – СПб.: Питер, 2009.

² Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008.

ступно написана. «Structure and Interpretation of Computer Programs»¹ Абельсона и Сассмана – мне очень понравилось. «Compiling with Continuations» (Компиляция на продолжениях) Эндрю Апеля – о том, как компилировать функциональную программу, используя стиль передачи продолжений. Тоже превосходная вещь.

Есть одна важная для меня книга, которую я читал мало, – «A Discipline of Programming» Дейкстры. Дейкстра заботится о красоте программ. Его программы полностью императивны, но обладают «свойством Хорара»: вместо того чтобы не иметь очевидных ошибок, они совершенно очевидно не имеют ошибок. Он очень хорошо и изящно рассуждает об этом. Я впервые понял, как это – рассуждать о программах, когда тебе невозможно возразить. Наконец, на меня сильно повлияла книга Пера Бринча Хансена о написании параллельных операционных систем. Я постоянно перечитываю ее.

Сейбел: Вы сейчас много программируете?

Пейтон-Джонс: Конечно. Ни дня без кода. Ну, не то чтобы так и было, но это моя мантра. Мне кажется, тем, кто что-то делает хорошо, угрожает движение вверх по карьерной лестнице, пока они совсем не перестанут заниматься тем, что делают хорошо. Поэтому работа в Microsoft Research и в исследовательской сфере вообще хороша тем, что я могу заниматься компилятором, над которым тружусь с 1990 года. Это большая по объему вещь, и есть длинные фрагменты кода, которые я знаю лучше всех остальных.

Много ли я пишу? Иногда я программирую целый день, иногда целый день не касаюсь кода. В среднем выходит по несколько часов в день. Мне очень нравится. Как можно вообще бросить программирование? И, кроме того, это помогает быть внутренне честным – работа с собственным компилятором и языком, который пропагандируешь.

Сейбел: Вам это доставляет такое же удовольствие, как и в начале?

Пейтон-Джонс: Да, конечно. Это лучшая в мире вещь. Мне кажется, у большинства программистов есть чутье: «здесь должен быть какой-то хороший выход». Работа в исследовательской сфере хороша еще и тем, что надо мной не стоит менеджер, которому нужен результат через неделю. Я могу спокойно сесть и подумать: «Здесь должен быть какой-то хороший выход».

Поэтому я много времени уделяю рефакторингу, вожусь с интерфейсами, создаю новые типы или полностью переписываю большие куски, чтобы их улучшить. GHC довольно велик – не по промышленным стан-

¹ Х. Абельсон, Д. Сассман «Структура и интерпретация компьютерных программ». – Добросвет, 2006.

дартам, а по понятиям функционального программирования: в нем 80 000 строк кода на Haskell, а может, и больше. Это компилятор-долгожитель – ему уже пятнадцать лет. Он активно развивается, большие куски переписываются, и нет мест, которые нельзя трогать. Поэтому меня так приятно возбуждает возможность сесть и сказать себе: «Здесь должен быть какой-то хороший выход». Иногда я оставляю что-нибудь на несколько недель – не могу найти хороший выход, зная, что он есть. Это мучительно. Потому что красивый способ должен быть.

Сейбел: Что происходит в эти несколько недель?

Пейтон-Джонс: Я размышляю о проблеме каким-то участком мозга. Иногда возвращаюсь к ней, пытаюсь одолеть этот подъем с разбегу. Тогда я вспоминаю, как тут все сложно, и опять переключаюсь на что-нибудь другое. Восхождение может повторяться несколько раз. Иногда я подсознательно думаю об этом, иногда решаю, что надо покончить с проблемой и найти какой-то выход. И он не всегда оказывается лучшим.

Сейбел: Как это происходит? Вы просыпаетесь утром и понимаете, что решение найдено? Или снова начинаете восхождение и в конце концов добираетесь до вершины?

Пейтон-Джонс: Скорее второе. Озарения по утрам случаются редко. На исследовательской работе есть время подумать над сделанным, записать свои мысли. Иногда получается настолько интересно, что я пытаюсь соорудить статью. Одна из таких статей называется «The Secrets of the GHC Inliner» (Секреты GHC). В ней описываются техники реализации, примененные нами для некоторых элементов GHC и полезные, как мы считаем, другим разработчикам. Как ученый, я имею возможность абстрагироваться от кода, которому мы на четвертый раз наконец-то придали нужный вид, и написать о нем, чтобы другие смогли применить наши методы.

Сейбел: Что для вас программирование? Вы считаете себя ученым, инженером, ремесленником или кем-нибудь еще?

Пейтон-Джонс: Вы знакомы со статьей Фреда Брукса на этот счет – «The Computer Scientist as Toolsmith» (Компьютерный исследователь как системный программист)? Я недавно ее перечитал – отлично написано. Нельзя забывать, что прежде всего мы конструируем разные вещи. Поэтому программировать так увлекательно.

Но одновременно мне очень хочется вывести некие постоянно действующие *принципы*. У меня есть статья о том, как написать хорошую статью или сделать хороший доклад об исследовании, и одно из главных правил в ней – «не описывайте артефакт». Артефакт есть продукт реализации некой идеи. А что такое идея, этот предмет многократного использования, который вы пытаетесь вручить своим читателям или

слушателям? Нечто такое, что может быть для них полезно. Задача ученого как раз в том, чтобы извлекать идеи, пригодные для многократного использования, из конкретных артефактов. Это не наука в смысле открытия законов. Но это перевод ежедневной жизненной суеты в абстракции многократного использования. И я считаю это важным.

Сейбел: Поговорим на тему «инженер или ремесленник». Должны ли мы уподобляться тем, кто строит мосты, которые, по большей части, не разваливаются? Или скорее имеем дело с обжиганием горшков – пусть даже очень сложных горшков, – когда можно лишь научиться этому делу у другого гончара?

Пейтон-Джонс: Это ложное противопоставление. Тут нет никакого «или-или». Для проектировщиков и разработчиков ПО очень сложно внутренне прочувствовать размер артефакта, над которым они работают. Если посмотреть на Эмпайр-стейт-билдинг через отверстие в один квадратный фут, очень непросто ощутить его истинные размеры и внутреннюю структуру.

Насколько велик ГНС? У меня нет внутреннего чувства на этот счет – такого, какое есть насчет здания. Поэтому, думаю, мы совсем не инженеры-мостостроители. Их шаблоны проектирования настолько отточены, что они действительно могут быть уверены, что мост не развалится. Мы даже близко к этому не стоим. Но это вовсе не значит, что мы не должны заботиться об этом.

И здесь, как мне кажется, функциональное программирование может многое нам дать. Оно позволяет создавать более крепкие конструкции, более легкие для понимания и рассуждения о них. И здесь мы, функциональные программисты, запаздываем – много говорим об описаниях функциональных программ, но мало делаем таких описаний. Я бы хотел видеть больше инструментов, позволяющих понимать программы на языке Haskell, формально рассуждать о них и давать гарантии за пределами его системы типизации. Мы уже поднялись выше других и способны подняться еще выше.

Это по поводу прочности. Чем прочнее ваши материалы, тем меньше вы отвлекаетесь на мелочи и больше думаете о масштабных конструкциях. Конечно, это подвигает нас на создание все более крупных структур, пока они не начинают распадаться.

Вероятно, это что-то вроде инварианта. Если что-то умеешь делать, то раздвигаешь это до тех пор, пока не перестает получаться. Поэтому в программировании всегда будет что-то от работы ремесленника, ведь наши амбиции непрерывно растут. А для инженерных сооружений всегда есть физические пределы, дальше которых наращивать структуру невозможно. В ближайшее время вряд ли построят мост через

Атлантику, а если его построить, он может рухнуть. Но его не строят просто потому, что это слишком дорого. А что в программировании? Мы научились быстро и с малыми затратами строить мосты через Ла-Манш, это пройденный этап – давайте попробуем взяться за мост через Атлантику. И он разваливается.

Сейбел: Гай Стил говорил, что закон Мура действовал на протяжении всей его карьеры программиста, но вот насчет всей карьеры своего сына он уже не уверен. Он также рассуждал насчет программирования в ближайшем будущем. Не пора ли уже прекращать эти разговоры – «Построили мост через Ла-Манш, давайте строить через Атлантику»?

Пейтон-Джонс: Нет-нет. С ПО все иначе. Если вы написали программу в десять раз больше предыдущей, это не значит, что вам нужно запускать ее на компьютере, работающем в десять раз быстрее. Программный счетчик тратит 90% времени на 10% кода или как-то так. Участки программы, критичные с точки зрения быстродействия, могут быть невелики по объему.

Хотя, правда, часто случается так, что громоздишь одну абстракцию на другую, так что при нажатии всего одной кнопки на экране происходит много разных вещей вниз по цепочке, пока дело не дойдет до реальных изменений в регистрах.

Поэтому иногда нужно схлопывать эти слои путем сложных преобразований с помощью компилятора, чтобы не делать так много вещей. Границы абстракций могут быть полезны для людей, но машины к ним безразличны. Поэтому я не думаю, что даже после достижения границ возможностей компьютеров программы перестанут усложняться. К тому времени они уже будут работать очень быстро. Думаю, главное ограничение здесь – не скорость процессора, а наша способность понять, что должна делать программа.

Сейбел: Что вам особенно нравится в программировании?

Джонс: Мне нравится писать внутренне логичные программы. Если прилепить к программе что-то наспех, она заработает, но это не лучший вариант. Поэтому мне кажется, что хороший программист – тот, который стремится найти красивое решение. Не каждый может позволить себе роскошь отложить сдачу готовой работы, потому что не нашел красивого решения.

Еще программы хороши своей пластичностью – с ними можно делать что угодно! Но это значит также, что вы можете делать программы уродливые, недолговечные, трудные в обслуживании. Меня иногда пугает индустрия ПО, где все диктуется, с одной стороны, желанием покупателя увидеть продукт на следующей неделе, а с другой – стремлением распространять системы вширь, а не вглубь.

Системы слишком уж распухли. Если вы хотите сделать веб-сервис на ASP.NET, то должны освоить API и инструменты, уметь писать на трех разных языках, знать Silverlight и LINQ – и вы можете создавать акронимы до конца света. И про каждый из них есть толстая книга.

Есть противоречие, неразрешимое для меня. Это полезные системы, придуманные не просто так. Их тщательно продумывали умные люди. Но у каждой – широкий интерфейс. Глубокий или нет, неважно – он широкий. И в голове приходится держать кучу разных программ. Все равно что учить обычный язык с очень богатым словарным запасом.

И это мне не нравится. Я так и не запомнил таблицу умножения. Я всегда все делал на основе нескольких простых правил и придумал много хитростей, чтобы делать это быстро. Все умножают семь на девять, а я умножаю семь на десять, вычитаю, получаю шестьдесят три. Другие просто запоминают таблицу. А она ведь совсем небольшая. Поэтому я не люблю системы, которые заставляют меня держать много чего в голове, и инстинктивно их сторонюсь. В то же время я признаю, что они полезны и важны в практическом плане. Но я задаю себе вопрос: если подумать над ними чуть дольше, нельзя ли сделать интерфейсы меньше, проще и универсальнее?

Сейбел: Мне иногда кажется, что вся эта сложность – именно *из-за того*, что системы разрабатывают умные люди. И им хочется играть в свои игры.

Пейтон-Джонс: Да, есть и такое. Но можно подойти к этому вопросу и более конструктивно. Перед нами – огромный, сложный мир, где так много надо сделать. Человек со зрением олимпийца, с могучим умом и невероятной проницательностью способен создать нечто стройное и цельное.

На практике нам приходится разбивать задачи на кусочки. За каждый кусочек отвечает человек, который исходит из собственного опыта и культурного багажа. То, что он создает на своем пространстве, хуже, чем могло бы быть, – человеку не хватает времени. В итоге весь ансамбль тоже выглядит хуже, чем мог бы. Плюс отрицательное влияние наследства прежних систем.

Это наследство – как ядро каторжника, которое мы волочим за собой. И в этом отношении удачей стал Haskell. Оглядываясь, я всегда вижу перед собой принцип, который мы в него заложили: «Избегать успеха любой ценой». Теперь это что-то вроде мема – люди запомнили эту фразу и мне же ее цитируют.

В этом есть доля правды – именно потому, что мы не хотели быть слишком успешными и слишком поспешными, у нас было какое-то время, чтобы изменять Haskell на протяжении его жизни. Я даже переживаю

из-за того, что Haskell стал более успешным, – мне приходит больше сообщений об ошибках, больше запросов новых функций. Все больше людей говорят: «Прошу вас, не ломайте мою программу». Раньше такого не было.

Сейбел: Вы несколько раз упоминали о красоте кода. Каковы признаки красивого кода?

Пейтон-Джонс: Тони Хоар замечательно сказал: нужен код, совершенно очевидно свободный от ошибок, а не код, свободный от очевидных ошибок. Думаю, красивый код – это код, который совершенно очевидно правилен. Абсолютно прозрачный код.

Сейбел: А как насчет перлов – небольших замысловатых, но интересных фрагментов кода, – они тоже красивы?

Пейтон-Джонс: Иногда что-то правильно, но этого не понять без умственных усилий. Иногда нужно предварительное знание. Если взглянуть на код для АВЛ-дерева, не зная, в чем его смысл, не понятно, зачем нужны все эти вращения. Но если вы знаете, какой инвариант тут поддерживается, то видите: ага, если поддерживается этот инвариант, мы получаем логарифмическое время поиска. Вы смотрите на каждую строку кода и понимаете, что здесь поддерживается инвариант. Инвариант дает нам предварительное знание и позволяет судить о том, что код совершенно очевидно правилен.

Я полностью согласен с тем, что простого изучения кода может быть недостаточно. Если вы смотрите на код и видите, что он правилен, это еще не характеристика красивого кода. Возможно, кто-то должен объяснить, почему код правилен. Но после этого, поняв, что тут делается, выработав свою точку зрения, вы говорите: да, это на самом деле правильно.

Сейбел: Есть ли верхняя граница, если говорить о размере, вплоть до которой код все еще может считаться красивым?

Пейтон-Джонс: Не знаю, связано ли это с размером. Знание, необходимое, чтобы убедить себя, что он правильный, направлено в сторону большей уверенности в его корректности. В очень больших программах всегда есть недочеты или даже несомненные ошибки. Но ликвидировать их сразу же бывает нецелесообразно. Это справедливо в отношении GNC и еще более – в отношении программ Microsoft.

Однако все эти большие программы управляемы – в них есть инварианты, есть общие спецификации, указывающие, как должна работать программа и что в ней должно быть верно. Возьмем опять для примера GNC, в котором есть такой инвариант: все промежуточные программы должны быть хорошо типизированы. При желании это можно проверить и во время работы программы. Мощный инвариант, позволяющий

отследить, что происходит с программой. Вот поэтому я не уверен, что дело в размере.

Конечно, внутренние связи в итоге приводят к тому, что большая программа рушится под собственным весом. Иногда в исследовательской работе есть еще и то преимущество, что можно переписать кусок кода на основе своего обогащенного знания о том, какова цель программы и как ее достичь. Мы говорили о рефакторинге бэкенда GHC. Будь я сильнее связан коммерческими соображениями, я бы не смог позволить его себе. Надеюсь, благодаря этому GHC в долгосрочном плане станет легче для понимания и обслуживания.

Есть ли верхняя граница, если говорить о размере? Не знаю. Но мне кажется, что продолжая создавать хорошие абстракции, можно строить мосты через Атлантику. У нас есть программы, которые работают, – пусть не идеально, но на удивление хорошо, учитывая их размер.

Сейбел: Вопрос в том, можно ли построить здание – большое, приспособленное для проживания и красивое одновременно.

Пейтон-Джонс: С красотой будет непросто. Код часто бывает красивым или, по крайней мере, не уродливым, сразу после рождения. Но это свойство трудно сохранить в течение всей его жизни, то есть при поддержке. Долгоживущие программы плохи тем, что постепенно становятся уродливыми. Не то чтобы безобразными – но негодными, бесполезными.

Сейбел: И единственный выход – признать, что программа отжила свое, и начать делать новую?

Пейтон-Джонс: Надо переработать какие-то ее куски. Если не можешь позволить себе переработку в то время, как программа разошлась и используется, то не исключено, что лет через десять придется ее выкинуть и задуматься о новой. А если можешь, если программа обновляется так, как самообновляются клетки нашего тела, получается то, что, надеюсь, происходит с GHC.

Самый угнетающий момент в жизни программиста – когда сталкиваешься с чужим кодом или, что еще хуже, со своим и не находишь сил его переделать. Это угнетает.

8

Питер Норвиг

Питер Норвиг – теоретик широкого профиля и хакер в душе. В свое время он написал программу для нахождения в истории поиска Google трех последовательных запросов от одного пользователя, так чтобы они складывались в хокку (один из знаменитых примеров: «Java ECC/эллиптическая криптография Java/FAQ „Плейбоя“»).

На сайте Норвига вы найдете самые обычные ссылки: написанные им книги, слайды выступлений, фрагменты его кода. Но там есть также ссылки на его работы, опубликованные в «McSweeney’s Quarterly Concern», на искрометный рассказ о создании программы для генерации самого длинного палиндрома и на пародийную PowerPoint-презентацию Геттисбергской речи Линкольна, отмеченную Эдвардом Тафти и появляющуюся на первых страницах результатов, если ввести «powerpoint» в строке поиска Google.

Сегодня Норвиг – глава исследовательского отделения Google, ранее руководил отделением качества поиска. До прихода в Google возглавлял подразделение вычислительной техники исследовательского центра НАСА, а до того, в конце 1990-х, был одним из первых сотрудников интернет-стартапа Junglee. Норвиг – лауреат Премии НАСА за выдающиеся достижения (2001), член Американской ассоциации искусственного интеллекта и Ассоциации вычислительной техники.

Благодаря опыту работы в Google, НАСА и Junglee Норвиг знаком как с «хакерским», так и с «инженерным» подходом к созданию ПО. В нашей беседе

он коснулся преимуществ и недостатков каждого из них. Как у бывшего профессора информатики, а ныне сотрудника одной из крупнейших корпораций по производству ПО, у него есть интересный взгляд на отношения между академической компьютерной наукой и промышленной практикой.

Мы говорили о том, как изменилось программирование за последние годы, почему никакие методы проектирования не помогут тому, кто не понимает, что делает, и в чем НАСА может выиграть, применяя менее надежное, но дешевое ПО.

Сейбел: Когда вы начали программировать?

Норвиг: В старших классах школы. У нас был PDP-8, кажется так, и уроки информатики – мы осваивали Бейсик. Там-то все и началось.

Сейбел: Какой это был год?

Норвиг: Я закончил школу в 1974 – значит, 1972 или 1973. Кое-что осталось в памяти. Помню, как учительница пыталась что-то объяснять, тасуя колоду карт. Алгоритм был примерно такой: используем генератор случайных чисел для выбора двух карт, поменяем их местами, отметив это в битовом векторе, и будем продолжать, пока не поменяются местами все карты. Помнится, я подумал: «Что за идиотизм, глупее не придумаешь». Так можно продолжать до бесконечности, потому что на какую-то пару карт можно никогда не попасть». Я еще не мог сказать тогда, что это n в квадрате, хотя такой алгоритм может быть порядка n . Я просто знал, что алгоритм неправильный. Потом я предлагал алгоритм перестановки, кажется, Кнута – от 0 до 52, затем от 0 до 51 и так далее – алгоритм порядка n . Но помню, что учительница отстаивала свой способ. Тогда я понял, что у меня, видимо, есть способности к программированию, а еще – что учителя тоже не все знают.

Сейбел: Вы сразу поняли, что ее алгоритм неправильный? Или сначала покрутили в голове, а потом уже осознали, что тут много лишней работы?

Норвиг: Сразу. Не помню в точности, о чем именно я думал, но я тут же заметил: определенно, так может продолжаться до бесконечности. Тогда я, кажется, еще не знал об ожидаемом времени выполнения.

Еще я полистывал найденные на антресолях старые номера «Scientific American» – этот журнал выписывал отец. Там была статья Кристофера Стрейчи о проектировании ПО: автор утверждал, что ожидается переход на высокоуровневые языки. Он придумал язык, компилятор для которого так и не появился, – чисто бумажный язык – и хотел писать

на нем программу для игры в шашки. Это была первая нетривиальная программа, о которой я узнал, – в школе мы не шли дальше тасования карт. Недавно я перечитывал ее и первое, что заметил, – ошибку. Просто здорово – ведь Стрейчи знал толк в программах, в «Scientific American» были редакторы, но ошибку никто не увидел. В пояснении он описывал функцию «сделать ход», которая получала позицию на доске и возвращала ход, но в коде у этой функции кроме позиции на доске был еще один параметр. Они явно начали с описания, а код писали позже. И выяснили, что бесконечный поиск невозможен, поэтому ввели дополнительный параметр – «глубину поиска»: вызывать функцию рекурсивно можно было только до установленного предела. Это добавили позже, а документацию не исправили.

Сейбел: Это была первая интересная программа, с которой вы познакомились. А какой была первая интересная программа, которую вы сами написали?

Норвиг: Наверное, Game of Life. Это было домашнее задание, которое я быстро сделал. Конечно, никаких 30-дюймовых мониторов не было – только телетайп с желтой бумагой. Я подумал, что расточительство – печатать на отдельном листе одно маленькое поле (кажется, 10×10), на следующем – еще одно и так далее. Поэтому я решил распечатать пять поколений подряд. В Бейсике не было трехмерных массивов, а я почему-то не мог просто взять несколько двумерных массивов – памяти не хватало или еще что-то. Мне нужно было пять или шесть двумерных массивов, и тогда я придумал битовые поля.

Сейбел: Итак из-за ограничений памяти вы изобрели собственный формат хранения данных. Вам рассказывали про битовые массивы, и вы просто догадались их применить? Или вы листали справочник и наткнулись на эти РЕЕК и РОКЕ или что там было?

Норвиг: Я хранил в каждой из ячеек ноль или единицу, а мне нужно было хранить где-то больше информации. И тогда я сказал: «Можно же хранить другие числа». Даже не помню, сделал ли я битовое хранилище. Кажется, я обошелся числами – скорее десятичными, чем двоичными, потому что двоичные нам преподавали плохо. Потом я еще добавил другие признаки – повторяется ли число, и если да, то с какой периодичностью. С одним предыдущим поколением этого не сделать.

Сейбел: В начале своей программистской карьеры вы что-нибудь делали, чтобы улучшить свои навыки? Или просто программировали?

Норвиг: Просто программировал. Конечно, я делал что-то просто для удовольствия, особенно на младших курсах, когда мог не сильно заботиться о расписании. Я думал: «Вот интересная задача, попробую-ка ее решить». Не потому что задали, а просто для удовольствия.

Сейбел: Вы изучали компьютеры в колледже, но по компьютерным наукам не специализировались?

Норвиг: Когда я поступал, компьютерные курсы были в ведении факультета прикладной математики. Когда я заканчивал колледж, уже был отдельный компьютерный факультет, но я специализировался по математике. Пойти на компьютерный факультет значило специализироваться по программам IBM. Надо было изучать их язык ассемблера, их операционную систему OS/360 и так далее. Не очень-то увлекательно. Я ходил на лекции, которые мне нравились, но погружаться во все это не хотел.

После колледжа я два года проработал в Кембридже, в одной компании, выпускавшей ПО. Потом я подумал: «Колледж надоел мне за четыре года, работа – за два, значит, колледж я люблю вдвое больше».

Сейбел: Чем вы там занимались?

Норвиг: Они разрабатывали инструменты для проектирования программ, а также занимались консалтингом. Основатели компании работали в Дрейперовской лаборатории в Кембридже над проектом «Аполлон» и еще над чем-то. Поэтому у них были связи в BBC и правительственные заказы. У них был интересный взгляд на то, как проектировать программы. Я не очень верил в саму идею, но это было занятно.

Одним из проектов была программа построения блок-схем – она должна была анализировать готовую программу и генерировать для нее блок-схему. Отличная мысль, потому что блок-схемы нужны. Но на самом деле их всегда рисуют не до, а после написания программ. Эта программа была умная, потому что умела работать с частичной грамматикой, так что при работе с синтаксически некорректной программой пропускала те ее части, которые не могла разобрать. Она должна была уметь разбирать, к примеру, операторы IF, поскольку те образовывали разные блоки, но по поводу других частей говорила: «Сбросим это все в один блок». И вот мы получили контракт на разработку, в котором было указано, что программа должна работать под UNIX. Мы одолжили машину в Массачусетском технологическом институте и применяли для компилятора все инструменты UNIX – уасс и так далее. В последний момент нам сказали: «Нет, использоваться будет VMS». Так что никакой уасс мы уже не могли применять, но решили, что он нам и не нужен – ведь мы брали его только для составления таблиц, а это мы уже сделали.

Сейбел: Пока грамматика не меняется, все в порядке.

Норвиг: Да, так что мы поставили программу, заказчики были счастливы, а потом, разумеется, грамматика изменилась. У нас больше не было доступа к машинам с UNIX. Мне пришлось править грамматику, роясь в таблицах, и я решил: если вот здесь у нас переход в новое состояние, я создам свое состояние, и переход будет к нему.

Сейбел: Это было правильное решение? Вы не думали о том, чтобы просто написать новый парсер?

Норвиг: Наверное, так и надо было поступить. Но ведь я сделал лишь одну небольшую правку.

Сейбел: А не получилось так, что им каждые три недели приходилось сталкиваться с очередным изменением в грамматике?

Норвиг: Я тогда уже ушел в магистратуру, и это уже была не моя проблема.

Сейбел: Уже не ваша... Итак, вы защитили диссертацию. Вы бы хотели, чтобы ваше обучение программированию шло по-другому?

Норвиг: В конце концов я занялся промышленной разработкой ПО. Возможно, стоило заняться этим раньше. Постепенно я всему научился, но много времени провел в колледже и магистратуре. Правда, мне нравилось, так что ни о чем не жалею.

Сейбел: А чему вам пришлось учиться, если говорить о промышленной разработке ПО?

Норвиг: Планировать работу и делать так, чтобы довольны были все – разработчики, клиенты, менеджеры. В магистратуре ведь все это не нужно – просто приходи к своему руководителю время от времени.

Важнейший шаг – переход от индивидуальной работы к командной: начинаешь понимать, как люди взаимодействуют друг с другом. Этому тоже не учат в колледже. Сейчас в некоторых колледжах обратили на это внимание, но в мои времена к командной работе относились плохо – считали ее жульничеством.

Сейбел: Какие навыки, кроме умения писать код, нужны тем, кто приходит в индустрию ПО?

Норвиг: Надо уметь ужиться с другими. Уметь понимать заказчика: знать, что именно ты создаешь и правильным ли путем идешь. Уметь взаимодействовать с клиентами и со своими коллегами. Уметь взаимодействовать с клиентом и своим начальством в одно и то же время. Это все разные социальные отношения, требующие разных навыков.

Сейбел: Программирование стало более социальным?

Норвиг: Думаю, да. Компьютерщики были больше обособлены от всех остальных. Раньше в основном шла пакетная обработка, интерфейсы были куда проще. Можно было работать по модели «водопада»: ввод – колода карт, вывод – отчет с таким-то номером в такой-то колонке.

Возможно, это была не лучшая модель для спецификаций, и с самого начала нужно было теснее взаимодействовать с клиентом. Но так сохранялась обособленность. Сейчас все намного подвижнее, больше вза-

имодействия, и лучше не составлять полную спецификацию в начале, а сесть вместе с клиентом и устроить мозговой штурм.

Сейбел: А были особенно памятные случаи, которые подчеркивают разницу между одиночной и командной работой?

Норвиг: Особенно памятных, пожалуй, не было. Просто пришло осознание того, что я не могу сделать все сам. Программирование – это в немалой степени способность держать в голове столько, сколько можешь, но это работает только до какого-то предела. По крайней мере, у меня это так. А потом надо полагаться на других, чтобы получить хорошие абстракции и использовать то, что у них есть. Теперь я все чаще думаю в терминах «Вероятно, это уже сделано. И как?», а не «Я знаю, как это сделано, потому что сам это сделал». По идее, вот так – а если не так, надо понять почему и научиться это использовать.

Сейбел: Освоение командного подхода позволяет самому браться за вещи большего размера, чем раньше? Вы как бы сам себе команда, только растянутая во времени?

Норвиг: Это так. И я наблюдаю это у молодых программистов. Еще одна разница между «тогда» и «сейчас»: программирование больше похоже на сборку из готовых модулей, чем на создание чего-то с нуля. Сейчас, делая домашнее задание – скажем, сайт, – школьник возьмет для одной части Ruby on Rails, для другой – Drupal, для третьей – скрипт на Python, а потом скачает статистическую программу. И все это связывается между собой с помощью скриптов, а не пишется с нуля. Сегодня важнее понимать интерфейсы и уметь их соединять, чем знать в деталях, что делается внутри ПО.

Сейбел: Значит ли это, что успешный программист сегодня – человек другого склада?

Норвиг: Успешные люди, по-моему, не изменились, насколько я вижу. Но в целом – да, в наши дни лучше уметь быстро понять, что тебе нужно, чем досконально знать все процессы. Тут, конечно, есть такая бравада: «Я просто беру и делаю это». Человек смело признается: «Я не понимаю, что тут внутри, но я залез в документацию и нашел там эти три штуки. Попробовал – работает. Значит, вперед».

Да, такой подход приносит свои плоды, но мне кажется, только за счет этого не станешь хорошим программистом. Надо знать немного больше. Безопасно ли то, что я делаю? В каких случаях это сломается? Попробовал один раз, работает – хорошо, но это должно работать всегда. Как написать тесты, чтобы проверить программу и самому лучше разобраться в ней? А когда они написаны, не взять ли мне то, что я сделал, и не опубликовать ли новый инструмент, чтобы другие им тоже пользовались?

Сейбел: Какие у вас были предпочтения в плане командной работы? Взять задачу и разделить ее между программистами? Или везде применять парное программирование с коллективным владением кодом?

Норвиг: Скорее первое. Стив Йегг написал статью «Good Agile, Bad Agile» (Agile-проектирование: хорошее и плохое), и думаю, в ней он прав. Десять процентов времени имеет смысл уделять коллективной работе, поскольку программисту необходимы понимающие слушатели. Больше – вряд ли стоит.

Двум хорошим программистам лучше работать отдельно, отлаживая затем сделанное друг другом, чем согласиться на 50% падения производительности только ради лишней пары глаз.

Иногда полезно собираться вместе для мозгового штурма, когда надо определить и задачу, и средства ее решения. Вы даже не знаете, что за продукт вам нужен, и об этом стоит подумать совместно. Затем вы делитесь с задачей, и надо решить, как разбить ее на части. Но вот после этого, когда идея пришла, лучше работать по отдельности. Нужна обратная связь, нужен другой человек, который внимательно просмотрит ваш код, – но только не в режиме реального времени, не в процессе написания.

Помнится, у IBM была модель «программиста-мастера» – глупее никто еще ничего не придумал. Кто захочет быть на побегушках у опытного программиста?

Сейбел: Удивительно, что вы считаете эту идею настолько глупой. В своей статье «Teach Yourself Programming in Ten Years» (Как самому научиться программировать за десять лет) вы утверждаете, что программирование – навык, который, как и многие другие, требует десятилетнего оттачивания, чтобы человек стал мастером. Во многих ремеслах работает модель мастер/подмастерье/ученик. Может, никому не нравится быть учеником, но что ужасного в том, что человек с десятилетним опытом будет выполнять другую работу, нежели желторотый новичок?

Норвиг: Лучшее в такой модели – возможность для ученика наблюдать за работой мастера. Вот таких возможностей должно быть больше. Это вариант парного программирования. Особенно полезно для начинающего наблюдать за тем, чему особо не учат, например за отладкой. Можно заучить алгоритмы, но так не научишься отладке. А вот смотреть, как опытный программист делает что-то, о чем ты даже не подозревал, очень полезно.

Но мне кажется, эта модель раньше была популярна из-за нехватки материалов. Допустим, у ювелира есть строго определенное количество золота. Или, скажем, когда идет операция на сердце, оно всего одно – нужен лучший специалист, остальные будут на подхвате. С програм-

мированием все иначе. Есть множество компьютеров, клавиатур, не надо распределять ресурсы.

Сейбел: К вопросу о том, чему особо не учат: вы занимались наукой, теперь работаете в индустрии. Как вы считаете, компьютерные науки и промышленное программирование правильно взаимодействуют между собой?

Норвиг: Сложный вопрос. Я не думаю, что заниматься наукой – большая потеря времени: есть шанс узнать много нужного. Но вы не узнаете всего, что вам требуется для разработки или производства ПО. Мне кажется, программы в высшей школе медленно адаптируются к реальности. Кое-где есть подвижки, но в целом работать в команде учат мало. Да и этому подходу со сборкой из готовых кусков тоже не особо учат. Однако ребята как-то набираются всего этого, так что в общем дело обстоит неплохо. У нас в Google много масштабных облачных вычислений, параллельного программирования и тому подобного. Индустрия во всем этом заинтересована, однако учат этому редко. Так что высшая школа немного запаздывает, но все равно остается полезной.

Сейбел: Есть ли области, где ученые находятся на переднем крае, где программная индустрия еще не подобралась к новейшим методам?

Норвиг: Отчасти да, есть. Лучший, наверное, пример – проверка моделей, которой Intel не уделила достаточно внимания, и на отзыве ПО из-за найденной ошибки в умножении они потеряли много денег. Тогда они всполошились и пошли на поклон к ученым. Теперь проверка моделей обязательно входит во все их программы. Другой пример, пожалуй, чуть менее яркий – языки программирования. В этой сфере идет напряженная работа, но она мало отражается на новых языках. Операционные системы, в какой-то мере. Мы финансируем лабораторию RAD в Беркли с Дэйвом Паттерсоном и другими. У них есть хорошие идеи относительно надежности систем. Но это тот случай, когда у индустрии есть куда более серьезные проблемы. Не каждую из них удастся решить, но поиски здесь идут более интенсивно, чем в университетах.

Сейбел: Значит, вы не считаете, что часть хороших идей, порожденных учеными, не используются просто из-за нежелания перемен? Ведь множество доморощенных РНР-программистов никогда не интересуются языком Haskell, пусть даже он будет удобнее?

Норвиг: Тут я настроен скептически. Будь у этого языка серьезные преимущества, его бы уже активно использовали. Не думаю, что у нас идеальный рынок информации, где все тут же бросаются применять новое оптимальное решение, но мы близки к этому. Ученые могут не видеть всю проблему, стоящую перед индустрией, и часть ее – проблема обуче-

ния. При множестве программистов, не знающих, что такое монада, не слушавших курса теории категорий, наступает разрыв.

Отчасти причина – в наследии прежних систем, которое нельзя просто так взять и отбросить, – необходим переход. Уверен, есть много областей, где промышленникам следовало бы смотреть дальше: если мы не можем осуществить переход сейчас, давайте хотя бы задумаемся, где мы будем через десять лет, в каком направлении нам двигаться.

Но мы хотим улучшений в тех областях, где это будет иметь большой резонанс. Во многих случаях языки нацелены на слишком низкий уровень, чтобы иметь тот эффект, на который рассчитывают их создатели. Человек говорит: «С моим новым прекрасным языком вот эти шесть строк кода заменяются двумя». Да, это неплохо: язык станет более эффективным, легче будет поддаваться отладке и обслуживанию. Но, возможно, ваш код – всего лишь часть большой системы. Настоящая головная боль начинается, когда надо ежедневно обновлять данные, рыться в Сети, добывать данные, переводить их в нужный формат. Нужно помнить, что вы решаете лишь небольшой подраздел громадной задачи, поэтому вам нужно преодолеть большой барьер, чтобы сделать переход на новый язык оправданным.

Сейбел: Оставляя в стороне исследования в области языков, как вы считаете – мы далеко продвинулись с тех пор, как компьютерные науки ограничивались постижением продуктов IBM?

Норвиг: Да. Сейчас она на хорошем уровне, и жаль, что ее выбирают так мало студентов – все меньше и меньше. Конечно, есть те, кто любит компьютеры и проектирование программ настолько, что в любом случае пойдут туда. Они – наша надежда. Но есть блестящие умы, которые предпочитают физику, или биологию, или еще какую-нибудь из актуальных дисциплин. А кто-то говорит: «Я вроде как люблю компьютеры, но здесь никаких перспектив – все программы пишутся в Индии. Пойду-ка лучше в юристы». Это очень обидно. Мне кажется, людей сбивают с толку.

Сейбел: Обидно потому, что иначе они получали бы удовольствие от программирования? Или потому, что эти мозги нужны отрасли?

Норвиг: И то и другое. Можно получать удовольствие от нескольких вещей сразу, в таком случае не обязательно идти в компьютерные науки. Но мне кажется, здесь просто есть некая недоработка. Нам нужны толковые люди, способные изменить мир. И если это действительно то, чего они хотят добиться, то в компьютерные науки должно идти больше студентов, чем сейчас.

Сейбел: В одной из статей Дейкстра утверждает, что компьютерные науки – это раздел математики. Поэтому те, кто их изучает, вначале – первые сколько-то лет – вообще не должны прикасаться к компьютеру,

а вместо этого осваивать обращение с системами формальных символов. Как по-вашему, сколько математики нужно грамотному программисту?

Норвиг: Ну, я не считаю, что надо учиться до уровня, о котором говорит Дейкстра. Кроме того, он сосредотачивает внимание на определенной области математики – дискретные, логические доказательства. Я пришел из сферы, где это не так важно, где все в большей мере основано на вероятности. Мне редко попадаются программы, которые можно проверить формально.

Код Google – он правилен или нет? Введя в Google этот запрос, вы получите десять страниц. Если поисковик даст сбой, код неправилен. А если он даст вот эти десять ссылок вместо вон тех десяти – правилен или нет? Можно говорить о том, что вам больше нравится, но дальше этого вы не пойдете. Это немного не то, о чем мы привыкли думать. Как только сталкиваешься с задачами такого рода или задачами вроде передвижения роботизированного автомобиля по улицам города, чтобы он никого не сбил, логические доказательства отбрасываются очень быстро.

Сейбел: Есть ли базовые навыки, необходимые хорошему программисту? В разных сферах, конечно, разные требования, но есть ли нечто общее в написании кода независимо от сферы деятельности?

Норвиг: Нужно уметь двигаться вперед и улучшать сделанное. Это все, что необходимо в жизни. Нужно порождать идеи, претворять их в жизнь, а потом совершенствовать сделанное. Совершенствовать можно по-разному. Можно сказать себе: «Я сделал это не совсем правильно, некоторые случаи не охвачены». А можно сказать себе так: «Теперь я понимаю это лучше, я создам более абстрактные инструменты, и в следующий раз мне будет легче создать такую систему». «В каком направлении я иду?», «Как я сделал это?», «Можно ли сделать это лучше?» – вот какие вопросы нужно перед собой ставить.

Сейбел: Считаете ли вы, что этот навык – по сути, сделал, отладил, повторил – стоит усвоить многим, и не только программистам? Если бы составляли программу для школы или колледжа, вы бы внесли в нее обязательное программирование для всех? Или это требует особых навыков?

Норвиг: Да, это требует особых навыков. Можно привести и другие примеры для этого типа мышления. Возьмем чисто механическую задачу: есть несколько деталей, и надо сделать так, чтобы вода в конце концов попадала вот в эту чашку. Речь не обязательно о программах – речь о том, чтобы соединять разрозненные элементы и проверять, как они работают в сборе.

Сейбел: Как глубоко нужно изучать программирование? В статье «Как самому научиться программировать за десять лет» вы говорите о том, сколько времени занимает выполнение инструкции по сравнению с чтением с диска, и так далее. Нужно ли программистам, как раньше, знать язык ассемблера?

Норвиг: Не знаю. Кнут советует делать все на языке ассемблера, поскольку Си слишком неэффективен. Я с ним не согласен. Нужно знать кое-что насчет эффективности и неэффективности инструкций, но это больше не относится к каждой конкретной инструкции. Теперь это не о том, исполняется последовательность из трех или из двух инструкций, а о том, случился ли у вас сбой по странице памяти или вы не попали в кэш. Мне кажется, знать язык ассемблера уже необязательно. Нужно понимать архитектуру. Нужно понимать, что такое язык ассемблера, понимать, что есть иерархия памяти и что сбой в переходе с одного уровня на другой сильно отражается на работе программы. Но это понимание может быть и на абстрактном уровне.

Сейбел: Есть ли, по-вашему, книги, которые должен прочесть каждый программист?

Норвиг: Выбор велик, тут можно пойти разными путями. Надо прочесть что-нибудь про алгоритмы – программист не должен становиться только склейщиком программ. Можно взять Кнута, а можно Кормена, Лейзерсона и Ривеста. Есть и другие авторы, например Салли Голдман. В ее последней книге алгоритмы рассматриваются с практической стороны. Довольно интересно написано. Кроме того, что-нибудь про идеи абстракции. Мне нравится труд Абельсона и Сассмана, но он не единственный.

Надо досконально знать свой язык. Читайте справочники. Читайте книги, где объясняется механика языка и одновременно способы отладки и тестирования, – «Beautiful Code»¹ или что-нибудь подобное. Но я бы не стал советовать читать исключительно это, а не что-нибудь другое.

Сейбел: Сейчас вы не так много программируете по работе, но пишете небольшие программы для статей, которые появляются на вашем сайте. Каков ваш сегодняшний подход к программированию?

Норвиг: По-моему, крайне важно умение держать все в голове. Тогда намного больше шансов на успех, легче написать маленькую программку. Для больших программ придется использовать дополнительные инструменты.

Еще очень важно понимать, что делаешь. Когда я писал программу для решения sudoku, некоторые блогеры комментировали ее в таком духе:

¹ Стивен Макконнелл «Совершенный код». – СПб.: Питер, 2007.

«Сравните, вот это написал Норвиг, а это другой парень, один из гуру разработки через тестирование (забыл, как его зовут). Он сказал, что сначала напишет несколько тестов». Но у него ничего не вышло. Он вывесил пять постов в своем блоге, там было много тестов, но ничего для решения задачи. Он не знал, как ее решать.

А я был знаком со сферой искусственного интеллекта, где есть то же самое распространение ограничивающих условий. Я знаю, как это работает. Рекурсивный поиск – я знаю, как это работает. И я с самого начала видел, как объединить их для решения sudoku. А тот парень блуждал в потемках, даже если благодаря своим тестам и создал «работающий» код.

Блогеры начали оживленно обсуждать, что это значит. По-моему, ничего не значит. Разработка через тестирование – отличная вещь, я сейчас применяю ее намного чаще, чем раньше. Но можно тестировать что угодно и при этом не знать, как подойти к решению задачи.

Сейбел: А что нужно, чтобы знать это? Защитить диссертацию и поработать с искусственным интеллектом? Невозможно ведь знать все алгоритмы. Сегодня у вас есть Google, но найти правильный подход к задаче – не то же самое, что найти веб-фреймворк.

Норвиг: Как узнать, чего именно не знаешь?

Сейбел: Вот именно.

Норвиг: Думаю, ответ состоит из двух частей. Первая – признать, что задача, возможно, уже решена. Можно думать, что, конечно же, никто в принципе не знает, как ее решать, поэтому действовать наудачу – метод не хуже любого другого. А можно предположить, что решение где-то есть, и нужно просто найти слова, которыми оно может быть описано. Это отчасти вопрос интуиции – нужно догадаться, что искать следует, скажем, в области искусственного интеллекта. Потом уже ищешь конкретные методы. Может быть, тот парень, искав по слову «судoku», нашел бы верный ответ. А может, он считал это жульничеством. Не знаю.

Сейбел: Предположим, что это так. Пусть вы были бы первым, кто пытался решить sudoku. Методы, которыми вы в итоге воспользовались, все так же существовали бы и ждали своего применения.

Норвиг: Допустим, я решаю некую задачу в области биологии. Я не знаю, какие алгоритмы применяются при генетическом секвенировании, но знаю, что они есть. И я оглядываюсь вокруг. На другом уровне некоторые из таких вещей довольно фундаментальны – если вы не знаете, что такое динамическое программирование, это большой минус для вас. И это будет постоянно проявляться, если у вас нет представления об этой общей модели поиска – сделать выбор и вернуться, когда необходимость отпадет. Это идеи родом из 1960-х. Люди открыли это, когда занимались программированием всего несколько лет. Это то, о чем

каждый обязан иметь представление. А идеи годовой давности, конечно, известны не каждому.

Сейбел: Значит, надо рыться в старых статьях?

Норвиг: Нет. Было много тупиковых направлений. Потом обнаружилось, что некоторые области, развивавшие каждая свою технологию и терминологию, на самом деле были очень близки друг другу. Лучше бросить взгляд на прошлое из современности, чем заново повторять все шаги. Но взглянуть туда все равно необходимо. Не знаю даже, какие книги посоветовать, — я осваивал все это по частям.

Сейбел: Вернемся к проектированию программ. Если вы работаете над большой программой и не можете вспомнить, как соединяются все фрагменты кода, то что тогда делаете?

Норвиг: Нужна хорошая документация на уровне системы в целом: что она должна делать, как этого добиться. Документировать каждую функцию обычно слишком утомительно. Большей частью лишь повторяется то, что и так можно понять из имени и параметров функции. Но общая документация очень важна, ее следует готовить в первую очередь. Она должна быть понятной всем и предлагать верные решения. Для успешного проекта нужны опытные люди, предлагающие верное решение. Кроме того, когда вы делаете что-то совсем для себя новое и не знаете, как к этому подступиться, лучше всего проявлять гибкость — настолько, чтобы неверное решение можно было пересмотреть.

Сейбел: Создавая что-то совсем новое для вас, как долго вы можете сидеть и обдумывать задачу? Или для лучшего понимания вам нужно начать писать код?

Норвиг: Иногда для осмысления задачи надо вернуться назад. Вы хотите получить на выходе что-то работающее — для некоторых задач есть только один путь к этому. Для других задач есть много равнозначных путей. Все зависит от вида задачи.

Следующий шаг — отделить сложные пути от легких. Вам придется очень туго при неверном выборе архитектуры — если вы нарушили встроенные ограничения или просто создаете не то, что нужно. В Google мы все время сталкиваемся с проблемами этого типа. Есть также проблема масштабирования. Исходя из сегодняшнего состояния вы говорите, что программа должна быть, скажем, в десять раз мощнее — через несколько лет этот показатель будет недостаточным и программу можно выбрасывать. Но важно, по крайней мере, сделать правильный выбор для заданных рабочих условий. Вы собираетесь обрабатывать от миллиарда до десяти миллиардов страниц. Как распределить это по машинам? Каков окажется объем входящего и исходящего трафика? Надо, чтобы на этом уровне все держалось крепко. Что-то можно рассчитать

на салфетке, что-то – при помощи симуляции, а в чем-то вам придется предсказать будущее.

Сейбел: Для такого рода задач подсчеты на салфетке или симуляции полезнее, чем написание кода.

Норвиг: Наверное, да. Здесь подсчеты будут лучшим выходом. Затем вам могут сказать, что в будущем году появится коммутатор, позволяющий пропускать вдесятеро больший трафик. Полагаться на это – или рассчитывать исходя из сегодняшних возможностей? И такие вопросы возникают на каждом шагу.

Есть пользовательские интерфейсы, про которые что-то можно понять только после их создания. Считаешь какое-то взаимодействие очень удобным, а половине пользователей оно оказывается недоступным. Тогда возвращаешься назад и придумываешь что-то новое.

Сейбел: Отвлечемся от пользовательских взаимодействий. Когда бывает полезным создавать прототипы, а не просто смотреть и обдумывать, как все должно работать?

Норвиг: Думаю, полезно придумать решение и посмотреть, как оно работает, удобно ли это. Вам нужны инструменты, которые позволят построить систему сейчас и совершенствовать ее потом. Тогда вы создаете прототип, и он оказывается неуклюжим – возможно, неверно задан набор примитивов. Лучше выяснить это как можно раньше.

Сейбел: Как насчет разработки через тестирование?

Норвиг: Тесты для меня – скорее метод исправления ошибок, чем разработки. Этот крайний подход к проектированию мне не кажется правильным. Мы пишем тест, на который знаем верный ответ, программа его не прошла – думаем, что делать дальше.

Это имело бы смысл только в случае одного-единственного заранее известного решения. Надо подумать прежде всего об этом. Каковы ваши строительные элементы? Как вы будете писать тесты для элементов, о которых еще ничего не знаете? Но если элементы известны, то полезно иметь тесты для каждого из них, чтобы понимать, как элементы взаимодействуют друг с другом, какие у вас пограничные случаи и так далее. Здесь тесты необходимы. Но нельзя построить на них проектирование целиком.

Еще мне не нравится вот что – мы в Google то и дело сталкиваемся с этим: программа не укладывается в простую булевскую модель теста. В тесте есть `assertEqual`, `assertNotEqual`, `assertTrue` и так далее. Это хорошо, но нам нужно также иметь `assertAsFastAsPossible`. Нам нужны утверждения относительно огромной базы данных возможных запросов: мы получаем результаты, которые оцениваются по стоимости до-

стижения определенной точности, стоимости повторного вызова – и хотим их оптимизировать. Но вот этих статистических или непрерывных значений, которые мы хотим оптимизировать, в тестах нет. А от булевского типа – «верно или нет» – проку мало.

Сейбел: Но в конце концов все можно свести к булевским типам – послать сколько-то запросов, получить все эти значения и посмотреть, находятся ли они в пределах допустимого.

Норвиг: Да. Но просто поглядев на методы, применяемые в тестах, вы поймете, что они не годятся для этого, что в них не предусмотрена такая возможность. Я поражаюсь тому, насколько этот подход распространен в Google. Работая в Jungle, я однажды читал об этих вещах лекцию в отделе контроля качества. Мы занимались исследованиями покупок и сказали тогда: «Нужен тест – можем ли мы по такому-то запросу получить 80% верных ответов». Они спросили: «Хорошо, а неверный ответ – это ошибка в программе?» Я сказал: «Нет, один неверный ответ здесь не будет ошибкой». И они удивились: «Как так, неверный ответ не считается ошибкой?» Как будто здесь был выбор только между «да» и «нет», в то время как это скорее напоминало компромисс.

Сейбел: Однако вы по-прежнему верите в модульные тесты. Как программисты должны продумывать тестирование?

Норвиг: Нужно писать много тестов, думая о разных условиях. Нужны и модульные тесты, и сложные регрессионные тесты. Нужно думать о вариантах отказа оборудования. Помнится, один из лучших уроков программирования я получил однажды в аэропорту Хитроу, когда там не было электричества и все компьютеры не работали. Тем не менее мой самолет улетел вовремя.

У них были распечатки для всех рейсов. Не знаю, откуда они взялись, возможно с какого-то компьютера за пределами здания. Может, они распечатали их именно в то утро, а может, делают это каждую ночь, а днем выбрасывают, если с электричеством все в порядке. Но распечатки были, и работники на каждом выходе пользовались ими вместо компьютеров.

Отличный урок проектирования программ. Мало кто задается вопросом: «Как будет работать моя программа, если отключат электричество?».

Сейбел: А как в этом случае идет работа в Google?

Норвиг: Работа в Google в этом случае идет так себе. Но у нас есть резервное питание и множество дата-центров. Мы продумываем такие варианты: что будет, если сервер, с которым идет соединение, недоступен? Что будет при каком-либо другом отказе? Программа выполняется

на тысячах машин – что будет, если одна выйдет из строя? Можно ли это вычисление перезапустить в другом месте?

Сейбел: В очерке о разработке TeX Кнут говорит о переключении сознания: как стать своим собственным инспектором по качеству и начать беспощадно крушить свой же код. Как по-вашему, разработчикам в целом это удается?

Норвиг: Нет. И в качестве примера могу привести свою программу проверки правописания. Я сделал ошибку в коде, который оценивал мое правописание, и одновременно немного изменил реальный код. Запустил его – и получил для себя результаты гораздо лучше обычных. И я поверил им! Будь результат очень плохим, я бы стал разбираться. Но тут я поверил в свои хорошие результаты, которые получились из-за небольшого изменения в программе. А надо было трезво все оценить и понять, что результаты не могут так сильно отличаться, что где-то допущена ошибка.

Сейбел: Как вы избегаете чрезмерной универсализации и создания того, что не понадобится, а только приведет к лишней трате ресурсов?

Норвиг: По этому вопросу идет борьба, даже горячая борьба. Но меня лучше не спрашивать – я всегда предпочитал изящные решения практичным. Поэтому я вынужден сражаться с собой, напоминая себе, что в своей работе не могу себе этого позволить. Приходится повторять – и себе, и своим коллегам: «Мы ищем самое разумное решение, и если есть идеально красивое, не факт, что оно применимо» или так: «Мы занимаемся тем, что важнее всего в данный момент». Есть поговорка «Лучшее – враг хорошего». Каждый инженер обязан ее помнить.

Сейбел: Почему так соблазнительно решать задачи, которые на самом деле не актуальны?

Норвиг: Мы любим чувствовать себя умными и любим завершенность. Мы хотим как можно скорее завершить работу – закончить и двигаться дальше. Мне кажется, так устроен человек – он может чем-то позаниматься, но потом ему хочется сказать: «Все, готово, выкидываю это из головы и иду дальше». Но надо еще подсчитать рентабельность полностью завершенной работы. Это кривая в форме S – после 80- или 90-процентной готовности рентабельность неуклонно снижается. И у вас есть сотня других проектов, где вы находитесь внизу кривой и рентабельность намного выше. В какой-то момент вы говорите: «Хватит с этим, переходим к более рентабельным вещам».

Сейбел: Как научить программистов понимать, на каком отрезке кривой они находятся?

Норвиг: Нужна правильная рабочая среда, ориентированная на результат. Думаю, люди способны учиться сами. Вы хотите оптимизировать

что-то, но предоставленные сами себе, вы оптимизируете ваше личное удобство. А надо иметь в виду другое, одни говорят – рентабельность, другие – удовлетворенность клиента. Что лучше для клиента – если я закончу эту программу с 95-процентной готовностью или примусь за десять других, где готовность 0%?

В Google с этим проще – здесь действует принцип «выпускать как можно раньше и чаще». Причин тому несколько. Прежде всего, большая часть наших продуктов распространяется бесплатно: значит, выпускаем как можно раньше – кто будет жаловаться? И потом, мы ведь не штампует и не расфасовываем компакт-диски, поэтому не совсем готовый продукт, даже продукт с ошибками – это не катастрофа. Программы в основном хранятся на наших серверах, можно исправить их завтра, и обновление будет у всех мгновенно. Нас не преследует кошмар установки обновлений. Мы рассуждаем так: «Запустим это, посмотрим на реакцию пользователей, исправим то, что нужно, и не будем волноваться об остальном».

Сейбел: Проектируя большую программу, чем вы пользуетесь – блокнотом, линованной бумагой, UML-программой для рисования?

Норвиг: Нет, все эти UML-инструменты мне никогда не нравились. Я всегда считал, что, если это нельзя выразить на самом языке, это недостаток языка. Многое приходится делать на более высоком уровне. В Google немалая часть работы связана с разбиением программ на модули и организацией их параллельного выполнения. Нам нужно запустить программу на большом числе машин, но у нас столько-то пользователей, столько-то данных для приложений – как все это будет работать? Поэтому мы скорее думаем в терминах машин и машинных комплексов, чем на уровне функций и взаимодействий. Если это улажено, можно переходить к частным функциям и методам.

Сейбел: Описания делаются на уровне простого языка?

Норвиг: В основном, да. Иногда кто-то рисует картинки, рассуждая в таком духе: «У нас есть сервер, который обрабатывает такие-то запросы, он подключается к другому серверу, мы используем различные средства хранения, большие распределенные хеш-таблицы и так далее. Мы берем три готовых инструмента и дальше выясняем, нужен ли новый инструмент, какой из этих трех будет работать, потребуется ли что-то еще».

Сейбел: Как оцениваются такие схемы?

Норвиг: Их показывают тем, кто уже этим занимался, и они говорят: «Похоже, здесь вам нужен кэш – система станет тормозить, но поскольку тут много повторяющихся запросов, кэш такого-то размера должен помочь». Есть стадия ревизии проекта – на ней решается, есть ли у него смысл, а потом начинается разработка и тестирование.

Сейбел: У вас принято устраивать формальные ревизии проекта? Вы ведь работали в НАСА, а там с этим строго.

Норвиг: Да, строже, чем в НАСА, не бывает. У нас планка ниже – как я уже говорил, мы можем допустить недочет и исправить его. В НАСА первый же недочет будет фатальным, и они вынуждены быть куда внимательнее. А мы не сильно на этот счет беспокоимся. Скорее консультации, чем ревизии.

Есть, конечно, те, кто официально занимается чтением предложений по проектам и одобряет или отклоняет их. Но это намного менее формальная процедура, чем в НАСА. Это происходит перед запуском. В ходе реализации бывают проверки, но в коде никто не копается, просто задают вопросы: «Как все идет? С отставанием от графика или с опережением? Есть ли большие проблемы?» Примерно на таком уровне.

Самая формальная часть – запуск проекта. Есть таблица контрольных проверок – в плане безопасности все проверяется очень тщательно. Если мы это запустим, сможет ли кто-то получить доступ через межсайтовый скриптинг? Тут все довольно строго.

Сейбел: Вы рассказывали, как проверяли Гвидо ван Россума на знание Python, а Кена Томпсона – на знание Си, выясняя, способны ли они придерживаться довольно жестких стандартов кодирования. Для проектирования у вас такие же жесткие стандарты?

Норвиг: Нет. Некоторые стандарты кодирования касаются вопросов проектирования, но все гораздо свободнее. Разумеется, есть определенная политика – нужно получить разрешение на участие в написании кода. И каждое изменение кода кем-то контролируется и проверяется.

Сейбел: Значит, любое изменение кода в хранилище р4 контролируется?

Норвиг: Можно писать экспериментальные фрагменты для себя. Есть исключения, когда ревизия выполняется позднее. Но лучше так делать пореже.

Сейбел: То есть это эквивалент классической проверки – кто-то смотрит ваш код и подтверждает, что в нем все в порядке.

Норвиг: Да. На самом деле это был первый проект Гвидо. Мы использовали для этого утилиту *diff*, но это слишком громоздкий способ. А Гвидо создал распределенную систему с красивым интерфейсом и подсветкой, так что ревизии кода облегчились.

Сейбел: Во многих компаниях говорят, что ревизии нужны, но не все этому следуют. Ведь этому людей надо обучать.

Норвиг: Думаю, такие вещи делались всегда, и люди к этому привыкают сразу. Ну, не все – некоторым требуется время. Вот типичный

случай: в компанию приходит не привыкший к этому новичок, создает экспериментальную ветвь и делает все в ней. Вы говорите: «Слушай, ты же не зафиксировал ни одного изменения». Он отвечает: «Да-да-да, я всего лишь кое-что чищу, зафиксирую завтра». Проходит неделя, другая, и в конце концов фиксируется одно гигантское изменение. Прошло много времени, весь этот объем изменений сложно оценить, да и то, с чем надо было сравнивать, тоже успело измениться. Новичок видит, какая это головная боль, и впредь так не поступает.

Сейбел: Понятно. А проверяющим нужны какие-то навыки?

Норвиг: Известно, что один проверяет лучше, другой – хуже. Приходится всегда делать выбор, кому отдать код на ревизию – тому, кто делает качественно, или тому, кто делает быстро.

Сейбел: Чем отличаются хорошие проверяющие?

Норвиг: Они отслеживают больше разных вещей. Иногда вы ошибаетесь в чем-то банальном – скажем, пробелы в отступах, но иногда вам предлагают изменить структуру кода – переставить такой-то кусок в другое место. Одни подходят к этому добросовестно, другие не заморачиваются.

Сейбел: В связи с этим хочу спросить: каждый ли хороший программист по мере роста становится хорошим проектировщиком? Или некоторые программисты блестящи только на своем уровне, и им никогда не дадут проектировать сложные программы?

Норвиг: У всех разные умения. Один из лучших наших специалистов по поиску не очень хорошо программирует – код выходит средний. Но, допустим, его спрашиваешь: «Вводится новый фактор – сколько раз человек щелкает на этой странице, сделав то и то. Как учесть его в наших результатах поиска?» И он отвечает: «В строке 427 есть альфа-переменная, возьмите новый фактор, возведите в квадрат, умножьте на 1,5 и прибавьте к переменной». Через несколько месяцев экспериментов с разными подходами выясняется, что он был прав, только умножать надо, скажем, на 1,3.

Сейбел: Значит, он четко представляет работу программы.

Норвиг: Он прекрасно понимает код. Другие пишут лучше, но он сразу прикидывает все последствия изменений в коде.

Сейбел: А это как-то связано между собой? Нередко тот, кто пишет самый жуткий спагетти-код, как раз и способен удержать его в голове целиком. Ведь только поэтому он так и пишет.

Норвиг: Да, не исключено.

Сейбел: Итак, проверки у вас не такие формальные, как в НАСА. Что еще скажете о разнице между культурой «инженеров» и «хакеров», в лучшем смысле обоих слов?

Норвич: Разница есть в организационной структуре и в отношении к ПО. Google начинался как компания, производящая ПО, и в те времена были наняты исполнительный директор – PhD компьютерных наук из Беркли, вице-президент по продажам – бывший компьютерный инженер, и так строилась вся фирма. А в НАСА все они специалисты по ракетам! Они мало соображают в программах, считая их необходимым злом. «Линейный код я еще понимаю, но вот циклы – это уже подозрительно. А если там еще и условие *внутри* цикла – о, это уже слишком далеко от того, что я могу решить через дифференциальное уравнение из теории управления!» Там все настроены очень недоверчиво.

Сейбел: Но так и нужно.

Норвич: Да, так и нужно. Еще они не любят инноваций. Если сказать кому-нибудь: «У меня есть новый отличный прототип, посмотри», – он ответит: «С удовольствием использую это в своем запуске, после того как он успешно отлетает два других». И все говорят одно и то же.

Дон Голдин пришел на административную работу в НАСА и сказал: «Лучше, быстрее, дешевле – вот что нам нужно. Запуски обходятся слишком дорого. Лучше делать больше запусков, пусть некоторые будут неудачными, но все равно мы сделаем больше за те же деньги». И поспорить было сложно. К сожалению, политическое решение оказалось неверным. Потерять спутник – совсем не здорово, потому что люди запоминают только одно: НАСА потеряла спутник. Разницы между спутником за 100 миллионов и за 1 миллиард они не видят. Так что потерять десять стомиллионных спутников и один миллиардный – разные вещи. Поэтому решение оказалось не совсем верным.

Сейбел: Какова худшая ошибка из всех найденных вами?

Норвич: С самыми большими ошибками (не моими) я имел дело, когда участвовал в чистке после программных сбоев на Марсе в 1998 году. Первый – из-за того, что вместо ньютонов были указаны футофунты. И второй – преждевременное отключение двигателей из-за сбоя программы, – мы так думали, но без полной уверенности.

Сейбел: Помню один из отчетов по Mars Climate Orbiter – как раз ту историю с футофунтами и ньютонами. Вы там были единственным специалистом по компьютерам. Вы тоже участвовали в разговоре с производителями ПО, выясняя проблему?

Норвич: Ну, после событий, когда нашли причину, все было просто. Они разобрались довольно быстро. А потом было расследование – как

такое могло случиться? Думаю, было несколько причин. Первая – аутсорсинг: одни программы разрабатывала JPL в Пасадене, другие – Lockheed-Martin в Колорадо. Два специалиста из двух разных команд попросту не пообедали вместе, иначе, убежден, этой проблемы бы не было. А так, один написал другому по электронной почте: «С этими измерениями что-то не так, кажется, мы слегка просчитались. Но не очень сильно, все должно быть в порядке».

Сейбел: Это происходило уже во время полета?

Норвиг: Да. И даже во время полета была возможность выловить ошибку. Они знали, что не все в порядке, поэтому и послали письмо, но не отметили это в системе отслеживания ошибок. Если бы они это сделали, в НАСА очень хорошо налажено отслеживание ошибок, и все можно было бы исправить даже на поздней стадии. А вместо этого – неформальное письмо, которое так и осталось без ответа. В JPL считали, что в Lockheed-Martin решили проблему, а в Lockheed считали, что раз JPL больше не спрашивает, значит, волноваться не о чем.

Так что перед нами проблема коммуникации. Но это еще и проблема повторного использования кода. В НАСА превосходная система проверок критически важных программ. При запуске предыдущего аппарата кое-что тоже записывалось в футофунтах, но это был просто лог, он не использовался для навигации. И проблему не отнесли к критически важным. А в марсианском проекте изменилась навигационная система, и лог-файл стал входными данными для навигации.

Сейбел: Выходит, в одном месте генерировались данные в футофунтах, а затем передавались в другую программу, которая обсчитывала ввод, ожидая, что данные будут в ньютонах?

Норвиг: Именно так. Другой ключевой проблемой было слишком большое число солнечных частиц. Космический аппарат асимметричен из-за солнечных батарей, он слегка вращается из-за этих частиц, так что необходимо для противовеса запускать ракетные двигатели. Сотрудник Lockheed, недавно принятый на работу, разговаривал с производителем ракетных двигателей, у которого все спецификации были в футофунтах. Так они и оказались записаны – сотрудник не знал, что НАСА требует данных в метрической системе.

Сейбел: Прочитав отчет, я был поражен позицией НАСА. Позиция примерно такая: «Проблема возникла из-за программной ошибки, но у нас было много других способов определить, что аппарат отклонился от заданного курса, и мы должны были это сделать. Мы в любом случае должны были выправить ситуацию, даже если наши данные оказались неверными из-за глупейшего программного глюка». Восхитительно.

Норвиг: Да, они заняли позицию стороннего наблюдателя.

Сейбел: Часто ли встречаются крупные ошибки, о которых мы ничего не знаем, потому что другие процессы позволяют системе работать?

Норвиг: Думаю, да. В вашем компьютере миллионы ошибок, но он тем не менее работает.

Сейбел: Говорят, программы для шаттлов стоят чуть ли не 1500 долларов за строку, поскольку пишутся очень тщательно и предположительно не содержат ошибок. Это просто слухи?

Норвиг: Нет, это похоже на правду. Но, по-моему, это не оптимальное решение. Программы с ошибками служили бы им лучше.

Сейбел: Более дешевые и производительные?

Норвиг: Да. Астронавтам приходится запоминать массу всего. А надо учить их обращаться лишь с тем, чего не может сделать программа. Астронавтов же помещают в симуляторы, прокручивают перед ними разные ситуации, и когда что-то не так, перед тобой мелькают картинки на экране. Ты не можешь остановиться, прокрутить назад, выявить самое важное. А учить надо так: «Если вы видите вот это, значит, происходит вон то». К ним приходят сотни сообщений подряд об отказе какой-нибудь электрической штуки, и они должны ответить наизусть: «Похоже, изначально отказало вот это, а остальные сообщения идут из-за каскадного отключения». Почему не передоверить все это программе? Но в НАСА не пытаются – не хотят связываться с этим.

Сейбел: Сменим тему. Какие методы и инструменты отладки вы предпочитаете? Операторы печати? Формальные доказательства? Символические отладчики?

Норвиг: Все вместе, по ситуации. Иногда я использую IDE, которая хорошо умеет трассировать, а иногда Emacs, в котором всего этого нет. Конечно, я трассирую и распечатаваю. И думаю. Пишу небольшие тесты, слежу за их выполнением, разбиваю функциональность на части, чтобы понять, где тест не прошел. Честно признаюсь: я часто переписываю код, порой даже когда не нахожу ошибок. Я просто чувствую, что вот в этой части она есть. Что-то меня в этой части беспокоит. Она слишком запутанная. Так не должно быть. Вместо того чтобы внести несколько мелких изменений, я переписываю несколько сотен строк за один раз, и ошибка уходит.

Иногда после этого я чувствую себя виноватым. Ведь я не понял, что это за ошибка, не нашел ее. Я просто разбомбил дом и выстроил новый. В каком-то смысле ошибка ускользнула от меня. Но если решение эффективно – что ж. Это быстрее, чем отыскивать ошибку.

Сейбел: Как насчет утверждений или инвариантов? Насколько формально вы подходите к ним при написании кода?

Норвиг: Скорее неформально. Я не пользуюсь языками со сложными формальными механизмами сверх описания типов, вроде инвариантов цикла. Мне всегда казалось, что хлопот от них больше, чем выгод. Иногда бывает, что цикл не завершается, но это редко, а формальная часть сильно все тормозит. Если же возникла проблема, отладчик сообщит, в каком вы сейчас цикле. Если вы пишете код, от которого зависит много другого кода, и он во что-то встроен, то, конечно, следует доказывать все. Но если встает выбор – выпускать первую версию программы или отлаживать ее, я стою за скорость. Формальная спецификация может подождать.

Сейбел: Вы делали что-нибудь специально, чтобы научиться на собственных ошибках?

Норвиг: Да, это очень интересно, и я хотел бы делать больше в этом смысле. Сейчас я веду переговоры об эксперименте, в масштабах компании или даже всего мира, который облегчил бы понимание таких вопросов. Как классифицировать ошибки и какие факторы надо учитывать в плане продуктивности? Возможно, тип личности программиста? Какие именно программисты более продуктивны? Интересно, какие поддающиеся учету факторы помогают одним программистам работать лучше других? Если это зависит от размера монитора, может, надо дать всем мониторы покрупнее?

Сейбел: Если выяснится, что чем меньше монитор, тем выше эффективность, вас возненавидят.

Норвиг: Верно. Если важна тишина, давайте ее добьемся, но если также важно взаимодействие между членами команды – как увязать одно с другим?

Я только недавно начал думать в этом направлении. Как поставить эксперимент? Что отслеживать? Есть ли пригодные для использования количественные данные, к которым надо только приложить опросник? Надо ли вообще ставить такой опыт?

Сейбел: Часто утверждают, что эффективность работы разных программистов может отличаться на порядок. Но это утверждение и критикуют, говоря, что соответствующие исследования проводились давно, что в программировании многое изменилось, что одни программисты использовали пакетную обработку данных, а другие – программные среды с разделением времени.

Норвиг: Наверное, дело не только в этом – думаю, отличались и программисты одной организации, которые пользовались одинаковыми инструментами. Критиковали и за использование корреляций при непонимании причин и следствий. Почему программисты в больших угловых кабинетах работают эффективнее? Потому что просторный кабинет

дают лучшему программисту или в нем лучше работает и обычный программист? Такое заключение сделать невозможно.

Сейбел: Сейчас вы получаете такое же удовольствие от программирования, как и вначале?

Норвиг: Да. Но тревожит то, что я знаю не все. Сегодня я программирую не так много и кое-что подзабыл. И столько нового! Обязательно надо переделать мой сайт и на клиентской стороне использовать JavaScript. Он должен быть на PHP или чем-то подобном, но я все не соберусь освоить их.

Сейбел: Как вы полагаете, программирование – удел молодых?

Норвиг: Молодость в некоторых отношениях – плюс. Конечно, у нас есть люди, прекрасно работающие на любом уровне и в любом возрасте. Дело в том, что надо держать в голове всю программу целиком, всю задачу, надо уметь концентрироваться. А это проще делать в молодости – лучше работают мозги или просто меньше отвлекаешься. Если есть семья, дети и так далее, просто нет возможности посвятить этому столько времени подряд, сколько нужно. Это да. С другой стороны, годы приносят опыт, и он отчасти уравнивает шансы: просто знаешь, что нужно делать.

Сейбел: Одна из особенностей нового стиля программирования, как вы подметили, в том, что программист должен впитывать все новое быстро. Что вы делаете, когда нужно прочесть и понять большой фрагмент совершенно незнакомого кода?

Норвиг: Надо применять и статические, и динамические методы. Начните читать код, постарайтесь понять, что откуда вызывается, где тратится большая часть времени, какие данные передаются. Потом попробуйте что-нибудь сделать – внесите хотя бы самое мелкое изменение. Или зайдите в базу данных возникающих проблем и выберите одну. Для этого надо изучить небольшой участок кода. А изучив его, можно двигаться дальше.

Сейбел: Вы занимались литературным программированием в духе Кнута?

Норвиг: Как таковым – нет. Конечно, я писал макросы и тому подобное, использовал документы Java. Программирование на Лиспе побуждает к созданию собственной системы по мере написания кода, в конце концов это выливается в литературное программирование. Вырабатываешь собственные макросы для программирования в контексте своего приложения – тут и документация, и данные, и код. Так что в принципе я занимался этим. Уже позднее, работая с Java, или Python, или другим языком, я внимательно относился к созданию тестов и документации для них.

На самом деле, в своей книге Кнут пытался рассказать, в каком порядке лучше всего писать книгу, полагая, что кто-то собирается читать книгу целиком и хочет, чтобы это происходило в определенном порядке. Но так уже никто не делает. Человек заглядывает в содержание и выясняет, какой наименьший по объему кусок ему нужно прочесть. Оказывается, ему нужно всего три абзаца. Он находит их и читает. Думаю, это серьезная перемена.

Сейбел: А нельзя ли писать литературные программы в более современном стиле? То, что есть у Кнута, позволяет создать указатель и прекрасную систему перекрестных ссылок. Может быть, если осовременить этот подход, мы получим книгу, построенную по-другому? Она будет читаться и как целая программа, и по кускам.

Норвиг: Не знаю. По-моему, он решал проблему, которой уже почти нет. Отчасти из-за того, что Кнут расположил все линейно, а не в веб-стиле или в порядке, удобном для поиска. Отчасти из-за ограничений – изначально он использовал Паскаль, а этот язык очень строг в отношении порядка определений, и он не всегда такой, как вам нужно. Современные языки более гибкие в этом смысле. Думаю, проблема сегодня не столь актуальна.

Сейбел: Вы говорили о том, как читали в «Scientific American» код программы для шашек. В книге «Как самому научиться программировать за десять лет» вы подчеркиваете важность чтения кода. Какой код вы еще читали, кроме кода Стрейчи?

Норвиг: Много кода Symbolics – он был под рукой, когда я работал в Беркли.

Сейбел: Из-за того, что он был под рукой и был интересным? Или вы пытались разобраться в том, что наблюдали?

Норвиг: И то и другое. Иногда я просто пытался понять, как все устроено, а иногда решал конкретную задачу.

Сейбел: Если вы читаете просто для самообразования, то что именно?

Норвиг: То, что мне интересно. «Смотри-ка, эта файловая система позволяет читать файлы по сети при помощи того же протокола, что я локально использую на своей машине. Как это? Может быть, это в функции open?» Смотришь на то, что эта функция вызывает, смотришь еще куда-нибудь и понимаешь, как это работает.

Сейбел: Вы читали книги с литературными программами Кнута?

Норвиг: Листал. Не скажу, что читал.

Сейбел: А как насчет «Искусства программирования»? Одни мои собеседники прочли ее от корки до корки, другие поставили на полку и используют для справок, третьи просто поставили на полку.

Норвиг: Как-то я сделал из нее подставку для монитора – это ведь громадный многотомник, как раз подошел по высоте. Думал, если он будет прямо передо мной, то я, наконец-то, начну его использовать.

Сейбел: Но ведь надо было всякий раз поднимать монитор?

Норвиг: Нет, тома были в коробке. Если сильно потянуть, можно достать. Сейчас я редко пользуюсь книгами для справки – все больше интернет-поиском.

Сейбел: Потому что так удобнее?

Норвиг: Удобнее. А еще потому, что я теперь больше ориентирован на цель. Кнут хорош, когда нужно узнать о каком-то предмете все. А мне нужно знать, чем А лучше Б, или выяснить приблизительную сложность чего-то, но подробности мне ни к чему.

Сейбел: Кем вы себя считаете – ученым, инженером, художником, ремесленником?

Норвиг: Если взять названия разных книг и тому подобного, больше всего подойдет слово «ремесленник». «Художник» – чуточку претенциозно, потому что искусство должно нести красоту, или устанавливать эмоциональный контакт, или вызывать эмоциональное воздействие. Это не имеет отношения к тому, что я пытаюсь сделать. Конечно, я хочу видеть в программах красоту и трачу на это, пожалуй, слишком много времени. У меня был период в жизни, когда я мог сказать себе: «А не вернуться ли назад, чтобы переписать вон тот кусок?» И когда писал что-то для публикации, тратил на это больше времени, чем ради только личного профессионального роста.

Но это, мне кажется, не искусство. «Ремесло» – вот самый подходящий термин. Вы можете сделать стул, на который приятно смотреть, но это все равно предмет функционального назначения.

Сейбел: Как вы распознаете хорошего программиста, особенно во время собеседования? Ваша компания нанимает много программистов, разумеется, лучших. Как вы их выбираете?

Норвиг: До сих пор не знаем.

Сейбел: Google известен тем, что на собеседованиях претендентам задают логические загадки. Как полагаете, это удачный подход?

Норвиг: Не думаю, что это важно – умеет человек решать такие задачи или нет. Я не люблю задавать головоломные вопросы. Важно дать претенденту техническую задачу, а не поболтать с ним и убедиться, что он – отличный парень. Правда, человек должен еще уметь работать в коллективе. Но прежде всего нужна техническая проверка: умеет ли он делать то, о чем заявляет. Для этого есть много методик. Часто все ясно из резюме. Лучшая рекомендация – если человек работал с кем-то

из наших сотрудников и тот может за него поручиться. Но мы стремимся также полноценно использовать собеседование. По большей части для того, чтобы почувствовать, как этот человек думает, как он работает вместе с кем-то. Владеет ли он базовыми понятиями? Может ли он сказать: «Для решения мне нужно знать А, Б и В», – и начать связывать их вместе? Думаю, это можно показать, даже не решив логическую задачу. Соискатель говорит, к примеру: «Я буду решать так: сначала подумаю об этом, потом сделаю это, потом это, а эту часть я, эээ, не совсем понимаю». Кто-то справляется с этой частью, кто-то нет. Но даже если человек не справился, он может произвести хорошее впечатление тем, насколько компетентно и уверенно подошел к делу. И, конечно, если берешь человека для написания кода, то просишь его написать код на доске. Кто-то подзабыл это дело или не очень хорошо знает – и это видно сразу.

Сейбел: Так это только негативный индикатор? Если человек не может написать вменяемый кусок кода, это плохой признак. Но если может, это вовсе не значит, что он сумеет решать более крупные задачи.

Норвиг: До некоторого уровня это так, дальше уже неясно. Мы всесторонне рассматривали эту проблему, потому что к нам приходит много резюме, и мы рассматриваем их на двух уровнях. Во-первых, чтобы правильно отбирать на собеседования людей из тех, кто прислал резюме. И во-вторых, чтобы правильно нанимать людей из тех, кто приходит на собеседования.

Сейбел: Как вы это определяете? Вы ведь ничего не знаете о тех, кого не позвали или кого не наняли.

Норвиг: Это нелегко решить. На обоих уровнях мы отбираем примерно половину народа. В целом, мне кажется, мы спрашиваем себя: «На что похожи резюме тех, кто успешно прошел собеседование?» – и стараемся найти именно таких. Важен ли многолетний опыт? Важна ли работа над проектом с открытым исходным кодом? Играет ли роль победа в конкурсе программистов?

Сейбел: Вы действительно заносите все это в базу данных?

Норвиг: Заносим. И при вопросе о приеме на работу смотрим, каковы показатели по резюме и собеседованию. Это не Библия, но это помогает принять решение – наряду с другими факторами.

Сейбел: Тем, кто проводит собеседование, эти данные сообщают заранее?

Норвиг: Нет, они передаются только в отдел кадров после сбора всей прочей информации. Мы обнаружили кое-что интересное: при оценке успехов сотрудников через год-два после их прихода лучше всего идут дела у тех, кто на одном из собеседований получил худший показатель.

Мы ставим оценки от 1 до 4. И «единица» – это надежный показатель будущего успеха.

Сейбел: Но ведь надо было в чем-то и отличаться, раз его приняли?

Норвиг: Ну, разумеется, 99%, получивших 1 на одном из собеседований, уходят ни с чем. А что касается оставшихся... Чтобы принять такого человека, кто-то должен стукнуть кулаком по столу и сказать: «Я беру его, потому что уверен в его успехе, а тот, кто счел его плохим программистом, ошибается. Я за него, ставлю на кон свою репутацию».

Сейбел: Значит, в Google собираются только крутые программисты. Сейчас компьютеры и программы заполонили все. Как вы думаете, должен ли каждый хоть немного разбираться в программировании, чтобы уживаться с ними или чтобы понимать мир, в котором мы живем?

Норвиг: Вероятно, вы имеете в виду, что взрослый человек должен разбираться в устройстве программ примерно так же, как в конструкции автомобиля. Но в какой мере этот человек должен *быть* программистом? Обычный человек сейчас может отредактировать документ, составить электронную таблицу. При некотором опыте обращения с таблицами вы уже немного программист.

«Программирование со стороны пользователя», «программирование для каждого» – все это не имело успеха. Не уверен, что это так легко. В чем тут дело? В том, что одним это дано, а другим нет? Или в том, что мы не создали модель, – одну простую модель вместо множества индивидуальных попыток программирования?

Сейбел: Многие из тех, у кого я брал интервью для этой книги или по другому поводу, пришли к компьютерам, потому что это доставляло им удовольствие и потому что компьютеры, как им казалось, изменят мир. И сейчас некоторые из моих собеседников разочарованы тем, как мало в результате изменился мир. А вы как думаете?

Норвиг: Я нахожусь в правильном месте. У нас сотни миллионов пользователей, мы можем принести им пользу и быстро запустить для них новые услуги. Это здорово. Не знаю, где еще я мог бы так воздействовать на жизнь людей.

9

Гай Стил

Гай Стил – настоящий программист-полиглот. На вопрос, какие языки он серьезно использовал, Гай выдал список: Кобол, Фортран, ассемблер IBM 1130, машинный язык PDP-10, APL, Си, C++, BLISS, GNAL, Common Lisp, Scheme, Maclisp, S-1 Lisp, *Lisp, C*, Java, JavaScript, Tcl, Haskell, Фокал, Бейсик, TECO и TeX. «Это основные», – прибавил он.

Он приложил руку к созданию двух основных существующих сейчас диалектов общего назначения языка Лисп: Common Lisp и Scheme. Он участвовал в работе комитетов по стандартам, определивших облик языков Common Lisp, Фортран, Си, ECMAScript и Scheme. Билл Джой нанял его, чтобы он помог в написании официальной спецификации языка Java. Сейчас он разрабатывает Fortress – новый язык для высокопроизводительного научного программирования.

Стил получил степень бакалавра в Гарварде, затем степень магистра и PhD в Массачусетском технологическом институте (MIT). В последнем он вместе с Джеральдом Сассменом выпустил серию основополагающих работ, ныне известных как «The Lambda Papers», где был впервые описан язык Scheme. Он также был летописцем хакерской культуры, одним из создателей Jargon File¹ и редактором книжной версии «The Hacker's Dictionary» –

¹ Jargon File – англоязычный онлайн-словарь сленга хакеров. – *Прим. перев.*

словаря хакера (впоследствии дополненный Эриком Рэймондом, он стал «The New Hacker's Dictionary»). Стил также играл заметную роль в создании Emacs и одним из первых занялся портированием программы TeX Дональда Кнута.

Стил – член Ассоциации вычислительной техники (ACM), Американской академии искусств и наук, Национальной академии инженерных наук. Он лауреат премии имени Грейс Мюррей Хоппер (1988) от ACM и премии Dr. Dobb's Excellence in Programming Award (2005).

В этом интервью он говорит о проектировании ПО и о сходстве программирования с написанием книг и дает одно из лучших слышанных мною объяснений ценности – и ограниченных возможностей – формальных доказательств корректности программ.

Сейбел: Как вы начали программировать?

Стил: В начальной школе я страшно интересовался разными науками, математикой, много читал, например, одна из моих любимых книг – «Magic House of Numbers» (Волшебный дом чисел) Ирвинга Адлера. Еще мне очень нравилась всякая детская фантастика, вроде серии книг про Дэнни Данна. Так что меня вообще влекло в научную, математическую сторону. Я читал все подряд про науку и математику, прочел кое-что и об этих новомодных компьютерах – тогда они только появились.

Сейбел: Когда это было?

Стил: В начальной школе я учился с 1960 по 1966 год. Но, думаю, поворотный момент случился, когда я уже был в Бостонской латинской школе, – примерно в девятом классе. Друг спросил меня: «Слышал о новом компьютере, который установили в подвале?» Я подумал, что это очередная его выдумка: до этого он говорил о бассейне на четвертом этаже, а в школе их было всего три. Но он сказал: «Нет, он правда есть».

Оказалось, Винсент Лирсон организовал для Бостонской латинской школы мини-компьютер IBM 1130 и установил его в подвале. Он сам был выпускником школы и очень ей помогал. Друг показал мне программу из пяти строк на Фортране, и она сразу же покорила меня.

Я спросил учителя математики, что можно почитать. Он дал мне кое-какие книги, думая, что мне хватит на месяц, но я проглотил их за выходные. Я самостоятельно выучил Фортран на День благодарения 1968 года, когда случились длинные выходные. После этого меня прочно зацепило.

Мне и моим школьным товарищам очень нравилось все от IBM из-за того компьютера IBM 1130. Раз в два месяца мы ходили в офис IBM, разговаривали с сотрудниками, покупали их издания, насколько хватало денег.

Кроме того, рядом был книжный магазин с книгами про разные экзотические языки вроде ПЛ/1, там мы тоже кое-что покупали. Вот так в школе мы и познакомились с техникой IBM. В самой школе была только IBM 1130, а мы мечтали о машине System 360. Мы читали о ней, но поработать на ней было негде.

А весной 1969-го я принял участие в программе MIT для старшеклассников. Здорово было – по утрам в субботу студенты объясняли нам все эти крутые штуки. Я слушал лекции по теории групп, программированию и чему-то там еще. И очень сильно прикипел тогда к MIT. Благодаря этой программе я смог поработать и на IBM 1130, и на PDP-10 фирмы DEC. Так мы и познакомились с компьютерами DEC.

Мы, старшеклассники, узнали, что на Централ-сквер есть офис DEC. Хотя он был рассчитан на студентов MIT, нам не глядя выдавали разные руководства. Просто здорово! В предпоследнем или последнем классе мы с приятелем предложили им реализовать APL на PDP-8, и они отнеслись к этому серьезно. А примерно через неделю пришел ответ: мол, мы считаем это нецелесообразным, но спасибо за предложение.

Сейбел: Вы помните свою первую интересную программу?

Стил: Так как моим первым языком был Фортран, интересное началось для меня с языка ассемблера для IBM 1130. Первым интересным произведением была программа для генерации контекстных указателей по ключевым словам. IBM делала «быстрые» алфавитные указатели к своим руководствам: ключевое слово можно было найти в алфавитном указателе, а по обе стороны от него располагались и другие слова из того отрывка, где оно встречалось.

Сейбел: Отрывка текста, где было слово?

Стил: Отрывка из собственно руководства. Итак, в средней колонке шли слова по алфавиту, а по бокам от них – слова из этих отрывков. Я подумал, что при помощи IBM 1130 справлюсь с этим. А так как в ее оперативной памяти помещалось только 4000 слов, было ясно, что придется работать с данными на диске. Так что мне пришлось изучить методы сортировки данных вне оперативной памяти. Интересной в этой программе была не столько возможность генерации ключевых слов в контексте, сколько именно эта возможность сортировки слиянием вне оперативной памяти. Программа оказалась более-менее эффективной. К сожалению, в оперативной памяти я использовал пузырьковую сор-

тировку данных. А надо было использовать также сортировку слиянием, но тогда я до этого не додумался.

Сейбел: А сколько времени прошло с того момента, как вы узнали о компьютере в подвале, до написания этой программы? Месяцы? Недели?

Стил: Это случилось в течение первых двух лет. Не уверен, был ли это первый год. Я выучил Фортран осенью 1968-го. И помню, что APL был моим третьим языком, значит, язык ассемблера я освоил на Рождество или чуть позже. Помню, что APL я выучил весной 1969-го, потому что тогда в Бостоне проходила Весенняя объединенная вычислительная конференция.

У IBM там была выставка, посвященная всем их продуктам, прежде всего APL-360, и я бродил вокруг их стенда. По окончании выставки они собирались выбросить демонстрационные распечатки от терминала Selectric, и тут я подошел и спросил: «Вы хотите это выкинуть?» Женщина, которая этим заведовала, удивленно поглядела на меня и протянула мне эту бумагу – так, словно это был рождественский подарок. Это и был рождественский подарок.

Сейбел: Что за бумага?

Стил: Фальцованная бумага от терминала Selectric, на котором они последние пару дней демонстрировали возможности APL. Небольшие примеры программ и все то, что они там набирали. Вот при помощи этих примеров да еще буклета IBM с этой выставки я и выучил APL.

Сейбел: Значит, в MIT вам было хорошо, но вы все же пошли учиться в Гарвард, и при этом работали в MIT. Почему?

Стил: Когда пришло время подавать документы в колледж, я подал бумаги в MIT, Гарвард и Принстон, но учиться хотел в MIT. Меня приняли во все три. Директором Бостонской латинской школы был Уилфрид О'Лири, классик старейшей школы, истинный джентльмен. И он сказал моим родителям: «Вы знаете, что вашего сына приняли в Гарвард, а он собирается пойти в MIT?» То есть он надавил на них, они надавили на меня, и в конце концов я решил пойти в Гарвард.

Родители были за то, чтобы я нашел работу на лето, а не болтался дома, – ну, знаете, этот классический синдром. Мне хотелось программировать и не хотелось работать за копейки. Я пытался наняться оператором клавишного перфоратора, думая, что более-менее квалифицирован для этой работы. Но меня нигде не брали – отчасти потому, что мне еще не было 18-ти. Но это я понял позже. Мне просто отвечали: «Не звоните нам, мы сами вам позвоним» – и все.

Где-то в начале июля я узнал, что Билл Мартин из MIT ищет Лисп-программистов. Я подумал: «Ага, я же знаю Лисп». Я так часто бывал

в МІТ, что раздобыл документацию Лиспа в лаборатории искусственного интеллекта МІТ. Я пробирався в лаборатории и возился с компьютерами. Везде было открыто – это было еще до протестов в связи с вьетнамской войной, из-за которых им пришлось врезать дверные замки. Так что в выпускном классе я писал свою собственную реализацию Лиспа для IBM 1130.

И вот я пришел в офис к Биллу Мартину, такой тощий парень из ниоткуда, сунул голову в дверь и спросил: «Я слышал, вы ищете Лисппрограммистов?» И он не рассмеялся мне в лицо, а посмотрел на меня и сказал: «Вам придется пройти мой тест по Лиспу». «ОК, прямо сейчас?» Я уселся и два часа разбирался со всеми вопросами и задачами. Закончив, я протянул ему бумаги, он минут десять смотрел то, что я написал, и наконец сказал: «Я вас беру».

Сейбел: Лисп вы тоже освоили на курсах МІТ для школьников?

Стил: Отчасти, но там больше преподавали Фортран и другое.

Сейбел: У вас в самом начале были сильные учителя?

Стил: В Латинской школе у меня были хорошие учителя по математике, они меня поощряли как раз в нужном направлении. Когда я был в девятом классе, Ральф Уэллингс – тот, кто дал мне книги накануне Дня благодарения, – предложил сделку. Он сказал: «Я заметил, что ты сдаешь все математические тесты со стопроцентным результатом. И я разрешу тебе четыре урока в неделю проводить в компьютерном классе, если на пятом ты будешь получать стопроцентный результат. Если когда-нибудь получишь меньше, договор отменяется». Это был отличный стимул. И до конца года я продолжал блестяще сдавать тесты. Чтобы иметь доступ к компьютеру, мне пришлось очень серьезно заниматься математикой. А на следующий год учитель уже ничего не предложил – и правильно, потому что с математикой у меня стало хуже. Так что у меня были хорошие учителя, они дали мне то, что нужно, чтобы все выучить.

Сейбел: А позже, когда вы уже увлеклись компьютерами, кто именно помогал вам в этом?

Стил: Билл Мартин, конечно, который нанял меня. И Джоэль Мозес, который руководил проектом Maccsuma, – для работы над ним меня и взяли в МІТ.

Сейбел: И в итоге вы проработали над этим проектом в течение всей учебы в колледже?

Стил: Да, я работал в МІТ все то время, пока учился в Гарварде. Летом это была работа на полную ставку, остальное время – на полставки.

Я очень старался подгадать так, чтобы с утра быть в Гарварде, потом заехать в MIT, поработать там часа два-три, ну, а после этого домой.

Сейбел: И вы работали только над проектом Macsyma на Лиспе?

Стил: Да. В частности, я отвечал за интерпретатор Macclisp. Йонл Уайт, ответственный за интерпретатор и компилятор, к тому времени стал уже чем-то вроде гуру. Я взял интерпретатор, и это разделение труда было почти идеальным. Моим наставником был Йонл Уайт. Но фактически все, кто работал над проектом, взяли меня под крыло. Еще я познакомился кое с кем из лаборатории искусственного интеллекта. Поэтому я без проблем поступил в магистратуру MIT – там уже знали, кто я и чем занимаюсь.

Сейбел: Вы стали бакалавром компьютерных наук?

Стил: Да. Вообще-то я хотел получить диплом по математике, слушал соответствующие лекции, но потом понял, что у меня нет способностей к изучению бесконечномерных банаховых пространств. Я просто изнывал над ними. Но, к счастью, я слушал достаточно лекций по компьютерам, просто из интереса, чтобы сменить специальность. На самом деле диплом был по прикладной математике – компьютерные науки были ее разделом, а прикладная математика относилась к инженерному факультету Гарварда.

Сейбел: С какими машинами вы имели дело в Гарварде?

Стил: С DEC PDP-10. В кампусе была PDP-10, но, кажется, к ней допускали только магистров. А нам давали доступ к телетайпным терминалам коммерческой системы, которую Гарвард арендовал или что-то в этом роде.

Сейбел: Вы бы хотели что-то изменить в том, каким путем пришли к программированию? Может быть, вы что-то упустили?

Стил: У меня никогда не было четко поставленной цели. И я не жалею о своем пути. Думаю, в моей жизни было много интересных совпадений и полезных напутствий.

Сейчас я понимаю, что посещать одновременно MIT и Гарвард было довольно необычно. Я курсировал туда-сюда и говорил, например: «Профессор с того берега утверждает то-то и то-то». А мне отвечали: «Чепуха, на самом деле вот как все обстоит». Так что я за короткий срок получил обширные знания.

Необычным было и то, что связь с MIT у меня появилась уже в старших классах. В 15 лет я мог возиться с компьютером за миллион долларов, а тогда миллион долларов был немалой суммой. Поэтому у меня нет сожалений насчет того, что жизнь могла бы сложиться иначе. И потом, я принимаю все таким, как есть.

Сейбел: В чем больше всего изменились ваши взгляды на программирование по сравнению с теми годами? Кроме того, что теперь вы понимаете, что пузырьковая сортировка – не лучший способ?

Стил: Пожалуй, вот что: сейчас невозможно знать абсолютно все, что происходит внутри компьютера. Есть вещи, вам неподвластные, – сегодня невозможно знать все о всем программном обеспечении. В 1970-е память компьютера вмещала только 4000 слов. Можно было сделать дампы ядра и проверить каждое слово. По распечаткам исходного кода операционной системы можно было понять, как она работает. Я изучал утилиты для работы с диском и с устройством для чтения перфокарт, создавал свои варианты. И мне казалось, что я понимаю, как работает вся IBM 1130. Ну, или понимал в этом достаточно для себя. Сейчас все не так.

Сейбел: Вам помогали книги?

Стил: В 1970-е – конечно, да. Например, «Искусство программирования» Кнута.

Сейбел: Вы прочли ее от корки до корки?

Стил: Почти. Я делал столько упражнений, сколько мог. Некоторые требовали знаний, которых у меня было, – скажем, высшей математики. Такие я пропускал или делал кое-как. Но первые два тома и солидный кусок третьего я прочел очень внимательно. Сортировке я учился по книге алгоритмов Ахо, Хопкрофта и Ульмана. Что до остальных, то надо поглядеть в моей библиотеке. Я ведь настоящий старьевщик и храню все свои книги. Но те, которые я назвал, сразу всплывают в памяти. И еще книги по Лиспу. Например, та, что издали Беркли и Боброу: это скорее собрание отдельных статей, но я многое вынес из него. Потом я начал читать журнал «SIGPLAN» и ежемесячный журнал «Communications of the ACM» («CACM»). Тогда в «CACM» было полно технических подробностей, их интересно было читать.

Вспоминаются два эпизода. В Латинской школе я участвовал в научных конкурсах, делал кое-какие компьютерные проекты. И вот член жюри одного из конкурсов спросил меня: «А почему бы тебе не вступить в ACM?» Не помню, как звали этого человека, но я последовал его совету, за что ему очень благодарен.

А в Гарварде, если у меня было утром окно, я шел в библиотеку и делал одно из двух: читал «Scientific American», узнавая что-нибудь из истории науки, или «CACM», заглядывая в будущее. Я не пропускал ни одной колонки Мартина Гарднера про математические игры. А в «CACM» я читал то, что меня интересовало. В 1972 году журналу исполнилось только 15 лет, и я прочитал все выпуски довольно быстро.

Сейбел: Должно быть, тогда было проще в том смысле, что если понимаешь, как работает система в целом, то можно охватить умом и целую отрасль знания.

Стил: Именно так: целую отрасль. Было полно одностраничных статей. Например: «Вот новый эффективный метод хеширования». Я много читал такого.

Сейбел: Мне кажется, сейчас понимать старые статьи сложновато, они завязаны на какие-нибудь особенности старых языков или железа.

Стил: Да. Необходимость – мать изобретения: идея появляется, когда в ней есть потребность. И только позже становится понятно, что идея эта очень важна. Чтобы убрать случайные обстоятельства, взглянуть на суть идеи, нужно несколько лет. Допустим, есть эффективный способ изменить порядок битов в слове на обратный, но реализован он на языке ассемблера 7090. В основе – интересная математическая идея, но она еще недостаточно абстрагирована.

Сейбел: Это ведь то, что делает Кнут?

Стил: Верно. Кнут и такие, как он.

Сейбел: Возможно, те, кто изучает компьютерные науки в колледже, проходят все это. Но многие программисты без официального образования учатся прямо в процессе работы. Что вы можете посоветовать в таком случае? От чего вы отталкиваетесь, как вам удастся читать и понимать эти статьи? Может быть, надо прочесть все до единого выпуска «САСМ», начиная с первых?

Стил: Прежде всего, я читал этот журнал, совершенно не собираясь прочесть все возможное и стать гением компьютерной науки. Просто это было мне интересно, я чувствовал потребность в этом чтении. Думаю, здесь две проблемы. Во-первых, нужно иметь внутреннюю мотивацию: вам либо интересно, либо хочется улучшить свои навыки.

Во-вторых, как найти хороший материал, тем более что само это понятие меняется со временем? То, что хорошо сегодня, через десять лет устареет. Надо спросить у того, кто в этом разбирается. Что было полезным мне? Кнут, Ахо, Хопкрофт и Ульман. Затем «The Psychology of Computer Programming» Джеральда Вайнберга – она все еще не утратила ценности. Кое-что мне дал «The Mythical Man-Month»¹ Фреда Брукса.

Я тогда заходил в отдел компьютерной литературы книжного магазина MIT – не реже раза в месяц я рылся там в книгах. Сейчас отделы компьютерной литературы в десять раз больше, но что вы там найдете?

¹ Ф. Брукс «Мифический человек-месяц или Как создаются программные системы». – СПб.: Символ-Плюс, 2000.

В основном учебники по Си и Java. Но если хорошо порыться, найдется немного книг по теории программирования, алгоритмам и так далее.

Сейбел: Есть другое чтение, и я знаю, что вы считаете его важным, – это чтение кода. Как вы ориентируетесь в чужом коде большого объема?

Стил: Если я знаю, для чего предназначен этот код, но не знаю его внутреннее устройство, я беру конкретную команду и начинаю распутывать.

Сейбел: Путь исполнения?

Стил: Да. Возьмем Emacs. Я говорю: посмотрим на код, который служит для перемещения курсора на один символ вперед. Я не понимаю его полностью, но с его помощью пойму какие-то структуры данных и способ представления буфера. Если повезет, то я найду место, где добавляется единица. Изучив это, я возьму код, перемещающий курсор на один символ назад. «Стереть строку». Так я отслеживаю все более сложные участки кода, пока не пойму, что нащупал важную его часть.

Сейбел: А что значит «отслеживаете»? Смотрите на код и пытаетесь выполнить его в уме? Или берете отладчик и проходите код шаг за шагом?

Стил: И то и другое. Отладчик применяю в основном для небольших объемов кода 1970–1980-х. Сейчас между запуском программы и моментом, когда она начинает делать что-то интересное, может быть долгая инициализация. Поэтому разумнее найти главный цикл программы или главную управляющую функцию и отслеживать уже оттуда.

Сейбел: И после этого вы зададите точку останова и начнете пошаговое исполнение от нее или просто исполните программу в уме?

Стил: Скорее второе – буду читать код и размышлять над тем, что он делает. Если очень понадобится, я могу засесть за код и начать подробно читать его целиком. Но прежде надо понять, как все вообще работает. Иногда везет, и программисты оставляют документацию или называют все понятными именами, или располагают все содержимое файла в правильном порядке. Если так, код прочесть легко.

Сейбел: А что такое правильный порядок содержимого файла?

Стил: Отличный вопрос. Удивительно, что одна из проблем такого языка, как Паскаль, связана с тем, что он задумывался для однопроходного компилятора: подпрограммы в файле располагаются «снизу вверх», так как перед использованием подпрограммы она должна быть определена. В итоге читать программы на Паскале необходимо задом наперед – только так получится взглянуть на них «сверху вниз». Сейчас все свободнее, и будет ли программа выстроена в удобном для вас порядке, зависит исключительно от вкуса программиста, который ее писал. Но вообще, сейчас есть удобные интегрированные среды разработки с перекрестными ссылками, и линейный порядок программ уже не столь важен.

Но я не очень люблю эти интегрированные среды – с ними трудно понять, все ли посмотрел. Перемещаясь по графу, трудно сказать, в какой момент обойдешь его весь. А при линейном порядке гарантированно проходишь всю программу.

Сейбел: Итак, при написании кода сегодня вы отдаете предпочтение выстраиванию его «сверху вниз»: сначала высокоуровневые функции, затем низкоуровневые, от которых те зависят?

Стил: Я стараюсь показывать высокоуровневые идеи. Наилучший для этого способ – показать центральную управляющую функцию со всем, чем она управляет, внизу. Или, возможно, важно показать сначала структуры данных, хотя бы наиболее важные из них. Идея в том, чтобы выстроить что-то вроде истории, а не сваливать в кучу фрагменты кода.

В MIT прекрасно было то, что человек мог знакомиться с кодом, написанным довольно умными хакерами, – код не держали под замком. Так я изучил код операционной системы ITS, реализации TESCO и Лиспа. И программу структурной печати Билла Госпера для Лиспа. Я освоил их еще в старших классах и попытался воспроизвести кое-что на своей IBM 1130.

Но я бы не смог реализовать Лисп на IBM 1130, не имея доступа к реализациям этого языка для других компьютеров. Я просто не знал бы, что делать. И это было важной частью моего образования. Отчасти проблема сегодня в том, что программы приобрели ценность, что большинство серьезных программ – коммерческие, и у нас не так много хороших примеров кода для изучения. Открытый исходный код отчасти решает эту проблему. Можно, к примеру, посмотреть код Linux. Для меня очень полезным оказалось чтение исходного кода TeX, потому что это был хорошо продуманный, хорошо отлаженный код.

Сейбел: Я обычно читаю код, если мне надо узнать, как работает программа. А что подвигло вас читать исходный код TeX?

Стил: Иногда у меня есть четко определенная цель, поскольку мне требуется решить проблему. Дважды я не мог найти ошибку в макросе TeX, читая книгу «Все про TeX»¹, и тогда пришлось читать «TeX: The Program»², чтобы точно понять, как что работает. И оба раза я справлялся с трудностями за четверть часа, так как исходный код TeX очень хорошо документирован, снабжен перекрестными ссылками. Это само

¹ Дональд Кнут «Все про TeX». – М.: «Вильямс», 2003.

² Второй том пятитомного труда Д. Кнута «Computers & Typesetting». Содержит исходный код TeX, написанный методом литературного программирования. – *Прим. ред.*

по себе откровение – то, что программа может быть так тщательно построена, документирована и индексируется, что все находится быстро.

Еще я узнал тогда, как нужно выстраивать структуры данных, как сделать код легче для чтения. «TeX: The Program» Кнута читается почти как роман, можно просто взять и читать подряд. Конечно, иногда возникает желание пролистать несколько страниц вперед или назад. Кнуту пришлось проделать огромную работу, поэтому мало кто поступает таким образом.

Сейбел: Добравшись до конца, что вы для себя выносите?

Стил: Я понимаю, как устроен код, и у меня могут возникнуть идеи, как рациональнее построить свой собственный. Вряд ли я смогу подражать Кнуту, как не смогу писать в стиле Фолкнера или Хемингуэя. Но чтение этих авторов воспитывает чувство стиля. Может быть, по той или иной причине я приму сознательное решение *не* писать, как Хемингуэй. Это ценный опыт. Ну и потом, читать хорошо написанный роман или код – само по себе удовольствие.

Сейбел: Вы занимались литературным программированием?

Стил: Последовательно, в духе Кнута – нет. Он повлиял на меня, заставив задумываться о таких вещах, и теперь, прежде чем написать подпрограмму, я обычно пишу абзац текста. Но последовательно литературным программированием я не занимался. Иногда мне интересно – пишет ли он литературно, занимаясь исследовательским программированием, до того как готовит свои программы для публикации? Я не знаю, как выглядит весь этот процесс.

Сейбел: Значит, вы пробовали, но не сочли этот способ делающим процесс программирования более эффективным или приятным?

Стил: Отчасти мне не хотелось самому делать утилиты для литературного программирования. У Кнута все утилиты были организованы сначала вокруг Паскаля, а потом Си. Паскаль еще ладно, но недостатки Си я видел ясно, и литературное программирование, по-моему, не позволяло их преодолеть. Вот если бы Кнут сделал утилиты литературного программирования для Common Lisp, возможно, я смог бы на них быстро переключиться.

Сейбел: Оставим литературное программирование и вернемся к чтению кода. Как по-вашему, хорошо написанную программу можно прочитать от начала и до конца – или она скорее напоминает гипертекст, и в нем надо разбираться?

Стил: Я вовсе не против гипертекста. Но если программа хорошо написана, ее структура такова, что по ней можно перемещаться в некоем логическом порядке. Это вопрос не о том, что делает программа, а о том,

как она построена, в каком контексте призвана работать. Хочется видеть комментарии в начале каждой подпрограммы, или отдельный сводный документ, или особое именование переменных, облегчающие понимание программы. И, пожалуй, хороший программист – действительно хороший – постарается сделать это независимо от того, *что* делает программа.

Сейбел: Какой код вы читали в последний раз для собственного удовольствия?

Стил: Трудно найти код, заслуживающий чтения. У нас нет списка исходных кодов, обязательных для чтения. «Это прекрасный код. Читать всем». Поэтому чаще всего попадают небольшие кусочки кода на одну страницу, скажем, в научных статьях, а не фрагменты реально работающих программ. Последним, скорее всего, был код, разработанный моей командой для реализации языка Fortress. Ну, и кое-что из Java-библиотек.

Наверное, последний крупный фрагмент кода, который я изучал для собственного удовольствия, – это код Джорджа Харта, математика, специалиста по многогранникам. Это был очень занятный фрагмент для генерации и отображения сложных многогранников в браузере с использованием VRML. У Харта получился огромный кусок кода на JavaScript, создающий VRML-код и передающий его в программу для отображения VRML.

Я попытался улучшить этот фрагмент, для чего пришлось тщательно изучить код Харта, потом я изменял его и смотрел, что получается: пытался сделать многогранники попричудливее. Кроме того, я умудрился сделать несколько грубых ошибок: там был релаксационный алгоритм, размывающий вершины многогранников, чтобы сделать их красивее и проще для отображения, и кое-где я случайно добавил математические нестабильности, которые приводили к забавным последствиям. Это было страшно занятно, – и все только ради самообразования. Это было лет шесть-семь назад.

Сейбел: Как связаны чтение и модификация кода? Вы можете сидеть с распечаткой или с кодом на экране компьютера – и, не исполняя код, понять, к чему приведет его изменение?

Стил: Обычно я печатаю код. И сижу с распечаткой за столом, часто делаю пометки, задаюсь вопросами и так далее. А потом иду к компьютеру, что-нибудь добавляю в код, и смотрю, как он себя ведет. Просматриваю его.

Сейбел: Ну, это в случае когда вам надо изменить код. Но приносит ли какую-нибудь пользу или удовольствие просто чтение кода? Распечатать, почитать, сделать какие-то пометки – и отложить в сторону?

Стил: Да. Если бы я этим и ограничился, просто чтение кода все равно было бы полезным упражнением. Я многое узнал о VRML, а в JavaScript, на мой вкус, маловато абстракций. Динамическая типизация в объектно-ориентированном языке, по-моему, все же лишена нужной строгости.

Сейбел: Поговорим о проектировании ПО. Сейчас вы не пишете столько кода, сколько раньше, но как вы подходили к разработке программы с нуля? Садись за компьютер и начинали писать код? Или брали разлинованный блокнот? Или еще как-то?

Стил: Здесь надо быть осторожнее, ведь память обычно нас подводит. Очень легко сказать, что я делал так-то, лишь потому, что я сделал бы так сейчас. Постараюсь припомнить.

Иногда я рисовал блок-схемы – у меня был шаблон для блок-схем IBM и специальная бумага. Я учился программировать до эпохи структурного программирования, поэтому среди моих программ были и структурированные, и нет. Потом я понял пользу структурного программирования, и в 1970-е мои программы на языке ассемблера стали более структурированными: я делал циклы, if-then-else, больше заботился о структуре своего кода.

Я составлял списки того, что хотел задать на входе и получить на выходе, иногда приводил краткие примеры. Недавно я нашел одну из своих ранних программ на APL. Мне тогда было лет 15–16. То был кусок кода на APL – я набросал его на бумаге, прежде чем попробовать запустить. Рядом лежал другой клочок бумаги – примеры того, что у меня должно было быть на входе и на выходе. Там были ошибки, примеры были несовместимы с кодом, но, по крайней мере, я пытался дать примеры использования этой программы. Примеры того, что как я думал, увижу на терминале.

Когда я стал работать в проекте MacLisp, структура была уже определена. Почти все, что я делал, было добавлением новых функций к их и без того солидному набору. Там уже были готовые примеры документирования функций, надо было просто добавлять свои в том же стиле.

Сейбел: Вы говорили, что Йонл передал вам работу над интерпретатором, – до того он занимался и интерпретатором, и компилятором.

Стил: Мы вместе занимались проектированием, я был младшим программистом. Он мог сказать: «Нам нужна такая-то функция, работающая так-то, – давай, напиши код». Но чаще мы получали запросы от разработчиков Маскута – мол, нам надо то-то и то-то, – и мы с Йонлом вместе выдумывали интерфейс, а потом я писал для него код.

Сейбел: Значит, в интерпретаторе и компиляторе надо было отразить новые языковые свойства, которые приобрел MacLisp?

Стил: Да, языковые свойства. Многие из них системно-ориентированного характера – управление ресурсами, выделение страниц. Я ввел новый тип данных, который назвал «hunk», и это оказалось настоящей катастрофой. Он представлял собой cons-ячейку более чем с двумя указателями. Это был акт отчаяния, потому что мы вышли за пределы адресного пространства PDP-10, которое, напомним, было 18-битным. Половина указателей в списке уходила на то, чтобы поддерживать структуру списка, в то время как в цепочке данных типа *hunk* на это уходила всего лишь восьмая часть указателей. В результате память использовалась лучше.

Сейбел: Получается, вас постоянно просили что-то добавить. Как же вам удавалось сохранять внутреннюю цельность программы? Если вы постоянно добавляете новые свойства самым очевидным способом, в конце концов выходит громоздкая программа, готовая развалиться.

Стил: Мы пару раз серьезно все переписывали. А однажды полностью пересмотрели структуру программы и реализацию всех операций ввода/вывода в языке – кажется, в 1975 или 1976 году. Старая система позволяла иметь один поток данных на входе и один на выходе и могла взаимодействовать с терминалом. Мы поняли, что получим гораздо более гибкую систему, если Лисп-объекты станут каналами ввода/вывода. И таких одновременно открытых каналов могло быть до 15.

Другим поводом было то, что Macclisp начали портировать на другие операционные системы. В разных местах использовался свой вариант операционной системы PDP-10, и, посмотрев на список клиентов, мы поняли, что придется поддерживать полдюжины различных операционных систем: TENEX, TWENEX, ITS, TOPS-10, WAITS и вариант CMU.

И вот тем летом мы с Джоном создали новый набор API. Тогда этот термин еще не употреблялся – были описания функций, пригодных для создания файловых объектов, их открытия и закрытия, систематизации процессов удаления и переименования и для получения перечня файлов в каталоге.

В какой-то момент я получил свежую распечатку всего Macclisp и на неделю засел в летнем доме родителей с шестью руководствами по ОС. Каждый день я шесть часов переделывал код.

Надо было разработать каждое свойство, например функцию «переименовать», потому что процесс взаимодействия с ОС при переименовании файла сильно различался во всех шести вариантах. В целом образовалось три основных группы вариантов – TOPS-20, TOPS-10 и ITS.

И я неделю занимался этим, проектировал, разрабатывал реализацию, и все это на бумаге, без компьютера. Потом я приехал в MIT и месяц набивал все это, занимался отладкой и тестированием.

Сейбел: Почему именно так?

Стил: Потому что каждой функции предшествовала громадная исследовательская работа. Как я уже говорил, надо было прочесть спецификации на шесть ОС. Один час в день я занимался этим, потом писал 30–90 строк кода. Не было особого смысла сидеть за компьютером – я ведь не мог ничего погуглить или найти онлайн-ую документацию. И лучше было заняться бумажной работой.

Сейбел: А сейчас бывают случаи, когда вам хочется выключить компьютер, положить на стол лист бумаги и взять карандаш?

Стил: Так я и поступаю время от времени. Иногда просто приходится его выключать – вентилятор словно нашептывает: «Проверь почту, проверь почту». Я выключаю его или перевожу в режим сна, сажусь за стол на другом конце комнаты, раскладываю бумаги и начинаю думать, или пишу что-нибудь на доске, и так далее.

Сейбел: Как-то раз вы переиначили фразу Фреда Брукса насчет блок-схем и таблиц, сказав примерно так: «Покажите мне ваш интерфейс – и мне не нужен будет ваш код: это будет уже лишним и не будет относиться к делу». Имея дело с языками вроде Java, вы начинаете проектирование с интерфейса?

Стил: Да, я сейчас больше внимания уделяю интерфейсам, чем в молодости. Описание того, что будет на входе, описание производимых операций, результатов работы методов безо всякого кода – это мне нравится. Писать соответствующий код – тоже, но сейчас я занимаюсь этим нечасто. И, конечно, нужен опыт, чтобы не создавать невыполнимые спецификации. Проектируя интерфейс, нужно представлять себе, как будет выглядеть реализация, – хотя бы приблизительно. Если затем кто-нибудь все сделает лучше, чем планировалось, – отлично.

Сейбел: Если не рассматривать возможность реализации, как вы оцениваете интерфейс?

Стил: По таким качествам, как универсальность, ортогональность и соответствие существующим правилам. Например, не стоит ставить делитель перед делимым без серьезных оснований, потому что в математике принято писать наоборот. Надо иметь в виду общепринятые нормы.

Я спроектировал много чего и теперь могу сказать, насколько хорошо это вышло. Сейчас я иногда проектирую вещи, родственные тем, что делал раньше. Например, создавая спецификации для числовых функций в Java, я вспоминаю, что уже делал это для Common Lisp. А еще я документировал числовые функции для Си. Я знаю, в чем подводные камни, если говорить о реализациях и спецификациях для таких вещей. Я потратил много времени, исследуя пограничные случаи. Об этом я вычитал у Тренчарда Мора, который разработал теорию масси-

вов данных для APL. Он считал, что если прояснить все с пограничными случаями, промежуточные прояснятся сами собой. Правда, Мор не говорил об этом прямо, этот вывод сделал я сам.

И наоборот, можно создать такую спецификацию того, что находится посередине, что она будет верна и для пограничных случаев, и не нужно будет рассматривать пограничные случаи особым образом.

Сейбел: Работая в MIT, вы были причастны к рождению Emacs. Но ранняя история Emacs несколько туманна. Можете ли вы что-нибудь прояснить?

Стил: Я занимался стандартизацией. Был избран такой способ отображения, что TECO становился чем-то вроде WYSIWYG-редактора. На наших экранах размером 24×80 21 строка предназначалась для отображения тех строк, что были в буфере, а нижние 3 строки относились к командной строке TECO. Там надо было вводить команды TECO, и они исполнялись лишь при двойном нажатии клавиши Alt. Был и режим редактирования в реальном времени – там этого нажатия не требовалось. TECO реагировал немедленно на каждый введенный символ как на команду. Вводишь один символ – исполняется команда, вводишь другой – исполняется другая. А большинство печатных символов вставлялось автоматически. Для перемещения вперед, назад, вверх и вниз служили управляющие символы. Это выглядело как очень примитивная версия Emacs.

Затем произошел прорыв. Идея была вот в чем: сейчас мы берем символ, ищем его в таблице, потом выполняем команду TECO. Почему не применить это к редактированию в реальном времени? Каждый символ, который может быть введен, ищется в таблице. Таблица по умолчанию говорит, что печатные символы вставляются в текст, а управляющие символы делают то-то и то-то. Давайте сделаем это программируемым и посмотрим, что получится. Что же получилось? Несколько ярких личностей, связанных с MIT, одновременно придумали, что с этим можно сделать. И через несколько месяцев мы получили пять взаимно несовместимых GUI-интерфейсов для TECO.

Сейбел: То есть они в основном настраивали клавиши?

Стил: Именно. И каждый имел свои идеи насчет удобных сочетаний, поскольку что-то ты делаешь чаще, а что-то реже, и это можно оставить длинным. Один из этих парней был очень озабочен вводом Лисп-кода и начал экспериментировать с выражениями в скобках. Другой больше интересовался текстом – возможностью перемещаться по нему, переключать регистр и так далее. Вот откуда взялись эти команды в Emacs.

Итак, несколько человек, и у каждого свои идеи насчет удобных сочетаний клавиш. Я оказывал системную поддержку по Лиспу, и меня часто

подзывали к себе, просили помочь. Вскоре я понял, что не могу помочь им, не могу менять их программы, потому что сочетания клавиш мне непонятны.

Сейбел: Одним из этих парней был Ричард Стеллмен?

Стил: Нет, Стеллмен занимался реализацией и поддержкой ТЕСО. Ричард снабдил его встроенной системой редактирования в реальном времени, хотя, кажется, над ранней версией работал Карл Миккельсен. Он реализовал функцию назначения сочетаний клавиш, которая сделала все это возможным.

Так или иначе, вышло, что есть четыре набора макросов, несовместимых друг с другом, и я решил заняться стандартизацией – или всеобщим примирением. Я заметил, что куда-то ушел дух товарищества, взаимопомощи, потому что уже нельзя было просто подсесть к чужому терминалу. И я сказал: «Ну вот, мы попробовали то и это, возникло много идей. Давайте придумаем общие и удобные для всех сочетания клавиш».

Я бегал с блокнотом по всему зданию, говорил со всеми этим парнями по несколько раз, пытаюсь найти общий знаменатель. Я пытался как-то организовать те сочетания клавиш, которые они выбрали, пытался сделать так, чтобы они легко запоминались и складывались хоть в какую-то систему. Я не заботился об удобстве печатания вслепую, меня такие вещи вообще мало интересуют: я заботился о легкости запоминания. Вот почему Meta-C, Meta-L, Meta-U означают «с заглавной буквы», «строчные буквы», «заглавные буквы»¹.

Сейбел: Это в своем роде ирония, учитывая то, каким образом команды направляются от мозга к пальцам. Вы, наверное, тоже сталкивались с этим: кто-нибудь спрашивает о сочетании клавиш, которое вы используете сто раз в день, – и вы не способны ответить.

Стил: С этим сталкивалась моя жена. Это как-то прошло мимо меня – я не слишком ловко управляюсь с клавиатурой. Жена 20 лет не пользовалась Emacs, пока я не поставил эту программу на ее Макинтош. Она что-то ввела и спросила: «А как сохранить? Я забыла, как сохраняют файлы». Ее пальцы делали это автоматически, и она не помнила само сочетание. Тогда она проделала это, следя за своими пальцами, и сказала: «Ах да, Ctrl-X, Ctrl-S». То есть она печатала неосознанно.

Сейбел: Вы установили стандартные сочетания клавиш. Как это прошло? Люди были довольны?

¹ По первым буквам слов capitalize (сделать заглавной), lowercase (нижний регистр – строчные), uppercase (верхний регистр – заглавные) – *Прим. науч. ред.*

Стил: Им пришлось, конечно, приложить усилия. Потом я занялся реализацией. Одновременно появилась и другая идея: можно заставить макросы ТЕСО работать быстрее, если сжать пробелы и убрать все комментарии. Интерпретатор ТЕСО обрабатывал символы последовательно, поэтому приходилось ждать, пока он пройдет очередной комментарий. Вот мы и решили создать примитивный компилятор ТЕСО, который бы сжимал пробелы, убирал комментарии и делал еще кое-что для ускорения работы.

И я поначалу решил разработать уплотнитель макросов на основе идеи Муна. То есть идея была не моя. Я стал думать, как организовать первые несколько функций, взял в качестве примера уже имевшиеся пакеты макросов, сделал своего рода синтез. Тут появился Стеллмен и сказал: «Чем ты занят? Выглядит интересно». Он подключился к этому делу, и все пошло в десять раз быстрее – Ричард знал ТЕСО досконально.

Выходит, я всерьез работал над реализацией Emacs 4–6 недель, не больше. Потом стало ясно, что Стеллмен все понял в этой программе, и я могу возвращаться к моим магистерским делам. Стеллмен сделал 99,999% всей работы. Но начал ее я.

Сейбел: Сменим тему. Компьютерные науки сегодня тесно связаны с математикой. Насколько важно для программиста-практика вникать, скажем, в математику из Кнута? Или лучше так: «Мне тут нужно отсортировать, пролистаю-ка я Кнута до того места, где он говорит „это наилучший алгоритм“, и реализую его»?

Стил: Не знаю... У меня нет математического образования, и мне понятно у Кнута не все, особенно из области высшей или непрерывной математики. Здесь я плаваю. Комбинаторика, перестановки, теория групп – другое дело, все это я часто использую. Это мое рабочее оружие. Не каждому программисту нужна математика, но она упорядочивает понятия, с которыми программисты сталкиваются ежедневно.

Приведу пример из моей недавней работы над параллельными языками в рамках проекта Fortress. Предположим, вам надо сложить сколько-то чисел. Вы можете начать с нуля и прибавлять числа одно за другим – это традиционный последовательный подход.

Заметим, что у операции сложения есть нейтральный элемент. Вы должны начать с нуля. Тривиальное замечание, но все же сделаем его.

А можно сделать по-другому – расположить все числа в ряд и складывать их попарно. Вы получите ряд сумм, и в конце концов останется лишь одна сумма. Если количество чисел нечетное, последнее из них прибавляется к общей сумме и так далее. Этот метод тоже вполне применим. Для чисел с плавающей запятой надо быть строже в расчетах, хотя порой это не требуется – слишком велики издержки.

Результат получится тот же самый, по крайней мере, если мы имеем дело с целыми числами, как если бы мы начинали с нуля и прибавляли числа по одному, – в силу ассоциативности сложения. Иными словами, неважно, каким образом вы группируете числа.

Есть и третий метод. Допустим, имеется сколько-то процессоров. Вы распределяете пары по процессорам. Алгоритм «начать с нуля и прибавлять числа по одному» трудно распараллелить, а вот при методе разбивки на пары у вас как бы образуется дерево, разные части которого обрабатываются разными процессорами. Они делают это независимо друг от друга, и лишь в конце операции должны взаимодействовать между собой, чтобы вы получили сумму.

И это круто. Вот еще одна стратегия распараллеливания, больше похожая на первую: инициализируем какой-нибудь регистр нулем, а потом процессоры борются за то, чтобы взять следующее число и прибавить его к этому регистру. Встает вопрос о синхронизации, но ответ-то будет тот же самый. Ведь сложение не только ассоциативно, но и коммутативно. То есть не имеет значения не только порядок группировки чисел, но и порядок их обработки.

У математиков для описания всего этого есть громоздкие устрашающие слова – «нейтральный элемент», «ассоциативность», «коммутативность». Я попытался растолковать понятнее. Программистам надо просто знать, что порядок операций не имеет значения и что можно перегруппировывать объекты. То есть в какой-то мере математические понятия, я считаю, важны для программистов.

Сейбел: Да, это хороший пример, потому что его поймет каждый, кто знает арифметику. Но не считаете ли вы, что таким же образом в программирование входят и понятия более высокого уровня?

Стил: Предположим, я генерирую отчет. Типичная ситуация: я делаю сколько-то выводов на печать, они должны быть выведены в определенном порядке. В многоядерном мире я, возможно, захочу делать это не в линейном порядке, а разложить на разные процессоры. Как мне связать все воедино? Можно ли использовать те же методы, что при сложении чисел? Выясняется, что здесь есть ассоциативность, но нет коммутативности, – тогда я знаю, какие приемы будут работать для строк, а какие не будут. Как разработчик языков программирования, имеющий дело с созданием параллельных языков, я нахожу эти понятия и прилагающийся к ним словарь очень полезными.

Сейбел: Кстати о создании языков: как менялся ваш подход с течением времени?

Стил: Главную переменую я описал в своем докладе «Growing a Language» (Выращивание языка) на конференции по объектно-ориентированному

программированию OOPSLA 1998 года. В 1970-е годы создатель языка готовил для него проект, реализовывал его и на этом всё. Или не всё; но вы оставались с мыслью, что язык завершен.

Возьмем Паскаль. Что-то по каким-то причинам туда вошло, что-то не вошло, но проектирование языка был завершено. Если бы выяснилось, что чего-то не хватает, что работа со строками никуда не годится, что ж, не повезло. Вирт создал такой язык. ПЛ/1 и Ада были спроектированы таким же образом. Возможно, Ада и C++ стали едва ли не последними языками такого рода. В меньшей степени C++ – он все-таки развивался с течением времени.

Затем языки становились все сложнее, было уже невозможно проектировать их за один раз, и эти языки уже эволюционировали с течением времени, потому что они были слишком велики, чтобы их спроектировать или реализовать за один раз. Вот это и определило смену моего подхода к проектированию языков.

Сейбел: Значит, по-вашему, Java проектировался уже по-другому?

Стил: Наверное, нет. Хотя должен был бы. Java менялся благодаря Java Community Process. Это касалось больше API, чем ядра. Последние 12–13 лет к языку добавлялись все новые свойства, но, наверное, его создатели в начале 1990-х думали, что создают совершенный язык для конкретных ограниченных целей. Вы знаете, их целью были ресиверы для телевизоров.

Сейбел: Верно.

Стил: Они не рассчитывали на Интернет или на такую широкую пользовательскую базу, как сейчас. По-моему, они хотели создать компактный, автономный базовый язык, над которым можно было бы надстраивать API для разных целей. Мой доклад «Выращивание языка» отчасти был посвящен этому процессу: тому, как Java оказался слишком маленьким, а пользователям нужно было больше свойств для их целей.

Особенно требовалось нечто вроде итерации с помощью цикла `for` путем перечисления. И это добавили в язык. Ученые, которые занимались высокопроизводительными научными вычислениями, требовали большей поддержки операций с плавающей запятой и тому подобного. Но участники Java Community Process это в целом отвергли – по причинам не столько техническим, сколько социальным, мне кажется.

Так или иначе, существовала потребность что-то добавлять к языку, и это вылилось в многоплановый социальный процесс. Думаю, для создания действительно успешного языка нужно планировать такой – социальный – процесс в той же мере, в какой и технические особенности языка. Нужно думать о взаимодействии этих двух вещей. Fortress стал

нашим – по крайней мере, моим – первым экспериментом подобного рода. И он еще не закончен, мы пока что на середине пути.

Сейбел: А Common Lisp, в создании которого вы участвовали, может служить в этом смысле примером?

Стил: Да, это еще один из «ранних» языков, который, будучи противопоставленным языкам вроде Java, заставил меня задуматься о «выращивании языка». Я хорошо знаю историю Лиспа, знаю, как возможности его макросов помогли ему безболезненно эволюционировать, помогли людям участвовать в добавлении новых свойств.

Сейбел: Недавно три языка, которые вы в той или иной степени проектировали, прошли или еще проходят болезненный редизайн. Scheme прошел R6RS; JavaScript – ECMAScript – проходит через споры «ES4 против ES3». И с Java спорят по поводу того, надо ли, а если надо, то как добавлять туда замыкания.

Стил: Кстати, да.

Сейбел: Это примеры языков, которым не хватило встроенных технических или социальных средств для того, чтобы развиваться нормально, и поэтому им пришлось пройти через этот болезненный процесс роста? Или это неизбежно?

Стил: Если язык не умирает, он растет. Он всегда испытывает давление – нужны изменения, люди хотят изменить свой инструмент, чтобы он лучше решал их сегодняшние задачи, а не те, которые были у них лет пять назад. Я говорю не о том, будет язык расти или нет. Я говорю о технических решениях, которые нужно принять на раннем этапе проектирования языка, чтобы облегчить дальнейший рост. И я думаю, что одним языкам легче расти именно за счет таких технических моментов. Но социальные обстоятельства тоже играют свою роль.

Сейбел: А есть примеры языков, которые развивались легко?

Стил: Ну, думаю, Лисп может служить примером языка, который легко вырос – благодаря гибкости системы макросов. А отчасти тут сыграла роль атмосфера, царившая в группе разработчиков.

Scheme, напротив, развивался с большим трудом – отчасти потому, что внутри сообщества разработчиков установилось правило: любое изменение должно быть одобрено всеми. Или почти всеми. Один черный шар – и все. А разработчики Common Lisp решили, что достаточно согласия большинства. Там народ не сходил с ума из-за каждой мелочи – люди знали, что взамен получают что-то полезное.

Сейбел: Выбор языка действительно важен? Есть ли серьезные доводы в пользу одного языка или другого? Или все это дело вкуса?

Стил: А что, вкус – плохой довод?

Сейбел: Допустим, я люблю ванильное мороженое, вы – шоколадное, но мы не станем из-за этого *спорить*. А люди спорят из-за языков программирования.

Стил: Это социальный феномен – желание быть на стороне победителей. Не думаю, что тут стоит спорить, но стоит иметь свое мнение о том, какой инструмент для какой цели подходит больше.

Единственное, в чем я реально убежден: ошибочно считать, будто один язык решит все проблемы лучше другого или хотя бы так же хорошо. В одной области лучше применять один язык, в другой – какой-то еще.

Например, проектируя алгоритмы, я смешиваю языки. Обдумывая что-то, я пишу на доске код на Java и тут же на Фортране, на APL. Я сам смогу разобрать, что у меня вышло, и это главное. Такая форма записи позволяет мне сделать каждый кусок алгоритма как можно более ясным и полезным.

Проблема вот в чем: даже если запись удобна для ограниченного числа целей, ее все равно надо встраивать в контекст языка, приходится что-то дописывать, и если ты где-то что-то упустил, то язык получается неоднородным – он хорошо подходит для одних вещей, но для работы с остальными слишком тяжел.

С другой стороны, очень сложно создать язык, пригодный для всего сразу, – отчасти потому, что есть много способов сократить запись. Это как код Хаффмана: если в одном месте удалось быть кратким, в другом придется быть многословным. И проектируя язык, надо думать вот о чем: что именно хочешь сделать кратким для изложения, а что – легко доступным для понимания. Если это осознать и использовать для этих целей некоторые символы, получится, что какие-то другие вещи сказать уже сложнее.

Сейбел: Одно из решений как раз предлагает Лисп, где все записи средней длины. И пользователи могут делать свои синтаксические расширения на базовом уровне языка – в том же духе, с записями средней длины. Тем не менее многие не любят S-выражения. Многие лисперы самоуверенно считают, что недовольные этим языком просто не видят всей прелести этого решения. А вы достаточно самоуверенны, чтобы полагать, что если человек действительно понимает Лисп, то скобки не будут его раздражать?

Стил: Нет. Я не считаю себя самоуверенным лиспером. Я изучил много языков и, пожалуй, лучше других понимаю, что разные языки пригодны для разных целей. И лучше пользоваться всеми, чем потрясать каким-то одним со словами: «Вот язык-победитель».

Есть проекты, для которых я обязательно возьму Лисп, потому что его инструменты мне подходят. Например, готовая система ввода/вывода – если я согласен пользоваться синтаксисом Лисп, то у меня сразу есть считывание и вывод на печать, пригодные для некоторых типов работ. Это, в свою очередь, позволяет делать быстрое прототипирование. С другой стороны, если мне нужно приспособить систему ввода/вывода к существующему специфичному формату, Лисп будет не столь хорош. Или я могу создать преобразователь на каком-нибудь языке, на Лисп или на любом другом, чтобы использовать его из Лисп.

Сейбел: Какими языками вы пользовались всерьез? Список, наверное, будет длинным...

Стил: Я заработал свои первые деньги, программируя на Коболе, еще в старшей школе. Я подрядился сделать систему по генерации таблиц успеваемости для какой-то другой школы. Потом были Фортран, язык ассемблера IBM 1130, машинный язык PDP-10, APL. Снобол всерьез я не использовал. Ну и, конечно, Си, C++, Bliss, язык реализации для DECsystems, разработанный в университете Карнеги-Меллона, GNAL, основанный на Red, – его я использовал довольно серьезно.

Кроме них всяческие разновидности Лиспа, включая Common Lisp, Scheme, MacLisp. И версию Лиспа, которую мы с Диком Гэбриелом создали для S-1 – S-1 Lisp; потом он стал одной из четырех или пяти частей, которые вместе образовали Common Lisp. Я разработал Connection Machine Lisp, но вроде ничего серьезного на нем не писал. Он, кажется, был реализован на *Lisp. Не надо путать *Lisp и Connection Machine Lisp – это два разных языка.

Довольно много я писал на C* – его мы тоже разработали для Connection Machine. Java, конечно же. Писал на некоторых скриптовых языках – JavaScript, Tcl.

Всерьез я имел дело также с Haskell – работал с ним больше месяца, написал длинный фрагмент кода. Фокал, ранний интерактивный язык для компьютеров DEC, похожий... немного на Бейсик, немного на JOSS. Помнится, на Бейсике я тоже писал. TECO (Text Editor and Corrector) – я пользовался им для создания ранней версии Emacs, значит, его тоже можно отнести к языкам программирования. На TECO пришлось писать очень много. И еще TeX, если и его рассматривать как язык программирования. Думаю, это основные.

Сейбел: Из сказанного вами я делаю вывод, что на вопрос «Какой ваш любимый язык программирования?» ответом будет дзэнское «му».

Стил: У меня трое детей: кого я люблю больше всех? Они все хорошие – это разные личности с разными способностями.

Сейбел: А есть языки, которыми вам просто не нравится пользоваться?

Стил: Удовольствие так или иначе доставляет любой язык. Но с некоторыми сложнее, чем с остальными. Когда-то мне очень нравился ТЕСО, но возвращаться к нему я не хочу. С ним были проблемы: если я через месяц брал собственный код на ТЕСО, то не понимал, что там написано.

На Perl я писал слишком мало, чтобы говорить уверенно, но он меня не привлек. И C++ тоже. Я писал код на C++. Думаю, все, что делается на C++, можно так же хорошо сделать на Java, но с меньшими усилиями. Если во главу угла не ставится эффективность.

Но я вовсе не хочу ставить под сомнение усилия Бьерна Страуструпа. Он ставил целью создать объектно-ориентированный язык, полностью обратно совместимый с Си. Трудная задача. Мне кажется, он с ней справился – по структуре язык великолепен. Но с учетом программистских задач, которые стоят передо мной, думаю, это желание добиться полной обратной совместимости с Си было роковой ошибкой. И исправить тут ничего нельзя. Система типов в Си никуда не годится. Она помогает кое в чем, но в ней есть слабые места и на нее нельзя полагаться.

Сейбел: Как по-вашему, языки становятся лучше? Вы продолжаете их проектировать, а значит, считаете, что это дело стоящее. Легче ли стало этим заниматься благодаря достигнутому прогрессу?

Стил: Да, сейчас намного легче писать те программы, которые мы пытались писать 30 лет назад. Но ведь и наши амбиции непомерно выросли. Поэтому программировать сейчас труднее, чем 30 лет назад.

Сейбел: Из-за чего именно труднее?

Стил: Думаю, сегодня есть такие же умные люди, как и 30 лет назад, которые используют свои возможности до последнего. «30 лет назад» – это такая произвольная дата, просто я тогда окончил школу. В чем разница? Я уже говорил: сейчас нельзя охватить все, что происходит в какой-нибудь области. Даже думать, что можешь, больше уже нельзя. Сегодняшние программисты противостоят более сложной среде, при этом проявляя такой же уровень мастерства, но в среде, которую все сложнее охватить. И мы создаем все более совершенные языки, чтобы помочь им справиться с изменчивостью этой среды.

Сейбел: Интересно, что вы сказали «все более совершенные языки». Есть такое течение – его можно назвать «поклонники стиля Scheme». Его представители считают, что единственный способ победить сложность – делать все, включая языки программирования, очень простым.

Стил: По-моему, язык должен передавать все, что программист хочет сообщить компьютеру, чтобы все было зафиксировано и учтено. Сейчас у разных программистов разные взгляды на то, что именно им нужно. Мое понимание этого менялось. Думаю, нужно гораздо больше сообщать о структурах данных и об их инвариантах. То, что есть

в Javadoc, – это то, что нужно сообщить компилятору. Мне кажется, все, что имеет смысл сообщить другому программисту, имеет смысл сообщить и компилятору.

Сейбел: Ведь в Javadoc мы видим главным образом вполне читаемую прозу, которая развилась из кода?

Стил: Отчасти да, отчасти нет. В коде на Java плохо улавливается связь между параметрами. Скажем, у нас есть массив данных и есть целое число, и это целое число должно быть корректным индексом этого массива. В Java выразить это не так просто. А это важно. В Fortress это можно сделать.

Сейбел: Это скомпилировано в утверждения (asserts) времени выполнения или проверяется статически?

Стил: Зависит от того, как удобнее. И так, и так. Мы стараемся, чтобы в Fortress такие связи улавливались. Ранее мы говорили об алгебраических связях, о том, что некоторые операции ассоциативны. Мы хотим, чтобы в Fortress это можно было указывать явно. Вряд ли каждый прикладной программист будет останавливаться и думать: «Ага, подпрограмма, которую я написал, ассоциативна».

Но создатели библиотек вынуждены думать об этом. Ведь если они используют изоопределенные алгоритмы, то правильность алгоритма сильно зависит от таких моментов. И если это так, то нам нужно выражать это на языке, понятном компилятору. По-моему, это важно для будущего – придать языку свойства, необходимые программисту.

Сейбел: А как насчет того, чтобы язык не позволял совершать ошибки? Одни думают так: «Если сделать язык достаточно закрытым, то невозможно будет писать плохой код». Другие же, наоборот, говорят: «Забудьте, такой подход обречен, мы можем с тем же успехом оставить все широко открытым, а программистам надо просто быть умнее». Как здесь найти баланс?

Стил: Важно понять, что тут все равно будет компромисс. Невозможно искоренить весь плохой код. Можно устранить самые вероятные ошибки, требуя, чтобы код спрашивал: «Мама, можно я?..»: когда что-то сделать чуть труднее, человек задумается и скажет себе: «Да, я имел в виду именно это». А можно некоторые вещи сделать намеренно трудными или невозможными, чтобы стало невозможно повредить, скажем, систему типов. Здесь есть плюсы и минусы – очень сложно писать драйверы устройств для голого железа на типобезопасном языке, потому что уровень абстракции будет слишком высок для голого железа. Можно попробовать добавить конструкции вида: «Эта переменная – действительно такой-то регистр устройства по абсолютному адресу XXXX». Но все равно это не очень безопасно.

Сейбел: Есть ли в современных языках какие-нибудь интересные сюрпризы?

Стил: Python очень неплох – в том смысле, что хорошо структурирован. Но Гвидо изначально не добавил сборку мусора, и мне это не нравилось. Потом он вроде бы пересмотрел свое решение, как я и предвидел. Там есть интересные синтаксические находки: отступы, например, и двоеточия в конце некоторых конструкций; это очень остроумно. Реализация объектов и замыканий тоже довольно любопытна.

Сейбел: Большинство лисперов, вероятно, думают, что замыкания там убогие, лямбда-выражения весьма ограниченные.

Стил: Это правда. Однако Гвидо приходилось идти на компромисс между ясностью, возможностью реализации и так далее. И все равно получился интересный набор свойств. Я сделал бы по-другому, но он работал для определенного пользовательского сообщества. Я понимаю, почему он в каждом случае поступил так, а не иначе, и уважаю его выбор. Haskell – красивый язык, я люблю его, хотя использую мало.

Сейбел: Итак, любя Haskell, вы сейчас проектируете язык, но Fortress – это не чисто функциональный язык?

Стил: Сейчас в Haskell используются монады: монада ввода/вывода, монада транзакционной памяти. Есть теория, что это очень функционально; может, это и правда помогает в работе. Но с другой стороны, кажется, что язык становится все более императивным. Как там говорил Белый Рыцарь из «Зазеркалья»? «Но я обдумывал свой план, как щеки мазать мелом, а у лица носить экран, чтоб не казаться белым»¹. Монады для меня – как раз такой экран: сначала вы вытаскиваете ввод/вывод, потом пытаетесь его скрыть обратно – и в итоге непонятно, есть побочные эффекты или нет.

Скажу вот что: примерно раз в месяц у меня возникает чувство, что в работе с Fortress надо было бы идти со стороны Haskell к Фортрану и Java, а не брать Фортран и Java и двигаться в сторону Haskell. Проектируя библиотеки для Fortress, мы все больше применяем функциональный подход – и все чаще встречаем трудности в создании эффективных параллельных структур данных.

Сейбел: Вы пишете много прозы, этот род деятельности вам также интересен. Как по-вашему, писать прозу и код – занятия одного порядка или нет?

Стил: Скорее разного: я четко сознаю, что у читателя прозы иная система обработки данных, чем у компьютера. И я не могу, скажем, в той

¹ Пер. Н. Демуровой.

же мере использовать рекурсию. Правда, для утонченных читателей я порой прибегаю к этому приему. Но всегда ясно сознаю, как читатель будет обрабатывать текст в уме и понимать его.

Иногда я очень беспокоюсь, когда пишу прозу, – с кодом я волнуюсь куда меньше. Это из-за неоднозначности слов. Мне все время кажется, будто меня поймут не так. И я провожу много времени, пытаюсь отточить стиль моей прозы, употребляю конструкции, которые сложно понять двояко.

У меня есть любимый скетч из передачи «Saturday Night Live» – тот, где Эд Эснер изображает сотрудника АЭС, который уезжает в отпуск на две недели. Перед уходом он говорит: «Всем пока! Помните, в ядерном реакторе не может быть слишком много теплоносителя». И дальше минуты три все обсуждают, что же он хотел сказать.

Сейбел: Итак, вы видите очевидный контраст между текстами для человека и текстами для компьютера. Но ведь многие, как Кнут, указывают, что написанный вами код обращен и к человеку – не меньше, чем к компьютеру.

Стил: О, это так.

Сейбел: Значит, в этом плане программисту полезно писать прозу?

Стил: Конечно. Работая над кодом, я все время думаю: поймет ли компьютер, чего я от него хочу? Скорее даже так: поймет ли он меня однозначно? А не в смысле, что совсем не поймет. Часто сказать что-то правильно можно разными способами. И тут я начинаю переживать за человека, читающего код. И одновременно за эффективность кода.

Это опять же компромисс: если важна эффективность, мы прибегаем к разным уловкам. Но тогда оказывается сбитым с толку человек. И надо добавлять комментарии или как-то еще делать код читаемым. Обычно выбор имен переменных и организация кода – забота скорее о читателе, о форматировании, которое безразлично компьютеру, но облегчает чтение человеку.

Сейбел: По мере того как языки улучшаются или хотя бы становятся дружелюбными к программисту – в сравнении с временами ассемблера и перфокарт, – избегать ошибок в программах вроде бы становится легче. Есть компиляторы, сигнализирующие об ошибках, и так далее. Так можно ли отдать предпочтение – пусть небольшое – читаемости кода перед правильностью? Как говорят разработчики на Haskell: «Если ваша программа на Haskell выполняет проверку типов, можно спать спокойно».

Стил: Думаю, это страшное заблуждение. В пропущенных через компилятор программах всегда слишком много ошибок, чтобы спать спо-

койно. А если есть ошибки, они собьют с толку не только компьютер, но и человека, читающего код.

Программирование – глубоко неестественный вид деятельности, и ему надо как следует учиться. Люди привыкли, что их слушатели сами восстанавливают лакуны в речи. И с компиляторами мы обращаемся в какой-то мере так же. Говоря: «Мне нужна переменная с именем `foo`», – вы не заботитесь о регистре имени. Люди неточны, часто неряшливы в своей речи. Но когда мы даем команды машине, детали важны, потому что мелкая погрешность может изменить ход всего процесса.

Мне кажется, люди привыкли к использованию рекурсии в ограниченном виде – кажется, это показал Ноам Хомский. Но на деле люди редко углубляются больше чем на три уровня, а если и углубляются, то обычно в стиле хвостовой рекурсии. Понимание рекурсии есть сложнейшее искусство. А ведь рекурсия – один из самых мощных инструментов программиста, если его как следует освоить. И поэтому заботиться о корректности надо всегда.

Сейбел: Тем не менее многие пытались разработать языки или системы, дающие возможность написать программу непрограммисту. Считаете ли вы эти попытки обреченными на неудачу – ведь дело не в правильности синтаксиса, а в том, что программирование неестественно по природе.

Стил: Да. И, кроме того, люди сосредоточены на главном, они не думают о пограничных случаях, о сложных случаях, о маловероятных случаях. А именно в таких случаях начинаются разногласия – как сделать правильно.

Иногда я спрашиваю студента: «Что будет в таком-то случае?» – «То-то и то-то». И тут же кто-нибудь вскакивает: «Нет, должно быть вот так!» Вот такие вещи и надо отражать в программной спецификации.

Неслучайно для описания процесса программирования мы пользуемся терминами из арсенала магии. Мы говорим: что-то произошло как по волшебству или автоматически. Думаю, это оттого, что заставить машину сделать нужную тебе вещь – почти то же самое, что добиться исполнения желания.

Посмотрите: героям волшебных сказок достаточно придумать желание, махнуть рукой – и вот оно выполнено. И, конечно, сказки полны поучительных ситуаций: герой забыл учесть пограничный случай – и из-за этого случилось что-то нехорошее.

Сейбел: Возьмем «Fantasia» – она в том числе об опасности рекурсии.

Стил: «Fantasia» и рекурсия, да. Или «Я хочу быть самым богатым человеком в стране», – в итоге все становятся бедняками, а он остается

при своем. Такое в волшебных сказках случается, потому что люди забывают о разных путях, ведущих к цели. Если думать только о главном желании, пренебрегая деталями, много чего не стыкуется.

Сейбел: Каков же урок волшебных сказок? Гэндальфы становятся великими магами путем тяжких трудов и зубрежки заклинаний, а легкого пути нет?

Стил: Да. Другой пример. Допустим, я говорю своему умному компьютеру: «Хочу, чтобы имена в моей телефонной книге шли по алфавиту», — и он выбирает все имена, кроме первого. Алфавитный порядок не нарушен, но это не то, чего мне хотелось. И оказывается, что спецификацию вида «хочу упорядочить имена по алфавиту, без потери данных, без дублирования» чертовски трудно написать.

Сейбел: Получается, есть свойства языков, которые делают труд программистов, освоивших эту неестественную работу, более продуктивным? Вы сейчас проектируете язык, у вас должно быть мнение на этот счет.

Стил: Я уже говорил: нельзя пренебрегать точностью. В то же время мы можем создавать инструменты для повышения точности. Мы не можем сделать эту процедуру тривиальной, но можем помочь избежать некоторых ошибок. Вместо того чтобы допускать циклическое переполнение 32-битных целых, можно сделать обнаружение арифметического переполнения или предоставить возможность работы с большими числами. Сейчас реализация всего этого обходится дороже, но я считаю, что работа с настоящими большими числами дает немного меньше ошибок для некоторых видов программирования. Программисты и создатели алгоритмов для операционных систем все время попадают в одну и ту же ловушку. Они говорят: «Нам нужно синхронизировать фазы, так что будем брать по одному числу. При начале новой фазы вычисления будем увеличивать на единицу какую-нибудь переменную, получать новое число — и тогда все участники будут уверены, что работают в одной и той же фазе, пока не начнется какая-нибудь операция». Это работает на практике, но с 32-битными числами вы досчитаете до 4 миллиардов довольно быстро. Что будет в случае циклического переполнения? Все будет в порядке? Или нет? Многие подобные алгоритмы в компьютерной литературе содержат эту небольшую ошибку. Что если какой-нибудь поток застопорится со 2-й по 32-ю итерацию? Маловероятно, но все-таки возможно. Надо или как-то смягчить эту проблему в смысле корректности, или все просчитать и показать, что такой вариант настолько маловероятен, что беспокоиться не о чем. Или, возможно, для вас приемлем один компьютерный сбой в день. Суть в том, что надо проанализировать проблему, а не игнорировать ее. Тот факт, что счетчик может переполниться, — проблема еле заметная, большинство

программистов она не затронет. Но для остальных это ловушка в их алгоритмах.

Сейбел: Кстати о сбоях. Какова худшая ошибка, с которой вы имели дело?

Стил: Не уверен, что назову худшую, но могу кое-что рассказать. Самые трудноуловимые ошибки порождаются параллельными процессами.

Когда я был зеленым программистом и работал на IBM 1130, решение, как исправить ошибку, однажды явилось мне во сне. Или сразу после пробуждения. Я бился над ней пару дней, ничего не получалось. И вот посреди ночи – озарение. Оказалось, я кое-что пропустил в спецификации интерфейса.

Это было связано с параллельными процессами. Я писал декомпилятор, чтобы декомпилировать и изучить дисковую оперативную систему машин IBM. Для этого надо было взять с диска данные в двоичном виде и распечатать их в разных форматах – как инструкции, как коды символов, как числа и так далее. Для преобразования символов я скамливал их разным функциям преобразования, одна из которых была предназначена для работы с кодом, считанным через устройство для чтения перфокарт. И я пропустил крохотное примечание в спецификации: «Прежде чем вызвать эту функцию, необходимо очистить младшие биты в буфере, в который будут считываться данные с перфокарты». Или, наоборот, их надо было установить.

Так или иначе, 12 бит с карты записывались в старшие 12 бит 16-битного слова, а младшие разряды использовались для хитрого трюка: можно было запустить функцию чтения перфокарты асинхронно, и тогда буфер заполнялся тоже асинхронно, и при этом выполнялась функция преобразования. И этот младший разряд определял, была ли считана следующая колонка перфокарты. Если была, то выполнялось преобразование. Таким образом, почти сразу после считывания всей перфокарты преобразование завершалось – за счет того, что эти процессы перекрывали друг друга, получался выигрыш во времени. Я же скамливал в функцию сырые двоичные данные, которые не подчинялись этим ограничениям. Я просто не обратил внимания на примечание. Я думал, что это обычная функция преобразования, а оказалось, что в интерфейсе этой функции есть особенность: она задействовала младшие разряды, о которых обычно думать не приходится. Она обрабатывала буфер и говорила мне: «Данные еще не поступили из устройства для чтения перфокарт». В принципе, я знал, что такое возможно, но тогда это мне в голову не пришло. А потом во сне меня озарило. Вот такой странный случай.

А вот другая занятная история. Я отвечал за MacIisp, а MacIisp поддерживал большие числа – целые числа произвольной точности. Они у нас были уже несколько лет, считалось, что они хорошо отлажены. Они широко использовались в Массума, пользователи Массума все время с ними работали. И вот приходит сообщение от Билла Госпера: «Частное двух этих целых чисел неверно». Он заметил это, поскольку частное примерно равнялось 10л.

В каждом числе было знаков по сто, и вручную выследить ошибку было невозможно – программа деления была сложной, а числа – большими. Я стал смотреть на код – с виду ничего такого. Но взгляд зацепился за условный оператор, который я не понял.

Эти функции базировались на алгоритмах Кнута. Я достал с полки его книгу, прочел спецификацию и начал переводить алгоритм в код на языке ассемблера. Я увидел комментарий Кнута о том, что этот шаг выполняется редко – с вероятностью примерно два в степени минус размер машинного слова. У нас должно было выходить примерно один раз на четыре миллиарда случаев.

Я подумал, что функции отлажены, ошибка должна проявляться редко, то есть она где-то в коде, который редко выполняется. Я стал изучать этот код и понял, что структура данных копируется неверно. В итоге дальше по коду обнаружилась ошибка, возникшая в результате побочного эффекта: там что-то затиралось. Я исправил это, пропустил числа через функцию и получил верное значение. Госпер был доволен.

А неделю спустя он пришел с двумя числами – они были еще больше – со словами: «Эти тоже делаются неправильно». Но я уже был готов: вернулся к тому самому маленькому куску из десятка инструкций и обнаружил вторую ошибку того же рода в том же самом коде. Я тщательно проверил весь код, убедился, что все копируется правильно, и больше проблем не было.

Сейбел: Как обычно – ошибка не ходит одна.

Стил: Вот я и вынес урок: ошибок может быть больше одной, и в первом случае надо было смотреть тщательнее на предмет других ошибок. Другой урок в том, что если ошибка проявляется редко, то надо смотреть участки кода, которые редко выполняются. И третий: желательно иметь хорошую документацию по алгоритму, в моем случае – книгу Кнута.

Сейбел: Кроме ночных озарений, каков ваш излюбленный метод отладки? Что вы предпочитаете – символьные отладчики, вывод на печать, утверждения, формальные доказательства или все сразу?

Стил: Честно говоря, я ленив и начинаю с вывода на печать, хотя при сложных ошибках это самый неэффективный метод. Но зато с про-

стыми работает хорошо, так что попробовать стоит. С другой стороны, на мое отношение к программированию сильно повлияла работа над проектом, написанном на Haskell. Так как это чисто функциональный язык, там нельзя было использовать вывод на печать.

И я полностью перешел на модульное тестирование. Пришлось придумывать модульные тесты для каждой из функций. Крайне полезный опыт.

Это повлияло на облик Fortress – я постарался ввести туда свойства, поощряющие модульное тестирование. И записать их не в отдельные файлы, а вместе с текстом программы. Мы заимствовали кое-что из контрактного программирования языка Eiffel – предусловия и постусловия для процедур. Есть места, где размещаются тестовые данные и процедуры модульных тестов, и тестовая программа запускает эти процедуры по первому вашему требованию.

Сейбел: Раз уж вы упомянули контрактное программирование: как вы используете утверждения в собственном коде?

Стил: Обычно я вставляю утверждения преимущественно в начале процедур и в самых важных местах по всему коду. И пытаюсь – «доказать», видимо, слишком сильное слово – убедить себя в правильности какого-то кода, я часто думаю в терминах инвариантов: слежу, чтобы инварианты поддерживались. Думаю, это плодотворный подход.

Сейбел: А как насчет пошаговой отладки? Если все прочее не помогает, вы прибегаете к ней?

Стил: Зависит от длины программы. И, конечно, помогают инструменты, которые позволяют пропускать целые заведомо безошибочные куски. В Common Lisp есть отличная функция STEP. Я много раз пошагово отлаживал код на Common Lisp. Это очень здорово – возможность пропустить функции, в которых уверен. И еще возможность расставить ловушки и сказать себе: «Сюда мне смотреть не нужно, до тех пор пока этот цикл не повторится семнадцать раз». На PDP-10 были для этого аппаратные средства, это было здорово, особенно в MIT. Тогда они любили модернизировать свои машины, добавляя в них что-нибудь. Можно долго рассказывать про выполнение одного и того же кода разными способами.

Сейбел: Вы применяете к своему коду формальные доказательства?

Стил: Зависит от кода. Если в нем есть сложные математические инварианты, я использую доказательства. Я не стану писать функцию сортировки, пока не подберу какой-нибудь инвариант и не докажу его.

Сейбел: Питер ван дер Линден в своей книге «Expert C Programming» (Программирование на Си для экспертов) отвел доказательствам целую

главу в пренебрежительном духе. Вот вам доказательство того-то, но смотрите – оно с ошибками! Ха-ха-ха!

Стил: Да, и доказательства могут быть с ошибками.

Сейбел: Но, по крайней мере, встретить ошибки в них меньше шансов, чем в проверяемом коде?

Стил: Думаю, да, ведь вы используете разные инструменты. Вы используете доказательства затем же, зачем и типы данных, – затем, зачем альпинист пользуется веревкой. Если все хорошо, они не нужны. Но если что-то не так, они увеличивают шанс найти ошибку.

Сейбел: Думаю, худший случай – это ошибка в программе, скрытая ошибкой в доказательстве. Но, к счастью, редкий.

Стил: Такое случается. Не уверен даже, что столь уж редко, ведь вы конструируете доказательства, отражающие структуру кода. И наоборот, если при написании кода вы держите в уме доказательства, это повлияет на структуру вашей программы. Поэтому нельзя говорить о взаимной независимости кода и доказательства в вероятностном смысле. Но можно задействовать разные инструменты, разные способы мышления.

Программируя, вы погружаетесь в разные локальные мелочи, а инварианты дают глобальный взгляд на программу. И доказательства заставляют эти два взгляда взаимодействовать. Вы видите, как локальные шаги воздействуют на глобальный инвариант, который нужно поддерживать.

Вот один из самых интересных случаев в моей практике. Меня попросили написать отзыв на статью Дэвида Гриса для «САСМ». Грис писал о доказательстве правильности параллельного алгоритма сборки мусора. Сюзен Овицки была студенткой Гриса и разработала кое-какие инструменты для доказательства корректности параллельных программ. А Грис решил применить их к параллельному сборщику мусора, разработанному Дейкстрой. Код занимал где-то полстраницы. А остальная часть статьи содержала доказательство его корректности.

Я зарылся в это доказательство и стал проверять его шаг за шагом. Трудность состояла в том, что каждое выражение в программе могло нарушить любой инвариант, поскольку программа была параллельной. Поэтому Овицки предлагала перекрестную проверку в каждой точке. Потратив на это около 25 часов, я обнаружил, что пару шагов я пройти не могу. И сообщил об этом – оказалось, что в этих местах алгоритма были ошибки.

Сейбел: Итак, алгоритм содержал ошибки, и доказательство их пропустило.

Стил: Да, получилось, что доказательство некорректно. Что-то где-то было упущено. Какие-то детали работы с формулой – формула была почти правильна, но именно «почти». Дело было лишь в том, чтобы поменять местами, например, две строки кода.

Сейбел: Итак, 25 часов ушло на то, чтобы проанализировать доказательство. Вы бы смогли найти ошибку в коде за 25 часов, имея перед собой только код и больше ничего?

Стил: Вряд ли бы я даже понял, что там есть ошибка. Алгоритм был довольно мудреным: я посмотрел бы на код, понял его общий смысл – и не заметил бы неточности. Нужна была очень маловероятная последовательность действий, чтобы она проявилась.

Сейбел: И все выявилось в процессе доказательства, то есть вам не надо было рассматривать различные сценарии типа «что, если» для обнаружения проблемы.

Стил: Верно. Доказательства позволяют представить глобальную точку зрения и предусмотреть все возможности, резюмируя их при помощи очень сложной формулы. И чтобы это получить, приходится проработать кучу формул. Автор переработал статью, она снова вернулась ко мне на рецензию – и хотя я уже проделал это упражнение, пришлось опять потратить 25 часов на проверку доказательства. На этот раз, кажется, все было нормально.

Статья вышла, никто больше не находил в этом алгоритме ошибок. Есть ли они там сейчас? Не знаю. Но доказательство дает мне куда больше уверенности в том, что с алгоритмом теперь все нормально. Надеюсь, я не единственный рецензент, разобравшийся в доказательстве.

Сейбел: Дейкстра говорил, что тестирование не позволяет считать программу свободной от ошибок. Вы всего лишь показали, что ваши тесты ошибок не нашли. Но ведь с доказательствами то же самое – вы всего лишь можете сказать, что ваше доказательство не выявило ошибок.

Стил: Это правда. И вот почему есть отрасль компьютерных наук, изучающая автоматическую проверку доказательств. Надежда на то, что программа проверки доказательства окажется корректной. А она будет корректной при небольшом размере. И сделать ее небольшой гораздо проще, чем проверять доказательство какой-либо достаточно большой программы.

Сейбел: И программа автоматической проверки, проверенная вручную, может сэкономить 25 часов, потраченные вами на изучение доказательства фрагмента кода?

Стил: Да. Точно.

Сейбел: О чем еще вы бы хотели поговорить?

Стил: Мы не коснулись такой темы, как красота программ. А мне хочется сказать об этом пару слов. Некоторые программы буквально поражали меня своей красотой. Например, TeX, то есть исходный код TeX. METAFONT – в меньшей степени, но не знаю, почему: то ли потому, что я работал с ним меньше, то ли потому, что мне и вправду меньше нравятся структура кода и архитектура программы.

Есть великолепные алгоритмы, которыми я просто восхищался. Я видел чудесные программы для сжатия кода – в те времена у тебя был всего один мегабайт памяти, и это имело значение. Было важно, сколько слов ты используешь – 40 или 30, и люди временами серьезно работали над тем, чтобы ужать программу. Билл Госпер писал эти шедевры из четырех строк, и они творили потрясающие вещи, например с усилителем, подключенным к младшим битам аккумулятора, который в это время тасовал биты.

Это может показаться бессмысленной тратой сил, но одним из лучших моментов в моей карьере был тот, когда я смог сократить программу Госпера из 11 слов до 10. И это – ценой лишь небольшого увеличения времени выполнения программы, ценой малой доли машинного цикла. Я нашел способ сократить код на одно слово, и на это у меня ушло всего лишь 20 лет.

Сейбел: И вот, спустя 20 лет, вы сказали: «Привет, Билл, а представляешь?..»

Стил: Ну, собственно, я не занимался этим все 20 лет, просто 20 лет спустя я вернулся к этой программе, и ко мне пришло озарение: я понял, что, изменив один код операции, я получу константу с плавающей запятой, близкую к тому, что мне нужно. И смогу использовать инструкцию и как инструкцию, и как константу с плавающей запятой.

Сейбел: Прямо «История Мэла, настоящего программиста».

Стил: Точно. Я не хотел бы заниматься этим в жизни, но это был единственный раз, когда мне удалось сократить код Госпера. Это была победа. И это был красивейший кусок кода – рекурсивная подпрограмма для вычисления синусов и косинусов.

Вот такие вещи нас тогда волновали. Во времена IBM 1130 были загрузочные перфокарты: это одна перфокарта, которую надо было класть на верх стопки. Надо было нажать кнопку запуска, машина считывала первую карту и размещала ее содержимое в первых 80 ячейках памяти. Потом она запускала выполнение по указанному адресу. На первой карте была программа для чтения остальных карт. Так и происходила загрузка.

С IBM 1130 трудность состояла в том, что на перфокарте 12 строк, а в машинном слове было 16 бит. Так что на 16-битную инструкцию приходи-

лось 12 бит, то есть некоторые инструкции не помещались на перфокарте. Такие инструкции приходилось собирать с помощью тех инструкций, которые помещались на перфокарте. Возникла сложная система компромиссов, какие инструкции можно использовать: если я использую вот эту инструкцию, мне нужны будут еще вот эти инструкции на перфокарте, просто чтобы собрать ее. Огромная нагрузка плюс размер функции не мог превышать 80 слов. Поэтому приходилось использовать некоторые инструкции и как данные, использовать данные повторно для других целей. Если удавалось впихнуть эту функцию в память, то ее адрес мог использоваться как константа. Такой вот стиль программирования – не то оригами, не то хайку. Я этим занимался несколько лет.

Сейбел: Как вы думаете, те, кто прошел через это, сейчас справляются лучше или хуже других программистов?

Стил: Они приучены работать с ограниченными ресурсами и умеют точно их оценивать.

Сейбел: Точная оценка – полезный навык. Но он может оказаться и вредным, в смысле развития ненужных сегодня навыков.

Стил: Да, можно легко заикнуться на оптимизации чего только можно, даже если такая задача не стоит. Я рад, что мой сын в старшей школе освоил программирование на калькуляторах TI, где тоже были серьезные ограничения по памяти. Он научился представлять данные в сжатом виде, чтобы они подходили для калькулятора. Я не хочу, чтобы ему пришлось заниматься этим все время, но все равно опыт ценный.

Сейбел: Вернемся к красоте кода. В чем прелесть этого стиля хайку-оригами? В том, что каждая сложная миниатюрная вещь кажется нам прекрасной?

Стил: Да. Но подчеркну, что красота вышеупомянутого фрагмента кода Госпера не только в том, что его можно сжать вот таким образом. Он изначально был невелик по размеру, потому что основан на красивой математической формуле – формуле тройного угла для синуса. И эта рекурсия может быть выражена очень лаконично в этой архитектуре, потому что эта архитектура спроектирована для поддержки рекурсии, а не как современные машины. Одна функция сочетает в себе самую разноплановую эстетику.

Сейбел: Вы говорили о TeX Кнута, – она существенно больше по размерам. Что придает ей красоту?

Стил: Он взял невероятно сложную программу со множеством особых случаев и свел ее к очень простой парадигме: склеить блоки воедино. Это был важнейший прорыв. Появилось множество возможностей не только для набора текста, но и для других вещей, например для отображения на странице текста в двух измерениях. Я за то, чтобы в гра-

фических интерфейсах пользователя этот принцип склеивания блоков действовал при расположении кнопок и тому подобного.

Сейбел: Значит, здесь красота в том ощущении, что есть блок и клей, и можно сказать: «Да, это глубоко правильная идея, я проникся ее красотой и хочу видеть ее в других программах». Вы проникаетесь ее красотой в процессе чтения кода, глядя на соотношение его элементов? Или же, прочитав код, говорите: «Великолепно, все основано на этой простой, но не упрощенной идее»?

Стил: И то и другое. Кнут замечательно умеет рассказывать о коде. Можно целый день читать «Искусство программирования», погружаться в алгоритмы. Он объясняет вам их, показывает, как их можно использовать, дает упражнения, и создается ощущение, что вас увлекают в интересное путешествие. И попутно показывают вам очень занятные вещи. Если пробираться через код TeX, ощущения сходные. Я много чего понял о программировании. Одни куски кода вполне обычные, порой даже поверхностны. А при виде других говоришь себе: «Даже не подозревал, что можно сделать так». Есть и то, и то.

Сейбел: А противоположностью красоте кода будут всякие несообразности, которые сложились исторически. Например, разные соглашения о конце строки.

Стил: Да. В комитете по разработке Common Lisp мы часами обсуждали конец строки, чтобы добиться совместимости с UNIX, где есть только символ перевода строки, и с системами PDP-10, где используется CR LF. Надо было приспособить перевод строки для обеих систем. Настоящий кошмар.

Сейбел: Что можно посоветовать читателям этой книги, тем, кто будет писать программы будущего, чтобы избежать этих проблем? Можем ли мы быть умнее наших предшественников? Или все это – неизбежное следствие эволюции программирования?

Стил: Да. Будущее неизвестно. Может, и глупо прозвучит, но если бы можно было забраться в прошлое и изменить только одну вещь, я бы попробовал убедить первобытных людей не использовать для счета пальцы. Насколько стало бы легче жить их потомкам. Хотя, с другой стороны, мы много вынесли из своих попыток совместить десятичную систему счета со степенями двойки.

10

Дэн Ингаллс

Если Алан Кэй – отец языка Smalltalk, то Дэн Ингаллс – его мать: может, идея Smalltalk и блеснула Алану Кэю, но именно Ингаллс немало потрудились, чтобы явить его миру. Начав с первой реализации Smalltalk, написанной на Бейсике на основе одной страницы заметок Кэя, Ингаллс участвовал в реализации семи поколений Smalltalk – от первого прототипа до нынешней версии с открытым исходным кодом – Squeak.

Физик по образованию, Ингаллс начал программировать на Фортране, зарабатывал на продаже профилировщика, который написал еще в школе, и со временем оказался в Xerox PARC, где присоединился к Кэю и его группе исследования обучения, которая создала Smalltalk и исследовала применение компьютеров в обучении детей.

Работая в PARC, Ингаллс также придумал BitBlit – операцию, используемую в растровой графике, и запрограммировал ее в виде микрокода для компьютера PARC Alto, обеспечив ускоренную обработку растровой графики, благодаря чему стали возможны такие новшества пользовательского интерфейса, как всплывающие меню, которые сейчас кажутся нам совершенно обыденными. (На одной из первых внутренних демонстраций системы Smalltalk всплывающее меню так впечатлило Питера Дойча – интервью с ним в следующей главе, – что он вскочил и закричал: «Ты и правда сделал это – или мне показалось?»)

Ныне Ингаллс, выдающийся инженер Sun Microsystems, работает над Lively Kernel – программной средой типа Smalltalk, которая полностью выполняется в броузере с использованием JavaScript и предоставляемой браузером графики. За работу над Smalltalk он получил от Ассоциации вычислительной техники в 1984 году премию имени Грейс Мюррей Хоппер, а в 1987 году – премию ACM Software System Award. В 2002 году был удостоен награды Dr. Dobb's Excellence in Programming Award.

В беседе мы обсудили важность интерактивных программных сред, почему ему повезло так и не выучить Лисп и почему лучше создавать гибкие, динамичные системы и затем ограничивать их, а не строить статичные системы и затем пытаться добавить к ним динамические свойства.

Сейбел: Для начала: как вы пришли в программирование?

Ингаллс: Давайте посмотрим. Я всегда хотел быть изобретателем, а для этого лучше всего подходит физика, так что в колледже выбрал ее в качестве специальности. Я пошел в Гарвард, и там был курс программирования на Фортране.

Сейбел: Когда это было?

Ингаллс: Я учился в Гарварде с 1962 по 1966 год. Программирование изучал по двум направлениям: курс Фортрана и курс по аналоговым компьютерам. В подвале одного из корпусов была отличная лаборатория, занимающаяся аналоговыми компьютерами, где приходилось думать совершенно иначе: вот большая коммутационная панель и куча схем – интеграторов, дифференциаторов, которые нужно связать воедино, чтобы решать все задачи в реальном времени. Но все начиналось с Фортрана, который я полюбил сразу же. Я все пытался понять, можно ли сделать программы короче и все такое.

В результате я решил, что, возможно, меня заинтересует электротехника. Я поступил на это направление в Стэнфорде, записался на несколько курсов по компьютерным наукам, и мне очень понравилось. Электротехника не занимала у меня много времени. Я попал на курс к Дональду Кнуту, это был курс по оценке программ, и он пришелся мне по душе. Я, собственно, уже работал над программой, которая анализировала бы другие программы, и ушел из Стэнфорда, чтобы сделать на ней бизнес. На следующий год я поступил в магистратуру, а потом ушел. В 1968 году, полагаю.

Сейбел: То есть вы бросили магистратуру?

Ингаллс: Да, на кафедре радиотехники факультета электротехники Стэнфорда.

Сейбел: Что представляла собой программа, с которой вы начали свое дело?

Ингаллс: Все началось еще у Кнута в рамках его семинара по оценке программ и их динамического поведения.

Сейбел: То есть по профилированию?

Ингаллс: Ну, да. Была программа, которая внедрялась в программу на Фортране и вставляла счетчики во все точки ветвления. Я сделал версию посимпатичнее, в ней было прерывание по таймеру, так что она могла записывать, сколько времени потрачено в разных частях программы.

Сейбел: То есть фактически дискретный профилировщик?

Ингаллс: Да. И что еще тут важно: раньше профилирование по времени обычно работало в терминах адресов памяти и требовалось чуть ли не квантовая механика, чтобы понять, что означают полученные результаты. Здесь же работа шла в терминах исходного кода, и вы видели: «Вот, все время расходуется здесь». То есть пользователь мог немедленно воспользоваться информацией. Я понял: «О! На этом можно сыграть».

Сейбел: Вы занимались этим, пока не пришли в Xerox PARC ?

Ингаллс: В общем, да. Вот как я оказался в PARC. Я много времени проводил в местных сервисных центрах: один принадлежал Control Data, а другой – IBM. И я носил свою программу в разные места, чтобы убедиться, что она запустится на конкретном компьютере.

Сейбел: Это по-прежнему был код профилирования Фортрана?

Ингаллс: Да. Но я выяснил кое-что интересное. Кто в основном использует Фортран? Те, кто работает с объемными научными вычислениями. А на кого они обычно работают? На правительство. Интересно им, насколько эффективна их программа? На самом деле, нет. Чаще всего они как раз хотят показать, что их компьютер перегружен, что им нужен новый компьютер и больше денег. Я показал свою программу в паре компаний, и там сказали: «Эх, вот для Кобола – было бы отлично».

Сейбел: Потому что больше денег для крутого железа под Кобол им бы никто не дал.

Ингаллс: Вот именно. И я написал такую же штуку для Кобола. Это было мое боевое крещение в плане Кобола. Помню, как писал завершающую процедуру, которая собирала статистику прерываний по таймеру. Завершающая процедура должна быть написана на том же языке,

что и оцениваемая программа, чтобы в нее встроиться. Так что, наверное, я единственный в истории человек, который написал хеш-таблицу на Коболе.

Продажи шли, во всяком случае, неплохо. Помню несколько выездных продаж, когда я прогонял на их программах демоверсию и, пока она работала, показывал им, как сэкономить больше, чем стоит сама программа.

Бегаю по этим сервисным центрам, я как-то попал в центр CDC в Стэнфордском индустриальном парке. Обычно работаешь поздно ночью, потому что дешевле. Там я познакомился с парнем, у которого была программа на Фортране для распознавания речи. У него были различные образцы речи, его программа анализировала спектр, группировала фоны и все такое прочее. Я разговорился с ним и спросил: «Слушай, а не хочешь запустить свою программу с моим профилировщиком?» Так мы и сделали, а потом расстались.

Через пару недель он позвонил мне и сказал: «Я нанялся в Херох, буду делать для них проект по распознаванию речи. Но тут никто не может мне помочь с разной рутинной, не хочешь со мной поработать?» И я согласился. Это был Джордж Уайт, который долгие годы занимался распознаванием речи. Так я оказался связан с Херох, а также с Аланом Кэем, поскольку мой офис оказался прямо напротив его, так что доносящиеся оттуда разговоры привлекали мое внимание больше, чем распознавание речи.

Сейбел: Это распознавание речи оказалось таким скучным – или тут что-то связано с программированием?

Ингаллс: Нет, оно было интересным, даже захватывающим. Я в конце концов построил целую среду для персонального компьютера на том мини-компьютере Sigma 3. Использовались пачки перфокарт, работа велась главным образом на Фортране. Из всего этого я создал интерактивную среду. Я написал на Фортране текстовый редактор, а затем средство удаленного управления с терминала. Получилась симпатичная вычислительная среда, довольно затейливая.

Сейбел: Стремление к интерактивным средам красной нитью проходит через вашу карьеру. Например, первый Smalltalk вы написали на Бейсике, потому что он обеспечивал интерактивную среду. Когда вы впервые поняли, что главное условие решения проблемы – это интерактивная программная среда?

Ингаллс: Хороший вопрос. Думаю, немедленная отдача хороша сама по себе.

Сейбел: Где же вы впервые получили немедленную отдачу?

Ингаллс: Вспоминаю пару примеров. У меня была возможность поработать с наполовину интерактивным PL/I. А мой друг работал на IBM, когда у них была интерактивная среда APL. Не помню, что из этого было сначала. Кажется, APL. Она повлияла на меня во многих отношениях: как из-за немедленного взаимодействия – результаты сразу же возвращались, – так и из-за вычисления выражений, которое очень отличалось от ориентированного на операторы программирования на Фортране.

И это сохранилось до сих пор, просто здорово, – вся традиция Си/C++/Java, хотя и развивается в объектно-ориентированном направлении, по-прежнему ориентирована на операторы. Но если есть возможность удобной работы с выражениями, то это дает совершенно другой опыт. Для меня это применение математики в жизни. Короче, вот один из этих двух примеров. Когда я пришел в Хероx, интерактивных языков, кроме Лиспа, не было. Но мне посчастливилось не подсесть на Лисп. Все сложилось бы иначе, если бы я на него подсел.

Сейбел: Как так?

Ингаллс: Думаю, я бы тогда полностью перешел на то направление. А поскольку я не перешел, мне хочется делать то же самое по-другому. Думаю, то, над чем мы работали с Аланом, было столь же симпатично и живо, но включало более естественное представление объектов и сообщений.

Думаю, если бы мне было так же удобно в системе вроде Лиспа, то я бы и не беспокоился. Я бы попытался построить работу с ней таким образом, чтобы получить объекты, но начать работать с объектами, а затем сделать эту работу интерактивной и удобной было, думаю, серьезным вкладом.

Сейбел: Алан Кэй сказал, что и у Лиспа, и у Smalltalk одна и та же проблема: они настолько хороши, что пожирают собственных детей. И если бы вы знали Лисп, то Smalltalk оказался бы первым съеденным ребенком.

Ингаллс: Возможно.

Сейбел: Так что перед нами отличный пример счастья в неведении: оно оставляет место для творчества. Но порой кажется, что неведение вообще свойственно данной отрасли: люди ни о чем не имеют понятия и постоянно изобретают колеса с острыми углами.

Ингаллс: Так и есть.

Сейбел: Нужно ли с этим бороться? Или это просто цена пространства для творчества? Или лучше бы нам больше понимать, что происходит в мире?

Ингаллс: Я за разнообразие. Из приведенного мною примера можно заключить: «Нет, пусть люди делают что хотят». Да, от незнания порой делаешь лишнее, но естественный отбор все исправит. И в конце концов все это помогает продвигаться в будущее.

Можно вспомнить множество областей, где попытки стандартизировать, привести все к общему знаменателю подавляли творческий импульс. Я работаю в компании, где используют Java, так что без подробностей. На конференции OOPSLA было хорошо видно, что когда появился язык Java, это не только замедлило и даже приостановило работу над другими объектно-ориентированными языками, но также сказалось на динамическом программировании в целом. Я считаю это большой потерей.

Но ничто не вечно. В итоге все поняли: «Так, минутку, Java имеет ряд преимуществ, из-за которых мы его используем, но у других динамических языков программирования есть и другие преимущества, так что пора вернуться к ним». Этот очевидный для меня пример, так как я сам был тому свидетелем.

В значительно большей степени я не люблю факультеты компьютерных наук, которые считают своей задачей готовить людей к работе в индустрии, причем если индустрия идет таким-то путем, то и наших студентов мы будем готовить к такому-то пути. Это как раз то, чего делать не надо. Студентов необходимо учить думать вообще, выходя за рамки, и прокладывать *другие* пути, по которым мы должны развиваться.

И это большая проблема, потому что у такого стандарта как раз есть серьезные преимущества. Если тысячи программистов работают с тысячами надежных процедур, то работа непременно будет сделана.

Есть некоторая разница между компьютерными науками в сфере производства и компьютерными науками в сфере интеллектуального развития. Так, мне нравилось работать над Lively Kernel, хотя это в каком-то смысле тривиальный проект. В нем нет ничего нового: я использую все, что уже было. Он построен на JavaScript и графике, которую можно получить в браузере. Но мне было интересно, поскольку это тоже ядро, как и Squeak. Поскольку JavaScript и графика уже есть в браузере, работы над ядром немного. Оживить графику и построить небольшую вычислительную среду.

Занимаясь чем-то подобным, небольшим, это может понять любой. Если что-то выкинуть, например язык и графику, то возникнет вопрос: где же тогда ядро? И это интересный вопрос, на мой взгляд.

Надеюсь, подобное исследование – хотя это и не совсем моя работа – может вдохновить компьютерные науки на изучение процесса создания ядра, на создание других ядер – еще более простых и унифицированных.

Точно так же, как и в математике. Математика обнаружила, что очень многое можно упростить с помощью символов. И после этого можно начинать думать о более крупных конструкциях. По крайней мере, я надеюсь.

Сейбел: Говоря о ядре, вы имеете в виду программное ядро. В чем ключевая идея Lively Kernel?

Ингаллс: Под ядром я понимаю примерно следующее: вы сводите вместе достаточно материала, который может в каком-то смысле создавать себя или другие полезные вещи. Squeak, например, действительно способен создавать себя. Lively Kernel предполагает наличие JavaScript и кое-какой графики, но в итоге она позволяет редактировать графику, и вы можете создавать новые графические штуки и редактировать программы, создавая таким образом новые программы. То есть можно в итоге создать все приложения, которые вы хотели бы встроить в браузер.

Думаю, играя в эту игру, начинаешь в конце концов скрывать ненужные уровни. Вопрос только в том, где игровое поле. В случае Squeak весь язык является частью ядра, с собственным компилятором и интерпретатором байт-кода. И у него есть собственная графическая система – BitBlit – и все, что с ней связано.

Кажется, это важнейшая часть любого ядра, но ее *можно* убрать. Можно сказать: «Предположим, у нас динамический язык; предположим, у нас есть графика». Раньше я бы подумал: «Ну и все тут». Но это не так. Остается еще кое-что: как свести графику, чтобы сделать интересную среду пользовательского интерфейса? И как перевести программы и скрипты на такой уровень, чтобы можно было их изменять?

В поисках того, что запускалось бы без установки в браузере, я был вынужден взять то, что было в самом браузере. А что в нем есть? JavaScript и графическая среда. Это был шанс отойти на шаг и сказать: «Что ж, есть ядра языка, есть графические ядра и еще один тип ядра – самостоятельное ядро среды пользовательского интерфейса».

Сейбел: И в Lively Kernel, и в Squeak, насколько я понял, немного с ними поиграв, часть этого ядра, которое не относится ни к языку, ни к графике, – это некое представление пользовательского интерфейса, которое можно программировать: все эти маленькие ручки, которыми можно управлять программно.

Ингаллс: Да.

Сейбел: Мне кажется, это запутывает. Я программирую или использую приложение? Хотелось бы, чтобы различия были более явными.

Ингаллс: Да, это палка о двух концах. И не думаю, что здесь есть простой ответ. Если говорить в целом, то прекрасно, что мы создаем ком-

пьютеры, которые позволяют любые изменения. Это и прямой доступ к памяти, и программирование. Мне важно, чтобы сохранялась живость, податливость, способность к изменениям. Если система динамичная и меняющаяся, то значительно проще ограничить ее и сказать: «Вот в этих рамках ничего изменить нельзя», – чем разработать что-то статичное и неизменное и пытаться потом это использовать.

Взгляните на современное положение в веб-программировании: все начиналось с языка разметки текста, а затем в игру вступил JavaScript и сделал картину динамичной. Было бы намного проще начинать с динамичной графики, о которой уже в те дни все знали, и затем уже фиксировать изображения и выводить их на печать при необходимости.

Сейбел: Да, проще для всех, кроме тех, кто просто хотел ввести в Сеть немного текста.

Ингаллс: Видимо, так и есть. Но проще тогда добавить сверху слой вроде HTML. Думаю, базовые системы должны быть динамичными, насколько возможно. Затем можно наложить ограничения по типу или синтаксису – то, другое, третье, чтобы ее зафиксировать. Конечно, бывают ситуации, когда люди просто пользуются системой, и тогда нужна неизменность, а не гибкость. И даже, когда до них доходит, что система гибкая, это несколько пугает. Возьмем Lively Kernel: в том виде, как сейчас, это не то, что хотел бы получить конечный пользователь. Вряд ли кому-то захочется увидеть, как окно интерфейса внезапно наклоняется на 20 градусов.

Сейбел: Или проверять код кнопки, на которую пытаешься нажать.

Ингаллс: Да-да. На самом деле, это демоверсия для тех, кто хотел бы двигаться в этом направлении. И очень простая, можно добавить уровень, который сделал бы ее полезной, чтобы она не изменялась так сильно и малопонятно. Но да, это действительно компромисс между гибкостью и универсальностью, с одной стороны, и кодифицированностью и способностью использовать программу как справочник, чтобы она делала именно то, что вы от нее ожидаете, – с другой.

Сейбел: Вы действительно полагаете, что нынешняя Lively Kernel или какая-то из ее ближайших модификаций способна стать образцом создания приложений, или это очередной мысленный эксперимент от Sun Labs, чтобы показать людям, в каком направлении думать?

Ингаллс: Конечно, это мысленный эксперимент. Он предлагает пару интересных решений, которые могут воплотиться в реальных продуктах. Он имеет возможность очень быстро делать определенные вещи. Допустим, вы хотите нарисовать красное сердце, написать на нем сообщение и заставить его биться, а потом сохранить как веб-страницу – все это можно сделать внутри броузера, не устанавливая никаких других

программ. Берете среду Lively Kernel, строите в ней эту динамическую штучку с небольшим скриптом, а среда создает новую веб-страницу и сохраняет ее по протоколу WebDAV.

Все это просто и удобно, и если скрипты тоже простые, как тайловые скрипты в eToys, думаю, многие были бы рады поиграть с такой системой. Но если подняться на пару уровней выше, там уже можно действительно чему-то научиться, пытаться строить простые динамические модели, с которыми можно будет взаимодействовать. Это что-то вроде флеш-анимации, но проще и больше похоже на программирование.

Полагаю, это неплохая среда для того, чтобы внедрить туда множество мелких динамических обучающих примеров. Десять-двадцать лет назад существовала система HyperCard, и многие преподаватели ее понимали и делали в ней разные полезные вещи. Довольно странно, что весь этот опыт не перешел в Сеть. Думаю, до сих пор имеется ниша для инструментов, которые были бы так же просты, как HyperCard, и так же быстры, как Сеть. Было бы здорово, если бы такие появились.

Сейбел: Общеизвестно, что вы принимали участие в создании не то пяти, не то семи, не то вообще не знаю скольких поколений реализаций Smalltalk. Начнем с первого Smalltalk, который вы сделали на Бейсике. У вас было несколько страниц записей Алана Кэя, которые нужно было превратить в жизнь. Что вы делали?

Ингаллс: Я просто начал набивать код. Первым делом, кажется, я проверил модель исполнения. Требовалось всего несколько базовых структур – эквиваленты стекового фрейма. И я сделал – кажется, это был массив на Бейсике, – чтобы все заработало и можно было прогнать кусок кода.

Обычно таким образом – здесь мне приходит на ум слово «макетирование» – делается только то, что нужно для создания структуры, той структуры, которую нужно интерпретировать, а затем заставить работать. Помню, первое, что мы запустили на той системе, был факториал шести. Это очень простой пример, но он задействует процессы динамического поиска и создания новых стековых фреймов. И как только это заработало, стало понятно, как все здесь будет развиваться и в чем состоят трудности.

Со временем все это улучшаешь. Затем в этом конкретном случае, уже после того как все заработало, нужно было внедрить уровень, по сути, парсер: в него вводился текст, и из него надо было получить структуру, макет которой был сделан. Потом добавлялась среда – и все малопомалу становилось понятным.

И тогда говоришь: «Так, теперь я вижу, как это работает, пора писать это на ассемблере», – примерно в этом духе. А потом вдруг осеняет:

«О, нам нужно автоматическое управление хранением данных. Как бы это устроить»? То есть одно следует за другим.

Сейбел: Бывали случаи, когда такая постепенная разработка не приводила к результату или вы понимали, что она к нему не приведет, и начинали работать в каком-то ином ключе?

Ингаллс: Ну, всегда стараешься делать все что можно, а когда не получается, отходишь в сторону и начинаешь соображать.

Если сравнивать с другими разработчиками, то я, возможно, ошибаюсь чаще всего на той стадии, когда надо заставить все заработать. Во многом это из-за того, что я настолько радуюсь, когда рождается что-то новое, что мне даже неважно, если поначалу это новое не работает. Дело в том, что как только программа оживает, она сама может рассказать, что с ней не так.

И обнаруживаешь, что да, возможно, стоило сделать управление хранением данных совершенно иначе, но действительно важные вещи, которые узнаешь по ходу работы, не имеют ничего общего с этими частными вопросами. Первые версии Smalltalk, которые я сделал, использовали для сборки мусора подсчет ссылок, хотя, наверное, следовало применить что-то иное. В результате возникли некоторые проблемы с подсчетом ссылок. Но это не играло большой роли: главное, что система родилась, стала жить и работать, и мы получили большой опыт в том, как работать с объектами, как обрабатывать числовые данные объектно-ориентированным образом, – и это действительно был прогресс.

Сейбел: Не знаю, насколько вы тут отличаетесь от большинства, по крайней мере от тех, у кого я брал интервью для этой книги. Хотя, Дон Кнут действительно написал TeX карандашом в блокноте за шесть месяцев, прежде чем начал набивать строки кода, притом он еще утверждал, что сэкономил время, поскольку ему не пришлось писать тесты для всего разрабатываемого кода, потому что он написал все оптом.

Ингаллс: Верю. Действительно, кто-то работает совершенно иначе, каждый по-своему. Я догадываюсь, что порой трачу время зря в том или ином отношении. Но такова уж архетипическая специфика исследовательского программирования: если быстрее получаешь среду, в которой можно узнать что-то новое, то в конце концов вполне может оказаться, что некоторые из твоих изначальных целей не имеют значения, а вон та штука гораздо важнее, и сосредоточиться в итоге надо именно на ней.

Возвращаясь к необходимости размышлений о том, как заставить программу работать правильно, не могу не вспомнить, что пару раз я так и делал. Первый пример, который приходит на ум, – это BitBlit. Когда я решил сделать то, что в итоге воплотилось в BitBlit, было несколько

вариантов, и потому пришлось сидеть и размышлять пару ночей. Как более эффективно перенести все эти биты по границам битов через границы слов? В этом случае ни одна из существующих в мире альтернатив для меня не годилась. Я думал-думал – и в итоге придумал простую модель. До того никто этого не применял, но я посмотрел на все случаи, где мы использовали похожие операции – графические изображения, текстовые дисплеи, прокрутку, – и придумал, что нужно делать.

Сейбел: Может быть, вы в двух словах расскажете, какую проблему главным образом должен был решить BitBlit?

Ингаллс: Проблему несоответствия между необходимостью думать о дисплее как о пиксельном экране размером 1000×1000 и тем фактом, что память организована пословно. Если вы хотите взять четыре бита и перенести их *туда*, они могут быть в другой части слова, чем та, в которую вы их переносите. Собственно, эти биты могут располагаться в разных словах. Если вы пытаетесь переместить биты на экране, бывает так, что вы берете куски двух разных слов *здесь* и помещаете их *туда*. А когда вы их уже расположили, нужно сохранить и целое слово. То есть ваша задача – вставить биты в то, что здесь было до того, и все вокруг замаскировать. Полный бардак.

Далее, у нас есть растровый экран – построчный экран, два измерения. BitBlit умеет работать с ситуацией, когда источник и цель имеют разное число слов на сканирующую строку.

Это было проблемой, пока не появилась четкая идея того, что нужно сделать. И именно здесь требовалось очень обобщенное ядро, потому что если бы эта идея удалась, можно было бы не просто переносить куски из одного места в другое, но и осуществлять перекрывающую прокрутку. Заодно можно было и смешивать пикселы. Все это давало возможности для обобщения.

Я протестировал и убедился, что эта идея работает, сначала на Smalltalk, затем на ассемблере, а потом перегнал все в микрокод для Alto. В итоге мы могли проводить эти операции на полной скорости памяти, без задержек из-за этой мерзкой маскировки и переносов, поскольку все это можно было спрятать за временем цикла памяти.

Появление микропрограммных компьютеров было отличной мотивацией, поскольку стало ясно, что если создать самое небольшое ядро, которое будет делать то, что требуется, его можно записать в микрокод, и все заработает быстрее. Так что я всегда был настроен делать именно так.

То, что я придумал, пришло ко мне скорее в виде образа, чем чего-то другого: это было похоже на колесо. Если посмотреть на источник, цель и границы слова, то кажется, что колесо забирает целые слова *здесь* и переносит их *туда*, так что требуется только один перенос, – вот та-

кая картина мне и явилась. Теперь оставалось только переложить это на язык кода.

Таким образом, в центре операции BitBlt находится один длинный процесс сдвига, который забирает слова в источнике и переносит их к цели. Над этим уже следовало подумать. Но как только понимаешь, что можно таким образом хранить константы, становится ясно, что так можно размещать текст, брать образы символа из шрифта и помещать их в любое место экрана.

Сейбел: Вернемся к варианту Smalltalk на Бейсике: получается, это был предтеча Smalltalk, даже еще до Smalltalk-72?

Ингаллс: Именно так. Как только он заработал, я тут же начал делать практически полную версию на языке ассемблера, который у меня был на Nova. Мы использовали ее для отладки, а параллельно создавался компьютер Alto. Как только он стал доступен, мы переключились на него. Так и появился Smalltalk-72.

Сейбел: Итак, Smalltalk-72 был написан на ассемблере – когда же, в таком случае, он стал самодостаточным? Часто можно услышать, что лучшее в Smalltalk – то, что многое в нем написано на нем самом.

Ингаллс: Это было уже намного позже. Smalltalk-72 содержал большое количество кода на ассемблере. Как, собственно говоря, и Smalltalk-76. Основная разница между Smalltalk-72 и Smalltalk-76 заключалась в том, что я предложил идею движка байт-кода для Smalltalk, у которого был синтаксис на основе ключевых слов и который компилировался. Так что классы и даже стековые фреймы были реальными объектами – это к вашему замечанию о самодостаточности.

Сейбел: Когда вы пришли к идее написать интерпретатор байт-кода?

Ингаллс: Меня захватила такая идея: синтаксический анализ (парсинг) Smalltalk-72 выполняется на лету, и как минимум по двум причинам нам нужно было уметь компилировать нечто с той же семантикой, но не требующее моментального парсинга.

И я предложил синтаксис Smalltalk-76, который во многом был близок синтаксису Smalltalk-80. Затем встал другой вопрос: как компилировать, чтобы эффективно заработало с новым синтаксисом? Впрочем, единственным моментом, с которым возникли сложности, были так называемые удаленные вычисления – переменные, которые вы объявляете в одном месте, вычисляются в другом. Так в Smalltalk появились блоки – эквиваленты замыканий в других системах.

Сейбел: А почему просто не компилировать в машинный код?

Ингаллс: Мы по-прежнему экономили память, и наш вариант выходил исключительно компактным по сравнению с любым другим. И он дол-

жен был быть компактным, потому что запускать это все равно предстояло на Alto с 96 Кбайт памяти. Потом вышли более объемные разновидности – 128 Кбайт. То есть была важна компактность.

Сейбел: Значит, порожденный код был меньше, потому что байт-код был богаче, чем машинные инструкции?

Ингаллс: Да. Я был в восторге от идеи, навеянной работой Питера Дойча над движком байт-кода для Лиспа. Это взаимодействие очень повлияло на меня: появилось еще одно ядро, которое могло поместиться в микрокод. С самого начала работы я старался уместить ее в микрокоде для Alto.

Сейбел: Микрокод был в памяти, так что вы могли бы внедрить туда ядро Smalltalk, а затем переключиться на Лисп и встроить туда интерпретатор байт-кода на Лиспе.

Ингаллс: Да.

Сейбел: Что стало следующим шагом?

Ингаллс: Smalltalk-76 унаследовал тот же графический багаж: много специального кода для отрисовки линий, вывода текста и так далее. Но в то же время я уже сделал BitBlt, так что я переписал ядро, чтобы вся графика использовалась только BitBlt и Smalltalk, в итоге ядро стало значительно меньше. Это был Smalltalk-78 – первый, который мы запустили на микропроцессоре – на 8086.

Но это по-прежнему не был Smalltalk на Smalltalk. Smalltalk на Smalltalk не существовал до появления Squeak. Для Smalltalk-80 были спецификации виртуальной машины – они были опубликованы в виде книги, но все реализации были на Си или на ассемблере.

Сейбел: А компилятор?

Ингаллс: Компилятор был написан на Smalltalk. Собственно, когда мы писали книги о Smalltalk-80, мы с Дэйвом Робсоном – в основном, впрочем, он – написали на Smalltalk эмуляцию интерпретатора байт-кода. Это должно было стать частью Smalltalk-80: мы хотели помочь людям создать собственные виртуальные машины. Мы обнаружили, что очень полезно знать, какие именно байт-коды и в каком порядке выполняются при первом запуске системы.

Итак, он написал эмулятор на Smalltalk, потому что наш Smalltalk становился все быстрее, так что вполне подходил для этой цели, и создал все необходимые инструменты слежения, которые помогали пользователям при отладке.

Сейбел: Итак, вы решили помочь людям создавать собственные виртуальные машины, потому что Smalltalk-80 был задуман как спасатель-

ный круг, и Smalltalk мог выйти в свет, даже если бы PARC решила от него отказаться?

Ингаллс: Именно так. Потом я ушел из индустрии, а когда вернулся, решил сделать Smalltalk для нового проекта. В то время все работало быстро: «Стоп, а почему бы не запустить версию этого на Smalltalk и не посмотреть, к чему это приведет?» Но главное «Ага!» заключалось в том, что нетрудно будет механически транслировать это в Си, и оно будет работать так же быстро, как и другие движки. Если вы хотели что-то изменить в виртуальной машине, можно было изменить это прямо в Smalltalk, нажать кнопку, и все сразу оказалось бы в интерпретаторе.

Сейбел: Итак, вы взяли интерпретатор Smalltalk, написанный на каком-то подмножестве Smalltalk, и сделали специальный компилятор, который умеет компилировать это подмножество в Си?

Ингаллс: А транслятор Си был просто частью компилятора Smalltalk – с его помощью печатались синтаксические деревья. Это мы, собственно, уже делали в Xerox – Тед Кэглер написал на Smalltalk виртуальную память, а потом мы использовали тот же прием для ее перевода на BCPL. То же самое.

Сейбел: Когда Smalltalk-80 вышел в свет, уже существовали компании, работающие со Smalltalk, объекты были модным увлечением, журнал «Byte» посвятил Smalltalk целый выпуск. Предполагалось, что объекты станут повторно используемыми компонентами: программисты будут заходить в «Магазин поддержанных объектов», покупать нужные и встраивать их в свою программу. Получилось ли так?

Ингаллс: И да и нет, думаю.

Сейбел: Что же произошло в итоге?

Ингаллс: Посмотрите на мир Java – там так все и есть. Есть большие массивы программ, которые хорошо работают вместе благодаря соответствующим интерфейсам. Думаю, это был значительный шаг вперед. В Smalltalk было несколько идей, которые более или менее укоренились в мире. Среди них – объектно-ориентированный дизайн и интерфейсы. Еще упомяну динамические языки и пользовательские интерфейсы. Они не победили, в истории можно найти случаи, когда можно было все сделать по-другому и притом лучше. Но не думаю, что из-за этого многое можно было потерять или приобрести. Мир движется вперед медленно. Победили другие принципы – значит, они были лучше. Обо всем заботится естественный отбор.

Сейбел: Но в результате естественного отбора могут появиться и довольно причудливые формы.

Ингаллс: Да, например Beta и VHS. Но ничто действительно хорошее не теряется.

Сейбел: Еще один аспект Smalltalk, что в последние годы постоянно подчеркивает Алан Кэй, – это язык не об объектах, а о передаче сообщений. C++ и Java не могут похвастаться такой передачей сообщений, как Smalltalk. Почему эта идея столь важна?

Ингаллс: Потому что она дает подлинную независимость. Последнее высказывание Алана – мне оно тоже нравится – гласит, что язык должен во многом напоминать Интернет. Мы всегда заботимся о безопасности и придумывает самые разные механизмы безопасности в программах, с которыми постоянно могут случаться всякие нехорошие вещи. Но независимость по типу Интернета – это реальный и наилучший вариант.

Итак, чем же так хороша пересылка сообщений? Дело вот в чем: она обеспечивает полное разделение внешнего и внутреннего, на 100%. По крайней мере, если ее сделать правильно. Есть и другие системы, которые в этой области продвинулись дальше, и думаю, что в этом направлении мы еще увидим многое.

Сейбел: Итак, ничто хорошее не забывается. Есть ли другие идеи из Smalltalk или откуда-то еще, которые, по вашему мнению, стоило бы взять на вооружение всем?

Ингаллс: Я вообще-то не думаю обо всех: у меня есть то, что я хотел бы сделать, или то, работу над чем хорошо бы облегчить. Единственное мое желание по этому поводу, с точки зрения компьютерных наук, состоит в том, чтобы люди вернулись к изначальным принципам, дабы хоть как-то удержать вычислительную технику и специалистов в интеллектуальном пространстве.

Мы добились огромного прогресса в программировании систем и в знании языков. Но что если бы мы были столь же хороши в логическом программировании? И смогли бы его лучше интегрировать? Думаю, в этом случае мы бы смогли сделать значительно больше в области, относящейся к взаимодействию с человеком. Сейчас же все движется в направлении искусственного интеллекта. Вы знаете, что в один прекрасный момент мы уже перейдем черту, за которой компьютеры будут думать лучше, чем мы сами.

Порой мне кажется, что мы сами бессознательно сдерживаем этот прогресс. До 1980 года в этой области было сделано многое. А компьютеры сейчас на порядки мощнее, умнее и быстрее. На моем последнем компьютере можно запустить музыкальный синтезатор на Smalltalk, способный рассчитать радиосигнал для радиостанции. Это трудно понять тому, кто еще помнит трудности простых арифметических вычислений.

Если сравнить нынешние достижения с возможностями в логическом программировании, в системах на основе продукционных правил, в искусственном интеллекте, то надо сказать, что тут еще есть куда идти. Я бы хотел проследить тот тип мышления, который привел в конце концов к Lively Kernel, – что такое ядро в отрыве от языка и пользовательского интерфейса? Какие еще ядра бывают? Что будет, если построить ядро на основе логического программирования, и что на основе этого можно сделать? Не думаю, что все, кто вокруг этого крутится, работают вхолостую. Господи, да с теми машинами, которые у нас есть сейчас, даже самое маленькое открытие ведет к совершенно невероятным вещам!

Сейбел: Smalltalk изначально планировался как образовательная платформа?

Ингаллс: Предполагалось, что это будет язык для детей – ребят любого возраста, если точно процитировать Алана. Думаю, всему проекту – и притом долгосрочному проекту, – среди прочего, помогло то, что мы не ставили задачу создать лучшую в мире среду для программирования. Мы собирались создать программное обеспечение для обучения, поэтому ориентировались главным образом на простоту и моделирование реального мира.

Эта образовательная цель побудила нас также сделать низкоуровневую часть насколько возможно проще. Поэтому там было множество вещей, которые работали не так быстро, как, возможно, вам бы этого хотелось. Первая система Smalltalk-72 была очень медленной; ее вторая редакция работала в 20–25 раз быстрее. Но у нас, слава богу, все заработало, мы использовали ее при работе с детьми и многому научились еще до того, как приступили ко второй версии.

Мы уделили серьезное внимание работе растровой графики и музыки, а также их сведению в относительно простом языке. Мы многому научились в ходе работы, и в итоге получился действительно хороший язык. После Smalltalk-72 мы сделали Smalltalk-76, который фактически представлял собой Smalltalk-80. И я понял, что это может быть серьезной программной средой для всей индустрии. У нас даже возник некий конфликт с Аланом, поскольку он не хотел углубляться в этом направлении.

Спустя некоторое время он ушел из Xerox, и наши пути разошлись. Это произошло, потому что мы кое-что выяснили. Например, время на внесение изменений в систему теперь исчислялось секундами, а вскоре должно было составить доли секунды. Система была жива и полностью функционировала. А это то, что мне, как и многим другим, очень нравится. Давайте строить такие же живые системы. Именно такой системой стал Smalltalk. Затем возникла новая цель, что породило

Smalltalk-80. Squeak был в какой-то мере возвращением к той же концепции, но с добавлением возможности писать его на нем самом.

Сейбел: Итак, вы с Кэем пошли по разным дорогам. Разочаровались ли вы в изначальной концепции Smalltalk?

Ингаллс: Нет, ни в коем случае. Я уже говорил, что по образованию я физик, и в моей натуре, думаю, заложена привычка глядеть на мир и думать, как он работает, каковы действующие в нем силы, как движутся планеты, как дуют ветры и все такое, задавать вопросы об этом и ощущать тесную связь со всеми явлениями. В физическом мире, во всяком случае, это просто. Можно войти в явление с нулевым пониманием и выйти, полностью представляя себе, как оно работает.

В компьютерах, полагаю, происходит то же самое. В вычислительной среде вы можете ничего не понимать в музыке и ее синтезе, но при этом понимать, как все работает. Это должно быть доступно. То же самое с графикой. Очень похожий вариант. У вас есть какие-то крошечные вещи – различные графические эффекты, есть какие-то более структурированные вещи, вы сводите это вместе. То же и с математическими вычислениями.

Когда я ввожу в Smalltalk новичка, я обычно говорю: «Что вас интересует здесь: разбор текста? Работа с числами? Графика? Или работа с музыкой?» И потом мы начинаем углубляться в материал. Это во многом по-прежнему моя страсть – уверен, и Алана тоже: мы любим погружаться с новичками в том направлении, которое им интересно, а выныривают они оттуда уже с такими идеями, которые Алан называет мощными, – теми «Ага!», позволяющими оценить поразительное разнообразие, которое на деле получается из пары-тройки простых, самых общих работающих элементов.

Это же можно видеть и в музыке, и в графике, и в численном анализе, и в операциях с текстом. И мне очень нравится делать это доступным.

Squeak – это среда для ученого, компьютерного специалиста. eToys – среда для детей, но, к сожалению, не столь понятная, как могла бы быть. Я до сих пор считаю, что мы ее не доработали, не добавив элемент, который мог позволить чисто интуитивно погрузиться внутрь и вынырнуть с физическим пониманием этих впечатляющих идей.

Я до сих пор ношусь с этим так же, как и со всем остальным. Почему я продолжаю заниматься подобными вещами – JavaScript и браузером? Мы уже очень близки к тому, чтобы поместить материал вроде Squeak на веб-страницу, которую можно просматривать любым браузером и с которой можно взаимодействовать каким-нибудь удобным и понятным образом. Это часть общей картины. Уверен, вскоре она изменится. Браузеры должны измениться. Мы возьмем другие языки, не только

JavaScript. Я по-прежнему поддерживаю контакт с Аланом и его группой, которые предпринимают новую попытку зайти дальше и решить некоторые из текущих проблем более серьезно. Но подход при этом остается совершенно тем же самым.

Сейбел: Вы упоминаете четыре дисциплины: музыка, графика, математика и текст. Они стары почти как само человечество. Безусловно, у них есть свои впечатляющие идеи вне зависимости от всяких компьютеров: компьютер просто обеспечивает способ их исследовать, что без него могло бы быть сложнее. Есть ли также интересные идеи, внутренне присущие компьютерам? Является ли программирование или компьютерные науки еще одним, пятым фактором, в котором мы можем разобраться только сейчас, уже владея компьютерами?

Ингаллс: Да, собственно, к этому я и подбираюсь. Вот план, который всегда казался мне наилучшим: начиная в одной области с настроем далеко в ней продвинуться, вы переходите затем в другую, менее вам знакомую, и то же самое делаете там. Урок здесь в том, что когда вы переходите к более простым и глубоким структурам, порождающим любую отрасль, то обнаруживаете, что в принципе все они сходны.

Вот алгебра графики: примитивные объекты, суперпозиция переносов, вращений. Или музыка: ноты, временные интервалы, аккорды – то же самое. И это, полагаю, то же самое, что знать, как дует ветер и как вращаются планеты. Это приглашение к исследованию и пониманию того, как все происходит, того, что составляет алгебру – процессов и примитивных объектов. Да, пятый фактор, как вы его называете, существует и взаимосвязан со всеми предыдущими.

Сейбел: Как вы считаете, появятся ли те, кто, преуспев в трех или четырех первых отраслях, затем выучатся благодаря этому программировать? Или это может случиться, только если их интерес пойдет по определенному пути?

Ингаллс: Думаю, все бывает, и это тоже. Тем более что вы оттачиваете их мыслительные способности, знакомя их с предметом и заражая энтузиазмом. Но одни хотят программировать, а другие нет. Например, мой 12-летний сын, по-моему, хочет только делать повороты на 540 градусов на лыжах – что ж, всему свое время и место.

Сейбел: Вернемся в специальные области: как вы тестируете свои программы?

Ингаллс: Это зависит от того, что я делаю. Я всегда стараюсь получить почти немедленную отдачу. И приступая к чему-то новому, всегда думаю, каким образом здесь можно достичь первых успехов. Каждый раз это бывает по-разному. Если бы моя жизнь была более упорядоченной и я работал бы в нормальных командах программистов, я бы полно-

стью был в курсе современных тенденций командного программирования. Но у меня программирование более самопроизвольно, что ли: например, я не могу долго сохранять концентрацию внимания. Если я считаю, что за выходные смогу заставить программу заработать, то для мира я потерян и занимаюсь только этим, игнорируя все прочее. Мне сложно обобщать идеи, разве что скажу так: есть цель, к которой вы хотите прийти, она тем или иным образом вознаградит вас, продемонстрирует, что вы на верном пути, и вы сможете уложиться в нужное время, пока вас не оторвут домашние или рабочие дела.

Сейбел: Допустим, вы начали работать с чем-то, что, по-вашему, принесет хороший результат. Первый тест – приемочный тест: вы рисуете окно на экране или делаете что-то другое? И как насчет более тонких тестов?

Ингаллс: Если вы только что сделали возможным что-то взять, переместить, положить, вам понадобится фреймворк, чтобы с этим работать. Такой ли это фреймворк, чтобы любой другой мог прийти и увидеть, что проводились такие-то тесты? У меня обычно не так. Это своего рода привилегия моего поколения, сейчас уже такое не пройдет. Я старпер, так что никто уже меня не заставит этим заниматься. Но, думаю, то, что я делаю, во многом приближается к такому варианту. Код Squeak, например, был полон комментариев о том, как что выполнить и как что проверить. А во многих тестах BitBlit имеются такие небольшие механизмы, которые берут что-то в одном месте экрана, проводят с этим некие операции и возвращают обратно, и если на экране что-то изменилось, значит, этот механизм не работает, и это написано в комментариях. Мне кажется, это и есть тест.

Сейбел: Поговорим о совместной работе. Группа по исследованию обучения в PARC была известна своей сплоченностью. Как вы сотрудничали в работе непосредственно над кодом?

Ингаллс: Просто постоянно были на связи и периодически вносили беспорядок в чужую работу. Мы никогда не были большой группой, у каждого была своя сфера деятельности. Сегодня очень развито командное программирование, но я, честно говоря, ничего в нем не понимаю. Вот сейчас, например, в Lively Kernel часть, относящуюся к ядру, сделали мы вдвоем с еще одним сотрудником, Кшиштофом Палачем. И мы сразу договорились о разделении сфер нашей деятельности. Да, мы действительно сейчас используем репозиторий кода, и в нашей команде есть другие ребята, которые занимаются более прикладными вещами, ядром совсем немного. И я вижу, что действительно очень удобно иметь общий репозиторий, просто замечательно. Следующий для Lively Kernel шаг – это интеграция с репозиторием: в Lively Kernel можно вносить изменения, но только в работающую версию; они не попадают в репозиторий

и не сохраняются на будущее. Это следующий шаг, который мы должны сделать.

Сейбел: Занимались ли вы когда-нибудь парным программированием?

Ингаллс: Сейчас подумаю. Обычно я работал один или в составе команды, но над своей изолированной частью. Но было и много проектов, где я работал вместе с другими, мы много и напряженно отлаживали код в парах.

Сейбел: Существуют ли какие-то общие технологии управления таким сотрудничеством? Когда один постоянно указывает другому, как делать его работу, или наоборот, всегда есть шанс, что в итоге у них ничего не сойдется.

Ингаллс: Либо мы сходились на каком-нибудь интерфейсе, либо я создавал какую-нибудь структуру – незаконченную, но работающую для нашего примера, так что другим было очевидно, куда что вставлять. Или же другие что-то делали так, что уже я понимал, куда должен встать мой кусок. Обычно это бывало таким вот конкретным образом, без разных инструкций, потому что мы часто работали над такими вещами, которые не запишешь. Все определялось нуждами момента.

Сейбел: Вы работали на многих уровнях: от побитового переноса в BitBlt и написания микрокода до довольно высокоуровневого программирования на Smalltalk. Насколько программистам необходимо знать о различных уровнях программного и аппаратного обеспечения, с которыми они имеют дело?

Ингаллс: Хороший вопрос. Чтобы думать не по шаблону, нужно выйти за рамки шаблона. Если в вашем обычном подходе к языку нет того, из чего можно извлечь преимущество, надо проявить интуицию, расширить обычные рамки, показать понимание и способность к контролю.

Если говорить о создании языка, то, возможно, вы захотите работать с процессорами, которые вам не очень знакомы; в этом случае вам не надо знать о них ничего, кроме того, как работает кэширование и все такое, – это необходимо для обеспечения нормальной производительности. Думаю, надо остановиться и спросить: «Какие границы я здесь пересекаю?»

Сейбел: Оставим вопрос о том, как много нужно знать, если речь идет о программе. Одни утверждают, что начинать нужно с высокоуровневого языка, чтобы ухватить некие общие идеи. Другие – что начинать надо с ассемблера и двигаться вверх, чтобы полностью понять, что же происходит. Что из этого вам ближе?

Ингаллс: Откровенно говоря, ничего. Я учился так, и мне нравились такие-то вещи. Но, думаю, всегда будут те, кому по душе придется тот

или иной уровень. Но не думаю, что существует какой-то один путь, как нет единой дороги в искусстве.

Есть и другие вещи, столь же захватывающие, а в наши дни, возможно, еще и более подходящие, которые еще только предстоит изучить. Господи, да мы задумывались о создании искусственного интеллекта четверть века назад! Машины стали неизмеримо быстрее, а мы почти ничего не делаем в этой области, мы по-прежнему остаемся почти на уровне Фортрана. Долгое время был популярен Пролог; с логическими программами можно сделать все, что угодно. Если требуется узнать об ассемблере и о том, как он работает, нужно немного погрузиться в то, что находится за рамками шаблона; возможно, это стоит отложить на будущее.

Так что я не говорю «нет» языку ассемблера – я говорю, что необходимо изучить другие действенные технологии, так чтобы вы могли извлечь из них выгоду, когда будете думать о том, куда двигаться дальше. Если говорить о том, как начать, то для меня всегда большую роль играла возможность немедленно получить какой-то результат. Когда я хочу научить кого-то языку Smalltalk, то обычно провожу небольшой опрос: «Что вас интересует больше всего? Игра с текстом? Или то, что можно сделать с числами, с графикой, с музыкой?» И начинаю с того, что выберут.

В разбиении и сведении текста много такого, от чего можно получить настоящее удовольствие. То же относится и к числам, различным системам счисления, плавающим и фиксированным запятым. То же самое с музыкой: берете ноты и объединяете их в мелодии и аккорды. То же самое с графикой: суперпозиции и повороты. Любой вариант из этого – интересное поле для исследования. Точно так же, если собираетесь научить кого-то программировать на компьютере, то можно начать с вычисления выражений, а можно с логического программирования. Можно начать и с пользовательского интерфейса. Нужно натаскать человека в одной отрасли, откуда он будет уже продвигаться вглубь.

Сейбел: Насколько я понимаю, изначальное предназначение Smalltalk состояло в том, чтобы научить своего рода азам программирования. Как вы считаете, это то, чем должен владеть каждый, подобно тому, как каждый должен уметь читать и знать что-то из математики? Должны ли все уметь немного программировать на компьютере, просто потому что это полезный способ мышления?

Ингаллс: Честно говоря, мне сложно говорить, что кто-то *должен* что-то делать, потому что я встречал людей, которые, на мой взгляд, в том или ином смысле лучше меня, а о программировании не знают ничего. Но если говорить о грамотности, то ведь программирование основано на логике и математике, а людям действительно неплохо было бы уметь

думать логично. Но я бы не сказал, что все обязаны уметь программировать. В повседневной жизни есть области, сходные с программированием. Нужно знать о пошаговых процедурах – вот это, пожалуй, правильная формулировка.

Компьютеры несут ряд впечатляющих идей, а также могут привнести в жизнь другие впечатляющие идеи. Самое чудесное в компьютерах – это то, что они воплощают в жизнь математику. В этом отношении они прекрасный инструмент. Если же говорить о впечатляющих идеях, необходимых в обычной жизни, то я не могу сказать, сколько таких идей заключено в данной области.

Сейбел: Сеймур Паперт писал в книге «Mindstorms» (Мозговые штурмы) об отладке как о важном элементе интеллектуального арсенала: идея программирования заключается не в том, чтобы найти обязательно верный ответ, но в том, чтобы найти *какой-нибудь* ответ и затем отладить его.

Ингаллс: Вот именно! Нам нужно научиться ясно мыслить и задавать вопросы. Для меня это просто основополагающее понятие. Если вы живете в семье, в которой, если дверца шкафа не закрывается, то ее осматривают и видят, что из петли выпал винтик, – это одно, а если в такой, где в данном случае просто говорят: «О, дверца сломалась, надо кого-то вызвать», – то это другое; чувствуете разницу? Думаю, не надо быть экспертом по компьютерам, чтобы понять, когда то, что вы видите, не работает. Спрашивайте. Смотрите. И думайте, как устранить проблему, если вы с ней столкнулись. Мне кажется, что это так просто и в духе человеческой натуры, что передается из поколения в поколение.

Компьютеры – безусловно, необходимый в этом случае посредник. Но это ведь только компьютеры. Они могут передать что угодно, но мне кажется, что все основное заключено в человеке, поэтому вряд ли мы сможем просветить весь мир, просто научив его пользоваться компьютерами.

Сейбел: Вы помните первую интересную программу, написанную вами?

Ингаллс: Дайте-ка подумать. В любом моем программистском опыте было нечто нешаблонное. Так, изучая VisiCalc, я написал табличную программу для перевода с помощью VisiCalc английских слов в «поросячью латынь». На мой взгляд, это интересно, поскольку табличная метафора была сродни параллельному программированию. Такой анализ текста был и интересен, и познавателен.

Сейбел: То есть в VisiCalc имелись примитивы для разбиения строк?

Ингаллс: Да, там можно было разбивать строки. Правда, может, на самом деле у меня был Lotus 1-2-3, а не VisiCalc, потому что я что-то не уверен, были ли в VisiCalc строковые примитивы. У меня был малень-

кий Poqet PC – один из первых настоящих наладонников. Он работал от двух батареек, я поставил на него 1-2-3, и когда летел на самолете через всю страну, то думал, что бы такого сделать.

Сейбел: Вероятно, это было гораздо позже того, как вы научились программировать, потому что когда вы начинали, никаких Poqet PC явно не было.

Ингаллс: Да, позже. Самое интересное из того, что я сделал на Фортране: я взял статью Вэла Шорра о МЕТА II – это отличный, очень простой компилятор компиляторов – и написал его реализацию на Фортране. Вдруг оказалось, что можно пользоваться другими языками в чисто фортрановой среде. Это было самое интересное, что я сделал на Фортране – ведь здесь Фортран использовался для освобождения от мира Фортрана.

Сейбел: Интересно получается: таблица с «пороссячьей латынью», потом это, потом хеш-таблица на Коболе для профилировщика – можно ли сказать, что какая-то часть вашей души любит идти против течения?

Ингаллс: Мне не кажется, что я иду против течения, но каждый раз, когда у меня есть возможность поиграть в вычислительной среде, я стараюсь пробовать что-то новое. Вот почему мне так нравилось делать все эти системы на Smalltalk. Начинаешь, можно сказать, с нуля, и первая задача – понять, что нужно ассемблировать и заставлять работать в первую очередь, что поможет сделать следующий шаг и определит направление, в котором двигаться.

В этих случаях можно говорить о выходе за рамки шаблона. Это способ убедиться в том, что ты овладел материалом, в том, что ты способен на то, о чем и подумать раньше не мог.

Сейбел: Сильно ли изменились со временем ваши представления о программировании?

Ингаллс: Хороший вопрос. С одной стороны, у нас теперь гораздо больше машинных циклов. То есть я спокойно могу, грубо говоря, профукать несколько циклов, но сделать все чистенько. Но основные вещи не изменились вообще: сначала надо разобраться в том, что такое ядро или ядра, с которыми я должен работать, и какой цели я пытаюсь достичь.

Что еще слегка изменилось: теперь я не занимаюсь в своей группе квантовой механикой. Я нахожусь на более высоком уровне. Это означает, что я провожу больше времени, определяя цели и занимаясь политикой, и меньше занимаюсь кодом. Я создаю контекст. Раньше мне везло, я работал с уже существующим контекстом, мне не нужно было его создавать. Но время от времени я все же сажусь и пишу довольно серьезный код.

Сейбел: Я читал одну из ваших работ в 1970-е о профилировщике для Фортрана. В предисловии вы с восторгом говорили о том, как этот инструмент изменил ваши представления о программировании: сначала вы обдумывали, как собираетесь писать, писали и отлаживали, а теперь, обдумав, что будете писать, пишете очень простую версию, затем профилируете ее и оптимизируете. Вы по-прежнему работаете таким же образом?

Ингаллс: Я действительно сначала ищу явные эффекты – пиксели на экране или что-то в этом духе, – потому что они очень мотивируют, и нередко узнаешь что-то новое о том, что хотел сделать, просто взглянув на то, что происходит.

Если нужно профилирование, то занимаюсь им. Или может оказаться, что все делается неправильно, не совсем то, что нужно, так что приходится менять цели или образ действий. Но когда речь идет о производительности, я по-прежнему работаю, как описал. Мы сделали действительно отличный профилировщик на Smalltalk, а затем на Squeak, с помощью которого можно получить качественную обратную связь. Отчасти здесь речь о производительности, но отчасти – о структуре и архитектуре. Можно обнаружить, что некоторые элементы почти совсем не используются, так что от них можно отказаться, сделав что-то иначе. Это просто другая точка зрения.

Сейбел: Наверное, почти у каждого программиста есть книга Кнута «The Art of Computer Programming»¹. У одних она просто стоит на полке, другие используют ее постоянно. А кое-кто вообще прочитал ее от корки до корки и знает наизусть. Собственно, вы-то общались с Кнутом в Стэнфорде. Вы читали книгу?

Ингаллс: Мне нравилось работать с Доном, один семестр я посещал его курс по MIX в Стэнфорде, и это было очень полезно. Думаю, мы с Доном серьезно отличаемся друг от друга, но что мне в нем нравится, так это то, что у него отличный математический ум, при этом он любит и глубоко погружаться в мелочи, то есть в прагматическую сторону вещей. Я тоже люблю прагматизм, но мне недостает строгости подхода Дона.

Я по образованию физик, и для меня проблемы, над которыми я работаю, выглядят физическими задачами. Говоря о других перспективах программ, я представляю их себе как вещи, которые можно потрогать – и почувствовать ответную вибрацию.

Возьмем то, как он работал над TeX: все это было математически выверено, красиво и элегантно. Сравним с этим, например, первые движки Smalltalk: они были сделаны почти как попало. Я просто свел вместе все, что было нужно для конкретной цели. Возможно, этому предшест-

¹ Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008 г.

вовало несколько неудачных опытов. Изначально я хотел сделать математическую модель, может, мы ее даже и сделали, но в итоге все получилось не так, как ожидалось.

Так что, честно говоря, я прочел довольно много его основополагающих работ о структурах данных, но я вообще не очень люблю читать, мне бы делать. Если у меня и есть какой-то недостаток, так это то, что я лучше изобрету собственный велосипед, чем прочту нужную литературу и узнаю, как его делать. Обычно это оказывалось моим преимуществом, но кто знает?

Сейбел: Как вы считаете, в каком объеме программистам необходима математика? Дейкстра заявил, что компьютерные науки – просто отрасль математики. А понимание «Искусства программирования» требует недюжинного знания математики.

Ингаллс: Нужно иметь логический склад ума. Но я провел много времени в Виргинии, в сельской местности, когда учился работать с компьютерами. Я всегда думал, что если бы захотел основать компьютерную компанию в горах Виргинии, то знал бы, что делать. За исключением заумных элементов, математика совсем не так важна для программирования, как логика и интуиция.

Большая часть программирования, на мой взгляд, скорее относится к архитектуре: то, как графика взаимодействует с моделями, то, как нужно кэшировать или обновлять информацию. Это не заумная математика. То есть я действительно считаю, что программирование – во многом одна из отраслей математики. На мой взгляд, компьютеры очень важны в том числе и потому, что позволяют математике быть синтетическим, а не только аналитическим искусством. То, чем я так люблю заниматься день за днем, безусловно, имеет отношение к математике, но это наука творческая, порождающая, синтетическая.

Сейбел: Вы сказали, что не очень-то любите читать. Есть ли тем не менее какие-то книги, которые вы могли бы порекомендовать?

Ингаллс: Нет. Уверен, что тут я в меньшинстве. Я и в детстве читать не любил. Иногда я читаю очень тщательно и не прерываясь, пока не закончу. Конечно, я могу назвать несколько статей, и, вероятно, некоторые из них выросли потом в книги. Одна из них – это статья Вэла Шорра о МЕТА II. Потом есть книга о Lisp 1.5. Есть еще APL, но книга Айверсона, по-моему, не лучшее пособие для обучения. Пусть ее математики читают. Даже не помню, что я про это читал, но мне понравилось. Думаю, вместо чтения книги можно потратить время на какой-нибудь язык. Например, на Smalltalk.

Сейбел: Вы все так же любите программировать, как и тогда, когда начинали?

Ингаллс: Да, я люблю само программирование. Последние пару лет были интересны, потому что я оторвался от той среды, к которой так привык, то есть Smalltalk и затем Squeak, где просто замечательный инструментарий. Пришлось сделать небольшой шаг назад, работая с JavaScript в браузерах и в обычных средах разработки. В итоге иногда отладка занимала у меня больше времени, чем я привык, но общий процесс появления идеи и воплощения ее в жизнь – это то, чем я продолжаю наслаждаться.

Сейбел: Как вы считаете, программирование – удел молодых?

Ингаллс: Нет, не думаю. Самое важное – это иметь способности к исследованию того, что происходит, и ту бесконечную энергию, которой, кажется, у меня больше нет. Но я все еще люблю взять проблему, посидеть над ней и сделать так, чтобы все заработало. Вот такая аналогия: я стал учиться играть на фортепьяно сравнительно поздно. Мне говорили: «О, вам надо было раньше начать, вы так быстро все схватываете». Хотя далеко я не продвинулся, но могу сделать вывод, что неверно, будто бы в молодости обучение всегда дается проще; у них просто больше времени. Когда я тратил достаточно времени, прогресс был.

Примерно то же самое я думаю и о программировании. Когда я оглядываюсь назад, на свою молодость, то понимаю, что тогда у меня времени было сколько угодно. Работай себе и работай. Сейчас же в моей жизни происходит и многое другое, у меня есть обязанности, которые не исчерпываются программированием. Это несколько отвлекает.

Сейбел: Программирование требует времени, но не нужны ли для него также и трудолюбие, и концентрация? Говорят, что это непрерывный процесс, что если отвлекать человека каждые пятнадцать минут, то он ничего не сделает, потому что требуется много времени даже для того, чтобы начать раскладывать идеи по полочкам в голове.

Ингаллс: Это мне напоминает то, что я когда-то сказал кому-то в PARC. У меня появились другие обязанности, кроме возни со Smalltalk, но мы уже сильно продвинулись на пути создания из него действительно очень продуктивной системы. Я пошутил, что спешу улучшить среду Smalltalk, чтобы в промежутках между этой деятельностью можно было наконец заняться нормальной работой. Так что я привык к тому, что у меня есть минут 15, и в это время я успевал посидеть и придумать что-нибудь полезное.

Другая сторона проблемы – то, что вы работаете с другими. Я работаю с молодежью, и это прекрасно. Возможно, я чуть больше размышляю о целях, политике и том, как заставить все работать, а они делают то, чего мне не хватает: у них достаточно времени для действительно глубокого проникновения в суть вещей.

Сейбел: О вас говорят как о хорошем руководителе проекта. Как вам удастся возглавлять команду, чтобы люди работали успешно и эффективно?

Ингаллс: Просто я люблю то, что делаю. И мне нравится сотрудничать с другими. Они могут работать вглубь или вширь, так что несложно найти рабочие задачи для разных людей. Мне всегда нравилось работать с другими. Работа идет то лучше, то хуже – бывают разные фазы. Иногда понимаешь буквально все, что надо сделать, и нужно только сказать людям, что делать. А бывает и по-другому: не знаешь, что нужно сделать, и пытаешься это выяснить. Это два очень разных состояния.

Сейбел: Можете ли вы что-нибудь посоветовать по поводу того, как стать хорошим техническим лидером команды?

Ингаллс: Во-первых, нужно ясно дать понять, что вы пытаетесь сделать. Нужно обеспечить всем недвусмысленное представление о работе. Если вы в курсе дела, то можете сами увидеть, как собираетесь реализовать ту или иную идею, чтобы начать понимать, как она будет работать, если этим займутся несколько человек одновременно, и как все это увязать.

Иногда я работал над проектом – и мне было все понятно. Это удача, потому что когда кто-либо начинал испытывать затруднения, я мог сразу ответить, какие должны быть следующие шаги или как обойти возникшее препятствия. И люди чувствуют, когда ты действительно знаешь то, чем руководишь. Они немедленно понимают: да, этот мужик специалист. И это придает команде уверенности.

Сейбел: Бывало ли так, что слишком ясное представление о цели, наоборот, *лишало* людей мотивации: мол, раз все и так понятно, то доделывать это неинтересно?

Ингаллс: Ну, тут можно просто не вмешиваться в то, как они работают над своей частью общей задачи, и только иногда осуществлять самое общее руководство, когда требуется. Нередко это и к лучшему. Мне повезло долгое время работать со множеством сотрудников, которым я доверял. Очень важно доверять тем, с кем вместе работаешь. Еще один важный фактор – уверенность. Когда цель ясна, быть уверенным легко. Плохим управление бывает как раз тогда, когда беспокоишься, не чувствуешь себя в безопасности и боишься, что нужно постоянно за всем и за всеми следить.

Сейбел: Когда вы еще были рядовой рабочей пчелой, попадался ли вам действительно выдающийся руководитель?

Ингаллс: Лучшим боссом в моей жизни, конечно, был Алан Кэй. Я работал под его началом в Хегох, пожалуй, в ключевые годы моей жизни, и это была интересная комбинация, потому что он знал, чего хочет, но почти никогда не говорил, как я должен работать. При этом у него было

столько технической смекалки во всем, что это делало его отличным критиком. И я, и мои коллеги отличались в то время большой продуктивностью, так что, полагаю, он чувствовал, что при таком уровне прогресса ему совершенно необязательно вмешиваться. Но он прикрывал и меня, и всех нас, и у него в голове была четкая картина того, что он хочет получить на выходе.

Сейбел: Когда люди работают в группе, какой подход лучше избрать? Тот, при котором у каждого разработчика есть свой кусок системы: «Вот это мой код, не трогайте его». Или вся команда занимается всем кодом, и любой может работать на всех участках?

Ингаллс: Не знаю. Вот как мы сейчас работаем над проектом Lively Kernel: у разных людей разные участки работы, но никаких преград при этом нет. Это во многом вопрос квалификации, концентрации или целей. Пытаюсь вспомнить, как мы работали в те времена, которые были действительно успешными. Никогда не работал в больших командах, так что действительно получается так, что в большинстве случаев люди единолично трудились каждый над своим кодом.

Сейбел: Если говорить об отладке, то какую самую страшную ошибку вам когда-либо удалось найти?

Ингаллс: Она была связана со сборкой мусора. Сборка мусора – самое плохое, потому что проблема проявляется намного позже того, как что-то случилось. Исправление таких сложных ошибок напоминает мне взлом кодов. Мой отец работал в Управлении стратегических служб, они там работали в командах и главным образом просто собирали информацию, чтобы быть в курсе дела. Затем появлялось закодированное сообщение с фрагментом чего-то, что они видели в новостях, и таким образом они его расшифровывали.

Примерно так же отслеживается ошибка. Когда я приступал к делу, мне помогала интуиция, чутье на то, что могло вызвать данную ошибку в данной ситуации. В тот раз, к примеру, я думал по меньшей мере сутки. Когда у меня наконец все получилось, я был просто в восторге, и мой сын, которому тогда было, кажется, четыре года, вручил мне «Приз великого отладчика».

Сейбел: Это было, полагаю, на Smalltalk. У вас был символический отладчик или вы смотрели шестнадцатеричные дампы памяти?

Ингаллс: Это был отладчик более низкого уровня, чем тот прекрасный отладчик Smalltalk. Деталей сейчас не приведу, но если находишь ошибку, она приводит к низкоуровневым отладчикам. То есть смотришь на память как на восьмеричные значения. И потом видишь, что один объект указывает на другой, а этого быть никак не может. И начинаешь думать: «Как такое могло случиться?» Там были закономерности, были

догадки по поводу того, что именно могло вызвать ту или иную неполадку, так что на их основе и нужно было работать.

Сейбел: Это было на самом низком уровне. Но работая в такой замечательной среде, как Smalltalk, вы, скорее всего, используете символические отладчики. Приходится ли вам прибегать к выводу на печать?

Ингаллс: Честно говоря, не знаю никого, кто бы так делал, имея возможность пользоваться хорошим отладчиком. Потому что куда вы помещаете `print`? *Туда-то*. А не лучше ли самому быть *там-то* и посмотреть все, а не распечатывать? Сейчас я довольно часто применяю отладку с помощью вывода на печать, но это потому, что нередко у меня нет под рукой хорошего отладчика для JavaScript.

Сейбел: Чем так хорош отладчик для Smalltalk?

Ингаллс: Можно остановиться на любом месте программы и посмотреть на все связывания всех переменных. Можно исполнять фрагменты кода и вычислять выражения прямо в контексте.

Сейбел: В любом месте стекового фрейма?

Ингаллс: Да, и далее можно внести существенные изменения и продолжить. Затем можно вернуться к ошибке, вывести ее на экран, сохранить полностью состояние системы, перебросить его кому-нибудь другому, кто сидит за компьютером с Windows, а не на Make; он может запустить тот же самый образ, попасть туда же, где сейчас находитесь вы, внести исправление и продолжить. Полное сохранение состояния, несмотря на различные варианты операционных систем.

Сейбел: Инварианты – это другой тип инструментов отладки. Многие очень озабочены формальными входными и выходными условиями всех их методов и инвариантами классов. Некоторые относятся к этим вопросам проще. Каково ваше мнение относительно инвариантов?

Ингаллс: Вероятно, я принадлежу к менее формальному лагерю. Во многом из-за того, что считаю, что вещи должны быть насколько возможно простыми. То же самое я думаю относительно типов. Типы по сути являются предположениями по поводу программы. И я думаю, что очень полезно делать все максимально простым, даже не заикаясь о каких-либо типах. Думаю, если серьезно подходить к системе, то неплохо иметь возможность добавлять подобные вещи: например, можно ввести подразумеваемые типы и видеть их только при желании.

Но типы – только одна вещь из многих, к которым относятся блоки и все другие варианты операторов утверждений, которые вы можете внедрить. Это часть той поразительно интересной отрасли, которую нам еще только предстоит изучить с помощью синтетической математики. Думаю, с течением времени мы сможем все лучше понимать

вычислительные сферы и документировать их с помощью живой документационной системы, настоящей программной документации. То есть существуют такие операторы утверждений, которые могут действительно помочь вам, притом что вы их обычно не видите. Но если у вас затруднения, то можно запустить их и тестировать с их помощью самые различные вещи.

Сейбел: Что вы думаете по поводу формального доказательства корректности программ?

Ингаллс: Я никогда этим не занимался. Я больше склонен к тому, чтобы сосредоточиться на архитектуре – так проще делать утверждения для разных вещей. Если внутри программы можно делать самые разные опасные вещи, то искать формальное доказательство оказывается очень трудно, так как на каждом шагу приходится говорить: «Это могло случиться, вот это могло случиться, да и вот это тоже». Если архитектура хорошая, то и формальное доказательство может быть столь же очевидным: достаточно прочесть код. Просто говоришь: «Так, это может произойти только из-за того-то. Мы в безопасности».

Сейбел: Вы когда-нибудь работали с C++?

Ингаллс: Нет. И с Си – тоже.

Сейбел: Но вы программировали на BCPL и ассемблере, то есть нельзя сказать, что низкоуровневым программированием вы не занимались вовсе.

Ингаллс: Да. Кстати, мы действительно немного писали на Си для отладки некоторых вещей, сгенерированных с помощью Squeak. Но я помню, что когда мы делали Squeak, то моей целью, среди всего прочего, было создать такую систему, для управления которой не нужно было бы знать ничего, *кроме* Squeak. Так что я решил даже ничего другого не учить. Джон Малоуни сделал транслятор с Squeak на Си в качестве практического примера. По правде говоря, я мог посмотреть код на Си, но сделал так, чтобы ничего этого делать было не нужно.

Сейбел: Но вы наверняка смотрели C++, когда он вышел, поскольку вы входили в группу, которая – возможно, наряду с изобретателями Симулы – может с наибольшими основаниями заявлять о том, что придумала объектно-ориентированное программирование.

Ингаллс: Я им особенно не интересовался. Конечно, во многих отношениях это был шаг вперед по сравнению с Си, но все же этот язык не оправдал тех надежд, которые мы на него возлагали. Если бы нужно было проводить еще одно внедрение с нуля, мы бы, возможно, использовали вместо машинного кода C++. Я знаю пару человек – настоящих мастеров C++, и мне нравится смотреть, как они с ним работают, пото-

му что, думаю, они не полагаются на него при работе с теми областями, в которых он не особенно хорош, часто используют его как язык мета-программирования.

Сейбел: Поговорим о чтении кода. Как вы работаете с новым куском кода?

Ингаллс: Мне сложно дать какой-то общий ответ. Когда начинаешь, то как-то сразу понимаешь, что вот это делает или должно делать. Думаю, я просто читаю сверху вниз и пытаюсь понять, каковы отдельные элементы и как они работают вместе. Смотрю, какие классы и методы определены и что они делают. Дальше все зависит от того, почему мы смотрим этот код. Возможно, это что-то новенькое, о нем хочется узнать побольше. Или тут проблемы с производительностью, тогда нужно профилировать и анализировать.

Сейбел: Мы уже упоминали интервью Кнута. Еще одна его любовь – литературное программирование. Занимались ли вы им когда-нибудь или, может быть, читали такой код?

Ингаллс: Мне нравится так работать, когда у меня достаточно времени. Когда я только начинаю писать, никаких комментариев не делаю. Когда все начинает работать, я пишу комментарии. Если мне нравится то, что я сделал, или мне кажется, что разобраться с этим будет сложно, я пишу более подробные комментарии. Но мне не нравится идея комментировать все подряд. И еще мне кажется, что чем лучше язык, тем меньше нужны комментарии. Лучше использовать разумные имена переменных. Вот почему мне нравятся именованные параметры в Smalltalk. Они действительно существенно облегчают читаемость. Есть еще такая отличная штука, которую можно использовать в различных местах JavaScript. Хотя это несколько расточительно, в JavaScript применяется запись объекта в фигурных скобках, так что можно использовать ключевые слова, и они действительно похожи на ключевые слова Smalltalk, потому что заканчиваются двоеточиями; поэтому можно применять объекты в фигурных скобках для передачи нескольких аргументов. Так получаются и впрямь очень симпатичные программы.

Сейбел: Гм. Выглядит одновременно и симпатично, и отвратительно.

Ингаллс: Именно.

Сейбел: Удалось ли вам кого-то еще убедить перенять этот стиль?

Ингаллс: Да я узнал, что так работают другие, еще до того, как начал делать это сам.

Сейбел: Кем вы себя считаете: инженером, ученым, художником или ремесленником?

Ингаллс: Всем сразу. Думаю, мое образование в области физики пошло мне на пользу. Я часто сталкиваюсь с проблемами, которые очень напоминают физические, – с проблемами приложения сил к телу. Это то, что вы используете по отношению к системе: смотрите, как на нее можно воздействовать. И очень похоже на физику то, что я испытываю по отношению к пространственным объектам: то, как они работают, что есть общего в разнородных вещах и как из них можно сделать лучшую архитектуру.

Помню одну из своих первых лекций о Smalltalk. Я сказал тогда: «То, чем занимается наша группа, напоминает научный метод: вы проводите наблюдения, выдвигаете теорию для объяснения результатов и ставите эксперимент для проверки». Именно так мы и работали со всеми поколениями Smalltalk. У нас была теория относительно того, как все это работает. Мы строили систему, она как-то работала. Мы использовали ее некоторое время, а потом выясняли: «Так, лучше бы мы сделали вот это, это и то по-другому», – и создавали новую систему. Все двигалось по спирали, как и должен развиваться научный поиск и прогресс.

Я чувствую себя художником, когда работаю, потому что в моей голове есть идея, которую я хочу воплотить. Полагаю, скульптор испытывает то же самое, вдыхая жизнь в кусок материала.

В данном контексте понятия инженер и ремесленник – практически идентичны. Инженер трудится на ниве технологий. Иногда я себя так и ощущаю, но это совсем особые случаи – когда я делаю что-то низкоуровневое. Помнится, я работал над самыми глубокими частями BitBlт или движком байт-кода Smalltalk – и это была действительно ремесленная работа. У меня была возможность несколько раз сделать то же самое, пока все не стало работать оптимально, а это ремесло.

Сейбел: Разница между инженером и ремесленником, на мой взгляд, заключается в том, что инженер – это тот, кто говорит: «Здесь как в мостостроении. Мосты не падают. В мостостроении есть повторяемый инженерный процесс». Ремесленник же говорит: «Нет, здесь больше от работы по дереву. Каждое дерево уникально, есть правила его обработки, но ни один метод не гарантирует результата».

Ингаллс: Тогда в этом отношении я скорее не инженер. Думаю, методы моей работы над различными системами различны. Я знаю, что есть люди, которые делают серьезные программные системы для корпораций. Это не мой путь, мне это неинтересно. Из четырех предложенных вами вариантов инженер, вероятно, на последнем месте, затем следует ремесленник, а лидирует забавное сочетание художника и ученого.

Сейбел: Вы сказали, что на время покинули индустрию, а потом вернулись. Устали от компьютеров или просто были изменения в жизни?

Ингаллс: В жизни были другие приоритеты. Кроме того, отдых от компьютеров пошел на пользу, потому что, вернувшись, я заметил, что почти ничего не изменилось, кроме того, что все теперь стало в сотни раз быстрее.

Сейбел: Есть ли у вас рекомендации для тех, кто хотел бы стать программистом?

Ингаллс: Думаю, прежде всего надо пройти хороший курс компьютерных наук. Я бы организовал его так: сначала нужно выучить несколько разных языков, у которых разные сильные стороны. Так, Smalltalk имеет множество преимуществ, но не является универсальным ответом. Есть логическое программирование. Есть функциональное программирование. Smalltalk можно использовать в функциональном стиле, эта сторона в нем отлично развита. Но помните, что я рассказывал про Lotus 1-2-3 и перевод с английского на «пороссячью латынь»? Думаю, полезно было бы взять несколько разных вычислительных сред и одну задачу и попытаться решить ее в каждой из них. Так проявится сила данного языка – или же это побудит вас от него отказаться.

Сейбел: Как вы считаете, сильно ли изменилось программирование, а следовательно, и те люди, которые могут в нем преуспеть? Можно ли быть отличным программистом, работая на высоком уровне, но без знания ассемблера, или Си, или даже тех алгоритмов, которые можно почерпнуть из чтения Кнута, поскольку сейчас мы используем высокоуровневые языки, у которых эти алгоритмы есть в библиотеках?

Ингаллс: У разных людей разные уровни, на которых они могут работать, чувствуя себя профессионалами. Думаю, в наши дни программист может с полной уверенностью пользоваться готовыми библиотеками, не программируя их сам. Это значит, что он просто работает на другом уровне. Честное слово, я вовсе не представляю, что кто-то, желая работать с графикой, станет писать BitBlt. Не обязательно делать элемент И-НЕ: можно просто использовать язык ассемблера. Думаю, работать можно на любом уровне. Если вы чувствуете потребность, то можете забраться глубже, а если вы полны энтузиазма, вы захотите забраться глубже.

Сейбел: Как вы считаете, нужно ли современным программистам довольно высокого уровня, работающим в различных средах, учить ассемблер и микрокод? Или набор навыков, помогающий стать успешным программистом, изменился?

Ингаллс: И да, и нет. В значительной степени он тот же самый, что, надеюсь, с течением времени будет только сильнее проявляться. Но сейчас есть такие сферы, где все исчерпывается работой по заданной формуле, и другие, которые имеют дело с примитивными вещами.

Сам я физик, друзья мои – математики, и я никогда не считал, что у меня такой же мозг, как у них. Но и у меня, и у них получались хорошие вещи. Думаю, в случае с компьютерами то же самое. Те, кто работает над доказательством программ, отличаются от специалистов по графическим системам. Поэтому нужно просто найти свои сильные стороны и то, над чем вы хотите работать и над чем не хотели бы. Это вопрос как природы, так и воспитания, и он всегда будет таковым.

Возможно, некоторые из этих систем имеют достаточно уровней и частей, чтобы отдельному человеку было комфортнее работать над одной из них, чем над другими, и он будет приносить на своем месте больше пользы. Думаю, все здесь однотипно. Есть логическое, есть структурное мышление. Есть еще человеческий фактор и творческий импульс. Конкретный человек имеет определенное сочетание природных и образовательных факторов, и, на мой взгляд, здесь мало что изменилось. Сейчас мы пытаемся работать больше и лучше, но мне кажется, что во многом все осталось по-прежнему.

Сейбел: В связи с этим вот какой вопрос: все больше и больше областей зависят от компьютеров, все по-разному, и поэтому некоторые хотели бы найти способ научить программировать «непрограммистов». Как вы считаете, будет ли так? Или специалисты другой отрасли, например биологи, всегда должны будут объединяться с программистами, чтобы те создавали ПО, которое будет решать их задачи?

Ингаллс: Думаю, такое сотрудничество будет всегда, просто потому, что биолога не интересует программирование. Ему нужно выяснить то-то и то-то. И есть такой человек, который понимает, как эта штука, которой ученый занимается, будет выглядеть на компьютерах, и этот человек может ему помочь. По-моему, такая вещь, которая позволяет программировать непрограммисту, называется приложением.

Сейбел: Я работал над проектом, призванным предоставить программную среду для биологов, притом им было нужно, чтобы каждый раз программа соответствовала их случаю. Нельзя создать приложение один раз и этим довольствоваться, потому что биологи не вполне знают, что им нужно, пока не придут к определенным биологическим данным и не скажут: «Я хочу знать X », – а единственный способ найти X – это, разумеется, написать программу.

Ингаллс: Да, было бы прекрасно, если бы мы могли создать какую-то компьютерную среду со всей вашей информацией, и вы могли бы сами получить то, что нужно, каким-нибудь очевидным способом. Но я думаю, будут люди, которые этим интересуются, и люди, которые этим не интересуются.

Сейбел: Есть ли что-то, что вы считаете важным, а я вас об этом не спросил?

Ингаллс: Нередко, читая об известных людях, я задаю себе вопрос: как им удалось обустроить свою жизнь? Как они относятся к вещам, которые не составляют предмет их интереса, как у них обстоят дела с семьей, с финансами, как они сочетают все эти факторы? Или просто залегают на дно и посылают все остальное к черту, пока не закончат с любимой работой?

Сейбел: Случалось ли вам забросить все остальные стороны жизни ради страсти к программированию?

Ингаллс: Да, порой так бывало, потому что я концентрировался на работе, и мне было необходимо поддерживать эту концентрацию. Это обычный риск для любого человека, который любит то, что делает. Либо учишься как-то умерять свой энтузиазм, либо договариваешься с близкими, что сейчас занят и освободишься где-то через неделю, а до того папу лучше не дергать.

Сейбел: Но в итоге-то вы получили от сына «Приз великого отладчика».

Ингаллс: Именно так. К тому же больше удовольствия получаешь от возвращения к друзьям и семье, которые чувствуют, что папа делает что-то хорошее и нужное, и все будут счастливы, когда он это доделает.

11

Питер Дойч

Питер Дойч – вундеркинд, начал заниматься программированием в конце 1950-х, когда ему было всего 11 лет, после того как его отец принес домой заметку о программировании проектных расчетов для Кембриджского ускорителя электронов в Гарварде. Вскоре после этого он оказался в Массачусетском технологическом институте, где работал над реализацией языка Лисп на компьютере PDP-1, разбирая и улучшая код, написанный программистами MIT, которые были практически вдвое старше его.

Учась на втором курсе в Беркли, Дойч принимал участие в проекте Genie, в рамках которого была создана одна из первых систем разделения времени на основе мини-ЭВМ. Дойч написал большую часть ядра операционной системы. (Кен Томпсон, автор операционной системы UNIX, которому посвящена глава 12, также принимал участие в работе над этим проектом, будучи старшекурсником Беркли. Эта работа нашла отражение в его деятельности, связанной с разработкой UNIX.) После неудачной попытки вывести систему Genie на рынок Дойч перешел в лабораторию Xerox PARC, где работал над средой Interlisp и создавал виртуальную машину Smalltalk, оказав существенное влияние на разработку технологии динамической компиляции (JIT).

Он отвечал за исследовательские работы в ParcPlace, подразделении PARC, и получил звание Fellow¹ в Sun Microsystems. Именно во время работы в Sun он написал знаменитые «Fallacies of Distributed Computing» (Заблуждения о распределенных вычислениях). Кроме того, он является автором Ghostscript, программного комплекта, позволяющего интерпретировать язык Postscript. В 1992 году он входил в группу программистов, получивших премию Ассоциации вычислительной техники (ACM) в номинации «Разработка ПО» за разработку Interlisp, а в 1994 году был избран почетным членом ACM.

В 2002 году Дойч отошел от разработки Ghostscript и стал вплотную заниматься музыкой. Теперь его можно застать скорее за созданием музыкального произведения, чем программы, но он по-прежнему периодически позволяет себе поиграть с кодом, в основном, собственного музыкального редактора.

В ходе интервью мы среди прочего обсудили серьезные проблемы, актуальные, по мнению Дойча, для любого языка программирования, который включает понятие указателя или ссылки; поговорили о том, почему ПО должно рассматриваться как основной актив, а не статья расхода, и почему он окончательно перестал заниматься программированием на профессиональном уровне.

Сейбел: Как вы начали заниматься программированием?

Дойч: Программированием я стал заниматься по чистой случайности в 11 лет. Мой отец принес домой какую-то заметку о Кембриджском ускорителе электронов, который в то время строили. Существовала некая группа, которая занималась проектными расчетами, и их заметка оказалась у моего отца. Я увидел ее у него в кабинете – там был какой-то компьютерный код, и что-то в нем было такое, что захватило мое воображение.

Оказалось, что эта заметка была лишь дополнением к другой заметке, поэтому я попросил отца найти эту заметку. Когда он принес ее домой, я взглянул на нее и сказал: «Ого, а ведь это на самом деле интересно». Наверное, я даже попросил отца познакомить меня с человеком, который написал эту заметку. Мы встретились. Уже не помню подробно

¹ Fellow – высшее звание в технической карьере. Обычно сочетается с постом вице-президента или исполнительного директора компании. – *Прим. ред.*

стей, это было 50 лет назад. Так или иначе, мне было поручено написать какую-то небольшую часть кода для одного из проектных расчетов для Кембриджского ускорителя электронов. Вот так я и начал.

Сейбел: Вам тогда было 11 лет. В 14 или 15 лет вы уже занимались с PDP-1 в MIT, в котором ваш отец был профессором.

Дойч: В 14 лет я добрался сначала до TX-0, а вскоре после этого и до PDP-1. Помнится, однажды ко мне попал экземпляр руководства для программистов на Лиспе версии 1.5. Не помню, как именно это произошло. Это была одна из самых первых версий – она была напечатана на mimeографе старыми фиолетовыми чернилами. Что-то в Лиспе захватило мое воображение. Я всегда любил математику, и Лисп показался мне очень клевым. Я хотел поработать с ним, но никак не мог добраться до мейнфрейма из здания #. Поэтому создал свою версию Лиспа на PDP-1.

Сейбел: Вы помните хотя бы приблизительно, как разрабатывали собственную версию Лиспа на PDP-1?

Дойч: Я улыбаюсь, поскольку это была очень небольшая программа. Вы видели ее листинг? Там всего около пары сотен строк кода на ассемблере.

Сейбел: Я видел его, но даже не пытался понять. То есть вы просто перевели написанное в руководстве по версии 1.5 в программу на ассемблере?

Дойч: Нет-нет. Все, что я взял из руководства по версии 1.5, – это интерпретатор. Мне нужно было написать считывание и токенайзер, разработать структуры данных и все такое. Насколько помню, все это я сделал так же, как делал большую часть своих программ – сначала занялся структурами данных. Когда я был молод, интуиция подсказывала мне – не все время, но практически всегда – действительно верные и удачные подходы к написанию программ.

В последние годы я замечаяю, что стал уже не тот – интуиция не работает так здорово, как раньше. Я уже несколько лет занимаюсь (нерегулярно, конечно) крупным проектом по созданию хорошего музыкального редактора с открытым исходным кодом и понимаю, что привычный метод работы – позволить интуиции привести меня к подходящей организации структуры данных, и уже потом просто написать код – больше не работает.

Сейбел: Вы считаете, что все дело именно в том, что ваша интуиция стала хуже работать, или, может быть, вы раньше прикладывали больше сил, чтобы все заработало так, как надо, даже если интуиция вас подводила?

Дойч: Думаю, что истина где-то посередине, хотя первое утверждение все-таки более верно. Как мне представляется, интуиция – это бессознательный процесс поиска решения на огромном поле исходной информации. И чем меньше я погружен в работу над ПО, тем меньше в моем распоряжении исходной информации, которая используется при поиске решений.

Я где-то слышал мнение, согласно которому профессионал в любом деле должен иметь в своем распоряжении около 20 000 конкретных примеров. И дело в том, что 20 000 конкретных примеров из области разработки ПО, которые прошли перед моими глазами за 45 лет работы в этой индустрии, постепенно выветриваются из моей памяти, как это всегда бывает. Мне кажется, по большому счету дело именно в этом.

Сейбел: Вы помните, что именно привлекло вас в программировании?

Дойч: Имея возможность окинуть ретроспективным взглядом 50 лет, вижу, что меня всегда интересовали системы денотативных символов – языки. И не только «словесные» (человеческие) языки, но и языки, в которых все высказывания имеют определенные результаты. Языки программирования идеально соответствуют этому требованию.

Мне кажется, этим же в какой-то мере объясняется и то, почему я сейчас занимаюсь именно сочинением музыки. Музыка – это язык, или семья языков, но у высказываний на этих языках есть не только значения, они еще и воздействуют определенным образом на людей. Музыка представляет для меня интерес, поскольку по степени формализации она находится между естественными языками и языками программирования. Музыка более формализованна и упорядоченна в сравнении с естественными языками, но до языков программирования ей в этом отношении очень далеко. Именно этим, наверное, объясняется, почему я выбрал музыку, а не поэзию. Мне кажется, что поэзия для меня недостаточно формализована.

Но если говорить коротко, то меня туда просто сразу потянуло.

Сейбел: Помните первую интересную программу, которую вы написали?

Дойч: Первой программой, которую я написал, потому что тема заинтересовала меня, стала моя вторая по счету программа. Первой программой был небольшой фрагмент кода, имевший отношение к Кембриджскому ускорителю электронов. Второй же стала программа для форматирования чисел с плавающей запятой.

Сейбел: Это достаточно сложная задача.

Дойч: Для бинарной машины – конечно. Но работая на десятичной машине, эту задачу решить вовсе не сложно. А я работал на десятичной машине.

Нужно просто двигаться по строке и решать, где поставить десятичную запятую. Нужно решить, какой формат использовать – Е или F. Но в те дни все было намного сложнее – я писал на ассемблере на машине пакетной обработки, поэтому эта задача все-таки не была столь уж тривиальной. Это была первая программа, которую я написал, потому что хотел написать.

Сейбел: В старших классах вы много времени проводили в МИТ, после чего поступили в колледж в Беркли. Вы хотели сбежать с восточного побережья?

Дойч: Что-то вроде того. Я решил, что мне будет полезно уехать куда-нибудь подальше от своих родителей. Серьезно я рассматривал три места: университеты в Рочестере, Чикаго и Беркли. Думать тут особенно было нечего: лишь в одном из этих трех городов сносная погода. Именно так я оказался в Беркли. И это событие было одним из лучших в моей жизни.

Я учился в Беркли и нашел там – достаточно быстро – проект Genie, и продолжал заниматься им до тех пор, пока... Сначала был проект Genie, затем Berkeley Computer Corporation, потом Xerox.

Сейбел: Судя по всему, именно в Беркли вы стали работать над гораздо более крупными проектами, чем разработка Лиспа на PDP-1.

Дойч: О, да. В рамках проекта Genie я работал над гораздо более крупными проектами. Для начала я написал ядро операционной системы, практически целиком. А это больше 10 000 строк.

Сейбел: Каким образом это изменение масштаба работы повлияло на процесс написания кода?

Дойч: Пытаюсь вспомнить, из чего состояло ядро. Это была достаточно небольшая программа, поэтому я мог воспринимать ее как единое целое. Очевидно, там были некие функциональные части. Помню, что у меня было ясное представление того, какие разделы программы и ключевые структуры данных могли вступать во взаимодействие друг с другом. Но на самом-то деле, черт возьми, этих самых структур данных было не так уж много. Была таблица процессов, были таблицы готовности. Были буферы ввода/вывода и было что-то, что отвечало за отслеживание виртуальной памяти. Кроме того, была таблица открытых файлов – для каждого процесса. Но описания всех системных структур данных, видимо, можно было бы уместить – в терминах структур языка Си – на

двух страницах. Поэтому, как вы понимаете, это была не очень сложная система.

Сейбел: Какой была самая большая программа, над созданием которой вы работали, и чье устройство вы помните?

Дойч: Я был основным вдохновителем трех крупных систем. В Ghostscript – не считая драйверов устройств, большинство из которых написал не я, – мною было написано порядка 50–100 тысяч строк на Си.

Что касается ParcPlace, виртуальной машины Smalltalk, то в этом проекте я работал лишь над JIT-компилятором, составлявшим лишь 20% от всего проекта. Количество написанных мною строк исчисляется четырехзначными числами, где-то 3000–5000.

Что касается реализации Interlisp – той ее части, к которой я имел отношение, – то я написал около пары тысяч строк микрокода, и возможно – сейчас я могу только гадать, – еще около 5000 строк на Лиспе. Получается, что Ghostscript – самая большая система, над которой я когда-либо работал.

Сейбел: И не считая драйверов устройств, написанных другими, вы создали его практически в одиночку.

Дойч: До конца 1999 года я написал код практически целиком – до единой строки. Сначала я принял несколько архитектурных решений, первым из которых стало решение полностью развести языковой интерпретатор и графику.

Сейбел: Язык, о котором идет речь, – Postscript?

Дойч: Именно он. То есть интерпретатор языка ничего не знал о структурах данных, которые использовались для создания графики. Они обращались к графической библиотеке, у которой был API.

Второе решение, которое я принял, заключалось в структуризации графической библиотеки с помощью интерфейса драйвера. Таким образом, графическая библиотека знала все про пиксели, про визуализацию кривых и текста, но ничего не понимала в том, каким образом пиксели были закодированы для данного конкретного устройства и каким образом они на него передавались.

Третье решение заключалось в том, что драйверы должны выполнять основные графические команды, которые вначале сводились к `draw-pixmap` и `fill-rectangle`.

То есть библиотека визуализации передавала прямоугольники и массивы пикселей драйверу. А драйвер мог либо составить полностраничное изображение, если хотел, либо мог передать их напрямую в Xlib,

GDI и так далее. Я принял эти три глобальных архитектурных решения – и это были верные решения. В этом, по большому счету, и состоял процесс создания этой системы. Мне кажется, что я придерживался следующего принципа: если есть что-то, функционирующее во множестве областей, и эти области по сути не склонны к взаимодействию друг с другом, то в этом случае лучше всего установить достаточно мощные программные ограничения.

То есть интерпретация языка и графика по сути не очень-то пересекаются и не взаимодействуют друг с другом. Процессы визуализации графики и представления пиксельных изображений взаимодействуют больше, но мне показалось хорошей идеей установить там еще и границу абстрактности.

На самом деле, интерпретатор Postscript первого уровня я написал без графики – лишь после я написал первую строку кода графики. Откройте руководство и просто пройдитесь по всем операторам, никак не связанным с графикой, – я реализовал их все до того, как начал разрабатывать графику. Мне нужно было разработать токенайзер; мне нужно было определиться с представлением всех типов данных Postscript и всего того, что, согласно руководству, интерпретатор должен делать. Мне нужно было вернуться и переделать многие из них, когда мы добрались до разработки Postscript второго уровня, в котором была функция сборки мусора. Но именно с этого я начинал.

После чего я стал разрабатывать структуры данных для интерпретатора, просто опираясь на свой опыт работы с языковыми интерпретаторами. Между моментом начала и моментом, когда я мог набрать 3 4 add equals и получить на выходе 7, прошло около трех недель. Это было очень легкой работой. Кстати говоря, среда, в которой я работал, – MS-DOS. MS-DOS с упрощенным Emacs и чьим-то компилятором Си – не помню точно, чьим.

Сейбел: Подобную работу вам приходилось много раз выполнять до этого: реализовывать интерпретатор для языка. Вы просто сели и начали писать код на Си? Или предварительно набросали в блокноте схемы структур данных?

Дойч: Эта задача казалась мне достаточно простой, поэтому я не заморачивался со схемами. Насколько сейчас помню, сначала я внимательно изучил руководство Postscript. Затем, возможно, сделал несколько заметок в блокноте, но, скорее всего, просто начал писать заголовочные файлы в Си. Поскольку, как я уже говорил, мне нравится начинать разработку программы с данных.

Затем мне пришла идея, что, наверное, должен быть файл с главным циклом интерпретатора. Каким-то образом должна была происходить

инициализация. Нужен был токенайзер. Нужен был менеджер памяти. Нужно было как-то управлять представлением файлов в Postscript. Нужно было реализовать отдельные операторы Postscript. Поэтому я разделил все это на несколько файлов, можно сказать, функционально.

Когда я озаботился проблемой получения авторских прав на код Ghostscript, мне пришлось отправить полный листинг самой первой его реализации. На тот момент уже прошло около десяти лет – мне было интересно взглянуть на первые варианты кода, структуры и названий для разных вещей. Примечательно также то, что около 70–80% структуры и принципов наименования остались теми же – спустя десять лет и после двух серьезных переработок языка Postscript.

По большому счету, этим я и занимался – в первую очередь структурами данных. Предварительным разделением на модули. Я до сих пор убежден, что если сделать правильно структуры данных и их инварианты, то большая часть кода напишется сама собой.

Сейбел: То есть говоря о написании заголовочного файла, вы имеете в виду создание сигнатуры функций или структур – или и то, и другое?

Дойч: Речь идет о структурах. Это был 1988 год, еще не было ANSI C и не было сигнатур функций. Как только компиляторы ANSI C стали более или менее нормой, я потратил два месяца, прошелся по всей программе и сделал сигнатуры для каждой функции в Ghostscript.

Сейбел: Каким образом ваши идеи о программировании или ваш подход к программированию изменились с того далекого времени?

Дойч: Они изменились очень сильно, поскольку очень сильно изменились те программы, которые мне интересны. Думаю, не погрешу против истины, если скажу, что программы, которые я писал первые пару лет, были всего лишь небольшими фрагментами кода.

Я все время размышлял над проблемами, связанными с тем, как нужно браться за программу, которая делает что-то более глобальное и интересное, как ее структурировать, как с ней работать, как нужно работать с языками, которые используешь, чтобы написать эту программу в соответствии с тобою же установленными критериями практичности, надежности, эффективности, прозрачности.

Теперь я знаю намного больше критериев, применяемых для оценки ПО. И воспринимаю эти критерии в контексте намного более масштабных и сложных программ – программ, в которых наибольшую сложность представляют архитектурные и системные задачи. Речь не о том, что в них не осталось сложностей, связанных с отдельными алгоритмами, но не это привлекает меня в них в первую очередь – и уже давно.

Сейбел: По-вашему, все программисты должны стремиться к работе на подобном уровне?

Дойч: Нет. На самом деле, я буквально на днях прочитал, что мой давний знакомый по работе в Xerox PARC Лео Гуибас только что получил достаточно престижную профессиональную премию. Он никогда не был по-настоящему системным программистом – таким, каким был я; он всегда был алгоритмовым программистом и всегда работал блестяще. Он нашел подход к определенным классам анализа и оптимизационным алгоритмам, который позволил применять их для решения множества различных задач, и создал новые инструменты для работы с этими задачами. Это прекрасная работа. К тому, чтобы работать на уровне Лео Гуибаса, программисты тоже должны стремиться.

Есть некое сходство между архитектурными принципами и теми принципами алгоритмической разработки, которые Лео и программисты его типа применяют для решения сложных оптимизационных и аналитических задач. А разница в том, что принципы работы с алгоритмическими задачами основаны на 5000–10 000-летнем опыте математической науки. В современном же программировании у нас нет подобной базы. Возможно, именно в этом одна из причин того, что сейчас так много плохого ПО; мы не понимаем толком, что делаем.

Сейбел: То есть ничего страшного, если человек, не обладающий системным мышлением, будет работать над более мелкими элементами ПО? Можно ли разделить специалистов на программистов и архитекторов? Или вы действительно хотите, чтобы любой человек, работающий над ПО, выполненным в системном стиле, был способен мыслить в масштабе целой системы, поскольку такие программы достаточно фрактальны по своей сути?

Дойч: Я не считаю, что ПО является фракталом. Было бы неплохо, если бы это действительно было так, но мне кажется, что это не так, поскольку я не считаю, что у нас есть хорошие инструменты для работы с теми явлениями, которые происходят, когда системы становятся больше. Мне кажется, что явления, которые происходят, когда системы становятся большими, качественно отличаются от тех явлений, которые происходят, когда системы из небольших становятся системами средних размеров.

Но если говорить о том, кто должен заниматься разработкой ПО, то на этот вопрос у меня нет однозначного ответа. Единственное, что я знаю, – это то, что чем сложнее программа, тем важнее, чтобы ее делали действительно хорошие программисты. Я знаю, что такая точка зрения – это элитизм, и убежден в своей правоте.

Сейчас происходит множество разных вещей – в частности, размывается граница между представлениями о том, чем ПО является и чем оно не является. Если кто-то разрабатывает веб-сайт, на котором есть хотя бы отчасти сложные моменты, касающиеся взаимодействия с пользователем или отслеживания состояния, то есть и инструменты для создания подобных сайтов. Человек, работающий с подобными инструментами – насколько я их понимаю, хотя я с ними никогда не работал, – решает некоторые из тех задач, которые стоят перед программистом, однако средства решения этих задач не очень-то похожи на написание программ.

Поэтому, пожалуй, на ваш вопрос можно ответить и так: с течением времени все большее количество задач, решение которых раньше требовало программирования, теперь уже не будет решаться с помощью программирования, и практически кто угодно сможет справляться с решением этих задач – и на хорошем уровне.

Знаете старую байку о телефоне и телефонных операторах? Дело в том, что когда телефон только входил в обиход и когда стало ясно, что популярность телефонов стремительно растет, требовалось все больше и больше телефонных операторов, поскольку тогда еще не было автоматических телефонных аппаратов. Кто-то подсчитал уровень роста и воскликнул: «Боже, через 20–30 лет абсолютно всем людям придется стать телефонными операторами». Собственно говоря, так все и произошло. Мне кажется, нечто подобное происходит сейчас и в некоторых крупных областях программирования.

Сейбел: Возможен ли исход, при котором та же история произойдет и с программистами, – что их заменят?

Дойч: Это зависит от того, какую программу нужно написать. Один из вопросов, который я себе постоянно задавал на протяжении последних пяти с лишним лет: «Почему программирование – такая сложная штука?»

Занимаясь программированием, приходится работать в том числе с алгоритмами, по сути достаточно близкими к математике, чтобы ее можно было при желании использовать как некую базовую модель для работы с алгоритмами. Можно применять математические методы и математические подходы. Это не делает программирование легким, но никто и не думает, что заниматься математикой легко. То есть существует достаточно высокий уровень совпадения между материалом, с которым вы работаете, нашим представлением об этом материале и нашим представлением об уровне мастерства, который необходим для работы с ним.

Думаю, отчасти проблема с иным методом программирования заключается в том, что мир практически всех известных нам языков про-

граммирования в корне отличается от физического мира, с которым наши чувства, наш разум и наше общество прочно связаны, и было бы безумием ожидать, что люди станут с ним хорошо справляться. С человеком что-то должно быть слегка не так, чтобы он стал хорошим программистом. Может быть, «слегка не так» – это чересчур, но качества, необходимые для того, чтобы быть полноценно функционирующим человеком, и качества, необходимые действительно хорошему программисту, – эти множества, конечно, пересекаются, но не так чтобы очень сильно. И это говорю я, человек, который когда-то был очень хорошим программистом.

Мир фон-неймановских вычислений и языков семейства Алгол настолько непохож на физический мир, что меня на самом-то деле сильно удивляет, что мы еще умудряемся создавать крупные системы, которые хоть как-то работают – даже так несовершенно, как сейчас.

Возможно, кому-то это представляется не более удивительным, чем тот факт, что мы можем строить реактивные самолеты, но реактивные самолеты существуют в физическом мире, и у нас за плечами сотни тысяч лет опыта в машиностроении. Если же говорить о ПО, то перед нами странный мир со странными, причудливыми свойствами. Свойства физического мира неразрывно связаны с субатомной физикой, и есть несколько уровней: субатомная физика, атомная физика и химия. Есть множество неожиданно обнаруживающихся свойств, обусловленных этими уровнями, и у нас есть все инструменты для успешного функционирования в этом мире.

Глядя по сторонам, я не вижу ничего, что напоминало бы мне адрес или указатель. Я вижу объекты – не те странные вещи, которые специалисты в области информационных технологий по какому-то недоразумению называют «объектами».

Сейбел: Не говоря о масштабе. 264 чего бы то ни было – это много, а когда какое-то действие происходит несколько миллиардов раз в секунду – это быстро.

Дойч: Но в реальном мире нас этот масштаб не заботит. Вы ведь знаете число Авогадро – 1023? Мы живем в мире, где одновременно сосуществует невероятное количество мелочей. Но нас это не заботит, потому что мир устроен таким образом, что нам не нужно понимать, к примеру, вот этот стол на субатомном уровне.

Физические свойства материи таковы, что в 99,9% случаев мы можем воспринимать ее в целом. Все, что нам необходимо о ней знать, мы можем узнать, воспринимая ее целиком. И по большому счету этот принцип не работает в мире ПО.

Постоянно предпринимаются поиски подходов к модуляризации ПО. Нужно отметить, что с течением времени эти попытки становятся все более успешными, но, на мой взгляд, еще не скоро будет достигнута та же легкость, с которой мы смотрим по сторонам и видим перед собой вещи – что бы это ни было, – в которых содержатся 1023 атомов, и это нас нисколько не занимает.

Разработка ПО – это наука о деталях, и это самая глубокая, самая ужасная фундаментальная проблема ПО. До тех пор пока мы не научимся понимать и организовывать ПО так, что нам не придется думать о том, каким образом каждая мельчайшая деталь взаимодействует со всеми другими мельчайшими деталями, ничего коренным образом не изменится в лучшую сторону. И нам еще очень далеко до подобных открытий.

Сейбел: То есть нужно лишь преодолеть технические проблемы или просто все дело в сущности самого процесса?

Дойч: Нужно все начать заново. Для начала нужно забыть обо всех языках, в которых существует понятие указателя, потому что в реальном мире нет такого понятия. Нужно смириться с фактом, что информация занимает пространство, существует какое-то время и размещена в определенном месте.

Сейбел: По мере того как вы переходили от работы над небольшими фрагментами кода к созданию крупных систем, вы писали эти маленькие фрагменты с помощью прежних методов и просто приобрели новый взгляд на более крупные системы или это повлияло на ваш подход ко всей работе в целом?

Дойч: Это изменило мой подход к работе в целом. Первыми значительными программами, которые я создал, стали программы на UNIVAC в Гарварде. Следующие несколько программ я сделал на PDP-1 в MIT. В то время – 1960-е, когда я учился в старших классах, – мною были созданы три действительно разные программы, или системы.

Для серийного PDP-1 я написал интерпретатор Лиспа. Я написал какой-то кусок кода операционной системы для причудливо модифицированного PDP-1 Джека Денниса. Кроме того, я написал текстовый редактор для PDP-1 Денниса.

Эти три системы я сделал в целом монолитными. Отличие от моих предыдущих программ на UNIVAC заключалось в том, что здесь мне уже пришлось начать разрабатывать структуры данных. Это было первое серьезное изменение относительно типа программирования, которым я тогда занимался.

Я начал осознавать существование того явления, которое я назову функциональным сегментированием, но тогда я не придавал ему особого зна-

чения. Я знал, что можно писать определенные части программы и не заботиться в этот момент о других частях программы, но проблемы с интерфейсами, которые приобретают гигантское значение по мере того, как программа увеличивается в объеме, насколько я помню, не представлялись мне тогда важными.

Переход случился во время работы над моим следующим крупным проектом – во время учебы на старших курсах в Беркли в рамках проекта Genie – над системой разделения времени 940 и над текстовым редактором QED. Еще я написал программу отладки для ассемблера, но почти ничего о ней не помню.

Наиболее системной частью того проекта была операционная система. Я бы погрешил против истины, если бы сказал, что написал всю операционную систему целиком, но это не так. Но я по большому счету написал все ядро на ассемблере. Сейчас речь уже идет о немного более крупных программах – возможно, порядка 10 000 на ассемблере. Там были планировщик процессов, виртуальная память, файловая система. На самом деле там было несколько файловых систем.

И здесь уже передо мной возникли более сложные проблемы в связи с организацией структур данных. Например, из того, что я помню, там была таблица активных процессов. И передо мной встал вопрос: как ее организовать и каким образом операционная система должна решать, когда процесс работоспособен, и прочее. Были и структуры для отслеживания системы виртуальной памяти. Но стали появляться и некоторые проблемы, связанные с интерфейсом. Не в рамках самой операционной системы, нет, поскольку она была настолько мала, что ядро было создано как единый кусок, монолит.

Но были две важные области, в которых стали появляться проблемы программного интерфейса. Одна из них – интерфейс между пользовательскими программами и ядром. Какими должны быть системные вызовы? Как должны быть размещены параметры? Я знаю, что в первых версиях системы 940 основные операции для чтения и записи файлов были эквиваленты вызовам `read` и `write` в UNIX, когда вы просто даете базовый адрес и смещение. Это все было очень хорошо, но большую часть времени это было не тем, что нужно. А нужен был попросту потоковый интерфейс. Но в те дни мы и понятия не имели, что можно взять функции операционной системы и затем обернуть их кодом пользовательского уровня, чтобы получился интерфейс получше – вроде того, когда `getc` и `putc` надстраиваются над `read` и `write`. То есть в более поздних версиях операционной системы мы просто добавили системные вызовы, эквивалентные `getc` и `putc`.

Другое место, в котором стали появляться проблемы, связанные с интерфейсом, – вновь на базе режима MULTICS – с самого начала мы

строго различали ядро и то, что сегодня называется оболочкой. Это был достаточно ранний этап развития операционных систем, и мы не понимали, что можно на самом-то деле создать оболочку, не обладающую никакими особыми привилегиями. Оболочка была программой, работающей в пользовательском режиме, но у нее было множество особых привилегий. Были небольшие вопросы касательно того, какие функции ядро должно было передать оболочке, – что оболочка должна была делать сама, а что – через вызовы ядра.

Принимая решения по организации интерфейсов, мы исходили из каждой отдельной задачи. Именно в этот момент карьеры ко мне начало постепенно приходить понимание того, что интерфейсы между сущностями нужно разрабатывать по отдельности, что интерфейсы между ними были действительно важной задачей для разработчика.

Поэтому ответ на ваш вопрос о том, изменился ли мой метод программирования после того, как я перестал работать с небольшими кусками кода и перешел к работе с более крупными системами, – да, изменился. По мере того как я создавал все более крупные системы, я стал замечать, что, садясь писать кусок кода, я все чаще и чаще первым делом задавался вопросами: «Каким будет интерфейс между вот этим и всем остальным?», «Что будет на входе?», «Что будет на выходе?», «Какая часть задачи будет отдаваться каждой из сторон этого интерфейса?». Подобные вопросы начали становиться все более крупной частью моей работы. И это, видимо, оказывало влияние на то, как я писал отдельные, более маленькие куски кода.

Сейбел: И это было естественным следствием работы с более крупными системами – в конце концов системы становятся настолько большими, что приходится думать, как разбить их на части.

Дойч: Именно. В этом смысле я согласен, что ПО фрактально, поскольку декомпозиция – процесс многоуровневый. Я хотел сказать, что не уверен в том, что декомпозиция на более высоких уровнях качественно не отличается от декомпозиции на более низких уровнях. Когда занимаешься декомпозицией на более низком уровне, можешь не думать, например, о выделении ресурсов; когда же занимаешься декомпозицией на более высоком уровне, от этого никуда не деться.

Сейбел: Приходилось ли вам работать с людьми, которые, на ваш взгляд, были очень хорошими программистами и тем не менее могли работать только с задачами определенного уровня? Например, они могли хорошо справляться с задачами определенного масштаба, но их склад ума не позволял им разбирать системы на составные элементы, видеть их насквозь?

Дойч: Мне приходилось работать с очень сильными программистами, которым не хватало опыта, чтобы работать на более высоком, системном уровне. Например, во время работы над Ghostscript у меня были достаточно серьезные разногласия с двумя инженерами, которые попали в мою команду, когда началась ее передача компании, которую я создал. Оба инженера были очень умными, очень трудолюбивыми, очень опытными. Мне они казались очень хорошими программистами, хорошими разработчиками. Но оказалось, что они не могли мыслить системно. Они не только не могли думать в терминах влияния или ветвления изменений – им даже в голову не приходило, что подобными вопросами вообще нужно было задаваться. На мой взгляд, одни понимают, какие вопросы нужно задавать при более масштабном подходе к разработке ПО, а другие – по какой бы то ни было причине – вообще не задаются такими вопросами.

Сейбел: Но как вам кажется, эти люди – когда они не пытаются разработать всю систему целиком – делают свою работу хорошо?

Дойч: Да. Те два инженера, о которых я сейчас говорил, приносили огромную пользу компании. Первый работал над одним крупным проектом – это была достаточно неблагодарная работа, но весьма важная с коммерческой точки зрения. А второй переделал крупные фрагменты моего графического кода, и с его кодом мы получили более приятное глазу визуальное воплощение. Они оба хорошие, умные, опытные ребята. Просто они видят не все части картины – по крайней мере, у меня осталось такое впечатление о них.

Сейбел: Есть ли у вас какие-то особенные навыки, которые, по вашему мнению, помогли вам стать хорошим программистом?

Дойч: Я отвечу вам сейчас в духе Нью-эйдж. Вообще говоря, меня нельзя назвать приверженцем Нью-эйджа, хотя и я в свое время носил длинные волосы. Будучи (по собственным оценкам) на пике своих возможностей, я обладал чрезвычайно надежной интуицией. Я просто делал какие-то вещи, и само собой получалось, что я делал их правильно. Где-то мне везло. В каких-то случаях, уверен, дело было в том, что я настолько слился со своим опытом, что мне даже не нужно было сознательно участвовать в процессе принятия решения. Но мне кажется, что у меня просто был к этому талант. Понимаю, что объяснение не очень-то убедительно, но я действительно верю, что кое-что из того, благодаря чему я стал хорош в своем деле, просто было у меня.

Сейбел: Когда вы были не по годам развитым подростком и проводили много времени в MIT, думали ли вы о ком-нибудь: «Да, этот парень очень умен, но он не знает, как делать вот эту штуку, а я знаю»?

Дойч: Нет, такого не было. Хотя, нет, помню один случай, когда я начал переписывать текстовый редактор на PDP-1 Денниса – мне, наверное, было лет 15 или 16. Исходный код был написан одним или двумя парнями из Клуба технического моделирования железной дороги. Это были умные ребята. Я смотрел на этот код и часто думал про себя: насколько же он ужасен!

Я бы не сказал, что речь о разнице между мной и теми, с кем я работал. Речь о разнице между тем, каким код должен был быть в моем представлении, и тем кодом, который я видел перед собой. Я бы не стал исходя из этого наблюдения делать какие-то обобщающие выводы о людях.

Я всегда достаточно комфортно чувствовал себя в том, что называю символическим миром, в мире символов. Символы и их сочетания – вот что я обычно ем на завтрак. Этого же не скажешь о других. Например, о моем партнере. Мы оба музыканты. Мы оба композиторы. Мы оба вокалисты. Но я рассматриваю музыку в первую очередь с точки зрения символов. Я часто сочиняю, просто сидя за столом с ручкой и листом бумаги. Я записываю ноты, но не играю их тут же на фортепиано. Я их слышу, и у меня есть план.

В то время как у него большая часть процесса сочинения музыки происходит непосредственно с гитарой в руках. Он что-то на ней играет, дурачится с ней, иногда побацает на фортепиано немного, потом играет все сначала. И никогда ничего не записывает. Может, если на него надавить, то он и запишет последовательность аккордов, и, насколько я понимаю, в какой-то момент он записывает слова. Но он подходит к процессу сочинения музыки не с символической точки зрения.

То есть кто-то устроен так, кто-то иначе. Если бы мне нужно было сделать из этого наблюдения какой-то вывод – что ж, повторюсь еще раз, я немного элитист, – я бы сказал, что программированием должны заниматься люди, которые чувствуют себя в мире символов как рыба в воде. Если вы не очень-то комфортно ощущаете себя в этом мире, что ж, может быть, программирование – это просто не ваше.

Сейбел: Были ли у вас учителя, которым вы многим обязаны?

Дойч: Их было двое. Один из них уже не с нами – его звали Кэлвин Муэрс. Он был первопроходцем в исследовании информационных систем. Кажется, это он придумал термин «информационный поиск». По образованию он был библиотеквед. Я познакомился с ним в старших классах или уже в колледже. Он тогда занимался разработкой языка программирования, который по его задумке мог бы использоваться неспециалистами. Но он ничего не знал о языках программирования. А я к тому моменту уже кое-что знал, поскольку уже успел создать Лисп-систему и изучал кое-какие другие языки программирования.

Мы сошлись, и в конце концов он сделал язык TRAC – думаю, можно сказать, что мы его создали вместе. Он в тот момент оказывал мне очень серьезную поддержку.

Другой человек, которого я всегда воспринимал как своего наставника, – это Дэнни Боброу. Мы очень долго дружили. И на протяжении своей профессиональной деятельности я всегда считал его своим наставником.

Но если говорить непосредственно о программировании, о разработке ПО, то никого такого в MIT не было. Никого, по большому счету, не было и в Беркли. В PARC был лишь один человек, который по-настоящему повлиял на то, как я разрабатывал ПО, – и то он даже не был программистом. Это был Джерри Элкин, менеджер лаборатории информационных технологий в PARC.

Вот его слова, которые значительно на меня повлияли: очень важно всегда проводить количественные оценки; будут моменты – даже больше моментов, чем сейчас кажется, – когда твои убеждения и твоя интуиция окажутся неверны, поэтому все измеряй. Иногда даже нужно измерять вещи, которые, как тебе кажется, и измерять-то не надо. Эта мысль очень сильно на меня повлияла.

Когда я хочу заняться чем-то, неизбежно включающим значительный объем вычислений или обработку значительного объема данных, помимо прочего я теперь всегда все измеряю. И это началось с того времени, когда я работал в PARC, то есть около 35 лет назад.

Сейбел: Вы единственный из тех, к кому я обращался по поводу этой книги, весьма резко отреагировали на слово «кодер» в ее названии. Как бы вы предпочли себя называть?

Дойч: Должен сказать, что сейчас у меня даже к слову «программист» слегка негативное отношение. Если вы посмотрите на процесс создания ПО, которое действительно работает, выполняет полезные функции, то увидите, что для достижения этой цели применяется множество разных ролей, процессов и навыков. Кто-то называет себя программистом, но это не очень-то много скажет вам о том, какие навыки они на самом деле используют во время своей работы.

Но, по крайней мере, слово «программист» – достаточно устоявшийся термин для описания достаточно широкого круга специалистов. Тогда как слово «кодер» ассоциируется исключительно с самой незначительной и наиболее узконаправленной частью этого огромного предприятия. Можно сказать, по отношению к процессу создания ПО, которое действительно работает и выполняет полезные функции, кодер – то же, что каменщик для процесса постройки действительно хорошего здания.

Нет ничего плохого в том, чтобы быть кодером. Как нет ничего плохого в том, чтобы быть каменщиком. Чтобы делать то и другое хорошо, требуется множество навыков. Но это лишь очень маленькая часть всего процесса.

Сейбел: Какой же обобщающий термин вас устроит? Разработчик ПО? Специалист в области компьютерных наук?

Дойч: Против термина «компьютерные науки» у меня тоже есть небольшое предубеждение. Могу очень убедительно показать вам, что слово «наука» вообще не должно применяться к программированию. На мой взгляд, по большому счету, все, что называется термином «компьютерные науки», – это сочетание машиностроения и прикладной математики. Думаю, лишь малую часть этого можно назвать наукой, если говорить о наличии истинно научного процесса, то есть когда вы получаете более качественные описания наблюдаемых явлений.

Думаю, если бы мне нужно было выбрать короткое, броское определение, я, пожалуй, остановился бы на «разработчике ПО». Этот термин учитывает практически все – от проектирования архитектуры до кодирования. Он не учитывает некоторые вещи, которые необходимо выполнить для создания ПО, которое действительно работает и выполняет полезные функции, но он описывает практически все, чем занимался я.

Сейбел: А что он не описывает?

Дойч: Он не описывает процесс понимания области задачи, а также определения и понимания требований. Он не учитывает процесс – по крайней мере, весь процесс – получения обратной связи, начиная от тестирования и заканчивая теми вещами, которые происходят после выпуска ПО. По сути, термин «разработчик ПО» описывает мир, ограниченный рамками организации, которая занимается разработкой ПО. Он практически ничего не говорит о связях между этой организацией и ее клиентами по всему миру, которые, по большому счету, и являются исходной причиной появления этого самого ПО.

Сейбел: Как вам кажется, в этой области происходят какие-то изменения? Некоторые сегодня ратуют за то, чтобы связываться с клиентом или пользователем на ранней стадии процесса разработки, и на самом деле хотят сделать это частью работы разработчика ПО.

Дойч: Да, именно этим занимается экстремальное программирование (XP). Я не являюсь большим поклонником этого подхода. Экстремальное программирование ратует за тесную связь с клиентом во время процесса разработки по двум, как мне кажется, причинам. Первая – таким образом нужды клиента можно лучше понять и выполнить. Может быть, это действительно так. Я не обладаю информацией из первых

рук, но отношусь к этому с небольшим скепсисом, поскольку клиенты не всегда знают, чего хотят.

Вторая причина, по которой, как мне кажется, экстремальное программирование стоит за подобное тесное взаимодействие с клиентом, – стремление избежать поспешных обобщений или специализаций. Полагаю, это палка о двух концах, потому что я видел, как реализовывались оба нежелательных сценария – и поспешные обобщения, и поспешные специализации.

Поэтому здесь у меня есть несколько вопросов к экстремальному программированию. Что происходит после того, как проект «завершен»? Оказывается ли техническая поддержка? Выходят ли дополнения и улучшения? Что происходит, когда уходят разработчики, сделавшие исходный вариант проекта? Поскольку экстремальное программирование панически боится всякой документации, я весьма скептически ко всему этому отношусь.

С подобной проблемой я сталкивался при общении с теми, кто очень любит быстрое прототипирование или занимается любой формой разработки ПО, не считая это занятие инженерной дисциплиной. Очень сильно сомневаюсь в том, что ПО, разработанное без применения инженерных подходов, может хоть сколько-нибудь долго проработать.

Сейбел: Вы можете привести пример неудачного обобщения или специализации из своего опыта?

Дойч: Когда я был на пике своей карьеры, одна из вещей, что удавались мне очень хорошо (не утверждаю, что всегда), – я умел выбирать верную степень универсальности, охватывающую несколько последующих лет развития в абсолютно неочевидных направлениях.

Но теперь, вспоминая, я могу назвать один пример поспешной специализации на уровне архитектуры – когда я, занимаясь Ghostscript, принял решение использовать пиксельное, а не плоскостное представление цветовой схемы. Использовать побитовое изображение и соответственно делать так, чтобы представление пиксела укладывалось в long.

Тот факт, что там использовалось точечное, а не планарное представление, означал, что возникали большие трудности, когда приходилось взаимодействовать с дополнительными цветами – при использовании специальных принтеров, в которых применяются цвета, не входящие в стандартный набор CMYK. Например, серебряный, золотой или специальные оттенки, которые должны были быть точно подобраны.

Если вы взглянете на цветное изображение, разбитое на пиксеты, то поймете, что в памяти его можно представить несколькими способами. Можно представить его в памяти как массив пикселей, где каждый

пиксел (точка изображения) содержит данные о цветах из схемы RGB или CMYK. Например, так работают обычные контроллеры дисплеев.

Другой способ, наиболее распространенный в полиграфической промышленности, – взять массив, который содержит значение красного цвета для каждого пиксела, затем другой массив, который содержит значение зеленого для каждого пиксела, затем тот, который содержит значение голубого, и так далее. Если изображения обрабатываются по-пиксельно, этот способ наименее удобен. С другой стороны, он не накладывает никаких изначальных ограничений на количество краски или пластин, которые могут быть использованы при создании того или иного изображения.

Сейбел: То есть если у вас есть печатное устройство, которое использует золотую краску, то вы просто добавляете дополнительную пластину.

Дойч: Верно. Это, как правило, не характерно для обычных пользовательских принтеров и даже для офисных принтеров. Но в офсетной печати это сравнительно общепринятая практика – использовать отдельные слои. Это был пример недостаточного обобщения.

И пример того, как, даже обладая неплохими навыками и аналитическими способностями, я просчитался. Вообще-то он не доказывает мой тезис, а наоборот, опровергает его, потому что в этом случае даже внимательные предварительные расчеты привели в итоге к недостаточно-му обобщению. И я могу указать точную причину этого недостаточно-го обобщения – дело в том, что Ghostscript создавался, по сути, одним очень смысленным парнем, который понятия не имел о полиграфии.

Сейбел: То есть вами.

Дойч: Именно. Ghostscript задумывался исключительно для предварительного просмотра файлов в Postscript, потому что тогда других таких программ не было, а PDF еще не существовал. Если мне и предстояло извлечь урок из той истории, то вот он: требования всегда меняются, они всегда как минимум предпринимают попытку измениться в направлении, о котором ты даже не подозреваешь.

Есть две философские школы, два взгляда на то, как нужно себя к этому готовить. Одна школа, которая, как мне кажется, весьма близка к воззрениям экстремального программирования, по сути говорит, что поскольку требования все равно постоянно меняются, не стоит ожидать от ПО долговечности. Если требования изменятся, вы создадите что-нибудь новое взамен. В этой мысли, на мой взгляд, есть определенная доля мудрости.

Но есть одна проблема. В бизнесе популярен такой старый афоризм: «Быстро, дешево, качественно – выбери любые два». Если вы делаете что-то быстро и знаете, как сделать это недорого, очень маловероятно,

что ваш продукт будет хорошим. Но эта философская школа говорит, что не стоит ждать от ПО долговечности.

Мне кажется, суть проблемы заключается в противоборстве двух философий – ПО как статья расхода и ПО как долгосрочный актив. Я убежденный сторонник второй философии. Когда я еще работал в ParcPlace и Адель Голдберг проповедовала объектно-ориентированный подход к разработке, мы либо говорили об объектах, либо пропагандировали объектно-ориентированные языки и объектно-ориентированный подход к разработке нашим клиентам и потенциальным клиентам, и суть сводилась к следующему: «Вы должны рассматривать ПО как долгосрочный актив».

Но нет таких долгосрочных активов, которые не требуют постоянных затрат на содержание. Необходимо ожидать, что если вы собираетесь поддерживать разрастающуюся библиотеку ПО многократного использования, то это потребует определенных расходов. И это сделает вашу бухгалтерскую отчетность более сложной, поскольку вы не можете списывать стоимость создания элемента ПО на проект или клиента, для которого необходимо создать данное ПО в данный момент. Вам нужно думать о нем как о долгосрочном активе.

Сейбел: Как о строительстве нового завода.

Дойч: Именно. Объекты большей частью продавались из-за того, что хорошо спроектированные объекты можно использовать многократно, поэтому ваши вложения в их разработку окупаются с меньшими усилиями.

Я по-прежнему убежден в этом, хотя, возможно, и не так сильно, как прежде. В наши дни вещи, создаваемые пригодными для многократного использования, – это либо очень большие вещи, либо очень маленькие. Масштаб многократного использования, о котором шла речь, когда мы продвигали объекты, – классы и группы классов. Кроме случаев, когда имеется коллекция классов, воплощающих настоящие знания в области, – подобного я сейчас не вижу.

По моим наблюдениям, многократно использовать теперь можно либо очень маленькие вещи, например отдельные значки или отдельные дизайны веб-страниц, либо очень большие вещи, например целые языки или крупные приложения с расширенными архитектурами вроде Apache или Mozilla.

Сейбел: То есть сейчас вы уже не так твердо убеждены в истинности исходной идеи о многократном использовании объектов. Что-нибудь было не так с самой теорией или она просто не заработала в силу исторических причин?

Дойч: Причина, по которой я больше не называю себя специалистом в области компьютерных наук, частично заключается в том, что я наблюдал практику разработки ПО около 50 лет и за последние 30 лет не увидел значительных изменений в лучшую сторону.

Если взять языки программирования, могу вам убедительно доказать, что они не улучшились в качественном плане за последние 40 лет. Сейчас нет языка программирования, который превосходил бы по качественным характеристикам Симулу-67. Я знаю, что это звучит достаточно смешно, но я действительно так считаю. Java не намного лучше Симулы-67.

Сейбел: A Smalltalk?

Дойч: Smalltalk немного лучше Симулы-67. Но Smalltalk в его нынешнем виде ничем не отличается от себя же образца 1976 года. Я не говорю, что языки, которые существуют сегодня, не лучше языков, которые существовали 30 лет назад. Язык, на котором я пишу все свои программы сегодня – Python, – превосходит, на мой взгляд, любой из языков, что были доступны 30 лет назад. Мне он нравится намного больше, чем Smalltalk.

Я не зря сказал о *качественных характеристиках*. В каждом распространенном в наши дни языке программирования, известном мне, используется понятие указателя. И я не представляю, каким образом можно качественно улучшить ПО, созданное с применением этого фундаментального понятия.

Сейбел: Вы относите ссылки в Python и Java к указателям?

Дойч: Безусловно. Да, у всех программ на Python или Java, не считая очень небольших программ, те же проблемы, за исключением проблем с порчей данных, как в случае Си или C++.

Суть проблемы в том, что не существует языкового механизма понимания, постулирования, управления или вывода шаблонов доступа и разделения информации в системе. Передача указателя и хранение указателя – это локализованные операции, но их последствия неявно создают целый граф. Не говоря о многопоточных приложениях, даже в однопоточных приложениях данные перемещаются между разными частями программы. Есть и ссылки, которые распространяются на другие части программы. Даже в хорошо спроектированных программах найдутся две, три или даже четыре различные сложные модели протекающих процессов и не найдется способа, которым можно было бы описать, объяснить или охарактеризовать крупные блоки, не касаясь при этом того, что происходит на нижнем уровне. Предпринималось множество попыток решить эту проблему. Но я не припомню никаких

открытий и прорывов в этой области. Думаю, в этой области нет каких-то общепринятых или общеиспользуемых решений.

Сейбел: А как насчет чисто функциональных языков – хотя они, наверное, и не являются широко используемыми?

Дойч: Да, у чистых функциональных языков совсем другие проблемы, но этот Гордиев узел они, конечно же, разрубаят.

Время от времени у меня появляется соблазн разработать язык программирования, но я ничего не делаю и жду, когда этот порыв пройдет. Но если я поддамся этому соблазну, то в моем языке будет фундаментальное разделение между функциональной частью, отвечающей исключительно за значения и в которой не будет понятия указателя, и другой областью, которая будет отвечать за схемы совместного владения, ссылки и управление.

Поскольку я занимался написанием как компиляторов, так и интерпретаторов, то могу придумать множество способов реализации языка без необходимости постоянно копировать большие массивы. Но те, кто занимается функциональными языками, понимают в этом намного больше меня. Есть множество умных людей, которые работали и работают над Haskell и подобными языками.

Сейбел: А эти ребята не придут и не скажут: «Да, именно так это и есть у нас в монадах, и это разделение у нас реализовано в системе типов»?

Дойч: Знаете, я никогда не понимал монады в Haskell. Я, наверное, перестал следить за функциональными языками после ML.

Если вы посмотрите на E – это не тот язык, о котором все знают, чтобы о нем можно было говорить, – он из тех языков, что основаны на очень четком понятии «способности». Он связан с акторными языками Хьюитта и с операционными системами, основанными на этом понятии. В нем есть порты, или коммуникационные каналы, для обеспечения фундаментальной связи между двумя объектами. Основная идея заключается в том, что ни один из участников коммуникации не знает другого участника коммуникации. То есть это очень сильно отличается от понятия указателя, который направлен в одну сторону, и где объект, держащий указатель, достаточно хорошо представляет себе, что находится на другом конце. В нем очень важную роль играет элемент непрозрачности.

У меня есть идея – предварительная, непроработанная, – согласно которой в языке должны быть функциональные вычисления и не должно быть совместного владения объектами. Должны быть своего рода сериализованные порты. Каждый раз, когда нужно обратиться к тому, что знаешь, только по ссылке, в соответствии с природой самого языка,

понимаешь: что бы это ни было, это что-то взаимодействует с многочисленными источниками коммуникации и соответственно следует ожидать, что оно должно уметь сериализовывать данные или что-нибудь в этом роде. Не должно быть понятия доступа к атрибуту и уже точно не должно быть возможности записи в атрибут.

Есть языки, в которых API непрозрачны, чтобы реализации могли иметь инварианты; но это опять же ничего не говорит о более глобальных моделях коммуникации. Например, одна распространенная модель: у тебя есть объект, ты передаешь его кому-то еще, просишь этого кого-то выполнить с этим объектом определенное действие и затем в какой-то момент просишь вернуть этот объект. Это модель совместного владения. Ты, вызывающий, можешь никогда на самом деле не отдавать все указатели на объект, который ты передал. Но ты говоришь себе, что не будешь ссылаться через этот указатель до тех пор, пока это третье лицо не выполнит те действия, о которых ты просил.

Это очень простой пример модели организации программы – если бы был способ выразить его языковыми средствами, это помогло бы людям обеспечивать соответствие кода цели, которую они для себя установили.

Возможно, самая серьезная причина, по которой я на самом деле не предпринимаю попытку создать язык, – я не уверен, что знаю, как нужно описывать модели совместного владения и коммуникации на достаточно высоком уровне и так, чтобы их можно было реализовать. Но я считаю, что именно поэтому индустрия разработки ПО так мало продвинулась за последние 30 лет.

Моя диссертация была посвящена доказательствам *корректности программы* – сейчас я этот термин больше не использую. Смысл его в том, чтобы система разработки давала как можно больше уверенности в том, что ваш код делает именно то, что вы от него хотите.

Раньше идея корректности программы заключалась в существовании неких утверждений, которые являлись выражениями того, что вы хотели от своего кода, – причем чтобы это можно было механическим способом проверить в самом коде. Этот подход был связан с множеством проблем. Сейчас я считаю, что путь к ПО, которое с большей вероятностью будет делать то, чего мы от него хотим, лежит не через использование утверждений или индуктивных утверждений, а через использование более качественных, более мощных и глубоких декларативных систем обозначения.

Джим Моррис, автор одних из самых остроумных высказываний касательно IT-индустрии, как-то сказал, что проверка типов – это первобытный способ доказательства корректности. Если и стоит ожидать прорыва в этой области, то он может произойти только тогда, когда появятся

более мощные методы декларативных высказываний о том, как наши программы должны быть организованы и что наши программы должны делать.

Сейбел: То есть, например, можно будет каким-либо образом выразить мысль «Я передаю ссылку на этот объект вот этой подсистеме, которая с ним повозится сколько-то времени, и я ничего не буду с ним делать, пока не получу его обратно»?

Дойч: Да. Когда в начале 1990-х я работал в Sun, там проводились экспериментальные исследования по созданию языка, в котором использовалось схожее понятие. И достаточно много исследований проводил в MIT Дэйв Гиффорд по созданию языка FX – в нем тоже старались сделать более очевидной разницу между функциональными и нефункциональными частями процесса вычисления и сделать более очевидным смысл передвижений указателя от одного места к другому.

Но мне все эти подходы кажутся чрезмерно узкими. Если случится прорыв, после которого уже будет либо невозможно, либо не нужно создавать чудовища вроде Windows Vista, то нам придется начать по-новому воспринимать программы – что они из себя представляют и как их нужно создавать.

Сейбел: Поэтому, несмотря на то что Python качественно не превосходит Smalltalk, вы все равно предпочитаете именно Python?

Дойч: Это так. И на это есть несколько причин. Что касается Python, то там предельно ясно, что такое программа, что значит запустить программу и что значит быть частью программы. Там есть понятие модуля, и модули объявляют, какая информация им нужна от других модулей. Поэтому там можно разработать модуль или группу модулей и давать их другим людям, и эти другие люди смогут работать и смотреть на эти модули, и они будут знать довольно точно, от чего модули зависят и в каких пределах.

Что касается Smalltalk, то там это делать неудобно – если вы работаете в Smalltalk в режиме образов, то там не бывает программы как таковой. В VisualWorks – Smalltalk от ParcPlace – есть три или четыре разных понятия того, как сделать так, чтобы вещи превосходили лишь один класс, и они изменились со временем, и они не очень-то хорошо поддерживаются инструментами разработки, по крайней мере визуально. Есть несколько инструментов, позволяющих сделать ясными, очевидными и механически проверяемыми взаимозависимости в программе. То есть работая в режиме образов, вы не можете никому ничего передать – только образ целиком.

Если вы делаете то, что называется *filing out*, то есть пишете программу в текстовой форме, у вас нет абсолютно никакого способа узнать, воз-

можно ли считать только что записанную программу еще раз, вернуть ее и сделать ту же операцию еще раз, потому что состояние образа совсем не обязательно совпадает с тем, которое было произведено или которое могло быть произведено путем считывания из набора в исходном коде. Вы могли сделать любые вещи в рабочем пространстве, у вас могли быть статические переменные, значения которых изменились с течением времени. Вы этого просто не знаете. Вы не можете уверенно выполнить ни одно действие.

Я подписан на рассылку разработчиков VisualWorks, и темы, которые там постоянно обсуждаются, – их просто нет в языках, не использующих понятие образа. Понятие образа похоже на множество других вещей, характерных для мира, в котором все стремятся быстро набросать опытную модель, быстро сделать программу. Это идеальный вариант для проектов, которые ведутся одним человеком и которые навсегда останутся только в собственном пользовании этого человека. Но это ужасный вариант, если вы хотите сделать ПО активом, если вы хотите, чтобы вашим ПО пользовались другие. Мне кажется, что именно в этом заключается настоящий минус подхода к разработке ПО, исповедуемого Smalltalk, – и минус очень серьезный.

Вторая причина, по которой мне нравится Python, – и, может быть, дело просто в том, как мой мозг изменился за эти годы, – что я больше не могу удерживать столько информации в голове, сколько раньше. Теперь для меня более важно, чтобы вся информация была перед моими глазами. Поэтому тот факт, что в Smalltalk вы фактически не можете использовать больше одного метода на экране, меня просто бесит. По мне, тот факт, что я редактирую программы, написанные на Python, с помощью Emacs, является преимуществом, поскольку я могу видеть больше 10 строк одновременно.

Я говорил с друзьями, которые до сих пор работают в VisualWorks, о том, чтобы открыть код ядра, JIT-генератора кода, который, несмотря на то, что его написал я, превосходит, по моему мнению, многие из современных аналогичных программ. Подумать только, у нас есть Smalltalk, у которого действительно великолепные инструменты для генерации кода, это проверенная технология – ей уже около 20 лет, и она очень надежна. Это сравнительно простой, сравнительно легко перенастраиваемый и достаточно эффективный JIT-генератор кода, который разработан для эффективной работы с языками без объявления типа. С другой стороны, у нас есть Python – прекрасный язык с прекрасными библиотеками и очень медленной реализацией. Было бы неплохо совместить их, не так ли?

Сейбел: Разве не эта же идея легла в основу вашего проекта ruscore – переписать Python на Smalltalk?

Дойч: Эта. Я дошел с этим проектом до того этапа, когда понял, что мне нужно будет сделать намного больше работы, чем я думал, чтобы эта затея по-настоящему удалась. Несоответствия между объектными моделями в Python и в Smalltalk были достаточно серьезными, поэтому были вещи, которые нельзя было просто отобразить один к одному, это нужно было делать с помощью дополнительных уровней вызовов методов, и еще много чего нужно было сделать.

Даже тогда Smalltalk с JIT-генерацией кода был – для кода, только что написанного на Python, – того же уровня, что и интерпретатор, написанный на Си. Я-то думал, что если бы была возможность открыть исходный код генератора кода Smalltalk, то задача совместить этот генератор кода с объектной моделью и представлением данных в Python не должна быть особенно сложной.

Но это нельзя сделать. Элиот Миранда, возможно, наиболее радикально настроенный из всех моих знакомых, связанных с VisualWorks, попытался, но Cincom заявила: «Нет, это стратегический актив, мы не можем открыть его исходный код».

Сейбел: Что ж, вы и сами говорили, что ПО нужно рассматривать как долгосрочный актив.

Дойч: Но это не должно означать, что ваша лучшая стратегия – не давать другим пользоваться этим активом.

Сейбел: Помимо того что вы еще в незапамятные времена были приверженцем Smalltalk, вы еще были и одним из первых фанатов Лиспа. Но и его вы сейчас уже не используете.

Дойч: Моя диссертация представляла из себя 600-страничную программу на Лиспе. Я преданный фанат Лиспа, начиная с PDP-1, Alto, Byte и заканчивая Interlisp. Причина, по которой я больше не программирую на Лиспе, – мне ненавистен его синтаксис. Синтаксис имеет значение, это правда жизни.

Языковые системы держатся на трех китах – язык, библиотеки и инструменты. Успешность языка зависит от сложного взаимодействия между этими тремя вещами. Python – отличный язык, у него отличные библиотеки и практически никаких инструментов.

Сейбел: Под «инструментами» вы в том числе подразумеваете и непосредственную реализацию языка?

Дойч: Конечно, можно и это сюда включить. Лисп как язык фантастически гибок, но его невозможно читать. Не знаю, как сегодня обстоят дела с библиотеками для Common Lisp, но мне кажется, что синтаксис – это очень важно.

Сейбел: Кому-то синтаксис Лиспа очень нравится, кто-то его на дух не переносит. Почему так?

Дойч: Я говорю только за себя. И могу сказать, почему я больше не хочу работать с синтаксисом Лиспа. На это есть две причины. Во-первых, и я об этом уже говорил, чем старше я становлюсь, тем для меня важнее высокая плотность информации на квадратный дюйм экрана перед моими глазами. Плотность информации на квадратный дюйм в инфиксных языках выше, чем в Лиспе.

Сейбел: Но практически все языки на самом деле префиксные, не считая небольшого количества арифметических операторов.

Дойч: Это не совсем так. Если говорить о Python, например, то это неверно для создания списков, кортежей и словарей. Это делается с помощью скобок. Построковое форматирование – инфиксное.

Сейбел: Так же, как и в Common Lisp – с помощью команды `FORMAT`.

Дойч: Ну хорошо, ладно. Но есть вещи, которые не делаются с помощью инфиксов; самые простые – циклы и условные операторы – не префиксные. Они выполняются с помощью чередующихся ключевых слов и того, к чему они применяются. В этом отношении они даже более многословны, чем Лисп. И здесь я перехожу к другой части, другой причине, по которой мне больше нравится синтаксис Python, – Лисп очень однообразен в лексическом плане.

Сейбел: Кажется, Ларри Уолл сравнил этот язык с тарелкой овсяных хлопьев, перемешанных с обрезками ногтей.

Дойч: Ну, я бы описал подобным образом Perl, только еще добавил бы, что все это вышло из собаки, причем понятно, откуда именно. На мой взгляд, силе духа Ларри Уолла можно позавидовать, если он еще позволяет себе высказывать какие-то мысли о разработке языков программирования, ведь Perl – просто отвратительный язык. Но не будем об этом.

Если есть кусок кода на Лиспе и нужно понять его значение, то необходимо сделать две вещи, которые не нужно делать в схожей ситуации при работе с языком вроде Python.

Во-первых, придется отфильтровать все эти чертовы скобки. Это не интеллектуальная работа, но понимание в вашем мозге происходит на многих уровнях, и мне кажется, первое, что он делает, – это распознавание символов. Поэтому он воспринимает все эти скобки, и затем нужно отфильтровать их уже на более высоком уровне. Таким образом, механизму распознавания символов мозга приходится выполнять дополнительную работу.

Возможно, сейчас арифметические функции в Лиспе записываются с помощью их общепринятых наименований, то есть знака плюс, знака умножения и так далее.

Сейбел: Да, именно так.

Дойч: Хорошо, вторую вещь, о которой я хотел сказать, сейчас уже делать не нужно – понимать все эти вещи с помощью распознавания токенов, а не символов, что также происходит на более высоком уровне мозга.

Есть еще третья вещь, которая может показаться незначительной, но мне она таковой не кажется. В инфиксных языках каждый оператор располагается рядом с двумя своими операндами. В префиксных языках это не так. Для того чтобы увидеть другой операнд, приходится делать больше работы. Знаете, все это кажется незначительным. Но для меня самым важным и значительным является плотность информации на квадратный дюйм.

Сейбел: Но тот факт, что базовый синтаксис Лиспа, лексический синтаксис, достаточно близок абстрактному синтаксическому дереву программы, позволяет языку поддерживать макросы. И макросы позволяют создавать синтаксические абстракции, а это лучший способ уплотнить информацию, на которую вы смотрите.

Дойч: Да, это так.

Сейбел: В моей книге о Лиспе есть глава, посвященная синтаксическому анализу двоичных файлов, и я привожу в качестве примера ID3-теги в MP3-файлах. Что здесь интересно – можно программировать так: выбираете спецификацию, в этом случае ID3, помещаете ее в скобки и делаете из нее код, который вам нужен.

Дойч: Верно.

Сейбел: То есть мое описание парсинга ID3-заголовка по сути содержит столько же токенов, сколько и спецификация ID3-заголовка.

Дойч: Примечательно, что я сделал точно то же самое, работая с Python. Мне как-то пришлось анализировать один очень сложный файловый формат – один из наиболее сложных музыкальных файловых форматов. Поэтому я написал в Python набор классов, которые отвечали и за парсинг, и за красивую картинку.

Связь между созданием класса и наименованием метода осуществляется в общем родителе. То есть это все делается в объектно-ориентированном стиле – макросы не требуются. Это выглядит не так красиво, как можно было бы сделать иным способом, но на выходе получается примерно так же читаемо, как и аналогичные макросы Лиспа. Есть вещи, которые на Лиспе делаются более гладко и внятно. С этим я не спорю.

Если вы взглянете на код Ghostscript, он целиком написан на Си. Но это Си, дополненный сотнями макросов препроцессора. В итоге, поэтому, для того чтобы написать код, который станет частью Ghostscript, нужно знать не только Си, но и то, что необходимо для создания расширенной версии этого языка. Вы можете делать подобные вещи в Си – и делаете их, когда это нужно. И подобное происходит во всех языках.

Что касается Python, то и здесь мне приходилось делать небольшие дополнения. Это не синтаксические дополнения; это классы, примеси¹ – многие из этих примесей дополняют то, что большинство людей привыкло считать семантикой языка. В Python за это отвечает один набор инструментов, в Лиспе – другой. Кому-то нравится одно, кому-то другое.

Сейбел: Что заставило вас отказаться от программирования и заняться сочинением музыки?

Дойч: По большому счету, я «перегорел» во время работы над Ghostscript. Ghostscript был одним из моих основных технических интересов с 1986 года, а где-то с 1992–1993 года был моим единственным крупным техническим проектом. Примерно в 1998 году я почувствовал, что сгораю, потому что не только делал всю техническую работу, но и занимался всей технической поддержкой, всеми административными вопросами. Это была компания из одного человека, и рано или поздно это должно было превысить мои возможности. Я нанял человека, чтобы создать компанию, и он стал нанимать инженеров.

После чего понадобилось еще два года, чтобы найти человека, который смог бы меня заменить. После чего потребовалось еще два года, чтобы по-настоящему передать ему все дела. К 2002 году я уже «наелся». Я уже не мог больше видеть Ghostscript.

Поэтому я сказал: «Ладно, отдохну полгода, посмотрю и решу, чем заняться дальше». Мне тогда было 55 лет, и я не чувствовал себя старым. Я чувствовал, что могу сделать еще один крупный проект, если захочу что-то сделать. Поэтому я стал смотреть, что же происходит вокруг.

Один проект, который меня одно время интересовал, был связан с моим давним приятелем по работе в Хероx, Джем Стротером Муром, который является – или являлся – заведующим кафедры информатики в Техасском университете в Остине. Одним из его величайших достижений стало то, что он вместе с еще одним парнем из SRI по имени Боб Бойер построил просто отпадную программу по доказыванию теорем. Вокруг этой программы он создал целую группу разработчиков, по-

¹ В английской литературе принято название *mixin*. – Прим. науч. ред.

строил большие библиотеки теорем и лемм по отдельным предметным областям.

И вот была эта небольшая преуспевающая группа, которая доказывала теоремы; я о них писал в своей диссертации, и мне всегда это было интересно. И они добились потрясающего результата, работая над арифметическим блоком центрального процессора AMD. И я подумал: «Хм, эта группа обладает множеством хороших качеств: они занимаются интересными мне вещами; ими управляет человек, с которым я знаком и который мне нравится; их технология базируется на Лиспе. Это действительно по мне».

Поэтому я отправился туда и прочитал там лекцию о том, как доказывание теорем могло бы – если, конечно, могло в принципе – помочь повысить уровень надежности Ghostscript. К тому времени у системы отслеживания ошибок Ghostscript была уже богатая история. Я нашел 20 ошибок, более или менее в произвольном порядке, взглянул на каждую из них и сказал: «Так, для того чтобы технологию доказывания теорем можно было бы с пользой применять для обнаружения или предотвращения этой проблемы, что должно еще произойти? Что здесь еще должно было бы появиться?»

Вывод, к которому я пришел, был следующим: технология доказывания теорем, скорее всего, не очень сильно бы помогла, потому что даже для того, чтобы она заработала в тех немногих местах, где могла принести пользу, нужно было бы проделать поистине гигантскую работу по формализации того, что именно должно было выполнять ПО.

Именно в этом заключается причина, по которой технология доказывания теорем в общем-то, по моему мнению, провалилась как прикладная технология по повышению надежности ПО. Просто чертовски трудно формализовать те свойства, которые вам нужны.

Я прочитал об этом лекцию и был достаточно хорошо там принят. Пообщался с несколькими выпускниками, поговорил немного с Джемом и ушел. И подумал про себя: «По пунктам все выглядело неплохо. Просто мне это не очень интересно».

Я не мог ни на что решиться. Я много лет пел в хоре. Летом 2003 года мы отправились в турне – дали шесть настоящих концертов в старых церквях в Италии. Вместе со мной ездил мой партнер, и мы решили остаться после этого турне в Европе на 2–3 недели.

Мы отправились в Вену и делали то, что люди делают в Вене. Дворец Хофбург был к тому моменту разделен на десять отдельных небольших специализированных музеев. Я прочитал в путеводителе, что там есть музей старинных музыкальных инструментов.

Я пошел в этот музей, состоящий из длинной анфилады старинных гостиных с высокими потолками. Он начинается с очень старых музыкальных инструментов – может быть, эпохи неолита – и дальше до современных. Естественно, большинство музыкальных инструментов там датированы последними двумя столетиями и сделаны в Западной Европе. Я на самом деле не все там посмотрел; я не дошел до конца одну-две гостиные, когда увидел фортепиано, принадлежавшее Леопольду Моцарту. И фортепиано, на котором репетировал Брамс. И фортепиано, которое стояло дома у Гайдна.

И тогда меня осенило: причина, по которой я не мог найти другой проект, связанный с ПО, который меня бы заинтересовал, была не в том, что я не мог найти проект. Дело было в том, что ПО меня больше не интересовало. Каким бы безумием это ни казалось сейчас, но главная причина, по которой я в первую очередь когда-то стал заниматься разработкой ПО, заключалась в том, что я думал: это поможет мне сделать мир лучше. Сейчас я в это больше не верю. Не очень-то верю. Уже не так сильно верю.

Благодаря этому небольшому озарению я внезапно понял, что для того чтобы не изменить, конечно, мир к лучшему, но дать этому миру что-то, что просуществует больше нескольких лет, я должен заниматься музыкой. Именно в этот момент я решил, что пришло время сделать глубокий вдох и закончить заниматься тем, чем я занимался 50 лет.

Сейбел: Но вы до сих пор программируете.

Дойч: Я не могу ничего с собой поделать – не могу запретить себе хотеть делать вещи прикольными и интересными. Я сделал несколько разных небольших проектов, связанных с ПО, но только двум из них я уделяю постоянное внимание на протяжении последних нескольких лет.

Первый – это технология спам-фильтра для моего почтового сервера. Я бы сказал, что это не очень прикольно, но достаточно интересно. Судя по логам, которые я время от времени просматриваю, фильтр действительно отсеивает – в зависимости от того, кто в тот или иной момент выигрывает в гонке вооружений, – от 80 до 95% входящего спама.

Другой значительный проект, связанный с ПО, к которому я постоянно возвращаюсь, – это музыкальный редактор. И делаю я это потому, что достаточно глубоко исследовал эту область. Мне приходилось пару раз работать с Finale – в гостях у одного знакомого. Полный отстой. Качество этой системы настолько низкое, что даже не знаю, как вам его описать. Я раздобыл диск с Sibelius. Нет, сначала я обзавелся ноутбуком Макинтош, чтобы запустить этот диск. И обнаружил, что пользовательский интерфейс там устроен так, что без клавиши Num Lock с этой программой практически невозможно работать. В ноутбуках Макин-

тош нет клавиши Num Lock. В их пользовательском интерфейсе были и другие моменты, которые мне не понравились. Поэтому я решил сделать собственный редактор.

Я перепробовал четыре разные архитектуры и в итоге пришел к варианту, который мне в принципе нравится. Но это был достаточно поучительный опыт. Это интерактивное приложение, достаточно большое и сложное, поэтому возникают системные проблемы, связанные с интерфейсами.

Итак, в результате я пришел к архитектуре для части, отвечающей за визуализацию программы – что, по моему мнению, является самым сложным, – и основанной на программировании уравнений. Вы определяете значения переменных в терминах уравнений, после чего позволяете реализации решать, когда их нужно вычислять. Оказывается, это не так уж сложно сделать на Python. Насколько я знаю, подобное уже делали как минимум дважды. Мне нравится этот подход, потому что при таком варианте используется меньше повторяющегося кода.

Да, я до сих пор немного программирую и получаю от этого удовольствие. Но я не делаю это для кого-то, и если я не программирую несколько недель подряд, – это нормально. Когда это было моей профессией, я всегда хотел быть в курсе всех дел проекта. Теперь же я хочу быть в курсе как минимум одной или двух композиций.

Сейбел: Вы сказали, что раньше думали, будто разрабатывая ПО, сможете изменить мир к лучшему. Как вы себе это представляли?

Дойч: Отчасти это не имело ничего общего с ПО как таковым; просто каждый раз, когда я видел что-то плохо сделанное, это меня невероятно оскорбляло, и я думал, что смогу сделать лучше. Так думают подростки. Сейчас это все кажется немного нереальным.

Конечно, в то время, когда я начал заниматься программированием, и даже в 1980-х компьютерные технологии прочно ассоциировались с миром корпораций. Я же, в соответствии со своими политическими убеждениями, был настроен против этого мира корпораций. Вид программирования, которым я всегда занимался, сегодня бы назвали личным программированием, интерактивным. Мне кажется, отчасти мной двигала мысль о том, что если каким-то образом множество людей овладеет мощностью компьютеров, это создаст определенный противовес власти корпораций.

Даже в самых смелых своих мечтах я не мог бы предсказать того, к чему придет в своем развитии Интернет. И я никогда не смог бы предсказать, до чего влияние крупных корпораций на Интернет изменит его характер с течением времени. Тогда я был склонен считать Интернет по сути

абсолютно неуправляемым, теперь же я так не думаю. Пример Китая показывает, что им можно управлять, и еще как.

И мне кажется, весьма вероятно, что если Microsoft правильно распорядится своими возможностями, то единолично завладеет Интернетом. Я уверен, что Microsoft не против такого развития событий, но мне кажется, им может хватить ума увидеть связь между тем положением, в котором они находятся сейчас, и положением, в котором у них, по сути, будет контроль над всем ПО, используемым в Интернете.

Поэтому я без большого оптимизма смотрю в будущее информационных технологий. Если быть откровенным, то это именно одна из тех причин, по которой мне было несложно уйти из этой профессии. Я видел мир, который находился в единоличном владении монополиста, не имеющего никаких представлений об этике, – и я не видел себе места в этом мире.

12

Кен Томпсон

Кен Томпсон – один из первых бородатых великих хакеров UNIX. На протяжении своей карьеры он занимался всем, что только казалось ему интересным, в том числе – в разное время – аналоговыми вычислениями, системным программированием, регулярными выражениями и компьютерными шахматами.

Он работал исследователем в проекте MULTICS Bell Labs. Когда Bell Labs решила прекратить разработку MULTICS, Томпсон вместе с Денисом Ричи продолжил работу над UNIX – за что его вполне могли уволить. Также он изобрел язык программирования Би – предшественник языка Си, созданного Денисом Ричи.

Затем он заинтересовался компьютерными шахматами и создал Belle – первый специализированный шахматный компьютер, сильнейший среди компьютерных шахматистов того времени. Он также помог довести эндшпильные шахматные таблицы до такого уровня, что они охватили все четырех- и пятифигурные окончания.

Работая над операционной системой Plan 9 компании Bell Labs, он предложил ныне повсеместно используемую кодировку Юникода UTF-8.

В 1983 году Томпсон и Ричи получили премию Тьюринга «за развитие теории порождающих операционных систем и в особенности за внедрение операционной системы UNIX». Также Томпсон был награжден Националь-

ной медалью за технологии и премией Tsutomu Kanai Института инженеров электротехники и электроники (IEEE) – оба раза за работу над UNIX.

В этом интервью он рассказал о своей давней любви к электронике, о довольно нетривиальной академической карьере, в ходе которой он еще студентом вел несколько курсов, а также о своих опасениях по поводу современного программирования.

Сейбел: Как вы научились программировать?

Томпсон: Меня всегда завораживала логика, и даже в школе я предпочитал решать арифметические примеры в двоичном исчислении и тому подобное. Мне это просто нравилось.

Сейбел: То есть так вы делали их для себя более интересными?

Томпсон: Нет-нет. Я разработал таким образом алгоритмы для сложения в разных базисах, узнал, что значит перенос, что обозначает каждый столбец и так далее. И у меня было небольшое десятичное арифметическое устройство, вроде счетов¹. Вместо каждого разряда там был ползунок от нуля до девяти. Вычитать надо было с помощью нижней колонки, а добавлять – с помощью верхней. Берешь стилус и сдвигаешь им ползунок на четыре, например, деления, при переполнении разряда переходя в соседнюю колонку. Позже, основываясь на этом варианте, я сделал двоичное устройство, а потом обобщил его до n -ичного.

Сейбел: Как вы впервые узнали о двоичной арифметике?

Томпсон: В классе, как раз когда я начал этим заниматься, нас познакомили с двоичным счислением.

Сейбел: Вы стали жертвой «новой математики»?

Томпсон: Нет-нет. Я был жертвой плохой математики. Мы переезжали каждый год, я учился в очень, очень плохих школах, а потом в хороших. Так что мне иногда приходилось за год проходить двухлетнюю программу, а потом я год ничего не делал. На математике я просто бездельничал, так что начальное математическое образование у меня просто ужасное. И вот в одном классе как раз рассказывали про двоичную арифметику. Я заинтересовался, расширил ее до любого базиса и стал с ней играть. Вот так все и началось.

Сейбел: И это было еще в школе?

¹ Арифметическая линейка – одна из модификаций счислителя Куммера. – *Прим. ред.*

Томпсон: Да, в седьмом классе. Позже, где-то в выпускном классе, я сильно увлекся электроникой – собирал радио, усилители, осцилляторы и терменвоксы. И очень заинтересовался аналоговыми вычислениями. Это и впрямь было потрясающе. Все это время электроника была моей страстью. Я поступил на соответствующий факультет в Беркли и только там на первом курсе впервые увидел настоящие цифровые компьютеры.

Сейбел: В каком году это было?

Томпсон: Я поступил рано, и тогда в году было три семестра. Поступил я в сентябре 1960 года, так что это, видимо, весна или осень 1962-го. У них был аналоговый компьютер, за которым мне нравилось работать. И была еще барабанная вычислительная машина – G15. Была одна учебная лаборатория с ними, и тогда она была всегда открыта. Там мог заниматься любой, но почти никто не хотел, так что всегда было свободно. И главным образом сидел там я один. Я писал на ней собственные программы по масштабированию – собственно, почти все аналоговое вычисление сводится к масштабированию.

Сейбел: В каком смысле «масштабирование»?

Томпсон: Масштабирование времени и амплитуды. Обычно все, что нужно было сделать, – это построить функцию. Вводишь некоторые данные, затем получаешь функцию для этих данных и связываешь результаты. И ни в один момент нельзя забираться слишком высоко, иначе наткнешься на отсечку.

Точно так же измеряешь время: делишь на два частоту в одних случаях или удваиваешь ее в других. И когда так делаешь, меняется и линейное масштабирование. Так что если есть несложная задача, при которой масштабирование не нужно, то аналог – это отличное решение. Но когда появится масштабирование, все становится очень, очень сложным. И вот я написал цифровые компьютерные программы для масштабирования установок аналогового компьютера. Без вычисления точной формы сигнала считаешь амплитуду и частоту формы сигнала в каждой точке. И таким образом понимаешь, где что-то идет не так и какая операция выполняется в данный момент.

Сейбел: А программы для цифрового компьютера были написаны на ассемблере или на Фортране?

Томпсон: В основном это был ассемблер. Интерпретируемый язык, который там был, оказался очень медленным. Поэтому пришлось перейти на ассемблер, и так я на самом деле узнал, что такое компьютер.

Сейбел: То есть загружаем программу, нажимаем на кнопку «пуск» и уходим. Это работало на перфокартах?

Томпсон: Нет-нет. Это был флексорайтер, то есть нечто вроде телетайпа или бумажной перфоленты. Записываешь на бумажную перфоленту и суешь во флексорайтер.

Сейбел: В этой лаборатории вас учили ассемблеру?

Томпсон: Нет.

Сейбел: Когда вы в следующий раз встретились с программированием?

Томпсон: На той машине G15 был установлен интерпретатор Intercom 501. И группа инженеров на нем программировала. У меня был приятель-выпускник, который написал интерпретатор для Intercom на большой машине IBM, предоставляющей лучшие вычислительные возможности во всем кампусе. Я получил листинг этого кода и однажды на каникулах, рождественских или еще каких-то, прочитал и проанализировал его. Я не знал язык, на котором он был написан, – это оказался NELIAC. И написана программа была просто отлично. И по ней я изучал программирование, NELIAC, Intercom и то, как разобраться в чем угодно. Я сидел и читал все каникулы – неделю. Потом я вернулся и стал задавать приятелю вопросы, иногда находя некоторые ошибки. Теперь я знал, как программировать, и добился в этом успехов. Потом я получил свою первую работу как программист.

Я учился, работал в лаборатории и брался за странные подработки. Я был помощником исследователя – должность мелкой сошки для студента, которая помогала написать программу для дипломной работы. И я был ассистентом, вел занятия. Программировал для компьютерного центра. Главным образом в компьютерном центре надо было сидеть в маленькой такой будке и слушать, как люди заходят и говорят: «Я изменил только в одном месте». «Ну, давайте посмотрим, что это за место и что у вас в итоге случилось».

Сейбел: Это помогло вам отточить мастерство отладки или все это было сплошной глупостью?

Томпсон: Отточило отладку только в том отношении, что после этого хорошо начинаешь понимать стандартные ошибки. Человек несколько дней корпел над своей программой, он приходит к тебе, а ты так небрежно ему: «Вот тут!»

Сейбел: По образованию вы инженер-электронщик? По компьютерным наукам в то время еще никого не готовили?

Томпсон: Нет, в то время на всей территории Соединенных Штатов компьютерные науки только зарождались, двумя путями. Теоретически – через математику и практически – через электронику. В Беркли компьютерные науки в то время преподавались в рамках курса по электроинжинирингу. Математика тоже пыталась прорваться, но у них не

было такого политического влияния, чтобы соревноваться с умудренными жизнью конкурентами.

Сейбел: Университет Беркли в итоге стал известен прежде всего такими вещами, как Системная лаборатория Беркли, то есть практикой, а не вкладом в теорию.

Томпсон: Именно так. Есть источники зарождения компьютерных наук, например Корнелл, и есть подход Беркли к компьютерным наукам. Сама атмосфера места очень значима. И я провел год в Беркли не потому, что имел какие-то амбиции. Просто мне больше нечего было делать, а это нравилось.

Сейбел: Сразу после учебы?

Томпсон: Да. Честно говоря, я работал в университете и не собирался продолжать обучение, даже не подавал заявку. Это за меня сделал один из преподавателей, который потом и сообщил мне, что я теперь в магистратуре.

Сейбел: По-прежнему по направлению электроинжиниринга?

Томпсон: Да. И мои последние годы были просто замечательными. Я не делал ничего, чего не хотел бы делать. Никаких требований, ничего такого. Чтобы нормально выпуститься, я летом прослушал курс, кажется, по американской истории – было какое-то требование по получению степени. Но за исключением этого я преподавал почти половину всех курсов, которые должен был изучать.

Общая теория вычислений в то время как раз оформлялась как самостоятельная дисциплина. Сортировка методом Шелла только что появилась, и никто не мог понять, почему она быстрее, чем n^2 . Все гоняли тесты и пытались выяснить, в чем же дело; очень просто было увидеть, что она действительно сортирует, но было непонятно, почему же, собственно, она такая быстрая. Брали асимптоту и считали, почему тут $n^{1.3}$, и так далее. А это же не натуральное число. И из всего этого – сортировки Шелла, интеллектуальной привлекательности сортировки Шелла и попытки выяснить, почему она такая быстрая, – вышла вся скорость вычислений. Разделяй и властвуй¹, все эти $n \log n$ и так далее. Поразительное, захватывающее было время.

У меня были друзья, в том числе весьма молодые преподаватели – по математике, я с ним очень дружил, и по электроинжинирингу, а также

¹ «Разделяй и властвуй» – важная парадигма разработки алгоритмов, заключающаяся в рекурсивном разбиении решаемой задачи на две и более подзадачи того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче. – *Прим. ред.*

выпускник, на которого я работал. Для меня придумывали класс, и я его обучал.

Сейбел: Вы официально изучали предмет или только по документам числились его преподавателем?

Томпсон: Нет, конечно, нигде я преподавателем не числился. Везде стоял код электроинжиниринга 199, который означал индивидуальные или групповые исследования и все такое. Они изобретали предмет, давали ему название и передавали мне. И на него приходили три-четыре студента.

Сейбел: Одним из которых на бумаге были вы.

Томпсон: Да.

Сейбел: Вам нравилось преподавать?

Томпсон: В некотором отношении. Мне приходилось преподавать в своей жизни дважды. Один раз я взял отпуск на год и преподавал в Беркли в 1975/76 гг., а еще год я читал лекции в Сиднее в 1988 г. Это очень интересно. Мне очень, очень нравится. Я занимался лабораторными исследованиями, а потом поехал в Беркли преподавать и сам изучал те предметы, которые преподавал, поскольку у меня ведь не было образования в области компьютерных наук. Обычно внештатный преподаватель ведет один предмет. У меня их было пять. Некоторые курсы я читал по два раза, и они, думаю, были наилучшими, потому что в первый год я учился, а на второй уже понял, как преподавать, и организовал свои занятия более успешно, опережая на два шага студентов в этом предмете. На третий год курс становился просто унылым. Я взялся вести один предмет третий раз подряд, и это было ошибкой. Я никогда не смог бы стать педагогом, потому что там нужно преподавать каждый раз одно и то же. Это не по мне. Но я люблю преподавание: сперва тяжело, потом весело и интересно. И затем унылый третий раз.

Сейбел: Какую самую первую интересную программу вы написали?

Томпсон: Первая длинная вычислительная программа, которую я написал, относилась к проблеме игры в пентамино. Знаете такую?

Сейбел: С кирпичиками?

Томпсон: Да, с кирпичиками. Я запустил программу на компьютере IBM 1620, который имелся на физическом факультете. Я знал, где находятся самые лучшие компьютеры, и все их запускал на ночь для своих целей. Да и в главном компьютерном центре у меня было порядка двадцати учетных записей под разными псевдонимами. Имеется 12 пентамино – это разные типы фигур из пяти кирпичиков. И таких фигур может быть 12.

Сейбел: Примерно как в тетрисе.

Томпсон: Да. Но здесь каждая фигура состоит из пяти клеток. Если их все разложить на доске, то есть две, скажем так, наиболее приятные конфигурации. Одна – прямоугольник 10 на 6, а вторая – 8 на 8 с отверстием 2 на 2 посередине. И я решил все конфигурации для этих двух досок, то есть как разместить на них фигуры. В общем случае я решал это, сравнивая шаблон досок и шаблон фигур: как фигуры вкладываются в шаблон доски. Программа не знала, что это пентамино.

Сейбел: В принципе, это был поиск методом грубой силы?

Томпсон: Так и есть.

Сейбел: Наверное, на ассемблере?

Томпсон: Видимо, так. Да, наверное, на ассемблере. Не помню, честно говоря.

Сейбел: Где-то в это время вы должны были изучить Фортран.

Томпсон: Да, конечно, в компьютерном центре мне пришлось преподавать Фортран и отлаживать программы на нем. Но я никогда на Фортране не программировал. Довольно давно я написал компилятор Фортрана для UNIX, язык Би был неудачной попыткой написать компилятор Фортрана.

Сейбел: А я думал, Би – это ваша версия BCPL.

Томпсон: Отчасти. Начиналось все, как... впрочем, не знаю как. С семантической точки зрения, однако, это оказался BCPL. А когда я начинал, то это должен был быть Фортран. И как раз тогда у меня впервые появилось описание BCPL. Мне понравилась его четкая семантика. Я отложил Фортран, и в итоге вышел синтаксис Си и семантика BCPL.

Сейбел: Есть ли большие различия в том, как вы занимались программированием и теоретизировали по его поводу в начале карьеры и сегодня? Считаете ли вы, что программирование в каком-то смысле повзросло или что вы повысили класс или узнали что-то такое, что заставляет вас оглянуться и сказать: «Господи, я же просто не знал, что творил!»?

Томпсон: Вовсе нет. Иногда я действительно оглядываюсь назад, но говорю: «Эх, а тогда ведь я был намного сильнее». Начиная с момента, когда я неделю читал ту программу, и до 30–35 лет я глубоко понимал каждую строку написанного мною кода. Я мог писать программу весь день, а ночью продолжать сидеть и читать строку за строкой в поисках ошибок. На следующий день я возвращался к нему и, разумеется, находил ляп.

Сейбел: А в 35 лет вы могли вспомнить то, что написали десятью годами раньше?

Томпсон: Да. Это позже моя память стала более избирательной.

Сейбел: Есть ли в изучении программирования то, что сейчас вы бы сделали по-другому? Сожалеете ли вы о пути, который избрали; может быть, хотели бы, чтобы что-то было сделано вами раньше?

Томпсон: Да-да, конечно. В школе мне обязательно следовало заниматься машинописью. Я плохо печатаю даже сейчас, но кто знал. Я никогда ничего не планировал. У меня нет дисциплины. Я делал то, что хотел делать завтра, на следующей неделе, всю жизнь. Если бы у меня были способности к планированию или предвидению, то курс машинописи я бы непременно по возможности прошел. И я бы, конечно, более углубленно изучил математику, потому что часто сталкивался с такими вещами, где помощи приходилось ждать только от математики. Вот такие частности, словом. Но если вернуться назад, то я уверен, что меня не хватило бы на то, чтобы сделать что-то принципиально по-иному. Обычно я ничего не планировал – просто делал следующий шаг. И начав заново, я точно так же делал бы следующий шаг.

Сейбел: Сразу после выпуска вы попали прямо в Bell Labs – как это вышло? На том этапе карьеры вы вроде не были классическим академическим исследователем.

Томпсон: Так уж меня занесло. Трудно сказать. На самом деле я, можно сказать, не учился. Формально – да, конечно. Один из моих преподавателей – собственно, мой хороший друг – прямо-таки натравил на меня рекрутера из Bell Labs. Но я не искал работу. У меня не было абсолютно никаких амбиций. И он назначил мне встречу на его маленьком рекрутерском стенде, а я то ли проспал, то ли сказал ему, что мне неинтересно. Но он не сдавался. В какой-то момент он позвонил мне и сказал, что хотел бы зайти. И пришел ко мне домой. И сказал, что хотел бы, чтобы я прошел собеседование в Bell Labs. Я отказался. Но он сказал: «Поездка бесплатная. Ты можешь делать все что угодно». А я ему: «Ладно, тогда вот что я тебе скажу. Работа мне не нужна. Но бесплатная поездка – это прикольно, у меня есть друзья на Восточном побережье. К ним и поеду». А он мне: «Отлично». Вот так я и попал на собеседование. Я приехал, два дня потратил на Bell Labs, а потом взял напрокат машину и поехал по Восточному побережью повидать школьных друзей, которые разъехались кто куда.

Сейбел: Очевидно, в вас было что-то, что заметили парни из Bell Labs и сказали: «Этот человек должен работать у нас».

Томпсон: С их точки зрения я смотреть не пытался. Я видел в них авторов статей, по которым я готовился к тем занятиям, что вел и где учился. Я знал их имена и репутацию. И они продолжали делать отличные вещи. По мне работа была работой, а эти парни не работали. Они просто развлекались. Как в школе.

Сейбел: Чем вы начали заниматься, поступив на работу?

Томпсон: Bell Labs занималась проектом MULTICS, и меня брали работать над MULTICS. Это я и делал. Я работал с машинами, загружал MULTICS, немного писал. В какой-то момент Bell Labs решила, что MULTICS не для них, и они отказались от проекта.

Но у них были машины специально под MULTICS. Они стояли без дела и ждали, пока их кто-нибудь не увезет. И примерно год я работал на одной из этих машин, она была чудовищная. Их использовали у нас только двое или трое. Я начал разрабатывать операционную систему, пытался создать маленькую операционную систему и запустить ее.

Это было безумно сложно, потому что сам компьютер был очень сложен. Но у меня получилось, и как-то я по этому компьютеру передал привет на 50 телетайпов по всему зданию. В итоге моя работа стала известна. Я побегал по помещению, нашел еще несколько неиспользуемых машин и, в принципе, таким образом и создал UNIX – на этих очень-очень маленьких компьютерах PDP.

Сейбел: Время на это у вас было – ваши боссы знали, что вы делаете, и одобряли этот проект как хорошее исследование, или вы занимались операционной системой сверхурочно?

Томпсон: Нет, честно говоря, я был просто неисправим. В принципе, меня могли даже и уволить, но меня это не беспокоило. Предполагалось, что мы должны заниматься фундаментальными исследованиями, но на деле выходило, что одними фундаментальными исследованиями мы заниматься будем, а другими нет. Только что мы выбрались из руин работы над MULTICS, так что операционные системы были одним из тех видов фундаментальных исследований, которые делать *не надо*. Поскольку мы пытались, не получилось, был большой провал, это дорого обошлось, так что давайте пошлем к черту. Так что я, можно сказать, ожидал, что за свои действия могу быть уволен. Но не уволили.

Сейбел: Как вы создаете программы? Царапаете на миллиметровке, запускаете утилиту для работы с UML или просто начинаете писать?

Томпсон: Зависит от масштаба проекта. Большую часть времени все хранится у меня в голове, никаких бумаг, и я концентрируюсь на сложных частях. Простые части отходят на второй план, их достаточно записать: они так и слетают с кончиков пальцев, когда все готово. Но над сложными я сижу и даю им некоторое время, чтобы созреть, – примерно месяц. В какой-то момент начинает складываться основание, образуется пирамида. И как только пирамида у меня в голове делается достаточно высокой, я начинаю с ее основания.

Сейбел: Но вы не просто создаете отдельные элементы – вы знаете, какова должна быть итоговая структура.

Томпсон: Допустим, кто-нибудь мне описывает что-то таким образом: «Вот это компьютер, а вот коды операций». Я могу представить себе структуру программ и то, насколько в соответствии с ней эффективны или неэффективны действия, основанные на этих кодах операций, потому что вижу основание и представляю себе иерархию. То же самое я могу делать и с программами. Если мне показывают функции из библиотеки или базовые вещи из нижнего уровня, я могу понять, как построить на этом разные программы и чего не хватает, какие программы будет сложно написать. Так что я могу представить себе всю пирамиду – останется только разобрать ее и определить, где основание.

Современное программирование во многих отношениях пугает меня: они пишут слой, за ним еще один, потом еще один, и все эти слои только и делают, что обращаются друг к другу. Меня смущает программа, которую *обязательно* читать сверху вниз. Там написано: «Сделай то-то». Отправляешься искать «то-то», находишь, а там написано: «Сделай еще что-то». Ищешь еще что-то, а там надо сделать еще вот это, и так до самого верха. И ничего не делается. Это просто перенос задачи на все более и более глубокий уровень. Я не могу этого ни понять, ни принять.

Сейбел: Почему же тогда не читать снизу вверх? Листья ведь где-то есть.

Томпсон: Но вы же не знаете, что является в данном случае листьями, а что нет. Если описание хорошее, то можно прочесть написанное английским языком и все понять, так что код можно не читать. Но если дается просто какой-то кусок кода и говорят: «Прочитай и сделай его лучше» или: «Прочитай и заставь его делать еще что-то». Тогда обычно приходится читать сверху вниз.

Сейбел: Вы что-нибудь записываете, прежде чем начать писать код?

Томпсон: Да, обычно это структуры данных. Я не записываю никаких алгоритмов или блок-схем, если вы это имеете в виду. Только то, к чему приходится обращаться практически в каждой строке кода, – структуры данных.

Сейбел: Если вы пишете программу на Си, значит ли это, что код на Си будет определять эти структуры данных?

Томпсон: Нет, это будут квадратики со стрелками и так далее.

Сейбел: Итак, у вас большая общая картина – пирамида. Насколько вы следуете плану в процессе написания кода?

Томпсон: Я не привязываюсь к коду. Если на полпути я понимаю, что другая декомпозиция лучше, то переключаюсь на нее. Я знаю многих, у которых написанная строка кода остается такой до конца жизни, если там, конечно, нет ошибки. Особенно если они пишут функцию для какого-то API, набросают этот API где-нибудь на бумажке или в спи-

ске рассылки – и все. Он никогда не меняется, как бы плох ни был. А я всегда с удовольствием все менял, если находил другой, более подходящий путь или иную декомпозицию. Я никогда не питал большой любви к имеющемуся коду. Код сам по себе – почти чепуха, его можно переписывать. Даже если ничего не изменяется, он все равно по какой-то причине портится.

Сейбел: Как вы понимаете, что вот этот код нужно отбросить?

Томпсон: Когда с ним становится тяжело работать. Я отбрасываю код гораздо быстрее, чем другие. Я отказываюсь от него, как только хочу к нему что-то добавить, но возникает ощущение, что добавить к нему что-то будет непросто. Тогда я выбрасываю его, начинаю заново и нахожу другую декомпозицию, при которой куда проще сделать то, что я собирался. Я не думаю долго над тем, выбросить код или нет.

Сейбел: Это же справедливо и для работы с чужим кодом?

Томпсон: Зависит от того, есть ли у меня на это право. Если да, то не важно, чей это код. Если нет и код чужой, то приходится терпеть. Или не делать это.

Сейбел: Если вы унаследовали чей-то код, то при его переписывании может возникнуть такая опасность: возможно, вы упустили какую-то тонкость в его работе или просмотрели какой-то функциональный элемент, который раньше был, а сейчас его не осталось. У вас так бывало?

Томпсон: Ну да, бывало, но это неизбежный элемент отладки. Если что-то забыл или не сделал, то сразу, как только это понял, доделываешь. Это просто элемент отладки. Код не получается с первого раза. Его расширяешь.

Сейбел: Построив систему, возвращаетесь ли вы, чтобы каким-то образом ее документировать?

Томпсон: Зависит от того, для чего она предназначена. Если она написана исключительно для меня, то нет, не возвращаюсь. Если я забуду аргументы, то добавлю строку о том, как это использовать. И в комментарии заголовка поясняю, что вся функция делает. Но очень кратко. Если это часть системы, или библиотеки, или еще чего-то, что должно быть опубликовано, то я потрачу время на документацию. В других же случаях нет.

Документация – это такое же искусство, как и само программирование. Редко когда документация оказывается на том уровне, какого бы мне хотелось. Обычно она намного изощреннее, чем нужно. Она содержит кучу маловажных подробностей и манящих возможностей, которые в данном случае неприменимы. Документирование – весьма сложный процесс, требующий больших временных затрат. Чтобы выполнить его

правильно, нужно относиться к нему как к программированию. Нужно разобрать и снова собрать, но уже лучше, переписать, если дело идет неправильно. А этого не происходит.

Кроме того, я предпочитаю документирование снизу вверх, а как раз так мало кто делает. Если программа опирается на другие программы, файлы или структуры данных, мне хочется видеть ясную ссылку на них, чтобы я мог пройти по ней и их почитать, а такие ссылки нередко отсутствуют.

Сейбел: То есть вы бы хотели читать код так же, как его пишете, а именно снизу вверх?

Томпсон: Да. Таким образом я могу удержать его у себя в голове и запомнить. В противном случае я читаю его и могу даже понять, но сразу после чтения он выветривается у меня из головы. Если я понимаю структуру кода, то он становится частью меня и я воспринимаю его полностью.

Сейбел: В своей речи при получении премии Тьюринга вы упомянули, что если бы Дэна Боброу заставили использовать PDP-11, а не более мощный PDP-10, то в тот день награду вместо вас и Дениса Ричи мог бы получать он.

Томпсон: Я просто пытался сказать, что наша награда – результат счастливого случая.

Сейбел: Думаете, вам повезло быть привязанными к менее мощной машине?

Томпсон: Безусловной удачей оказалось то, что компьютер был мал и эффективен. Но, думаю, этот код мы бы написали в любом случае. Нам сильно помогло то, что дело происходило в самый разгар революции мини-компьютеров. «Десятка» была большим мэйнфреймом, управляемым компьютерным центром. Автономные вычисления вместо централизованных, думаю, и явились элементом счастливого случая. И это сыграло свою роль в PDP-11.

Сейбел: Выиграла ли UNIX от того, что была написана на Си, в то время как другие ОС – например TENEX и ITS – создавалась на языке ассемблера и, следовательно, их нельзя было так запросто переносить на другое оборудование, как UNIX?

Томпсон: Были и другие хорошие языки системного программирования, на которых писались такие вещи.

Сейбел: Например?

Томпсон: NELIAC был версией Алгола-58 для системного программирования.

Сейбел: BLISS тоже относится к той эре?

Томпсон: BLISS, кажется, был позже. В этих языках акцент делался на то, чтобы они хорошо компилировались. Думаю, с самого начала было вполне ясно, что из-за хорошей компиляции нельзя так уж убиваться. Нужно делать это хорошо, но вовсе не обязательно безупречно. Дело в том, что пока вы дорастаете до безупречной компиляции, закон Мура вас все равно обгонит. Вы можете повысить качество на 10%, но пока вы это делали, быстродействие компьютеров выросло вдвое и, возможно, появилось еще что-то более значимое для оптимизации, вроде кэшей. Думаю, по большей части стремление к совершенству здесь – пустая трата времени. Это очень тяжело: вы порождаете столько же ошибок, сколько устраняете. Нужно остановиться и не тратить 100% времени на 10% работы.

Сейбел: Вы, наверное, слышали о статье Ричарда Гэбриела «Worse Is Better» (Чем хуже, тем лучше).

Томпсон: Нет.

Сейбел: Он сравнивал два стиля – стиль Массачусетского технологического института (MIT), где на первом месте правильность, и стиль Нью-Джерси (то есть Bell Labs), где высоко ценят простоту реализации. Его теория состоит в том, что стиль Нью-Джерси, который он также называет «Чем хуже, тем лучше», обеспечивает возможность немедленного запуска проекта и исправления в нем ошибок на ходу.

Томпсон: Думаю, у MIT всегда был некий комплекс неполноценности по поводу UNIX. Я выступал с лекцией о UNIX в MIT, и меня представлял, кажется, Майкл Детрузос. Он объяснял, почему UNIX не была написана в MIT, хотя именно так и должно было быть. Почему у них были возможности, люди и все такое, но тем не менее ничего не произошло. И тут меня осенило, что все это время в них сидит дух соперничества. У меня ничего подобного не было. Мы сделали UNIX, а они сделали MULTICS, настоящего монстра. Это явно синдром второй системы.

Сейбел: Где MULTICS была вторичной по отношению к операционной системе MIT CTSS?

Томпсон: Да. Слишком много лишнего. Почти неприменимо. Они по-прежнему уверяют, что это был потрясающий успех, хотя очевидно обратное.

Сейбел: Как я понимаю, большинство специалистов MIT придерживается именно такого мнения о MULTICS. Они предпочитали построенные ими же системы ITS и те, что были основаны на Лиспе. Похоже, после MULTICS получилась вилка. UNIX вышла в свет, как вам хорошо известно, и в MIT все эти любители Лиспа стали срочно работать на PDP-10 и строить основанные на Лиспе системы, которые в конце концов, думаю, и породили Лисп-машины.

Томпсон: Да-да. Я знал всех этих ребят. Я считал, что это сумасшедшая работа. Не думал, что Лисп – настолько уникальный язык, чтобы делать под него свою машину. И я, видимо, был прав. Все это время я говорил: «Вы с ума сошли». Вот PDP-11 – отличная Лисп-машина. PDP-10 – отличная Лисп-машина. Зачем строить машину под Лисп, которая не будет быстрее? Да и вообще нет никакого смысла строить машину под Лисп. Это глупо, в конце концов.

Сейбел: Были ли у MULTICS такие черты, которые вам нравились, но тем не менее не были перенесены в UNIX?

Томпсон: Вещи, которые мне понравились настолько, что я их оттуда позаимствовал, – это иерархическая файловая система и оболочка – отдельный процесс, который можно было заменить другим процессом. До того во всех системах был «исполнительный компонент» – это так тогда называлось, – который был встроенным языком обработки. Он исполнялся в каждом процессе отдельно. А каждый раз когда вы что-то печатаете в оболочке, она порождает новый процесс и запускает то, что вы напечатали, а когда тот заканчивается, то вы возвращаетесь, и таким образом всегда находитесь недалеко от оболочки.

Сейбел: И все это вы перенесли в свою систему, там не осталось ничего, о чем бы вы жалели?

Томпсон: Нет.

Сейбел: Судя по тому, что я читал по истории UNIX, вы действительно использовали тот процесс проектирования, который здесь описали. Вы долго думали, а потом, когда жена и ребенок на месяц уехали, сказали: «Отлично! Теперь я могу написать код».

Томпсон: Да... Мы собрались группой и поговорили о файловой системе. Нас было трое или четверо. Единственный, кто сейчас не очень хорошо известен, – это Радд Кэнэди (Rudd Canady). В те дни в Bell Labs были потрясающие удобства: можно было позвонить по телефону и потом получить распечатку вызова. То есть оставляешь заявку на то, чтобы оно было записано, и на следующий день в твоём почтовом ящике появится распечатка. И после того как мы обсудили файловую систему с помощью доски и мела, Кэнэди снял трубку, набрал номер и зачитал данные с доски.

Распечатка пришла, и получившееся оказалось так близко к проектной документации, которую мы тоже сделали, что были просто невероятные совпадения. И я взял и реализовал эту файловую систему, только на PDP-7. В какой-то момент я решил, что мне необходимо тестирование. И написал разные загрузочные штуки. Но у меня были проблемы при написании программ для файловой системы. Хотелось чего-нибудь интерактивного.

Сейбел: И вам просто хотелось поиграть с написанием файловой системы? В тот момент, получается, вы не собирались писать именно ОС?

Томпсон: Нет, это была просто файловая система.

Сейбел: И вы, таким образом, написали ОС, чтобы создать наилучшую среду для тестирования файловой системы.

Томпсон: Да. Примерно на полпути я понял, что это была настоящая система разделения времени. Я писал оболочку для запуска файловой системы. И еще пару программ, запускавших ее. Тут я подумал: «Осталось только сделать редактор, и у меня есть целая операционная система».

Сейбел: Какую самую страшную ошибку вам приходилось отлавливать?

Томпсон: В основном это были ошибки, связанные с порчей памяти. Больше такого не бывает. Не знаю почему. Но поначалу мы постоянно работали с различным экспериментальным оборудованием, и в нем были ошибки.

Сейбел: То есть память портилась из-за аппаратных ошибок, а не из-за вышедшего из-под контроля указателя?

Томпсон: Мог быть указатель. Могло быть оборудование. Или и то, и другое. Та ошибка, которая мне сейчас вспомнилась как один из худших примеров, случилась на PDP-11. Там не было блока умножения, но его можно было докупить отдельно и подключить как периферию ввода/вывода. Сохраняешь числитель и знаменатель и запускаешь. Проходит цикл ожидания, затем выдается ответ – частное и остаток. И это было сделано под PDP-11 без управления памятью, а потом к нам пришло первое экспериментальное оборудование для PDP-11 с управлением памятью, и в результате блок умножения конфликтовал с управлением памятью.

Например, сохраняем, а потом прогоняем тест занятости. А в некоторых участках теста занятости отсылался физический адрес вместо виртуального, и часть памяти в итоге затиралась числителем того, на что мы пытаемся поделить. А обнаруживалось это далеко не сразу, причем в самых разных местах. Безусловно, это самая сложная ошибка, которую мне пришлось искать.

Сейбел: Как же в итоге удалось ее отследить?

Томпсон: Я написал программу, которая должна была поставить мировой рекорд по нахождению знаков числа e . Предыдущие мировые рекорды были ограничены не вычислениями – количеством циклов в секунду, – а вводом/выводом. Я придумал новый алгоритм, узким местом которого были вычисления (ввод/вывод был тривиальным). Он был феерически тяжелым в отношении умножения и деления. И мы за-

метили, что машина каждый раз буквально обрушивалась, когда я запускал на ней эту свою программу. А потом и установили связь.

Сейбел: И это дало вам возможность понять, что проблема в умножителе. Вы сразу же нашли причину?

Томпсон: Сначала мы решили, что ошибка возникает, когда сохраняешь множитель в блоке умножения, потом обращаешься за ним, а его там нет. Мы обратились в компанию DEC, там ничего не нашли и не захотели связываться. Их слишком нормальным сотрудникам не хотелось иметь дело с гибридной машиной. В те дни у нас были принципиальные схемы машин, и мы, собственно, нашли ошибку по этой схеме. Потом мы позвонили в DEC и сказали им: «Соедините вот этот и вон тот проводки».

Сейбел: К счастью, сейчас оборудование так не сыплется.

Томпсон: Да. Думаю, такие ошибки сейчас редки. К тому же сейчас компоненты гораздо более изолированы друг от друга: ошибку можно сделать, только если совсем уж безумствовать. Просто ошибиться на ассемблере: очень несложно что-то перепутать в каком-нибудь регистре при вызове подпрограммы. Но если вы имеете дело с высокоуровневым языком, где все аргументы должны подходить, такие ошибки становятся все более и более редкими.

Раньше в языках ассемблера их было множество. Если речь шла только о программах, а не о комбинации программ и оборудования, обычно они случались в одном месте и повреждалось только это место. Ошибка была с чем-то связана. Нужно было сидеть и мониторить операционную систему. И часто – или очень часто – мы видели, что случалась ошибка, как можно быстрее останавливали процесс и смотрели, что происходит в других местах, таким образом отслеживая ошибки. Их можно было так поймать.

Но вот ту, о которой я говорю, поймать было нельзя. Пока я не написал ту программу с тяжелым умножением/делением, благодаря которой стало видно, с какой частотой происходит ошибка. Вместо того чтобы падать раз в пару дней, теперь система рушилась раз в пару минут. И как только мы получили то, что обрушивает машину, у нас появился шанс это найти.

Сейбел: Сегодня некоторые говорят: «Да, конечно, у ассемблера больше всего шансов повредить память из-за программных ошибок, но и Си склонен к этому больше, чем некоторые другие языки». Можно сделать так, что указатели будут указывать черт знает куда, и получится выход за границу массива. Вы не считаете, что это проблема?

Томпсон: Нет, ее можно обойти языковыми оборотами. Некоторые пишут хрупкий код, а некоторые – очень крепкий структурно, и тут все зависит от человека. Думаю, хрупкий код можно написать на любом

языке. Я определяю хрупкий код так: допустим, требуется добавить функциональность, так вот в хорошем коде ее нужно добавить только в одно место, а в хрупком придется затронуть сразу десять мест.

Сейбел: Если появляется брешь в безопасности из-за переполнения буфера, то что вы можете ответить на критику Си и C++, где утверждается, что они частично за это ответственны, что многих проблем можно было бы избежать, если использовать язык, который проверяет границы массивов или в котором есть сборка мусора?

Томпсон: Ошибки есть ошибки. Вы пишете код с ошибками, просто потому что вы так пишете. Если язык безопасен в смысле безопасности времени выполнения, то операционная система упадет, а не переполнит буфер, став при этом уязвимой. Например, *ping of death* – атака на IP-стек операционной системы. Мне кажется, что будут еще атаки такого рода. Это будут не атаки типа «полностью захватить машину и стать суперпользователем». Это будут *ping of death*.

Сейбел: Но ведь есть разница между отказом в обслуживании и уязвимостью, когда вы можете стать суперпользователем и делать, что захотите.

Томпсон: Есть две возможности стать суперпользователем: одна состоит в переполнении буфера, а другая – в том, чтобы указать программе сделать то, чего она делать не должна. В большинстве случаев реализуется вторая возможность, а не переполнение буфера. Вы можете стать суперпользователем без всяких переполнений. Так что ваш аргумент не работает. Все, что нужно сделать, – это уговорить *su*¹ дать вам оболочку; все пути и так уже там есть, без каких-либо ошибок времени выполнения.

Сейбел: Ладно. Не будем говорить о том, что ведет к краху программы, уязвимости или чему-то подобному; есть такой класс ошибок, которые случаются в Си и C++, но почему-то не случаются, скажем, в Java. Есть ли для определенных типов приложений какие-то преимущества, весомые по сравнению с тем, что эти ошибки будут происходить и вызывать неприятности?

Томпсон: Думаю, этот класс ошибок доставляет не так уж много проблем. Разумеется, каждый раз когда я пишу один из вызовов функции, которая не проверяет длину строки, *strcpy* и тому подобное, я знаю, что фактически пишу ошибку. Но каким-то образом я принимаю решение о том, оправданна ли эта ошибка, нужно ли экономить аргументы. Сейчас обычно я их экономлю. Но имеется семантическая проблема: если вы обрезаете строку и используете обрезанную строку, то появляется новая проблема. Ошибка по-прежнему там, просто буфер еще не переполнен.

¹ Команда UNIX-подобных операционных систем, позволяющая пользователю войти в систему под другим именем. – *Прим. науч. ред.*

Сейбел: Какие инструменты вы используете при отладке?

Томпсон: В основном вывод на печать. При разработке программы я размещаю очень много операторов вывода на печать. И пока я их не убрал или не закомментировал, вывод на печать достаточно надежен. Мне редко приходится возвращаться к этому снова.

Сейбел: И что именно вы печатаете?

Томпсон: Все, что нужно, все, что удобно иметь под рукой. Инварианты. Но по большей части я печатаю во время разработки. Так я и выполняю отладку. Я не пишу программы с чистого листа – я беру программу и изменяю ее. Даже в большой программе я вывожу: «main, left, right, print, hello». Да, «hello» – это не то, что я ожидаю от этой программы. Я вывожу на печать то, что ожидаю увидеть, и отлаживаю эту часть. При разработке я запускаю программу двадцать раз в час.

Сейбел: Вы печатаете инварианты; а используете ли вы утверждения, которые проверяют эти инварианты?

Томпсон: Редко. Мне проще убедить себя, что они правильны, и либо закомментировать вывод на печать, либо отбросить их.

Сейбел: Почему же тогда вам проще напечатать, что инвариант верен, чем использовать `assert` для автоматической проверки?

Томпсон: Потому что при печати вы действительно видите, что происходит, а не частные значения, и вы печатаете к тому же много всего, не только инварианты. Просто я делаю вот так. Я не предлагаю это как общую парадигму. Но сам я всегда делаю так.

Сейбел: Когда мы говорили о том, как вы создаете программы, вы упоминали процесс разработки снизу вверх. Вы строите эти кирпичики отдельно друг от друга?

Томпсон: Иногда.

Сейбел: А вы пишете тестовые модули для тестирования низкоуровневых функций?

Томпсон: Да, я так часто поступаю. Все зависит от программы, над которой я работаю. Если программа – это транслятор из A в B , то я предложу целый спектр возможных A и соответствующих B . Для регрессионного тестирования исполню все варианты A и проверю, насколько им будут соответствовать B . Компилятор, транслятор, поиск регулярных выражений. Что-то вроде этого. Но есть программы, которые совсем не похожи на эти. Я никогда много не занимался тестированием и в этих программах мало понимаю. Я проведу несколько проверок, но они редко будут значительными, потому что, например, их будет тяжело проводить внутри самой программы. Главным образом это будут просто регрессионные тесты.

Сейбел: Под более сложными для тестирования программами вы понимаете, например, драйверы устройств и сетевые протоколы?

Томпсон: Вообще, они ведь запускаются каждый раз, когда вы запускаете систему.

Сейбел: То есть, по-вашему, таким образом можно избавиться от части ошибок?

Томпсон: Да, конечно. Что может быть лучше для тестирования операционной системы, чем постоянно запускать ее?

Сейбел: Еще один важный аспект программирования – оптимизация. Одни приступают к оптимизации с самого начала. Другие предпочитают сначала написать код, а потом уже думать о дальнейшей оптимизации. Каков ваш подход?

Томпсон: В первый раз я делаю программу насколько возможно простой. И очень часто этого достаточно. Строить очень сложный алгоритм для того, что никогда не будет запущено, просто глупо. Это трата времени и порождение ошибок. И поддерживать это невозможно, потому что придется исписать математическими формулами 50 страниц, чтобы рассказать другому парню, что ты, собственно, делаешь.

То, что сделано просто и методом грубой силы, 99% времени будет работать отлично. Если вы действительно создаете инструмент, который используется часто и местами проявляет квадратичную сложность, то иногда приходится вложиться в него как следует. Но чаще всего нет. Чем проще, тем лучше.

Сейбел: Некоторым просто нравится доводить код до блеска – так сказать, код для кода.

Томпсон: Да мне тоже нравится, но здесь во многом ради кода приходится жертвовать алгоритмом. Я имею в виду, что обычно сложный алгоритм требует сложного кода. А я лучше буду использовать простой алгоритм и простой код, чем какой-нибудь ужас. А если нужно как-то кратко охарактеризовать мой код, то могу сказать, что он простой, легко меняющийся и маленький. Ничего особо интересного. Его может прочесть любой.

Сейбел: Есть ли еще такие задачи, для решения которых в целях производительности приходится работать вручную с ассемблером?

Томпсон: Это редкость. Это музейная редкость – разве что вы можете таким образом получить порядковую разницу. Если вы можете усердно работать и заставить маленький кусочек большой программы исполняться в два раза быстрее, то и всю программу можно заставить исполняться в два раза быстрее, если только подождать год-другой. Если вы пишете компилятор, то, безусловно, 99% кода, который вы пишете,

будут исполнены один или два раза. Но будет крохотная часть в операционной системе, которая работает 24 часа в день. А еще более крохотная – в самом внутреннем цикле такой системы. Так что, возможно, только 0,1% оптимизации, которую вы применили к компилятору, обернется для ваших пользователей каким-то эффектом. Но притом эффект может оказаться глубоким, так что, возможно, вы все равно захотите эту оптимизацию выполнить.

Сейбел: Но это будет результатом порождения лучшего кода в компиляторе, а не переписыванием всего компилятора на ассемблере.

Томпсон: Да, да.

Сейбел: Так что, возможно, часть причин, по которым программы пишут прямо на ассемблере, ныне менее важна, потому что компиляторы стали лучше.

Томпсон: Нет. Думаю, главным образом машины стали намного лучше. А компиляторы – все такой же отстой. Посмотрите на код, который выходит из GCC, это же ужасно. Просто плохо. И еще чертовски медленно. В самом компиляторе более 20 проходов. Это просто чудовищно медленно, но компьютеры-то стали в 1000 раз быстрее, с тех пор как вышел GCC. Поэтому кажется, что он стал быстрее: он ведь не стал настолько медленнее, насколько повысилось быстродействие компьютеров, на которых он запущен.

Сейбел: Кстати было бы упомянуть сборку мусора. С появлением Java сборка мусора наконец-то пошла в массы. Как однажды сказал Деннис Ричи, Си активно сопротивляется сборке мусора. Хорошо ли, что сейчас наблюдается явная тенденция к языкам со сборкой мусора? Заслуживает ли эта технология настолько массового использования?

Томпсон: Не знаю. Здесь я настоящий шизофреник. Если вы пишете операционную систему, компилятор Си или что-то еще, что должны использовать многие и многие, то сборку мусора я почти всегда считаю ошибкой. Дело в том, что здесь это всегда можно сделать вручную и лучше, притом намного лучше. Вы просто ухудшаете свою задачу, свою работу, делаете ее более медленной для ваших пользователей. Поэтому для операционной системы это, полагаю, ошибка. Почти никогда операционной системе это не подходит. Но если вы пишете программу под конкретный случай, которая должна сделать свою работу, после чего ее можно выкинуть, то это прекрасное решение. Сборка мусора берет на себя часть работы, о которой вам не хочется думать, и по разумной цене, потому что компьютеры пошли быстрые, так что тут сплошные преимущества. Теперь вы видите, что я занимаю в этом вопросе двойную позицию.

Отчасти проблема в том, что есть различные алгоритмы сборки мусора, а у них есть разные, существенно различные свойства. Пишете вы какую-нибудь вещь общего пользования, например операционную систему. И если вы пишете ее на языке, в который встроена сборка мусора, то у вас нет возможности выбора алгоритма. Допустим, у вас недопустимы временные задержки, а ваш сборщик мусора работает до некоего критического уровня, а потом начинает все подчищать. Вы обречены еще до начала работы.

Так что если вы работаете над задачей общего назначения, не зная, кто ваши пользователи, так делать не надо. К тому же сборка мусора в значительной степени конфликтует с когерентностью кэша. И нет такого алгоритма сборки мусора, который подходил бы всем машинам без исключения. Есть машины, где можно увеличить его быстродействие в пять раз и больше, повозившись с кэшем. Сборщики мусора должны быть в значительно большей степени привязаны к машине. Обычно они трактуются как отдельные алгоритмы, не зависящие от машин, но когерентность кэша очень важна для таких алгоритмов.

Сейбел: Кем вы себя считаете – ученым, инженером, художником, ремесленником или кем-то еще?

Томпсон: Не знаю. Ненавижу слово *ученый* – оно слишком отдает элитарностью. И предполагает наличие степени. Нет дипломов, в которых было бы написано «ученый», нельзя закончить курсы ученых, так что мне не нравится этот термин, я его не использую. Инженер – что ж, у меня есть диплом, в котором написано «инженер», так что я вправе называться *инженером*. И заполняя пункт «профессия» в документах, я обычно именую себя инженером или программистом, поскольку имею на то основания. Но обычно я просто ни о чем таком не думаю.

Сейбел: Ладно, если не учитывать то, как вы сами себя называете, то с какой профессией больше всего связана ваша? Физик, мостостроитель, художник, плотник?

Томпсон: Что-то ближе к низкому уровню. Например, ремесленник, но с творческой жилкой.

Сейбел: Как вы определяете талантливую программиста?

Томпсон: Во многом это вопрос энтузиазма. Спрашиваешь его, над какой самой интересной программой он работал. Просишь описать ее в целом, ее алгоритмы и принцип действия. Если он не может подобрать достойного ответа на мои вопросы, ничего хорошего в нем нет. Если я могу атаковать его, найти ошибки в его алгоритмах и решениях, так чтобы он не смог защитить себя, будучи значительно лучше меня знакомым с собственной программой, это плохой программист. В то же время можно оценить его энтузиазм. Об энтузиазме, разумеется, никто напрямую

не спрашивает, но в диалоге можно использовать своеобразную мерку энтузиазма, и мне это, например, всегда очень помогало. Так я и провожу собеседования. Мне говорили, что собеседование со мной просто опустошает.

Сейбел: Представляю себе. Это что-то вроде устного экзамена. А случилось ли вам встречаться с людьми, которые просто по личным качествам не могли такой экзамен выдержать, но при этом обладали весьма приличными способностями к программированию?

Томпсон: Нет, я не считаю, что эти два качества не зависят друг от друга. Так было бы, если бы я стал задавать им классические экзаменационные вопросы по компьютерным наукам, но я же так не делаю. Я прошу их описать то, что они сделали сами, над чем корпели. Мне никогда не встречался тот, кто потом и кровью что-то сделал, а потом не жаждал описать, что именно было сделано, как именно и почему так, а не иначе. Я позволяю им перехватывать инициативу. Я в их области любитель, а они в данном предмете профессионалы. Тот, кто не может ответить на вопросы любителя о его профессиональной сфере, неудачно выбрал профессию.

Сейбел: Чем вы занимаетесь в Google?

Томпсон: Инфраструктурой. Всем, что связано с операционными системами. Склеиваю разные куски. Я могу делать вообще что захочу. Основная задача – заставить кучу ненадежных машин работать как одна большая надежная мультипроцессорная машина. Думаю, это самое близкое описание.

Сейбел: Это ведь ключевая идея знаменитой гугловской системы обработки MapReduce, которая заключается в том, что нет совместного владения, есть пересылка сообщений, а не разделяемая память?

Томпсон: Ну, это процесс с хорошо известной семантикой, не включающий цикл обратной связи. Если у вас есть для него надежная инфраструктура, то с ее помощью можно решить множество задач.

Сейбел: И вы работаете в рамках этой структуры?

Томпсон: Нет, мы просто хотим переложить бремя ответственности с плеч отдельных программистов. Это сложная задача. Все оборудование здесь имеет массу слоев, обрабатывающих ситуации, когда не работает то-то и то-то. Что будет, если я не работаю, – кто меня убьет, кто стартует, кто что будет делать. Думаю, больше 50% создаваемого кода имеет вид «что, если».

Сейбел: То есть ваша задача – сделать так, чтобы половины такого кода больше не было?

Томпсон: Чтобы он был где-то скрыт. Он будет систематическим образом прилагаться к другому коду. То есть мы на это надеемся. Задача-то трудная.

Сейбел: Вам нравится работать в Google?

Томпсон: В чем-то очень нравится. Но в основном это очень громоздкая работа: деньги вкладываются в отладку, во многое другое. Масштабы я даже не в силах вообразить. Например, сегодня все только-только начинается, а завтра у вас уже два миллиона пользователей. Такие вещи мне представить просто тяжело.

Сейбел: То есть вы главным образом заняты производством. Можете ли вы сравнить работу в Google Labs с вашей прежней деятельностью в Bell Labs? Что между ними общего?

Томпсон: Не то чтобы я занимался производством. Я занимаюсь проектами, которые переходят в производство. Но я не нянчусь с ними до самого завершения. Возможно, описание моей работы – насколько я ему удовлетворяю сам, это другой вопрос – состоит в том, что я ищу, как сделать жизнь лучше. Придумать новую идею, что-то новое взамен старого. Постараться улучшить качество. Устранить то, что сделано неправильно, что отнимает много времени, вызывает ошибки. Если в Google я нахожу то, что может быть сделано лучше, то и пытаюсь сделать это.

Сейбел: Я знаю, что в Google есть правило, когда каждый новый сотрудник проверяется на знание языков, прежде чем ему разрешают вносить код в репозиторий. То есть вас должны были проверять на знание Си.

Томпсон: Да, но меня не проверяли.

Сейбел: Не проверяли! Вы не можете вносить код в репозиторий?

Томпсон: Да, не могу.

Сейбел: Вы просто не стали этим заниматься или у вас есть какие-то философские возражения относительно стандарта кодирования Google?

Томпсон: Не занимался. А потом понял, что мне это не нужно.

Сейбел: То есть вы работаете в своей песочнице? Большей частью пишите на Си?

Томпсон: Да, в основном на Си. И тестирование, и так поиграться – на Си, а стандарт Google – C++, и только C++. Не велико искусство программировать на C++, но мне не нравится. Вызывает внутреннее сопротивление.

Сейбел: Вы работали в AT&T с Бьерном Страуструпом. Вы каким-то образом участвовали в разработке C++?

Томпсон: Похоже, у меня сейчас будут проблемы.

Сейбел: Не без того.

Томпсон: Я тестировал этот язык в процессе разработки и делал к нему комментарии. Это было в рамках тамошней рабочей атмосферы. И бывало так, что пишешь что-нибудь, а на следующий день это не работает, потому что язык изменился. Он очень долго был крайне нестабилен. В какой-то момент я сказал: ну нет уж, хватит.

В интервью я именно так и сказал, что не использовал этот язык просто потому, что он был нестабилен и менялся каждый день. Когда это интервью прочитал Страуструп, он с криком ворвался в мою комнату, вопил, что я его унизил тем, что там сказал, а сказал я, оказывается, что это плохой язык. Хотя я ничего подобного не говорил. И опять по кругу. С того раза я таких вещей стараюсь избегать.

Сейбел: Как вы сейчас считаете: хороший это язык или плохой?

Томпсон: Безусловно, у него есть сильные стороны. Но в принципе я считаю, что этот язык плохой. Много он делает только наполовину, к тому же это просто куча взаимоисключающих идей. Все, кого я знаю лично или по работе в компании, выбирают подмножество этого языка, и это разные подмножества. Так что это не лучший язык для передачи алгоритма – для того чтобы сказать: «Вот, я написал, пользуйтесь». Он слишком велик, слишком сложен. И, очевидно, порожден комиссиями.

Страуструп многие годы вел кампанию – и она значила намного больше, чем весь его технический вклад в язык, – за то, чтобы C++ стал общепринятым стандартом. Он фактически руководил всеми комиссиями по стандартам и никому не отказывал. Он снабдил свой язык всей существующей функциональностью. Проектирования как такового не было – просто объединили все, что попало под руку. И, разумеется, я считаю, что языку это никак не пошло на пользу.

Сейбел: Как вы считаете, это связано с тем, что ему нравились все идеи, или же это был способ заставить принять язык, который каждому давал то, что он хочет?

Томпсон: Скорее второе, чем первое.

Сейбел: Похоже, многие говорят: «Черт возьми, C++ – это ужас!» При этом все его используют. Например, это один из четырех официальных языков Google. Почему же тогда его продолжают использовать, раз он так плох?

Томпсон: Не знаю. Думаю, он доживает последние дни в Google. Сейчас здесь больше его противников, чем сторонников.

Сейбел: И они переходят на Java?

Томпсон: Не знаю. Почти нет. Они жалуются, но не меняют язык. Вчерашние студенты – новобранцы Google – об этом знают. Поэтому слож-

но заниматься чем-нибудь другим. Вот почему его продолжают использовать – не нужно тратить массу времени на обучение и переобучение. С ним отдача быстрее.

Сейбел: Есть ли другие языки, на которых вам нравилось программировать?

Томпсон: В общем, все забавные языки, с которыми мне доводилось иметь дело. Например, для решения уравнений и всего такого – такие как Марле и Массута. Для строк – Снобол. В общем, я имел дело с десятками языков, если только с их помощью могло получиться что-то интересное мне.

Сейбел: А с какими инструментами разработки вам больше всего нравилось работать?

Томпсон: Я люблю уасс. Мне очень нравится уасс. Она делает как раз то, чего от нее хочешь. А вот ее дополнение Lex – просто ужас. Не делает ничего из того, что надо.

Сейбел: А вы им все равно пользуетесь или пишете лексические анализаторы вручную?

Томпсон: Пишу вручную. Так намного проще.

Сейбел: Доводилось ли вам заниматься литературным программированием по примеру Дональда Кнута?

Томпсон: Нет. Это отличная идея, но на практике реализовать ее почти невозможно.

Сейбел: Почему?

Томпсон: Получаются две версии одной и той же программы, которые часто рассинхронизируются и начинают конфликтовать. И исправить это нет никакой возможности. Если что-то хорошо написано на языке программирования, то это читабельно. И достаточно. Комментарии здесь не нужны. Комментарии, пожалуй, хороши для алгоритмов, или, например, если вы делаете что-то очень хитроумное, тогда они должны быть скорее в форме предупреждений. Я не любитель развернутых комментариев, это общеизвестно.

Сейбел: Когда я беседовал с Кнутом, он сказал, что ключ к техническому писательству – говорить одно и то же дважды взаимно дополняющими друг друга способами. Думаю, он считает это достоинством, а не изъяном литературного программирования.

Томпсон: Вот вы сделали два варианта. Один из них – реальный, тот, который исполняет машина. А второй – нет. И он нужен, только когда он значительно короче первого. Если же объем одинаковый, то можно читать тот, который работает. Если один вариант значительно короче,

менее детализирован и из него можно извлечь все что надо – отлично. Но очень часто его недостаточно, и за деталями приходится обращаться к другому. В зависимости от того, что вам нужно, вы читаете либо один, либо второй. Но пытаться сделать два микроскопических описания алгоритма – на языке программирования и на английском – может, Кнут это и умеет, а я так не могу.

Сейбел: Вы читали какую-нибудь из его литературных программ?

Томпсон: Только то, что было в его ранних статьях. Из недавнего – ничего.

Сейбел: Есть ли книги, которые вы считаете особенно важными для себя либо можете порекомендовать другим?

Томпсон: Я не читаю книги по программированию для начинающих, так что затрудняюсь что-то рекомендовать. Если мне приходится учить новый язык или что-то в этом духе, я стараюсь найти книгу по нему. Я предпочитаю сжатые информативные книги, которые описывают только синтаксис и семантику, а не такие, где сплошная болтовня и где мне пытаются объяснить, какой стиль хороший, а какой плохой.

Когда я был преподавателем, мне нужно было выбрать базовый учебник по своему предмету, так что приходилось читать все учебники подряд, чтобы сделать выбор. Поэтому дважды в жизни я знал практически всю базовую литературу по своим курсам. Но в остальное время я читал нечасто.

Сейбел: Когда вы придумывали UNIX, у вас был план сделать четыре части, из которых впоследствии возникнет операционная система. Тогда ваши жена и ребенок уехали на месяц, не мешая вам работать. Предполагаю, что в тот месяц вы работали не разгибаясь. Зачем мы делаем это? По необходимости? Или просто получаем от этого удовольствие?

Томпсон: Так поступаешь, когда увлечен. Вообще не представляю, чтобы я мог *не написать* UNIX. Кроме того, когда жена и ребенок рядом, ты прикован к 24-часовому циклу. Когда они уехали, 24-часовой цикл стал для меня менее обязательным. Мне вовсе ни к чему следовать за солнцем. Поэтому я обычно сплю по 6 часов в соответствии с 27- или 28-часовым циклом. В плавающем режиме. Когда я сплю до естественного пробуждения, я в лучшей форме, чем когда просыпаюсь от детского плача.

Сейбел: Так бывает, когда увлекаешься проектом и встаешь с желанием поскорее сесть за компьютер и писать код. Но иногда люди перерабатывают, потому что охвачены идеей срочно выпустить продукт, так что всем приходится работать по 80–100 часов в неделю.

Томпсон: Так сгорают на работе. Когда я был увлечен, то получал от программирования удовольствие и никогда не испытывал стресса.

И бывал в других ситуациях, когда установленные кем-то жесткие сроки порождают стресс. В этом нет ничего забавного, мне это не нравится.

Сейбел: Да, сгораешь на работе, и это, безусловно, плохо, но ведь в итоге работа сделана за более короткое время – может, оно того стоит?

Томпсон: Обычно получается, что так происходит непрерывно. И как только проходит один дедлайн, на горизонте появляется другой. Если все время находишься под угрозой дедлайна, то к следующему энтузиазма уже поубавится, и очень скоро просто не сможешь жить в таких условиях. Я вот не могу.

Сейбел: Со стремлением успеть к дедлайну напрямую связано умение оценить, сколько времени у вас отнимет та или иная деятельность. Можете ли вы оценить, сколько времени будете писать тот или иной кусок кода?

Томпсон: Зависит от того, для себя я пишу или для производства. Если пишу для себя, то могу. Я могу смириться со странностями. Я могу не делать лишние десять процентов. Я могу избегать зияющих дыр, которые, как я знаю, там есть. И все такое. Я могу сделать продукт, потом подчистить его на досуге и продолжать использовать. Возможно, это другое определение *готовности*. Но если делаешь продукт для производства, то тут вмешиваются другие люди, нужна координация усилий – этого я просчитать не могу.

Сейбел: В одном из интервью 1999 года вы сказали, что невысоко ставите Linux, и фанаты Linux приняли это в штыки. Что вы можете сказать сейчас, спустя десятилетие, когда эта система почти что завоевала мир?

Томпсон: Она гораздо надежнее – здесь нет никаких сомнений. И я как-то заглянул в ее код. Не так углубленно, как, например, в Plan 9. Они всегда нас опережали, просто у них намного больше ресурсов для работы с оборудованием. Так что, когда мы работали с каким-то элементом оборудования, я смотрел на драйверы Linux и писал в сравнении с ними драйверы для Plan 9. Сейчас у меня нет причин заглядывать в драйверы, я запускаю Linux. И иногда смотрю в код, но редко, так что не могу сказать, выросло качество или нет. Но надежность, несомненно, значительно возросла.

Сейбел: Вы когда-нибудь читаете код просто ради интереса?

Томпсон: Раньше я часто так делал. Когда я только пришел сюда, делал это, чтобы попрактиковаться и уловить рабочую атмосферу. Думаю, это нужно делать. Есть много того, что не говорится, но тем не менее следует знать.

Сейбел: Вы брали программу, чтобы полностью понять ее, или просто просматривали, как и что сделано?

Томпсон: То и другое. Сначала я пытался разобраться в больших библиотеках, рассматривал основные программы. В Google очень странный стиль программирования. Они берут вызов подпрограммы, запаковывают его как RPC и сохраняют как нечто статичное. То есть каждый может обратиться к нему в любой момент по любому поводу. И потом они вызывают групповой слушающий код и кто-то где-то получает сообщение, идет и находит вот это и выполняет этот вызов подпрограммы.

Сейбел: Это механизм распределенных вычислений.

Томпсон: Да. И это все, что делает это место. Его очень сложно читать. Выругаешься и начинаешь читать связующий код. А потом этот код. А потом общий код межпроцессного взаимодействия. Так и начинаешь осознавать, как читать код и понимать его. А до этого ничего не понимаешь.

Сейбел: Работая в команде, какую структуру вы предпочитаете?

Томпсон: Просто работать с хорошими и совместимыми людьми.

Сейбел: Работая с совместимыми людьми, предпочитаете ли вы четкое распределение ответственности: «Этот кусок кода пишу я, он мой и я несу за него ответственность», – или коллективную ответственность: «Мы все владеем этим кодом, и любой может делать то, что считает подходящим»?

Томпсон: Я всегда работал с чем-то средним. Есть один владелец, и если с каким-то куском кода проблема, пишешь ему письмо или просто говоришь при личной встрече, и исправить ошибку – его задача. Если же он в какой-то момент исчезает, больше не хочет работать с этим кодом и чинить его, действует безответственно – тогда чинишь его сам. Ключевая фраза: «Ты трогал его последним». Тебе и владеть. Так что это некий промежуточный подход. Не то чтобы толпа народу влезала в файл и изменяла код как заблагорассудится. Все проходит через единого владельца. Но владельцу гораздо проще все изменить.

Сейбел: Сегодня кое-кто ратует за парное программирование – двое за одной клавиатурой. Вы работали таким образом?

Томпсон: Так можно сделать что-то небольшое. Мне трудно быстро печатать, значит, тот, у кого это хорошо получается, может сесть за клавиатуру и печатать под мою диктовку. Я делал это для заданий, которые нужно было выполнить за несколько минут или часов, но каждый из нас мог бы это сделать и сам.

Сейбел: И как, результат был лучше или дела шли быстрее?

Томпсон: Результат точно не лучше. Возможно, ускоряется отладка: пока печатаешь, кто-то может отловить ошибку из-за твоего плеча. Так

что ошибок должно получаться меньше. Но я не принимаю это за принцип – просто так получается.

Сейбел: Вам по-прежнему нравится программировать?

Томпсон: Да, я люблю небольшие программы. Небольшие – то есть те, с которыми можно управиться за месяц. А взяться за какую-нибудь чудовищную задачу, на которую нужен год, – тут я пас.

Сейбел: Так было всегда или просто уже не хватает энергии на долгосрочные проекты?

Томпсон: Не знаю. Зависит от конкретного случая. Если огромный проект рассчитан на несколько лет, например, операционная система, его можно разбить на части, и эти части будут очень интересными, так что можно рассматривать их как несколько маленьких программ, а не как одну большую. Но есть и здоровенные неделимые проекты, пожалуй, мне они всегда казались трудными. Мне нужна отдача, обратная связь. А если приходится сидеть и работать, работать, работать целыми днями, не видя ничего, кроме кучи кода, то с этим у меня проблемы.

Сейбел: Вы в основном занимались исследованиями, то есть у вас был простор для работы. Изменилась ли ситуация, когда это стало работой? Не пропал ли интерес?

Томпсон: Нет. Интерес всегда был, главным образом потому, что я выбрал то, что хотел. И даже когда это стало профессией, в колледже-то было доступно множество вариантов профессий. Мне казалось, что многие вокруг что-то делают, чем-то заняты, и им можно помочь, написав небольшую программу. Это было для меня идеально. Я мог заниматься небольшими подработками, выбирая себе время и тот тип работы, который мне был особенно интересен.

Одна из первых подработок была у профессора-гуманитария, который каталогизировал труды Гомера. У него были «Илиада» и «Одиссея» на перфокартах. Ему были нужны частота употребления слов и их количество, то есть сравнительный статистический анализ этих двух поэм. И это было интересно. Это была работа с текстом, которую в те дни компьютеры просто не выполняли. Да, это и была моя первая странная подработка.

Сейбел: В интервью 1999 года вы рассказали, как убедили сына заниматься биологией, а не компьютерами, потому что те исчерпали себя. Это было почти десять лет назад. Что вы об этом думаете сейчас?

Томпсон: То же самое. В компьютерной сфере не происходит ничего, что нельзя было бы предсказать. Последним значительным событием стал, думаю, Интернет, а в 1999 году он уже работал. Да, все растет,

скорость персональных компьютеров продолжает экспоненциально расти, но что радикально нового?

Сейбел: Когда читаешь историю UNIX, кажется, что вы, ребята, изобрели операционную систему, просто чтобы поиграть с этим компьютером. То есть для того, чтобы сделать какую-то простую штуку – написать игру или что-то подобное на компьютере, пришлось сначала писать целую операционную систему. Чтобы хоть что-то начать делать, нужны были компиляторы и серьезная инфраструктура. Уверен, все это само по себе было интересно. Но не является ли сложность современного программирования, о которой мы уже говорили, нынешним эквивалентом все того же «Итак, в первую очередь ты должен создать собственную операционную систему!» По крайней мере, сейчас этого делать не нужно.

Томпсон: Сейчас все еще хуже. Операционная система не только дана – она обязательна. Поговорив сейчас с выпускниками компьютерных специальностей, вы увидите, что они совсем не понимают базового уровня. Очень, очень страшно за то, как они далеки от понимания того, что такое компьютер или хотя бы теория вычислений. Они просто ничего не понимают.

Сейбел: Я все думаю о вашем совете сыну – заняться биологией вместо компьютеров. Нет ли в программировании чего-то такого – интеллектуального удовольствия от определения процесса, который эти волшебные машины могут для вас выполнить, – вроде как ты работаешь с железом, но на очень абстрактном уровне?

Томпсон: Это затягивает. Но вы ведь не предложите своему ребенку попробовать крэк. Думаю, все изменилось. Может, я просто старею, но кажется, что когда накладываешь слой на слой, а затем еще слой, это не приносит такой радости, как, например, создание детерминированного конечного автомата. Да, разумеется, дело и в алгоритмах: со временем они становятся все сложнее. Новый алгоритм – это нечто основанное на пятидесяти мелких алгоритмах. А когда я был молодым, достаточно было писать один такой маленький алгоритм – и это было интересно. Их можно было понять, не прибегая к сложным вычислениям, когда алгоритмы надо делить на блоки, когда вот этот блок решается тем алгоритмом, о котором читал, но на самом деле почти ничего не знаешь, и так далее. Так что все изменилось. Я действительно так считаю и во многом из-за того, что постоянно надстраиваются новые слои, с которыми мы и имеем дело. А может, я просто слишком стар, чтобы их понять.

13

Фрэн Аллен

В 1957 году Фрэн Аллен собиралась стать учителем математики, но поскольку ей нужны были деньги, чтобы выплатить кредиты на обучение в университете, она временно (как тогда казалось) устроилась программистом в исследовательский центр IBM. Началом ее деятельности стало преподавание ученым IBM недавно созданного языка Фортран.

Учителем Аллен так и не стала – она проработала в IBM 45 лет и занималась рядом проектов по созданию компиляторов, в том числе компиляторов для компьютера STRETCH-HARVEST и для грандиозного, но так никогда и не законченного суперкомпьютера ACS-1. Кроме того, она работала над собственным проектом PTRAN, в рамках которого были разработаны технологии автоматической параллелизации программ на Фортране и создана система промежуточного представления SSA (Static Single Assignment), которая сейчас широко используется как в статических, так и в динамических (JIT) компиляторах.

В 2002 году Аллен была удостоена премии Тьюринга «За новаторский вклад в теорию и практику технологий оптимизационных компиляторов», став, таким образом, первой женщиной, получившей эту награду, за 40 лет существования премии. Кроме того, она стала первой женщиной, удостоенной звания действительного члена научного общества IBM – высшего почетного звания в компании для технического сотрудника. Кроме того, она является почетным членом научного общества Института инженеров электро-

техники и электроники (IEEE) и Ассоциации вычислительной техники (ACM), а также членом Национальной академии инженерных наук, Американской академии искусств и наук и Американского философского общества.

На протяжении всей своей карьеры Аллен видела, как менялась роль женщины в сфере информационных технологий – от первых лет, когда компании, вроде IBM, нанимали именно женщин на новую и непонятную должность «программиста», до последних десятилетий, когда в этой области стали преобладать мужчины.

В беседе мы обсудили особенности этих изменений, необходимость предоставления равных возможностей в сфере информационных технологий и пагубное влияние, которое язык Си оказал на компьютерные науки.

Сейбел: Как вы начали заниматься программированием? Я знаю, что в начале вы собирались стать учителем математики, но устроились на работу в IBM, потому что вам нужно было погасить кредиты на учебу.

Аллен: Для того чтобы получить лицензию учителя в штате Нью-Йорк, нужна была степень магистра. У меня были степень бакалавра по математике, вторая специальность – физика и двухлетний опыт преподавания. Затем я поступила в университет штата Мичиган, где углубилась в изучение математики. Для того чтобы получить степень магистра в университете Мичигана, нужна была вторая дополнительная специальность – и я выбрала вычислительные системы. Компьютерных наук тогда, в 1957 году, еще не было. Лишь 10 лет спустя они всерьез заявили о себе. Но в инженерной школе было несколько курсов.

Сейбел: Что вы там изучали?

Аллен: Там был компьютер IBM 650 – он весьма отличался от привычных нам сегодня компьютеров, и студенты учились на нем программировать. Это включало не только изучение всех свойств компьютера и написание кода – на ассемблере, естественно, – но и запуск своих программ на компьютере. Результат был абсолютно наглядным.

Сейбел: То есть вы набивали перфокарты, сами несли их к компьютеру и пропускали через него?

Аллен: Верно. А потом шли и исправляли. Это была машина с магнитным барабаном – барабан постоянно вращался, на нем и были записаны твои инструкции. Поэтому, чтобы все работало быстро, нужно было рассчитывать положение программ на барабане таким образом, чтобы при каждом новом обороте каждая новая программа располагалась на нужном месте.

Сейбел: А потом появились агенты по найму из IBM. Чем вас привлекла работа на IBM?

Аллен: Если честно, мне просто нужна была работа. У меня были долги, агент пришел прямо в кампус, и располагались они удачно – в штате Нью-Йорк. Так что я заполнила анкету, на самом-то деле толком не понимая, что за организация предлагает мне работу, что это за исследовательский центр IBM. Я тогда о них ничего не знала.

Несколько недель спустя мне позвонили – в то время я пыталась устроиться на работу в педагогический колледж в южном Иллинойсе. Я уже начинала отчаиваться – все никак не могла найти работу. Мне позвонили, когда я была в пути, и я согласилась – ровно ничего не зная о компании, и только заполняя бумаги выяснила, что речь идет об исследовательской лаборатории в Пафкипси.

Я устроилась туда и начала работать программистом. Компания IBM стремительно расширялась, их интересовала сфера информационных технологий, а поскольку не существовало никаких курсов по компьютерным наукам, то они нанимали людей отовсюду, откуда могли.

Сейбел: Чему вас там обучали?

Аллен: Насколько помню, это было своего рода обучение на ходу. Существовала определенная ориентация на нужды компании, но не могу припомнить, чтобы там были курсы по изучению программирования как таковые, что сейчас кажется немного странным. Кажется, были курсы, специфика которых зависела от подготовки того или иного сотрудника. Все проходило в очень неформальной обстановке.

Поскольку я была учителем математики, моим первым заданием стало обучение исследователей и других программистов языку Фортран. Я устроилась на работу в июле 1957 года, а Фортран был выпущен 15 апреля того же года. В моей же организации – исследовательском центре IBM – был выпущен указ, согласно которому к сентябрю все программы должны были писаться на Фортране. Это был их метод убедить собственных сотрудников – как и людей вне компании – использовать этот язык.

Сейбел: То есть речь идет об исследователях, работавших в IBM над собственными научными проектами?

Аллен: Да. У них был компьютер 704 – именно для работы на нем и был создан и оптимизирован Фортран. Они привыкли, что код на ассемблере можно было писать прямо на компьютере, то есть так же, как я делала в университете Мичигана, привыкли планировать время и запускать программы. Они не верили, что с помощью любого из высокоуровневых языков можно будет получать результаты, хотя бы сравнимые с получаемыми при программировании непосредственно самого компьютера.

Сейбел: И это был последний раз, когда ученые перешли на новый язык, – ведь они до сих пор используют Фортран, не так ли?

Аллен: Верно. Да, это был неудачный курс. Но в конце концов это был прекрасный опыт для всех нас, ведь Фортран был не только языком – в рамках этого проекта был разработан чрезвычайно передовой компилятор, который заложил структурные основы современных компиляторов.

Сейбел: Следующий ваш крупный проект, о котором я знаю, – это компьютер Stretch. Было ли что-нибудь в промежутке между этими двумя проектами?

Аллен: Да, были еще два проекта, над которыми я работала после Фортрана и перед компьютером Stretch. Один из них – разработка управляемой автоматической отладочной системы на ассемблере для компьютера 704. Мне очень нравилось работать над этим проектом.

Это была весьма незрелая операционная система. Мы ее делали втроем. Мы установили на компьютере несколько кнопок – тогда это можно было сделать, – одной из которых была аварийная кнопка. Когда казалось, что программа зависла, можно было просто нажать эту кнопку. Затем мы писали отладчик; одной из моих задач было написать на ассемблере программу, которая бы располагала двоичные данные по колонкам. Устройство считывания с перфокарт представляло данные в виде построчного двоичного кода, когда биты каждой инструкции располагаются в строке, но на ленте данные представлялись по-другому. Вот и требовалось организовать поколонное двоичное представление. Эту программу я храню до сих пор.

Среди прочего я с большим удовольствием вспоминаю чтение исходной программы – мне она показалась весьма изящной. Мне это запомнилось, потому что эта выдающаяся программа была написана не очень опытным на тот момент программистом – Роем Наттом. Она была прекрасна.

Сейбел: Что делает программу прекрасной?

Аллен: Она должна представлять собой простое прямолинейное решение поставленной задачи; при достаточно сложной структуре эта очевидность не вытекает непосредственно из поставленной задачи. Наверное, именно тогда я приобрела привычку учиться программированию и изучать новый язык посредством чтения и подробного изучения готовых программ.

Сейбел: Как вы читаете код? Предположим, собираясь изучить новый язык, вы находите готовую программу – и что с ней делаете?

Аллен: Ну, например, однажды один из моих сотрудников написал парсер. Это уже было в рамках проекта PTRAN. Мне хотелось понять его методы. Это, пожалуй, лучший парсер в мире (сейчас он есть в открытом доступе), действительно превосходный – он умеет исправлять ошибки на лету.

Мне хотелось понять его, поэтому я взяла и прочитала его. Я знала, что автором программы был прекрасный программист Филипп Чарльз. Для того чтобы понять новый язык или новую реализацию какой-либо весьма сложной задачи, я беру программу, написанную программистом, в чьей компетенции не сомневаюсь, и читаю ее.

Сейбел: Как вы изучаете фрагмент кода подобной программы? По шагам? Или просто читаете от начала и до конца, строя в уме структуру? Достаточно непростая задача.

Аллен: Задача и правда достаточно непростая, но, как правило, я интуитивно понимаю или предварительно узнаю общую структуру решения, поэтому могу просто начать читать с середины – с ядра. И это прекрасный способ изучить не только применяемые алгоритмы, но и то, как изящно можно использовать язык.

Сейбел: Есть ли у вас любимые истории, связанные с устранением неполадок в программах?

Аллен: Было несколько интересных случаев. Один из них произошел во время моей работы над системой MAD. Оператор машины позвонил мне посреди ночи – программу, которая должна была быть запущена на всю ночь, никак не удавалось запустить. У нас был особый способ автоматического подсчета контрольной суммы для проверки корректности данных, ведь сами машины толком не отслеживали и не исправляли ошибки. Говоря по телефону, я не могла сообразить, в чем ошибка. Но чуть позже вдруг поняла: в написанной мною части системы, отвечавшей за подсчет контрольной суммы, не учитывалась одна конкретная ситуация. Даже если программа исполнялась корректно, она не могла преодолеть этот барьер – из-за того способа, которым я осуществляла подсчет контрольной суммы. Я перезвонила оператору, и мы устранили эту неполадку.

Другой случай – его я запомнила, поскольку была очень довольна собой тогда, – был связан с одним моим коллегой, который тоже работал над Stretch, предпочитая делать это по ночам. Однажды он пришел утром, а вида он был внушительного – гигантского роста, очень серьезный. И бросил мне на стол распечатку отладочной информации – дампа программы – огромную, толстую стопку. Указывает мне на один какой-то бит в распечатке и говорит: «Почему этот бит установлен?» Он переживал всю ночь. Как ни странно, я знала ответ на его вопрос. Это не было

неполадкой – просто он не знал, что это, и решил, что именно этот бит был причиной ошибки.

Сейбел: Это случилось позже – вы говорили, что был еще один проект между MAD и Stretch.

Аллен: Да. Это был проект для одного ученого, который делал коммутационные схемы для аппаратного оборудования. Среди прочего он размещал схемы на том, что тогда считалось микросхемами. Мы разрабатывали математическое решение – было множество препятствий, конечно же, из-за физических размеров. Я работала над этим проектом в качестве программиста. Нас было двое, может быть, трое – все женщины.

Сейбел: А затем вы приступили к работе над проектом и весьма крупным – Stretch.

Аллен: Благодаря моему опыту работы с Фортраном и хорошему владению данным компилятором меня перевели из исследовательского центра IBM в очередной крупный проект компании – создание компьютера Stretch. Проект начался в 1955 году, а название Stretch появилось где-то в 1956 году; планировалось создать машину, которая работала бы в 100 раз быстрее любой другой машины, существовавшей тогда в мире: мы хотели создать абсолютно фантастический компьютер.

Все прекрасно понимали, что ключом к успеху данного предприятия будет компилятор и что главная задача, которую предстояло решить для достижения высокой производительности, – это доступ к памяти, для чего также очень важен компилятор.

Сейбел: Из-за того, что работа с латентностью памяти стала бы слишком неподъемной задачей для программистов, писавших на ассемблере?

Аллен: Да. И из-за того, что проблема латентности памяти решалась с помощью очень сложного аппаратного параллелизма. Применялись различные методики расслоения памяти, поэтому невозможно было предсказать, в каком порядке данные будут доставлены в вычислительный элемент. Шесть обращений могли происходить одновременно. В самом вычислительном элементе были конвейеры, что позволяло выполнять несколько инструкций одновременно. Самым сложным элементом машины был блок предварительного просмотра: в рамках архитектуры были предусмотрены точные сигналы прерывания, поэтому он должен был не только отслеживать весь перспективный параллелизм, но и откатывать его при поступлении сигнала прерывания.

Это была чрезвычайно сложная машина – и программировать ее было сплошным удовольствием. Компилятор представлял собой очень трудную задачу. Весь проект был восхитительно сложным.

Поэтому меня и еще нескольких сотрудников перевели из исследовательского центра IBM в этот проект для разработки компилятора и непосредственно операционной системы. Компилятор по размаху соответствовал самой машине. Из-за того что ранее мне приходилось работать с оптимизатором Фортрана, здесь я работала над оптимизатором для Stretch, который позже назовут Stretch Harvest. Общий план компилятора был утвержден другим комитетом, но мы четвером отвечали за детали, включая интерфейсы компилятора, их спецификации и прочие его компоненты. У меня был оптимизатор, у кого-то парсер, распределитель регистров и интерфейс с программой ассемблера.

Сейбел: Как проект был организован с точки зрения функций, отведенных разным работникам?

Аллен: Три человека разрабатывали общий план компилятора – у нас будет парсер, у нас будет то, у нас будет это, а это мы поставим сюда. И были люди, отвечавшие за продукт целиком, то есть было несколько уровней, на которых осуществлялось принятие решений и руководство проектом.

Затем им понадобилось назначить ответственных за каждый из крупных компонентов. Поэтому они попросили нас четверых, причем трое были женщины, заняться этим и совместно начать разработку интерфейсов.

Сейбел: Были ли помимо вас программисты, занимавшиеся непосредственно реализацией проекта?

Аллен: Да. У меня была группа из 17 программистов.

Сейбел: Как происходил переход от этапа разработки к этапу написания кода? Вы четвером собрались и установили интерфейсы между компонентами системы. Это произошло до того, как ваши 17 программистов начали писать код, или процесс написания кода влиял и на общую концепцию в целом?

Аллен: Все происходило, по большому счету, само собой. Ограничения были установлены вышестоящим начальством. Главы отделов, вроде меня, отчитывались перед одним человеком – Джорджем Гровером, который, собственно, и разработал всю глобальную концепцию, особенности которой, в свою очередь, были продиктованы ограничениями у потребителей. Это была командная работа, часто вносились изменения и поправки – отчасти из-за того, что мы изобретали эту систему на ходу. Но был и срок сдачи проекта. Корпоративная субординация не так много значила, работа в команде была важнее.

Сейбел: Случалось ли, что код, написанный кем-то из ваших подчиненных, заставлял пересмотреть какие-либо решения более высокого уровня о взаимодействии компонентов системы?

Аллен: Да, иногда становилось понятно, что задуманный интерфейс не будет работать. Это была часть работы – следить за тем, как все получалось на практике. Мы встречались командой – вчетвером. Но большей частью каждый из нас работал над созданием доверенного ему компонента системы – в этом смысле была огромная свобода.

Разработка ПО как таковая появилась позже. В то время еще не существовало разработки ПО, не создавалось крупных процессов. В следующем проекте (компьютер 360, руководитель Фред Брукс), в котором я не была занята, с ПО были большие проблемы. Технически с 360 все пошло на лад где-то в 1963 году. И некоторые разработчики аппаратного обеспечения перестали заниматься компьютером: ничего не зная о ПО, они занялись им, поскольку оно пребывало в полном хаосе. Это был настоящий бардак.

После того как 360 был закончен, один специалист – не знаю, работал ли он над 360, – написал письмо боссам IBM, предлагая ввести некий ряд мер по улучшению разработки ПО Cleanroom. Он заявлял, что если следовать всем изложенным им предписаниям, то можно будет создавать идеальные программы. И из-за всего того, что руководство компании уже пережило, – по крайней мере, мне так кажется, – они на это купились.

Сейбел: Вы имеете в виду из-за того, что проект 360 был таким тяжелым?

Аллен: Именно. Таким образом, отдел разработки продукции IBM стал придерживаться процессов Cleanroom – целого набора процессов. Например, постановкой целей и проектированием системы занимались разные группы. И проектировщики должны были продумать все до мельчайших деталей, чтобы программисты могли писать код в соответствии с разработанной концепцией. И эти группы не были открыты друг другу – вы просто делали все аккуратно и на выходе получали идеальное ПО.

Сейбел: Во время работы над проектом по созданию 360 Брукс отвечал и за программное, и за аппаратное обеспечение?

Аллен: Да, мне кажется, что он контролировал весь проект в целом. Но некоторых из глав ПО-отделов он заменил людьми с опытом создания аппаратного обеспечения. Разумный шаг, поскольку эти специалисты уже выработали прекрасную дисциплину вокруг создания «железа» – разработка микросхем, процесс тестирования и так далее. Это был более проверенный временем и традиционный метод разработки концепции. Мы – специалисты в ПО – все придумывали сами.

Сейбел: То есть вам кажется, что как минимум в рамках этого проекта внесенные ими в процесс разработки ПО изменения оказались спасительными?

Аллен: Этот шаг был абсолютно необходим, но разработчикам ПО было нелегко, когда все эти ребята, понятия не имевшие о ПО, пришли к ним и начали просто назначать проектные экспертизы, проектные спецификации и так далее.

Сейбел: Но все-таки эти изменения спасли проект. Это придало тем ребятам уверенности, и они сделали следующий шаг к процессу Cleanroom – шаг, который оказался не слишком удачным?

Аллен: Мне кажется, да. Так все и было. Процесс Cleanroom – «модель водопада» – во время представления руководству получил очень сильного защитника.

Сейбел: Этот защитник был из лагеря разработчиков аппаратного обеспечения?

Аллен: Нет, он был из разработчиков ПО. Но насколько я знаю, он не участвовал в проекте 360 и вряд ли хотел таким образом его спасти. И те из нас, кто уже имел кое-какой опыт работы в сфере ПО, были ошарашены некоторыми из произнесенных тогда заявлений. Но иногда громкие фразы говорят для того, чтобы что-то продать.

Сейбел: Вы когда-нибудь работали над проектом, в рамках которого применялся подобный процесс?

Аллен: О, да. И разочаровалась в нем, потому что на его ранних этапах проектировщик и программист не могли работать совместно. Одна из его проблем состояла – да и, пожалуй, состоит – в том, что жизненный цикл программы очень долг. В те времена для создания крупной программы требовались многие месяцы и даже годы. Менялась конъюнктура, менялись требования. И много значило мнение потребителя о том, что он хочет получить в результате.

Сейбел: И вы продвигали изменения по всей вертикали процесса? Или люди шли прямо к программистам: «В общем, мы поняли, заказчику нужен X»?

Аллен: Да. Никто не мог написать спецификации, которые соответствовали бы всем нормам и были бы полезны в отношении всех мелких деталей на протяжении всего жизненного цикла программы. В этом и заключалась проблема. Разумеется, сейчас мы работаем по другой схеме – просто сделай и выброси, что-то вроде того.

Сейбел: Ну, собственно, Брукс в своей знаменитой книге¹ и написал: «Планируйте выбросить первую версию – вам все равно придется это сделать».

¹ Ф. Брукс «Мифический человеко-месяц или Как создаются программные системы». – СПб.: Символ-Плюс, 2000.

Аллен: Да. На самом деле, это правда – я в этом убеждена. Но зачастую это приводит, на мой взгляд, к тому, что мы не думаем перед тем, как начать делать программу.

Мне всегда нравилось иметь перед собой общую картину – модель. Чаще всего я рисую блок-схему и пишу спецификацию об интерфейсах между частями. В то время мы, конечно, постоянно рисовали блок-схемы, потому что, во-первых, не всегда компьютер был под рукой, а во-вторых, потому что блок-схема – это прекрасный способ отражения взаимоотношений частей системы, планирования того, что и где будет выполняться, и отражения функциональности различных компонентов. Не знаю, что могло бы ее в этом отношении заменить.

Сейбел: Есть несколько разновидностей блок-схем: формальные блок-схемы – они являются частью документации, и схемы, которые чертишь на доске, пытаешься в чем-либо разобраться. Какого типа блок-схемы рисовали вы?

Аллен: В каких-то случаях – формальные блок-схемы. Часто в ядре программы находились очень сложные фрагменты, и было необходимо составить подобную блок-схему. В остальном же речь о схемах второго типа – это был способ выработки решения задачи. Доска изрисовывалась целиком – и становилась своего рода архивом месяца или любого другого периода времени.

Сейбел: Итак, вы тогда возглавили крупный проект по созданию компилятора PTRAN; вы впервые стали работать с явным параллелизмом, а не со скрытым параллелизмом в конвейерах центрального процессора и так далее. Тогда это было в новинку – как для вас, так и для IBM.

Аллен: Это было новым веянием для IBM, но мы уже очень запаздывали. Огромная работа, открывшая реальную практическую выгоду в этом плане, началась в Иллинойсе в 1969 или в 1970 году.

Сейбел: Для какого языка предназначался компилятор PTRAN? Был ли это простой Фортран, без дополнительных конструкций для параллелизма?

Аллен: Именно так, с этого мы начинали. Я хотела сделать то же самое, что мы сделали для оптимизации: пользователь пишет последовательный код на языке так, как это свойственно данному приложению, после чего компилятор оптимизирует и преобразует его для машины, применяя параллелизм.

Что касается PTRAN, то мы собирались использовать «пыльные колоды» (dusty decks), как мы их называли, то есть уже существующую базу кода, после чего автоматически задействовать предоставляемые железом параллельные компоненты.

Сейбел: То есть, по сути, речь шла о том, что сегодня мы называем симметричными мультипроцессорами?

Аллен: Да, возможно. Моделей параллелизма очень много – и в этом одна из трудностей. Думаю, здесь все могло быть гораздо проще. Но многоядерность – одна из наиболее интересных вещей, по крайней мере для меня. А моделей параллелизма много.

Собственно, мы строили это на основе уже существовавших работ, в частности Дэйва Кука. Некоторые работы были из Нью-Йоркского университета. Мы наняли группу свежееиспеченных PhD из ряда мест, в которых уже к тому моменту был наработан определенный опыт. У нас было довольно много значительных результатов – и в практической, и в теоретической части; мы работали одновременно и над тем, и над другим. Я убеждена в том, что практика должна быть выражаемой в узнаваемых алгоритмах, в теории и способах представления решения задач, а алгоритмы должны применяться на практике, для того чтобы понять, насколько они ценны и как их применять. Я считаю, что в нашей области при работе над проектом лучше всего сочетать теорию и практику.

Сейбел: Во время работы над проектом PTRAN вы руководили командой. Вы тогда все еще писали код?

Аллен: Я не писала код, но была очень близка к этому. Например, когда работа по SSA-представлению была закончена, я не видела, каким образом оно может быть реализовано в какое-либо разумное время. То есть это был очень хороший алгоритм, но я не могла представить себе реализацию с разумными ограничениями по времени и по памяти. Соответственно это был для меня вызов. Мне нужно было видеть этот код. Он был необходим мне. Его следовало реализовать. Он не мог остаться на бумаге – только как очень хорошее, даже отличное исследование с графиками и ограничениями по сложности.

Если мы не можем реализовать его по-настоящему, то этот вызов останется. И проделанная работа будет не столь полезна, как мне хотелось. В конце концов один из моих подчиненных все запрограммировал. Я прочла программу целиком, изучила каждый фрагмент кода, проанализировала все использованные структуры данных. Это было потрясающе. Я сказала: «То что надо. Это работает».

Сейбел: То есть вы просмотрели все фрагменты кода, которые должны были войти в систему?

Аллен: Да, да, да.

Сейбел: Вы также руководили всеми этими людьми? Или вы были просто главным техническим архитектором, а за руководство группой отвечал кто-то другой?

Аллен: Нет, я была научным руководителем группы. Было 10–12 основных сотрудников, и мы разделили всю работу таким образом, что каждый владел каким-то ее куском.

Сейбел: Как минимум начиная с выхода книги Джеральда Вайнберга «The Psychology of Computer Programming» (Психология программирования) не утихают споры, что лучше – когда кто-то один «владеет» кодом и соответственно отвечает за него или совместная работа, позволяющая избежать вещей, понятных только одному. Судя по всему вы решили, что оптимальным вариантом было разделить владение кодом?

Аллен: Мы работали командой, но это касалось состояния системы, реализации. Некоторые специалисты очень хорошо справлялись непосредственно с реализацией, поэтому владели определенным фрагментом – фрагмент оптимизатора или внутрипроцедурный анализ определенно делался кем-то одним или двумя. Но были и специалисты, которые делали огромную теоретическую, научную работу – писали доклады, документы и алгоритмы. Единство этих двух противоположностей и сделало нашу группу такой мощной и сплоченной.

В то время велась большая работа по анализу и трансформациям для параллелизма. Поэтому я пыталась сделать так, чтобы каждый специалист-теоретик написал какой-то фрагмент кода, выразил свои теоретические построения в коде как часть системы. А тех, кто занимался более практическими вещами, я старалась организовать так, чтобы они писали как можно доступнее для более широкого круга специалистов.

Сейбел: Многие программисты сделают все, что угодно, лишь бы избежать руководящей работы. Правилось ли вам быть руководителем?

Аллен: В первые годы существования исследовательского центра не было разделения – руководящая должность вовсе не означала продвижения по карьерной лестнице, повышения оклада. Просто кому-то нужно было руководить разработкой вот этого отрезка работы, поэтому говорилось что-то вроде: «Не хочешь ли взяться за это?» или «Очевидно, этим должен руководить ты». Это было техническое руководство – таких руководителей было немного. Но в исследовательском центре человек становится членом исследовательского центра (Research Staff Member, RSM) с момента прихода туда и практически до конца карьеры. Вот кем являемся мы с коллегами. Поэтому можно постоянно выдвигаться и уходить с руководящей должности – без какого-либо клейма.

Сейбел: Скорее всего, руководить доверяют тому, кто хорошо с этим справляется. Как вы приобрели подобные навыки?

Аллен: Ну, меня отправили на курсы менеджмента. В то время, когда меня впервые назначали руководителем, обычно делалось так. Но,

думаю, все началось еще когда я росла на ферме. Я была старшей из шести детей – пятеро младших были погодки, и мне, конечно, приходилось помогать родителям. Возможно, это естественная для меня роль, в каком-то смысле.

Сейбел: Одна из трудностей в техническом менеджменте, которым вы занимались, – уравновесить стремление самому направлять работу сотрудников и их желание думать самостоятельно.

Аллен: Как мне представляется, в рамках работы над проектом Stretch я получила ряд суровых уроков. Помню, однажды ко мне подошли несколько сотрудников проекта и сказали, что мы должны использовать списки и хеш-функции. Мы знали о списках, но хеширование было в новинку. И двое коллег говорят мне, что хотят использовать хеширование для таблицы символов. Я ответила: «Нет, мы не можем. Мы не знаем, как с этим работать». Бла-бла-бла. В следующий понедельник я прихожу и вижу – они сделали это. Они снесли систему целиком и сделали ее заново, добавив туда хеширование. Это сработало, система начала работать намного быстрее. Это был важный урок для меня. Я должна быть более открытой для совершенно новых идей.

Сейбел: То есть иногда – может быть, даже часто – подчиненные знают, о чем говорят, и не следует слишком уж вмешиваться в их работу, потому что вы можете зарубить на корню хорошую идею. Сложнее представляется ситуация, в которой вы действительно правы, а идея подчиненного в самом деле не без изъяна, но вы все равно не хотите на него давить.

Аллен: Да, бывало и так. Нередко приходил сотрудник, обладавший знаниями в определенной сфере, с желанием применить эти знания в работе над текущей частью проекта. Но поскольку он работал в проекте недавно, то не знал его специфику. И обычно это происходило незадолго до дедлайна.

С серьезной проблемой такого рода я столкнулась, выполняя какой-то проект по договору субподряда. У меня была группа специалистов, прекрасно справлявшихся с созданием оптимизатора на основе нашей же работы, которую мы проделали для ПЛ/1 – громоздкого, очень прихотливого языка. Но кто-то из работников субподрядчика, недавно открыв для себя объектно-ориентированное программирование, решил применить его здесь по полной. Я не смогла ему помешать, даже несмотря на то, что отвечала за контракт, и проект был загублен. Грубо говоря, проблема была в том, что в ПЛ/1 множество указателей, и каждый указатель постоянно отслеживался; а чтобы отслеживать значение, то есть находить значение по указателю, требовалось 11 инструкций.

Сейбел: То есть в уже сгенерированном коде.

Аллен: В сгенерированном коде и в самом компиляторе, поскольку все это запускало и сам компилятор. Каждый раз, сделав шаг, нужно было убедиться, что все в порядке. Проверить, перепроверить и снова перепроверить. Так делается и сегодня. Некоторые уроки мы так и не выучили. И мне кажется, что я не очень правильно поступила в той ситуации: нужно было указать на издержки, связанные с применением там объектно-ориентированной технологии. Все работало ужасно медленно, поэтому проект был закрыт.

Сейбел: Какие продукты для IBM, в разработке которых вы принимали участие, имели сроки сдачи, в которые надо было уложиться?

Аллен: Безусловно, здесь нужно упомянуть проект Stretch. Кроме того, я участвовала в разработке продукта еще 2–3 раза, с еженедельными ревизиями кода незадолго до сдачи проекта. Я с большим уважением отношусь к этим процессам, считаю их очень важными для конечного результата и для команды, работающей над проектом. Хотя это может быть очень утомительным – каждую пятницу садиться читать код, слушая объяснения, почему сделано так-то и так-то, и находить ошибки других.

Сейбел: Это утомительно, но стоит того?

Аллен: Безусловно, стоит. Во время работы над проектом PTRAN, ближе к завершению, мы включали фрагменты кода в другие продукты, полдня уходило на объяснение ошибок в нашем коде, их коде, неважно, – и так каждую неделю, около 10 месяцев. Полпятницы уходило на это.

Сейбел: Работая в подобных условиях, вы ощущали, что у вас есть процесс, позволяющий достаточно точно рассчитать время, необходимое для создания определенного объема ПО?

Аллен: Сотрудники отдела разработки продукции, конечно, ощущали. Все отслеживалось, и я уверена, что до сих пор все так и есть. В том числе было статистическое отслеживание качества кода. В смысле, сколько ошибок было найдено на этой неделе. Мне нравилась рабочая атмосфера в лаборатории разработки продукции – там, где все воплощалось.

Сейбел: На что вы обращали внимание, нанимая сотрудников?

Аллен: У меня было множество связей в университетах. В Нью-Йоркском университете был прекрасный факультет, где программистов учили писать компиляторы; эти ребята действительно хорошо писали компиляторы.

Сейбел: То есть можно нанять человека, рекомендованного вам профессором, которого знаешь и которому доверяешь. А если проводишь себе

седование с тем, у кого нет солидной рекомендации, – как за несколько часов понять, станет человек хорошим программистом или нет?

Аллен: Я всегда начинала собеседование с претендентом на место в исследовательском центре IBM, пытаюсь понять, что этого человека интересует в жизни. Для меня это был своего рода пороговый уровень. Было абсолютно не важно, имело ли это увлечение какое-то отношение к программированию или компьютерам. Если человек не может ни чем увлечься, то не будет выкладываться по полной, работая в группе.

Иногда приходится идти на большой риск. Однажды я серьезно рискнула, взяв на работу парня, хотя его научный руководитель написал мне, что тот страдает тяжелой формой дислексии. В IBM у него работа не заладилась, но после он основал собственную компанию, и я до сих пор обращаюсь к нему за советами, за рекомендациями по поводу технической реализации определенных идей. Он просто знает эти вещи. То есть то решение не было ошибкой. Оно было ошибкой для того проекта, но не в смысле его отношений со многими из нас.

Сейбел: В дальнейшем вы стали заниматься наставничеством – IBM даже присуждает премию лучшему наставнику, названную в вашу честь. Как сделать, чтобы неопытные программисты становились более квалифицированными программистами?

Аллен: Я и сейчас не очень-то понимаю, что такое наставничество. Главное – нужно посоветовать начинающему специалисту не занимать слишком рано руководящую должность, что весьма соблазнительно для тех, у кого есть соответствующие таланты. Для начала нужно заработать репутацию своей технической работой. Какая-то интересная научная работа, алгоритм, отлично написанный код – что угодно; сначала нужно заработать себе репутацию чем-то подобным. Это сослужит вам хорошую службу, если вы действительно хотите руководить проектами, понимая в деталях, что требуется для выполнения той или иной работы.

Сейбел: Возможно ли вообще быть хорошим руководителем, не имея выдающихся технических способностей, а просто умея хорошо координировать работу других?

Аллен: Да, только если этот руководитель не думает, что он хорошо разбирается в технологических процессах, и если он может понять, кто из его подчиненных разбирается и кто не разбирается в этих процессах.

Сейбел: Это, наверное, самое сложное. Как по-вашему, чем отличается по-настоящему хороший программист?

Аллен: Мне всегда нравилось находить людей, которые открывали бы мне глаза на что-то. Это очень важно для меня, поскольку я постоянно думаю о системах и поэтому хочу, чтобы у меня были люди – по крайней мере, в исследовательском центре, – которые могли бы показать

мне что-нибудь новое и интересное или новый подход к старому – новый способ решения какой-то задачи.

Кроме того, я опираюсь на мнение других. Я часто ошибаюсь, и бывает очень познавательным обнаружить, что думаешь о специалисте намного лучше, чем остальные участники коллектива. Если у вас хорошая группа, то прислушиваться к мнению ее участников – очень эффективный способ узнать, кто хорошо справляется со своей работой.

Сейбел: Можете ли вы сказать, когда последний раз программировали?

Аллен: О, довольно давно. Наверное, я прекратила писать код, когда появился язык Си. Это был серьезный удар. Мы так успешно работали над оптимизациями и трансформациями. Мы решали одну непростую задачу за другой. После появления Си на одной из конференций, посвященных компиляторам SIGPLAN, случился спор между Стивом Джонсоном из Bell Labs, который поддерживал Си, и моим сотрудником Биллом Харрисоном, участником одного из моих тогдашних проектов, который поддерживал автоматическую оптимизацию.

В частности, во время этого спора Стив говорил, что больше не нужно создавать оптимизаторы, потому что теперь обо всем позаботится сам программист. Что это, на самом деле, задача программиста. Стимулом к созданию языка Си послужили три проблемы, которые невозможно было разрешить в рамках высокоуровневых языков. Первая – управление прерываниями. Вторая – распределение ресурсов, возможность перехватить управление и передать его следующему процессу в очереди. Третья – распределение памяти. Ничего из этого нельзя было решить с помощью высокоуровневого языка. Что и стало поводом для появления Си.

Сейбел: Как вы считаете, с существованием языка Си можно было бы смириться, ограничив его применение ядром операционной системы?

Аллен: О, да. Это было бы прекрасно. Действительно, нечто подобное нужно, чтобы эксперты могли бы проводить действительно тонкую настройку без особых препятствий, ведь решение этих проблем действительно очень важно.

К 1960 году у нас было множество прекрасных языков: Лисп, APL, Фортран, Кобол, Алгол 60. Это языки более высокого уровня, чем Си. С момента появления Си мы серьезно деградировали. Си уничтожил нашу способность прогрессировать в автоматической оптимизации, автоматической параллелизации, автоматическом отображении языков высокого уровня в машинную архитектуру. Это одна из причин, по которым компиляторы, по сути, больше не изучаются в колледжах и университетах.

Сейбел: При том что курсы по созданию компиляторов определенно читаются?

Аллен: Не во многих университетах. Это ужасно. До сих пор проводятся конференции, создаются хорошие алгоритмы, проводятся хорошие исследования, но отдача, по моему мнению, абсолютно минимальна. Все из-за того, что языки вроде Си слишком уж локализуют решение проблем. Подобные языки – вот что уничтожает компьютерные науки как предмет.

Сейбел: Но большинство современных новейших языков – более высокого уровня, чем Си. Например, Java, C#, Python и Ruby.

Аллен: Да, но они все равно излишне локализуют решение. Суть в том, что они точно указывают местонахождение данных. Если вы посмотрите на упомянутые мною языки (Лисп, APL, Фортран, Кобол, Алгол 60), то они не указывали местонахождение данных и как их перемещать, где их размещать в компьютере. В конечном итоге ключевым является значение данных в любой момент.

Сейбел: Но лишь немногие языки, вроде Си или C++, по-прежнему используют указатели в чистом виде. В Java есть сборка мусора, и данные перемещаются. По-вашему, это тоже излишняя локализация?

Аллен: Да. Я убеждена, что есть возможность сделать с данными то же самое, что мы сделали с вычислениями в мире оптимизации. Мы не очень хорошо умеем управлять данными. У нас нет хороших методов автоматического управления данными – установления местонахождения данных, которые будут использоваться совместно.

В настоящее время ведется много захватывающих исследований в разных направлениях. Но, думаю, не хватает более глобальных, более смелых решений. Многие исследования происходят в пространстве, ограниченном современными реалиями или современными представлениями о проблеме. Ничего не изменится в мгновение ока – уже написаны миллионы строк кода. Но мы должны начать избавляться от оков, вроде «это будет сделано здесь, а вот это – тут».

Сейбел: Вся ваша профессиональная деятельность во многом была связана с высокопроизводительными вычислениями. Однако, скажем, к 2019 году у каждого ноутбука будет 1000 ядер. Будет ли это означать, что создание высокопроизводительных и пользовательских компьютеров станет одним и тем же? Или с высокопроизводительными вычислениями все всегда будет совсем не так, как везде?

Аллен: Это, наверное, зависит от того, где мы сейчас на этой шкале. Дойти до петафлопсов – текущая задача высокопроизводительных вычислений, и я не знаю, каким образом эта задача будет решена. Естественно, битва за производительность будет вестись с помощью многоядерности, поскольку ее движущие идеи – сокращение потребления

энергии и множество других благих идей, а также решение некоторых задач на уровне элементарной физики.

Поможет решению этих задач и соревновательный элемент. Но использование этого множества ядер переводит проблему из области аппаратного обеспечения в сферу ПО, где, как мне кажется, мы не готовы ни к какому прогрессу. Использовать эти ядра – вот задача, которую нужно решать новым уровнем языков. Нам нужно полностью исследовать эту проблему. Но для этого необходимо радикально новое мышление.

Думаю, за эти первые 50–60 лет (ENIAC появился в 1943 или 1944 году) мы создали не только превосходное, восхитительное – просто потрясающее – наследие, но и несколько вещей, от которых нужно избавляться. Чтобы их заменить, потребуется очень много времени, и, думаю, трудно предсказать, как все это будет развиваться. Но это развитие пойдет очень быстрыми темпами, если мы сможем применить новое мышление в нужных местах. Мы знаем, как проводить вычисления на большом объеме данных, но мы понятия не имеем, каким образом передать данные на вычислительные элементы компьютера.

Сейбел: Вы можете привести простой пример того, что понимаете под передачей данных на вычисление – в противоположность тому, что мы сегодня знаем, как делать?

Аллен: Для меня это означает собственноручно управлять данными. По сути, сегодня мы это делаем с помощью ссылок – они передаются посредством аппаратного обеспечения или фундаментальными операционными системами и системами поддержки. Зачастую эти ссылки располагаются на самом базовом уровне.

Сейбел: В том смысле, что возможен указатель внутрь структуры или массива?

Аллен: Да, на его элемент. После чего значение передается – в зависимости от протоколов аппаратного обеспечения или самой архитектуры – туда, где оно может быть использовано как часть вычислительного процесса.

Но это можно сделать иначе – организовать местонахождение данных в их относительных положениях как объект для оптимизации. Кроме того, зачастую то, что хорошо для одного вычисления, плохо для другого. Какая-либо организация, даже простейших вещей вроде матриц, не годится, если вы пытаетесь получить к ней доступ по-иному. Таким образом, речь идет о противопоставлении сочетания порядка доступа и местонахождения. Возможно, потребуется проделать какую-то работу по разработке архитектуры или аппаратного обеспечения, но мне кажется, что это возможно сделать, если мы вернем ряд функций, касающихся ссылок и адресации, аппаратному обеспечению. Есть компью-

теры, где в момент поступления данных в память можно выполнять достаточно много преобразований. Там можно осуществлять преобразование данных.

Скорость вычислений – вот, по большому счету, что важно в высокопроизводительных вычислениях, поэтому мы делаем все для того, чтобы увеличить эту скорость. Загрузка вычислительного устройства – одна из больших задач, стоящих перед нами, но мы никогда не делали ее первостепенной задачей. Мы оставляем ее аппаратному обеспечению.

Сейбел: В своей лекции после получения премии Тьюринга вы сказали что-то вроде: «Мы на распутье, и мы можем этого не заметить. Мы можем пойти не той дорогой и идти по ней достаточно долго».

Аллен: Да.

Сейбел: А верная дорога, по-вашему, – это вернуться к работе над автоматической параллелизацией?

Аллен: Да, но мы должны заниматься этим с учетом известных сегодня высокоуровневых языков.

Сейбел: А неверная дорога – это поиск лучших путей явного использования параллелизма?

Аллен: Да, думаю, в конце концов мы таким образом лишь усугубили свои проблемы.

Но нам нужны языки более высокого уровня, и, конечно, есть языки, отражающие специфику конкретной предметной области, и есть действительно превосходные методы разработки.

Но мы должны захотеть использовать и эти наработки, и интеграцию систем, не забывая тот факт, что данные поступают отовсюду. Они больше не инкапсулированы внутри программы, кода. Сейчас мы видим немислимо огромные массивы данных, которые стали доступными. Это цифровые и информационные данные, и они хранятся – и будут храниться – по всему миру, особенно если вы занимаетесь чем-то вроде биоинформатики. И нам нужно создать платформу, возможно, состоящую из множества частей, которая позволит соединить все эти вещи; для этого потребуются, вероятно, совсем другие вычислительные мощности. И рано или поздно нам придется решать проблемы удобства использования и цельности этих систем.

Сейбел: Вы говорите об удобстве для программиста или для конечных пользователей этих систем?

Аллен: Для пользователей. Это ресурс, гигантский ресурс. Также важна цельность корректности этих систем. Не так давно я работала над одним проектом по управлению рисками для Агенства национальной безопасности, и до меня вдруг дошло, что зачастую в высокопроизводи-

тельных вычислениях не нужны вычисления со всей возможной точностью. Не нужно использовать все данные, чтобы добиться прогресса в решении задачи. И поэтому в сфере изучения данных проводятся неплохие исследования, как мне кажется, с достаточно неплохими результатами, которые всех устраивают. Многоядерные компьютеры представляются мне прекрасной возможностью по-новому взглянуть на многое.

Сейбел: Кем вы себя ощущаете – ученым, инженером, художником или ремесленником?

Аллен: Я специалист в области компьютерных наук. Работая в своей узкой области, я помогала развиваться этой дисциплине. Это были интересные времена – когда компьютерные науки только появились, – потому что были вопросы «А наука ли это? Если в названии чего-то есть слово „наука“, наукой это не является». Тогда мне было совсем непонятно, что этот термин означает.

Но компиляторы – это очень старая область, старше разработки операционных систем. Как-то я решила узнать об этом побольше. Слово «компилировать» в исходном смысле означало вставку небольших отрывков инструкций для исполнения. Так, например, слово «add» (сложить) нужно было заменить примитивными машинными инструкциями. Если требуется выполнить сложение, то машина обращается к библиотеке, которая находит его определение и вставляет его.

Но ассемблеры также использовали символы. Не уверена, что это правда, но раньше я считала, что впервые символьные имена для переменных использовал человек по имени Нэт Рочестер, работавший над одной из первых машин (IBM 701) где-то в 1951 году. Он занимался ее тестированием, и они писали программы, чтобы тестировать компьютер. В процессе тестирования они стали использовать символические переменные. Но с тех пор мне приходилось сталкиваться со свидетельствами того, что информацию представляли в символическом виде и раньше. Мне кажется, это стали делать в начале 1950-х, или даже в 1940-х. Например, стоит найти и посмотреть, как все записывалось и выражалось в ENIAC.

Сейбел: То есть в какой-то момент вы поняли, что стали специалистом в области компьютерных наук, создателем теорий об оптимизации компиляторов и так далее. Но начинали вы как программист, вас нанимали для того, чтобы вы писали код. На момент работы над проектом PTRAN вы уже руководили командой программистов, непосредственно писавших ПО. Почему вы тогда решили все изменить?

Аллен: Что ж, вероятно, причин было две. Во-первых, я не была очень хорошим программистом. Я часто делала небольшие ошибки, опровер-

гая тогдашнее расхожее мнение, что из женщин получаются хорошие программисты, поскольку они уделяют внимание мельчайшим деталям. Я была не из их числа. Мне скучно было приводить в порядок все мелочи – гораздо интереснее было разбираться в том, как работают системы.

Математикой я заинтересовалась, потому что меня всегда привлекали абстрактные понятия. Если бы у меня тогда хватило денег на получение степени PhD, я бы стала заниматься геометрией. Мне нравились строгие процессы. Больше всего мне нравится разгадывать внутренние загадки систем – своего рода инженерные загадки, – не обладая детальными знаниями, которые требуются для того, чтобы быть собственно инженером, – это совсем другая область.

Сейбел: Ваш технический вклад в проект PTRAN, судя по всему, заключался в цельности вашего взгляда на архитектуру всего проекта, на его работу; кроме того, вы могли указать на те вещи, о которых нельзя было уверенно сказать, как они будут работать.

Аллен: Точно.

Сейбел: Как по-вашему, эта способность у вас была изначально или развилась со временем?

Аллен: Мне кажется, отчасти дело в том, что я выросла на ферме. Если вы посмотрите на многие интересные инженерные решения, появившиеся в нашей области – в то время или чуть пораньше, – то увидите, что большинство этих идей привнесено теми, кто рос на ферме. Я обнаружила это благодаря коллегам по Национальной академии машиностроения – очень многие из старшего поколения были выходцами с ферм на Среднем Западе. И они с большим интересом занимались разработкой ракет и прочих проектов, связанных с машиностроением, системным мышлением, с тем, что можно было потрогать руками. Думаю, живя на ферме, на природе, я всегда очень интересовалась тем, как делаются вещи и как они работают.

Сейбел: Кроме того, ферма – это большая система устройств ввода и вывода.

Аллен: Верно. И поскольку она тесно связана с природой, то у нее свои циклы, своя собственная система, с которой ничего не можешь сделать. Поэтому находишь себе в ней место – и весьма удобное.

Сейбел: Вы говорили о том, что когда пришли работать над компилятором для Stretch, три из четырех ключевых разработчиков этого проекта были женщины. Вы не могли бы рассказать, как это получилось?

Аллен: Дело было в 1959 году или около того. Женщины играли тогда очень важную роль в программировании. И IBM в этом плане всегда

была очень прогрессивной компанией. Недавно я посмотрела кое-какие архивы – политика равных возможностей IBM берет свое начало аж в 1899 году, то есть эта политика всегда была очень последовательной и открытой – даже тогда, когда этой проблеме почти не уделялось внимания.

Сейбел: Вы считаете, что количество женщин в этом проекте было прямым следствием заявленной политики компании – что в IBM должно работать больше женщин?

Аллен: Не думаю, что это было озвучено как «Мы должны нанимать больше женщин». Критерий был один – наличие квалификации, и под этот критерий попадали не только женщины. Те времена были тяжелыми для афроамериканцев, и IBM здесь проявила себя с лучшей стороны. Рассказывают, хотя это и не широко известно, что в Пафкипси (Poughkeepsie) в то время белые жили отдельно от афроамериканцев, и IBM изменила эту ситуацию.

Сейбел: В одном интервью вы рассказывали историю об одной конференции, когда вы пришли и представились «Фрэн Аллен».

Аллен: А они в ответ: «Вы женщина!»

Сейбел: «А мы вас поселили в одну комнату с Джином Амдалом».

Аллен: Да, помню. Это была конференция IBM, где мы переводили проект System Y, создававшийся здесь, на Западное побережье для разработки продукта и переименовали в ACS. Это была большая конференция – она проводилась в Harriman Estate на другом берегу. Все участники были – кроме, может быть, одного-двух человек – сотрудниками IBM. Администратором был человек с Западного побережья, который никого из нас не знал. Вот и расселил всех в алфавитном порядке.

Сейбел: Но все-таки они с вами разобрались и нашли вам комнату?

Аллен: Да, комнату для горничных, под крышей.

Сейбел: Вы подписывали свои публикации именем Фрэн или Фрэнсис?

Аллен: Ф. Е. Аллен. Не помню, почему я так делала. Думаю, так было принято – по крайней мере тогда – писать только инициалы.

Сейбел: Вы также сказали, что когда начинали работать, считалось, что из женщин получают хорошие программисты – из-за их склонности проверять все до мелочей. А сегодня бытует мнение, что именно мужчины могут сосредоточиться на чем-то одном, как правило, в ущерб всему остальному, и именно поэтому среди программистов так много мужчин.

Аллен: Верно.

Сейбел: Вы, должно быть, наблюдали немало подобных перемен.

Аллен: Да, сегодня уже не говорят, что из женщин получаются хорошие программисты, но они хорошо работают в команде, потому что любят совместную работу. То есть тогдашняя мысль о том, что женщины склонны проверять все до мелочей, сегодня трансформировалась в идею о том, что женщины склонны к совместной работе и работают вместе хорошо.

Сейбел: Несмотря на то что в группе, работавшей над компилятором для Stretch, было много женщин, вам наверняка приходилось работать и в командах, практически полностью состоявших из мужчин. Работа в женском коллективе и в мужском чем-то отличалась?

Аллен: Да. Мне кажется, что отличалась, но дело не только в том, что раньше со мной работало много женщин, – со мной работали мои сверстники, причем в истинном смысле этого слова, ведь мы были практически одного возраста, поскольку в то время, когда на работу была принята я, набирали много работников. Мы были практически ровесниками, с почти одинаковым образованием. Это была очень дружная группа. Кроме того, вся эта область была совершенно новой – в ней было много неизвестного. Чего-то мы не знали, но и вокруг не было много опытных профессионалов, которые знали намного больше нас.

Сейбел: Так что же случилось? В нашей сфере уже не так много женщин. Когда это изменилось?

Аллен: Прошло несколько лет, прежде чем я поняла, в чем дело. Это произошло в конце 1960-х – как минимум в тех областях, в которых я тогда работала. Я перешла из исследовательского центра в проект ACS – и переехала в Калифорнию. Затем вернулась в исследовательский отдел и обнаружила, что рабочая обстановка там очень сильно переменялась по сравнению с тем временем, когда я оттуда уходила восемь лет назад.

Был огромный «стеклянный потолок». Множество процессов в офисе, множество уровней руководства. Изменились структуры руководства, процесс принятия решений стал намного более формальным – особенно в отношении того, какими проектами заниматься и как их делать. Изменилось количество женщин в компании, при этом значительно изменилось и их положение, причем не в лучшую сторону. Разумеется, меня все это не обрадовало.

В начале 1970-х я уже 18 или 19 лет занималась делом, которое меня интересовало и открывало передо мной множество возможностей. Я никогда не стремилась к карьерному росту, но всегда чувствовала, что вольна заниматься тем, чем хочу заниматься, и работать над интересными проектами в подходящих мне ипостасях. И вдруг возвращаюсь и вижу, что это уже не так.

Сейбел: Вам не кажется, что этот стеклянный потолок на самом деле там уже был, просто тогда вы не ударились об него головой? Или что-то на самом деле изменилось?

Аллен: Его действительно там раньше не было. Недавно я поняла, в чем, возможно, был корень этой проблемы: компьютерные науки появились в 1960–1970 годах. И, по большому счету, они вышли из инженерных школ, отчасти из математики.

А в инженерных школах в то время учились в основном мужчины. Своим потенциальным сотрудникам IBM предъявляла определенные требования: наличие определенных степеней и знание определенных предметов в области компьютерных наук. Таким образом, их штат практически целиком состоял из мужчин, поскольку лишь они соответствовали всем этим требованиям – ведь компьютерные науки тогда уже были дисциплиной. Думаю, тогда произошло еще кое-что. Это занятие стало профессией: на рабочем месте шло множество процессов и существовало множество уровней руководства, реализующих эти процессы и следящих за идеальным выполнением всех дел. Это было уже совсем другое место.

Сейбел: Я практически уверен, что в 1950–1960 годы в обществе процветал сексизм. Тем не менее в то же время вы работали в группах, где было много женщин. Почему же эта сфера была тогда такой открытой для женщин?

Аллен: ПО тогда было абсолютно всем в новинку. Да и до сих пор нет твердого определения его как науки. Именно это и привлекало женщин. Раньше женщины работали программистами в ENIAC и в Блетчли-парк. Женщины были вычислителями – это было их название. Но в инженерии, физике и более тяжелых старых науках женщин было не так много. Так роли распределились с самого начала.

Затем женщины стали заканчивать инженерные школы. Сейчас количество студенток последних курсов инженерных специальностей составляет около 20%. У Карнеги–Меллона результат выше, но они прикладывали особые усилия. Что касается компьютерных наук, то здесь результат 8%. На данный момент для женщин нет области хуже компьютерных наук, если говорить о цифрах. «Хуже» не самое подходящее слово, лучше сказать «ниже».

Сейбел: Попробую выступить в роли адвоката дьявола. Почему это так важно, чтобы мы достигли, например, цели в «50 на 50 к 2020», сформулированной Анитой Борг, то есть чтобы к 2020 году в компьютерных науках было занято 50% женщин. Почему это так важно, почему именно эта сфера должна отражать распределение в объеме всего человеческого

Аллен: Эта сфера сильно влияет на изменения во всем обществе. Без участия широкого круга населения результаты нашей работы не станут полезными или привлекательными для всех слоев населения. Одна из наших задач – сделать вычислительную технику и все, что с ней связано, общедоступными. Это в идеале. Но именно к этому мы и идем – работа в MIT над компьютером за 100 долларов и то, как мы пытаемся поощрять торговлю в малом масштабе с помощью вычислительной техники в отдаленных уголках стран третьего мира.

Сейбел: То есть получается, чем ближе мы становимся к конечному пользователю, тем легче предположить, что люди с разным жизненным опытом смогут привнести разные идеи относительно того, как эти пользователи захотят взаимодействовать с компьютером. Опять же, выступая в роли адвоката дьявола, – что вы отвечаете тем, кто говорит: «Это все очень хорошо, когда речь идет о разработке приложений, но кому какое дело до разнообразия точек зрения, когда речь идет о разработке оптимизаций компиляторов?» Так ли важно иметь неоднородный штат сотрудников, даже если вы работаете над самыми что ни на есть техническими аспектами разработки ПО, например оптимизацией компиляторов?

Аллен: Да. На самом деле, это было одно из ключевых качеств группы PTRAN. Многих женщин привлекла работа в этой группе отчасти из-за того, что там уже работали другие женщины. Кроме того, именно это сплотило группу. Благодаря соединению противоположностей, а не тому, что в группе были женщины. Эти люди пришли из разных организаций, с разным образованием.

У меня были сотрудники из Нью-Йоркского университета – из института Куранта, которые по-своему представляли себе выполнение определенных задач, поскольку закончили одну и ту же магистратуру. Было несколько человек из MIT. Одна женщина была особенно сильным теоретиком и обладала очень нестандартным мышлением. У людей из Иллинойса тоже были свои особенности. Помимо половых или культурных различий, сам по себе тот факт, что они приехали из разных городов, усилил эту группу.

Сейбел: Мне кажется, что если когда-нибудь соотношение мужчин и женщин среди студентов последних курсов на специальностях, касающихся компьютерных наук, будет 50 на 50, то мы лишимся этого разнообразия жизненного опыта, ведь тогда все будут обучаться одним и тем же компьютерным наукам.

Аллен: Чем в особенности привлекательны некоторые молодые выпускники, например для компании IBM, так это тем, что не заикливаются на одной дисциплине, а переходят от одной области к другой. Это могут быть абсолютно технические, но очень разные дисциплины. Нередко это делается осознанно: человек хочет увязать несколько крупных

областей знания. Я общалась с рядом таких людей – они видели связь между лингвистикой и вычислительной техникой. Как потенциальные сотрудники компании они очень привлекательны.

Сейбел: Так как же вы сами относитесь к проекту «50 на 50 к 2020 году»?

Аллен: Не очень-то в него верю.

Сейбел: Какие шаги должны быть предприняты для его реализации? Что-то изменить в изучении математики в старших классах? Насколько я понимаю, именно в это время многие девочки перестают интересоваться математикой и наукой, хотя до этого математика им нравилась.

Аллен: Это распространенное мнение, но я так не считаю. Посмотрите на конкурс Вестингауза¹. Там постоянно побеждают девушки. Многие женщины работают инженерами – для этого им приходится изучать разные сложные науки и математику в школе. Ученик моей небольшой школы в Кротоне (Нью-Йорк) занял пятое место в конкурсе Вестингауза по стране. У них очень неплохая научная программа. Шесть из семи участников в старшей категории в этом году – девушки с превосходными научными результатами.

Что на самом деле происходит с этими девушками – так это то, что они идут заниматься общественно значимыми областями. Компьютерные науки, может быть, и являются чрезвычайно важными в общественном плане, но они идут заниматься геологией, биологией, медициной. Вот в медицине скоро будет соотношение 50 на 50. Во многих областях знаний эта теория была опровергнута – но только не в нашей.

Сейбел: Так что же такого непривлекательного в компьютерных науках?

Аллен: Многие считают, что дело в играх и ботаниках, сутками просиживающих перед компьютером. Интересно, изменят ли эту ситуацию новые социальные сети. Не знаю. Но я считаю, что эту проблему должны решать мы. Мы должны не просто требовать от системы образования изменить систему обучения, но сделать нашу профессию более привлекательной.

Мы должны сделать так, чтобы у профессии появилось новое лицо, позволяющее расширить аудиторию, – более человеческое лицо. Нужно сформулировать, почему мы любим свою профессию, почему нам интересно в ней работать, почему нам небезразлично ее будущее и почему работать в нашей профессии – это здорово.

¹ Westinghouse Science Talent Search (сейчас известен как Intel Science Talent Search) – конкурс научных работ для старших школьников. – *Прим. науч. ред.*

Сейбел: Так почему же вы любите ее?

Аллен: Отчасти потому, что здесь каждый день могут появляться новые идеи. Видишь что-то и говоришь себе: «О, этого еще не было». Вся наша сфера обновляется регулярно. Мне очень нравится представлять себе ее потенциал и воздействие, которое она может оказать.

Айзек Азимов как-то сказал о будущем компьютеров (не знаю, прав он или нет), что их деятельность сведется к повышению нашего творческого потенциала. Вычислительная техника запустит век творчества. Это уже можно видеть – особенно что касается средств аудиовизуальной информации. Подростки делают то, чего раньше не могли себе позволить, – снимают фильмы, создают картины. Мы привыкли считать творчеством некий дар, которым человек просто наделен; примерно так в Средние века относились к умению читать и писать, поскольку очень немногие обладали этими навыками. По-моему, мысль о том, что компьютеры повысят творческий потенциал человека, очень воодушевляет.

Сейбел: Вы стали первой женщиной во многом – первая женщина, получившая премию Тьюринга, первая женщина, ставшая почетным членом научного общества IBM. Как вам кажется, были ли женщины до вас, которых незаслуженно обошли этими наградами?

Аллен: Да, безусловно.

Сейбел: То есть, став лауреатом премии Тьюринга, вы подумали про себя: «А ведь есть другая женщина, которой этот приз должен был достаться много лет назад»?

Аллен: На самом деле, первое, что мне пришло в голову: «Как же это здорово!» А потом я стала вспоминать всех тех женщин, которые никогда не были отмечены за свою работу. Нередко их заслуги просто доставались другим. Я думала о женщинах, которые выполняли потрясающую работу, но не были оценены даже своими коллегами и друзьями. Когда я говорю им: «Тебе нужно вступить в профессиональное объединение – я напишу тебе рекомендацию», – они стеснительно отнекиваются.

Сейбел: То есть вы считаете, что проблема их непризнания отчасти заключается в том, что они не заявляют о себе таким образом, чтобы их можно было легко признать и отметить.

Аллен: Верно.

Сейбел: Есть ли какие-то конкретные люди, которых вы хотели бы назвать, чтобы прямо сейчас отдать им дань уважения?

Аллен: Да, например, Эдит Шенберг – она прекрасный специалист в области компьютерных наук. Если говорить о технической работе, неко-

торые из ее работ были по-настоящему новаторскими. У нее крали работы, причем совершенно циничным образом. Она написала работу об отладке параллельного кода, а это очень сложная проблема. На одной конференции эту статью не приняли, а кто-то из комитета конференции сделал из работы Эдит три статьи. Такого рода вещи. В нашей профессии это случается, и у нас нет эффективных способов с этим бороться.

Сейбел: И подобное чаще происходит с женщинами?

Аллен: Да, мне так кажется. Часто про женщин думают, что они просто не станут устраивать скандал, что у них нет связей и адвоката, который рискнул бы выступить против широко известного вора. Это был известный вор, все об этом знали, но никто не хотел иметь с ним дела. И их было много – во времена работы над Stretch. Одна женщина, по сути, изобрела мультипрограммирование, но все заслуги были приписаны человеку, который в итоге стал лауреатом премии Тьюринга.

Сейбел: Вы бы хотели выиграть премию Тьюринга, не становясь первой женщиной-лауреатом этой премии? Газеты кричали: «Женщина стала лауреатом премии Тьюринга», – и это, наверное, слегка раздражало. Если бы вы могли сделать так, чтобы за десять лет до вас премию получила другая женщина, став первой женщиной-лауреатом, а вы бы просто получили ее в свое время, – вы бы согласились?

Аллен: Не могу сказать, согласилась бы или нет. Мне кажется, я получила эту награду по ряду достойных причин. И получила не сразу, поскольку не всегда было ясно, чем же я на самом деле занималась. Я всегда работала в группе, нередко бок о бок с великими, известными людьми. Возможно, это были заслуги кого-то еще – Джона Кока, например, который всегда делился со всеми своими идеями. Многие были удостоены похвал и наград, потому что многому научились у него – как и все мы.

Но я была рада получить эту награду – отчасти потому, что до этого ни одна женщина не удостоивалась такой чести, и было уже пора. Мне казалось, что всему сообществу было слегка неловко – за 40 лет было 50 мужчин, примерно так. Поэтому я понимала, что уже без всяких сомнений настал срок вручить эту награду женщине, и я была абсолютно счастлива стать первой женщиной, ее получившей. Но я старалась не заострять внимание именно на этом аспекте награды. Я старалась говорить о своем опыте, о своей продолжительной карьере и обо всех достижениях.

Сейбел: Каково это – проработать всю жизнь в IBM?

Аллен: То, что я получила работу в исследовательском центре IBM, стало одним из самых счастливых событий в моей жизни, ведь исследовательский центр IBM стоит между производством и наукой. Я как будто

на каменной стене – могу посмотреть по любую ее сторону и найти интересные задачи и возможности и там, и там.

Сейбел: Как вам видится – с вашего наблюдательного пункта на стене: достаточно ли активно и тесно взаимодействуют производство и наука?

Аллен: Несколько лет назад национальный научный фонд опубликовал чудесный отчет с диаграммой на целую страницу, в котором было показано, откуда выросли самые доходные отрасли, такие как графика, Интернет, высокопроизводительные вычисления, транзисторы. Эти производства с оборотом в несколько миллиардов долларов были расположены по оси Y , а по оси X было показано время их зарождения, роль производства (по лаборатории) и роль науки.

Некоторые начинали с производства, некоторые – с науки. Вместе они примерно одинаково повлияли на создание этих отраслей с многомиллиардными оборотами. Мне кажется, что по-настоящему важным является сохранение этого взаимодействия – чтобы идеи, технологии, методы и вложения постоянно курсировали туда и обратно.

В настоящий момент, учитывая курс США на поддержку инноваций, на их важность, думаю, в плане взаимодействия у нас все хорошо – в плане совместной работы и совместного решения проблем. А также решения проблем, которые защищают нас от новых проблем, которые нужно решать, в том числе проблем интеллектуальной собственности.

Сейбел: Нельзя сказать, что IBM в этом отношении абсолютно чиста.

Аллен: Конечно, нет.

Сейбел: У вас наверняка есть несколько патентов.

Аллен: Нет, ни одного. Отчасти из-за того, что раньше на ПО нельзя было получить патент. Другая причина заключается в том, что я часто работала на переднем крае индустрии, а лучшим способом внедрить что-либо в IBM была публикация, и нередко другие компании подхватывали мои идеи. Я была больше заинтересована в том, чтобы реализовать идею в продукте, нежели в том, чтобы получить патент.

Сейбел: То есть это было проще, чем убедить кого-либо из IBM создать продукт на основе вашего исследования?

Аллен: Сейчас мы намного эффективнее работаем в этом плане. Но раньше от хорошей исследовательской идеи до продукта зачастую пролегал очень долгий путь.

Сейбел: Раз уж вы заговорили о том, что получение премии Тьюринга заставило вас поразмышлять над всей своей карьерой, есть ли одно понятие, объединяющее всю вашу деятельность?

Аллен: Думаю, мою карьеру и то, как я работаю, можно описать одним словом: исследование. Я люблю исследовать углы и края – идей, проектов, физической действительности, людей – чего бы то ни было – и мне это очень нравится.

Но есть и другая сторона – я хорошо стартую, но не финиширую. Меня привлекают новые вещи. Компиляторы были просто одной из превосходных областей применения сил, поскольку, работая с компьютерами, я постоянно наталкивалась на новые сложные задачи. И по мере их решения я встречала все больше новых сложных и интересных задач.

14

Берни Козелл

В 1969 году, когда первые два узла сети ARPANET – основы будущего Интернета – были введены в строй, каждый пакет данных, передававшийся по арендованным линиям с пропускной способностью 50 Кбит/с, направлялся через два специальных компьютера, носивших название Interface Message Processors (Интерфейсные процессоры сообщений), или IMP. Эти IMP были спроектированы и созданы компанией Bolt Beranek and Newman (BBN), а управлявшие ими программы писали три программиста, одним из которых был Берни Козелл, ушедший с третьего курса Массачусетского технологического института (MIT) ради работы в BBN.

Первоначально нанятый как программист приложений в проекте по созданию одной из первых систем разделения времени, Козелл быстро перешел к программированию самой системы и вскоре стал «царем системы разделения времени PDP-1», ответственным за завершение кода операционной системы и поддержку работы системы.

За свою 26-летнюю карьеру в BBN Козелл занимался практически всем, заслужив в компании репутацию мастера-отладчика и «умельца», который может спасти любой неудачный проект. Он программировал просто для развлечения: чтобы отточить свое мастерство владения Лиспом, он написал DOCTOR – версию знаменитого виртуального собеседника Элиза – по описанию в журнальной статье автора программы Джозефа Вейзенбаума. Написанный на языке BBN-LISP, широко распространившемся по сети

ARPANET вместе с операционной системой TENEX, DOCTOR Козелла стал популярнее оригинала Вейзенбаума, вдохновив других программистов на новые версии и похожие программы.

В 1991 году Козелл ушел из BBN и купил овцеводческую ферму в Виргинии, где и проживает сейчас с женой Линн, тремя собаками, бесчисленными котами и большим стадом овец. Он немного программирует для местного интернет-провайдера, занимается собственными проектами и читает несколько курсов по программированию и компьютерной безопасности в высшей школе, не желая больше быть профессиональным программистом. Ирония судьбы в том, что после переезда в сельскую местность Козеллу – одному из отцов современного Интернета – дома сейчас доступно только соединение по телефонной линии.

В своем интервью Козелл рассказал о том, как ему удалось заработать репутацию мастера-отладчика программ, о важности написания ясного кода и о том, как он убедил остальных программистов проекта IMP перестать латать дыры в двоичном коде.

Сейбел: Когда вы начали программировать?

Козелл: В старших классах. Не знаю, правда это или нет, но говорили, что наша школа чуть ли не первой в стране обзавелась собственным компьютером. IBM подарила нам свою модель 1620. Думаю, она появилась в нашей школе за год до меня или тогда же, то есть в 1959 году.

Сейбел: В какой школе вы учились?

Козелл: В научной школе Бронкса в Нью-Йорке. Думаю, те, кто учились до нас, работали еще на IBM 650 Колумбийского университета. Но декан математического факультета был чрезвычайно горд, что у них есть собственный компьютер. Он даже написал учебник по программированию, а в то время их было совсем мало. В итоге я отлаживал все примеры его программ. Практически все, что я помню о старших классах, – это то, как я учился программировать.

Сейбел: Как вы тогда программировали? Ассемблер на перфокартах?

Козелл: Да. В основном на перфокартах, но у модели 1620 уже была своя консоль ввода/вывода; ее роль играла электрическая пишущая машинка IBM Selectric. С ее помощью можно было вводить программы. Чтобы вы поняли, что это была за эпоха: в компьютере не было арифметического блока. Для арифметических действий использовалась таблица, хранящаяся в памяти машины. Например, при сложении двух чи-

сел одно из них давало номер ряда, другое – колонки, а результат находился на их пересечении. Частью создания любой программы был ввод в соответствующий раздел памяти таблиц сложения и умножения.

Так что, в принципе, можно было пользоваться пишущей машинкой, но чаще мы набивали перфокарты и загружали их. Для этого был и Фортран, но я его мало использовал и чаще пользовался ассемблером 1620.

Другой полезный навык, который я освоил в старших классах, – подключение оборудования через коммутационную панель. На каком-то этапе у нас появились, кажется, старые печатающие устройства модели 403, и я занимался их подключением. Это было просто, даже тогда, но десять лет спустя, уже во время работы в BBN, этот опыт мне пригодился. Как-то раз нам понадобилось собрать панель и я сказал: «Дайте инструкцию мне». В результате мне удалось подключить по простому протоколу старое печатающее устройство, чтобы оно служило строковым принтером для нашей PDP-1.

Сейбел: Между школой и работой в BBN еще был MIT?

Козелл: В 1963 году я закончил школу и сразу поступил в MIT. Там было сильное математическое направление, включающее довольно бессистемный компьютерный курс. Он все еще был узкой прикладной дисциплиной на факультете электротехники, специализироваться по компьютерам было нельзя. Ребята как раз начали разрабатывать первые системы разделения времени на модели 709 или 7094, или что у них там было в компьютерном центре, а я в то время с головой ушел в математику.

Я прослушал несколько курсов по электротехнике и логике и тот компьютерный курс и вроде во всем разобрался. Я не понимал, что делает действительно хороший программист, потому что был тогда еще ребенком. Но считал, что уже умею программировать.

По-настоящему я погрузился в это, присоединившись к кружку под названием «Клуб технического моделирования железной дороги». Я решил, что это очень интересно, – релейная логика была моим коньком. У них была модель железной дороги, вся построенная на реле и шаговых переключателях. Через этот клуб я познакомился с людьми из RLE – Исследовательской лаборатории электроники. В те времена мы целыми днями сидели в подвале 26 корпуса и набивали перфокарты, чтобы вечером отдать их «компьютерному шаману», который на следующий день вручал нам стопку распечаток. Потом я вошел в проект MAC. И заметил, что вместо математики все больше времени уделяю компьютерам.

А из RLE открывалась прямая дорога в научно-технологический центр Tech Square. Там я познакомился с такими людьми, как Ричард Грин-

блатт и Билл Госпер. Тогда я только начинал познавать этот мир и едва ли представлял из себя что-то как программист. Например, в проект MAC я попал, увлекшись компьютерной игрой «Spacewar!». Причем я интересовался ею не как программист: «Дайте взглянуть на исходный код. Как вы это сделали?» На том этапе я был просто геймером. И мне сказали, что ребята из проекта MAC разработали потрясающую версию Spacewar!, что у них крутые консоли и они могут сколько угодно использовать PDP, так что мне немедленно захотелось стать одним из них. Так я познакомился с Питером Сэмсоном в тот момент, когда он пытался (неудачно) решить грандиозную задачу, связанную с метрополитеном Нью-Йорка, – чтобы по одному билету можно было объехать его весь с максимально возможной скоростью.

Я был, наверное, типичным второкурсником, полностью погруженным в типичные для второкурсника проблемы, и в то же время видел всех этих ребят, специалистов, прекрасно понимающих, что они делают. Писал небольшие учебные программы; одна, помнится, была связана с поиском пути в лабиринте. Лягушка должна была, перепрыгивая с одного листа кувшинки на другой, выбраться из пруда. Помню, как писал эту программу и помогал другим студентам из нашего общежития. Вот чем я занимался в то время, не задумываясь, к чему это приведет.

Оглядываясь назад, я могу сказать, что тогда осваивал азы ремесла программиста. Учился добиваться от компьютера нужного мне результата. Но в этом еще не было искры. Я не осознавал, что делаю; фактически я даже не понимал, как это все работает. Что-то вроде волшебства. Вот чем это было для меня в колледже. Настоящим программистом меня сделала работа в BBN.

Один мой университетский знакомый, уже отучившийся и работавший в BBN, сказал мне: «Давай к нам». Помню, я впервые попал туда в полночь, потому что BBN была безумным местом, где работа кипела круглосуточно, семь дней в неделю. Компания была своеобразным продолжением лабораторий MIT. Каждый мог приходить и уходить когда хочет. И мой приятель как раз работал в ночную смену. Вот мы и наведались туда как-то поздно вечером. Все было загадочным и мистическим; я, по-моему, вообще не понимал, что мне показывают. Вскоре мне передали, что меня хотят нанять. Пригласили на интервью и взяли на работу.

Сейбел: Вы тогда были на третьем курсе MIT?

Козелл: Точно. В сентябре меня взяли на полставки. А уже в начале октября я отчислился и устроился в BBN на полный рабочий день.

Теперь мне кажется, что я не был так уж хорош. Я видел компьютер PDP-1, но понятия не имел, как на нем программировать. И ничего не знал о том, что такое разделение времени. Впрочем, это неудивительно, потому что об этом тогда знали хорошо если человек пятьдесят во всем мире.

В BBN тогда делали совместный проект с Главным госпиталем Массачусетса по автоматизации работы больниц, и меня взяли в этот проект. Начинать я как программист приложений, поскольку это было все, на что я тогда годился. Проработал в этом качестве недели три. Но вскоре стал системным программистом, писал библиотеки, которые они использовали. И некоторое время спустя два системных гуру, парни, создавшие большую часть системы разделения времени для PDP-1, взяли меня под опеку и подключили к своей работе. Той зимой они оба покинули BBN, чтобы продолжить свое университетское образование. И к январю я был «царем системы разделения времени PDP-1» – отвечал за весь проект целиком.

За короткое время меня посетила целая серия озарений. Я внезапно понял все про разделение времени. Понял все про системы реального времени. И как только я все это понял, система разделения времени сразу заработала у меня в голове. После этого все остальное уже было делом техники.

Для своего времени это был весьма амбициозный проект. Идея была в том, что телетайпы модели 33 – шумные, громоздкие и печатавшие только прописными буквами – должны быть установлены в каждой больничной палате. Телетайп в приемной у врача. Телетайп в аптеке. И телетайп в каждом офисе администратора, думаю, тоже. И наша маленькая система разделения времени должна была все это координировать.

Как только пациент входит, его приглашают лечь на кушетку. Доктор сразу назначает анализы. По телетайпу сестре приходит сообщение: «Возьмите такие-то анализы и пронумеруйте их». В лабораторию приходит сообщение: «Проанализируйте эти образцы». Если врач прописывает какое-то лекарство, в аптеке сразу об этом узнают и готовят его.

Дурацкий шумный телетайп в каждой палате был просто чудом. Возня с этими примитивными громоздкими аппаратами была довольно утомительной, и медики бунтовали против нее, но меня это мало волновало. Я витал в системных облаках.

Я решил, что вся система должна быть абсолютно безотказной. Не помню, требовало от меня этого начальство или нет, я сам поставил перед собой такую задачу. Решил, что должен доказать – именно я должен, –

что разделение времени может работать. Что это достаточно надежная вещь, чтобы медики могли ей довериться. Я представлял себе ситуацию, когда пациенту срочно нужно лекарство, а система вдруг «упала». Или, что еще хуже, она потеряла рецепт и пациент вообще никогда не получит свое лекарство. А что будет, если все рецепты перепутаются, а ничего не подозревающие врачи поверят электронике? Так что я решил, что система вообще не должна ломаться. Ее нужно было сделать столь же надежной, как UNIX 30 лет спустя.

Но выполнять отладку в реальном времени было невозможно. Когда система «падала», лампочка гасла, и все. Контрольная панель позволяла считывать и записывать содержимое памяти. Но отладить систему можно было, только узнав ответ на вопрос «Что она делала перед отключением?». Нужно было не запускать программу, а просмотреть список выполненных операций. Так что я рылся в памяти машины, просматривая рулоны распечаток. И постепенно учился делать это все лучше и лучше.

Сейчас даже страшно подумать, насколько лучше. В общем, в итоге начальству пришлось выдать мне пейджер. В те времена это был настолько крутой аппарат, что служебные давали только врачам. Громоздкая неудобная штука, которая умела только одно – пищать. Никакого обмена сообщениями. И он работал только в Бостоне и вокруг него, потому что передатчик был установлен на крыше Пруденшл-центр. Но в радиусе 50 миль сигнал был.

Так я стал чем-то вроде робота: если мой пейджер запищал, значит, возникла проблема и нужно срочно выяснять. Немыслимо, но не имея клочка бумаги под рукой, по телефону-автомату на парковке я мог разобраться в восьмеричных адресах и в конце концов сказать парням на том конце провода: «ОК, теперь вводите такой-то адрес и жмите RUN», – и все начинало работать. Черт меня возьми, если я сейчас понимаю, как мне это удавалось. Но удавалось. Я следил за нашей системой разделения времени добрых два или три года.

Сейбел: Наверное, вам приходилось писать значительную часть кода самому.

Козелл: Так и было. Операционная система была еще не готова, когда я принял проект. Когда Стив Вейсс и Боб Морган уволились, чтобы учиться, она была полна багов и просто недописанных кусков. Мне пришлось доделывать работу за них, и одна из главных вещей, которым я научился в BBN, – как заставлять такие программы работать.

Я действительно считал, что компьютеры устроены абсолютно логично, что всегда можно четко поставить им задачу и что неумению заставить их работать нет оправданий. Как я сейчас понимаю, мне удивительно

хорошо удавалось поддерживать работу системы, добавляя в нее новые куски кода и ничего не ломая при этом.

Так я и заслужил свою репутацию. Я знал, что мой босс и многие коллеги считают меня великим отладчиком. Отчасти это так и было. Но только отчасти.

На самом деле я был очень скрупулезным программистом, самонадеянно верившим, что в нашем деле довольно мало по-настоящему трудно-выполнимых задач. Обычно я просто брал участок кода, который, по-видимому, не работал, и начинал в нем разбираться. И если мне удавалось разобраться в нем, я быстро понимал, что с ним не так и как это исправить.

Иногда попадались такие участки кода – обычно как раз те, которые другим ну никак не давались, – на которые я смотрел и говорил: «Тут все слишком запутанно».

Так что я еще раз читал исходную задачу и переписывал код с нуля. Некоторым из моих коллег, потрясающим программистам, например Уиллу Кроутеру, это не нравилось. Они считали, что таким образом можно, исправив в коде 2 ошибки, добавить туда 27 новых. Но у меня все получалось. Я мог полностью переписать программу, иначе организовав ее, чем это сделал автор, потому что имел собственное видение. Обычно так было проще, во всяком случае для меня. И обновленная программа работала.

Вот так я и заработал свою репутацию, исправляя ошибки, которые больше никто не мог исправить. Хорошо, что меня никогда не спрашивали, как я это делал. Потому что честный ответ, как правило, был: «Я недостаточно разобрался в коде, поэтому просто взял и написал новый».

Я много раз делал так с системой разделения времени для PDP-1. Там были фрагменты кода, при взгляде на которые я понимал, что они либо не работают так, как должны, либо вообще написаны «криво». Так что я их переписывал. Если я не вылетел с работы, то только благодаря хорошей репутации. Если ты в таких вещах недостаточно хорош, то результатом твоей работы будет хаос. Но если ты хорош, люди будут думать, что ты умеешь даже больше, чем на самом деле.

Сейбел: Уход из MIT был трудным решением?

Козелл: Нет. Оглядываясь назад, я удивляюсь, насколько просто это вышло. Я ненавидел учебу, она меня бесила. MIT – место, где испытываешь очень сильное давление. BBN после этого была просто землей обетованной. Это было чудесно. Сотрудники играли с компьютерами, обстановка в компании была безмятежной. Это было похоже на проект MAC даже больше, чем сам проект MAC. В те времена человек запросто

брал на работу свою собаку. Так что по коридорам все время бегали домашние питомцы. Ну, а их хозяева работали днем и ночью.

Я начал работать на полставки, потому что, когда учился в MIT, у меня практически всегда была работа на полставки. И сразу же почувствовал себя на работе, как дома. Просто сам не мог в это поверить. Наплевав на учебу, я ушел из института и устроился на полную ставку. Поработав в BBN, я успокоился и лучше разобрался в себе. Так что следующей осенью восстановился в институте и вернулся к учебе. Вот как это было.

Сейбел: Как вы думаете, обучение в MIT было хорошим дополнением к вашему практическому опыту?

Козелл: Курсы программирования, которые я посещал, будучи старшекурсником MIT, были полезны для понимания общих принципов, но мало чему научили меня в практическом смысле. Работать я учился под руководством коллег в BBN. Целенаправленно меня учил, пожалуй, только Стив Вейсс. Но я понемногу набирался знаний и умений от всех.

Сейбел: Конечно, в те времена учебников по программированию было на порядок меньше, чем сейчас. Но, может, все же вспомните какие-нибудь книги, которые были вам особенно полезны или которые вы бы порекомендовали современным программистам?

Козелл: Мне трудно советовать сегодняшним программистам. На самом деле, я не могу вспомнить ни одного по-настоящему полезного учебника тех времен. Пожалуй, ближе всего было «The Art of Computer Programming»¹ Кнута, которое я когда-то прочитал от корки до корки. Но его едва ли можно рекомендовать как учебный текст.

Сейбел: Вы действительно прочли эту книгу от корки до корки?

Козелл: Это было потрясающе! Я тогда был на первом курсе. Как только выходила очередная часть, мы сразу же прочитывали ее целиком.

Сейбел: Это требует серьезной математической подготовки. Думаете, программистам действительно нужно разбирать Кнута досконально, как вы?

Козелл: Я привел эту книгу в качестве примера. Студентов я бы не стал учить по Кнуту по двум причинам. Во-первых, он использует много математики, не просто рассказывая о разных алгоритмах, но и подробно разбирая, какие из них хуже, а какие лучше. Не уверен, что всем это нужно. Я сам понял не все, но не захотел разбираться дальше. Но понять, хотя бы в общих чертах, когда что работает быстрее, когда медленнее и почему, очень полезно, даже если вы не знаете, насколько оно быстрее или медленнее.

¹ Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008 г.

Вторая проблема состоит в том, что студент, приобретая такое понимание, становится даже немного слишком умным. Он начинает оптимизировать мелкие детали программ, размышляя примерно так: «Здесь лучше применить несбалансированное двойное реверсивное кубическое преобразование A–B, и я как раз давно хотел это сделать». И так он тратит неделю или две, улучшая те куски, которые в этом не нуждаются, программа становится сложнее, что ей не на пользу. Программисту нужно скорее поверхностное понимание всех этих алгоритмов, принципов их работы и использования. Важнее знать, как выбрать правильный алгоритм и заставить его работать, чем то, что этот – порядка $n^3 + 3$, а тот – $4n^2$.

Если станет интересно, всегда можно заглянуть в книгу Кнута, но обычно это не требуется. Что действительно нужно изучить студенту, так это структуры данных. Его не должно шокировать, например, создание связанных списков в Perl. Если знаешь все основные структуры данных, всегда сможешь выбрать нужную. Не всегда самую быструю. Не всегда самую симпатичную. Но всегда ту, которая лучше всего подходит в данной ситуации, потому что тебе известны все альтернативы. Не говорите Дону, но я продрался через зубодробительные вычисления, которые он использовал, чтобы понизить комбинаторную сложность, и практически не нашел им достойного применения. Зато я всегда отлично ориентировался в структурах – и это приносило результат.

Сейбел: Что посоветуете программистам-самоучкам?

Козелл: Программируйте больше. Когда-то мне это принесло огромную пользу. Вспоминая свою учебу, могу сказать, что ничто так не продвигало меня вперед, как практика. Не программирование от нечего делать, а совсем наоборот. «Я должен разобраться в этом, и если так, то почему бы не написать с помощью этого программу?» – вот как нужно думать и поступать.

Никогда не поймешь, как разные части программ устроены и как они взаимодействуют друг с другом, пока не попробуешь сам. Никогда не узнаешь, как НЕ нужно программировать, пока не станешь неделями биться с собственными программами, пытаясь заставить их работать, и видя потом, как хороший программист исправляет все в пять минут. Не думаю, что подобный опыт можно получить в учебной аудитории. Теоретические занятия дают знания, но программирование – это ремесло, и единственным путем к мастерству является практика.

Если повезет, можно учиться на работе. Но даже в рабочей обстановке, когда опыт приносит решение текущих задач, думаю, нужно все время забегать вперед. Уметь делать больше, чем требуется по работе. Например, если требуется написать что-то на Tcl, то в первом приближении

достаточно разобраться в этом языке настолько, чтобы сделать нужный интерфейс. Но правильнее будет в выходные разобрать несколько серьезных программ на Tcl, чтобы к понедельнику уже понимать его устройство досконально.

Сейбел: Насколько часто вы сами программировали ради удовольствия, вместо того чтобы целенаправленно осваивать на практике конкретные методы?

Козелл: Я рассматривал программирование в основном как способ делать полезные вещи и старался развивать свои умения именно в этом направлении. Иногда эти вещи были неисправны, и тогда я мог их починить. Как-то я решил, что неплохо бы разобраться в Лиспе, потому что знал настоящих специалистов по этому языку, а для меня он был тогда по большей части загадкой. Так что я написал несколько программ на Лиспе, и это для меня было гораздо естественнее, чем сидеть под крылышком у Дэна Мерфи и ждать, пока он прочтет мне лекции по CONS, CDR и CAR.

Сейбел: Как вы считаете, есть ли разделы формальной компьютерной науки, которые непременно должен изучить будущий программист?

Козелл: Конечно, и таких немало. Какой-нибудь курс по объектно-ориентированному программированию в абстрактном ключе, хотя во многих учебных заведениях с этим очень плохо. В университете, где я читал лекции, мне пришлось повоевать с другими преподавателями по поводу использования C++ при обучении объектно-ориентированному программированию. Я спрашивал их, уверены ли они, что студенты смогут отделить философскую концепцию объектно-ориентированного программирования от странных особенностей того же C++ как ее прикладного воплощения.

В книге Кнута есть и много других полезных вещей. Я окружен людьми, для которых списки – это что-то из области магии. Они не знают ничего о 83 типах деревьев и о том, чем одни лучше других. Они не понимают ничего в сборке мусора. Они не разбираются в структурах и методах.

Или взять сортировку и поиск. Если в языке программирования нет функции сортировки, они ничего не будут знать о разных методах сортировки, о поиске, о том, когда надо строить индексы, не будут понимать, что значит «наша база данных хранит данные в виде В-дерева». В хорошем курсе обучения, я считаю, нужно не просто объяснить, как, к примеру, создать связный список в Си – это чисто техническая задача, – но и рассказать о принципах работы связных списков вообще.

Сейбел: Среди самых известных проектов, в которых вам приходилось участвовать, был ARPANET. Вы, Уилл Кроутер и Дэйв Уолден писали

программное обеспечение к первым компьютерам IMP новой сети. Как вы попали в этот проект?

Козелл: Наш руководитель Фрэнк Харт относился к своему отделу как к своеобразному инкубатору молодых программистов. Он передвигал людей из проекта в проект. Когда один мой проект заканчивался, он решал, куда меня затем направить. Настоящие консультанты начинали свою работу с командировок в Вашингтон и написания техзаданий; я, однако, был далек от этого. Тем не менее Фрэнк решил поставить меня третьим специалистом в проект IMP.

Осенью 1968 года, когда начали работать Дэйв, Уилли и остальные, я работал над другим проектом. Думаю, мой контракт был делом решенным, но к работе я приступил только в январе. К этому моменту сделано было не так много. Помнится, ребята написали часть кода, но ничего еще не работало. Когда я пришел, Дэйв и Уилли как раз начали вчерне разрабатывать алгоритм работы всей системы в целом и делить фронт работ. Я тоже попросил себе кусок-другой. Как программисты мы все были очень разные, но каждый должен был точно знать, как работает каждая строка кода, в том числе потому, что это была относительно небольшая программа. Сложная, но небольшая.

Я знал, что к моему приходу они не могли далеко продвинуться, потому что процесс ассемблирования у них не был автоматизирован. Им приходилось нести перфоленту в комнату с Honeywell 516 и прогонять через него всю эту перфоленту, чтобы получить распечатку, набивая для этого целую коробку перфоленты, которую они потом несли к другой машине, потому что к Honeywell не был подключен строковый принтер, чтобы распечатать результат ассемблирования. Программировать так было очень утомительно. Первое, что я сделал в рамках проекта, – написал кросс-ассемблер для нашего PDP-1.

После этого на PDP-1 появилась возможность редактировать файлы, ассемблировать их, распечатывать результат, запускать ТЕСО-макросы и так далее. Единственное, что надо было набивать, – относительно небольшая бинарная исполняемая программа, которая предназначалась для машины Honeywell.

Сейбел: Что было самым трудным в написании программ для IMP – заставить их работать быстрее?

Козелл: Интересный вопрос. Так, давайте посмотрим. Мы не слишком много думали о размере программы – предполагалось, что в системе будет достаточно памяти для буферизации. И что кода будет не так уж много. Если бы он был, скажем, на 10% больше, чем он получился в итоге после максимального сжатия, рабочих буферов памяти стало бы не-

многим меньше. Поэтому мы не беспокоились о том, сколько инструкций все займет.

Сейбел: То есть какой объем.

Козелл: Да. Какой объем. Но нас чрезвычайно заботила скорость, поскольку приходилось учитывать пропускную способность. Перед нами стоял вопрос: как организовать систему, чтобы она была отказоустойчивой и сама восстанавливала свою работу, вместо того чтобы умереть?

Второй проблемой было заставить все это работать. В системе было много новых, непроверенных элементов. Мы не знали заранее, будут ли работать протоколы. Уилл предложил несколько новых идей, связанных с алгоритмами маршрутизации, – будут ли они удачными? Таких вопросов было очень много. Например, по поводу отслеживания заторов. Могли ли мы быть уверены, например, в том, что если пользователи всего мира отправят пакеты одному бедолаге, то мы сможем отбрасывать их в правильном порядке, не дав ему утонуть?

Сейбел: С подобными проблемами до вас никто не сталкивался?

Козелл: Совершенно верно. Это был в то же время и исследовательский проект – много новой теории. Было написано несколько диссертаций, многие думали, что разбираются в вопросе. Пришло время воплотить эту теорию на практике. Нам предстояло увидеть, работают ли теория очередей и алгоритмы маршрутизации на самом деле.

Третьим большим вызовом была отладка системы, устранение неисправностей. Например, внезапно пропала связь с городом Цинциннати (штат Огайо). Что случилось? Как это исправить? Звонишь в Цинциннати, а тебе отвечает заспанный ночной охранник – там сейчас три часа ночи. Нужно, чтобы он поднялся и посмотрел на ту мигающую коробочку в углу. Что он расскажет? Что с этим делать? Даже если система вновь заработала, как выяснить, что пошло не так? Как избежать этой проблемы в будущем? Не забывайте, что я считался великим-отладчиком-всего-и-вся, мастером-на-все-руки.

Помню, как был удивлен Уилл, когда я нашел в программе ошибку, которую не мог найти никто. Она скрывалась в одном из модемных протоколов и посылала неправильный пакет в неправильное время. Я сделал несколько заплаток, так что смог поместить маркер в пакет, и когда программа замечала тот самый пакет, она устанавливала в систему заплатку, которая наблюдала за соответствующей операцией и при ее выполнении останавливала работу системы. После этого можно было запустить отладочные программы, чтобы выяснить, что произошло. Как только я все это сделал, поиск бага занял две минуты, так как вредоносный пакет все еще находился в памяти и не был затерт другой информацией.

Была еще одна проблема из разряда неприятных, но не смертельных. Порча памяти из-за неправильного указателя. Сама по себе порча была незначительной, но спустя несколько тысяч машинных циклов программа вылетала, потому что одна из структур данных оказывалась поврежденной. Та поврежденная структура данных использовалась постоянно, поэтому мы не могли просто добавить в код команду «Остановить программу, когда она изменится». Я немного поразмыслил и наконец придумал двух- или трехступенчатую систему заплаток. Как только память портилась, первая заплатка активировала вторую, которая работала с другой частью кода. Это, в свою очередь, активизировало запись третьей заплатки в еще один участок. Наконец, когда эта третья заплатка замечала, что происходит что-то не то, она останавливала систему. Я придумал, как откладывать проверку до нужного момента с помощью такой динамической системы заплаток. К счастью, это сработало, и мы смогли быстро устранить неполадку.

Сейбел: Что наталкивает на подобные удачные идеи?

Козелл: Если я досконально разобрался в системе, будь то ПО для IMP или система разделения времени для PDP-1, несмотря на то, что она мультипрограммная, многослойная и основана на работе с прерываниями, все равно вся ее динамика находится у меня в голове. Я знаю, что и в каком порядке должно происходить и что не должно происходить. Это позволяет всегда представить примерную модель происходящего.

Надо сказать, что многие из возникавших тогда проблем относились к двум разным моделям компьютеров, и их анализ требовал недюжинной фантазии. Например, что-то происходит не так на первой машине, а последствия проявляются на второй. Я не могу остановить работу, первая машина уже обработала на 6000 пакетов больше, чем вторая, и вдруг вторая говорит, что один из пакетов поврежден. Что тогда делать? Всем нам троим приходилось садиться и проследивать весь процесс в ретроспективе, чтобы найти ошибку и «вылечить» систему.

Сейбел: Был ли в систему встроен отладочный код?

Козелл: Нет.

Сейбел: Значит, с каждым сложным багом приходилось возиться отдельно?

Козелл: Насколько помню, мы не добавляли никакого отладочного кода. Сейчас я не устаю повторять: программу нужно писать так, чтобы она была тестируемой. А для того чтобы она была такой, нужно подумать об этом заранее, еще до того, как напишешь первую строку кода. Когда программа заработала, в нее уже поздно вставлять блокировки, утверждения или тестовые модули.

Но тогда мы даже не думали об этом. Мы просто старались написать эту чрезвычайно сложную систему реального времени так, чтобы она работала быстро. Это само по себе было трудной задачей. Мы не закладывали никаких проверок на непротиворечивость; к чему тратить на это время? Так что все исправления вносились потом бессистемно. Загляни в такой-то раздел памяти, просмотри код, чтобы разобраться с тем, или другим, или третьим, потом вернись и начни сначала.

Иногда эту работу удавалось несколько упорядочить. Точно помню, что написал однажды утилиту, позволявшую добавлять в систему заплатку, которая изымала один буфер из оборота и использовала его для фиксирования кода на определенных этапах. Делалось и такое, но тоже без всякой системы. Когда появлялся очередной баг, мы ломали головы, пытались его вычислить.

Нередко, разобравшись, что делает тот или иной баг, можно было сразу указать на часть кода, в которой он сидит. Так что достаточно было изучить ее подробно и подправить. В остальных случаях требовалось собрать больше информации. Подчас мы просто бились головой об стену, пытались вычленивать из всей горы данных ту деталь, что давала ответ. Но в конце концов нам обычно удавалось во всем разобраться.

Нужно помнить, что мы работали на машинах с чрезвычайно примитивным, по современным меркам, управлением. В общем случае заплатка собирала некоторые данные и после этого останавливала машину. После этого приходило время использовать переднюю панель – потому что вряд ли тогда существовал отладчик, который можно было запустить с терминала, не испортив машину. Так что мы считывали нужные участки памяти с помощью передней панели, делая выкладки и пометки, чтобы понять, что, собственно, происходит.

Сейбел: Это были буквально ряды лампочек?

Козелл: Да, ряды лампочек. Одна лампочка – один бит.

Сейбел: И тумблеры для ввода адресов?

Козелл: Да. Я считаю, это лучше. На PDP-1 были тумблеры. На той машине, о которой мы говорим сейчас, помнится, были кнопки.

Сейбел: Как вам троим работалось вместе?

Козелл: У нас были разные стили программирования, тому можно вспомнить множество примеров. Уилл был великолепным интуитивным программистом. Он мог найти способ решения сложнейших задач, которые многие даже не поняли бы.

Для меня он был чем-то вроде виртуального противника в компьютерной игре, причем использующего в качестве оружия Фортран. Дело в том, что и алгоритм маршрутизации, и всю динамическую часть сис-

темы управления IMP собирал Уилл. Одно из важных условий работы системы реального времени – каждой операции должна быть сопоставлена предельная продолжительность, тайм-аут. Нельзя дожидаться чего-то бесконечно долго, потому что понятия «бесконечно долго» в таких системах просто не существует.

И вот подобных тайм-аутов в программе набиралось все больше и больше. Я пытался понять все их в целом, и это было очень трудно. Однажды я решил посчитать все тайм-ауты. Например, тайм-аут подтверждения доставки сообщения должен был в восемь с небольшим раз превышать время прохождения одного пакета данных через сеть. Или тайм-аут прохождения сообщения через сеть должен был равняться максимальному диаметру сети, умноженному на время, за которое пакет передается от одного узла к другому.

И вот я думал: какими базовыми константами пользовался Уилл, собирая все это вместе? Когда два тайм-аута были одинаковыми по длине, было ли это совпадением или то был один и тот же тайм-аут? Кто знает? Сколько нужно внести изменений в код, чтобы изменить одну константу? Если в процессе работы замечаешь, что ждешь чего-то недостаточно долго и операция завершается по тайм-ауту, когда не должна, то понимаешь, что не можешь просто взять и изменить один тайм-аут в программе, потому что там все взаимосвязано.

И вот мне пришлось все точно измерить, чтобы определить минимальное число независимых констант. Отлично помню, как делал это, потому что занятие было довольно рискованным. Приходилось на ощупь пробираться там, где не проходил еще вообще никто, ведь многие из этих констант Уилл выбирал интуитивно, а теперь нужно было заставить все их работать, одну за другой. Если тайм-аут был недостаточно велик, мы увеличивали его – не высчитывали, а просто подбирали, пока все не заработает.

Сейбел: Таким образом вы исправляли ошибки или просто подводили под работу системы более прочную базу, чтобы в будущем не нужно было возвращаться к одному и тому же?

Козелл: Не помню, чтобы я там нашел какие-то ошибки. Но, конечно, в нескольких местах временные промежутки пришлось изменить – просто в качестве меры предосторожности, потому что все работало. Это позволило упростить систему. Для меня немисливо было оставить 200 разбросанных по коду случайных независимых констант, задающих ритм работы сети. Думаю, корректура упростила код, стало проще разобраться в происходящем. Это также позволило нам использовать более осмысленные константы. Диаметр, умноженный на восемь, плюс длительность импульса или что-то такое – гораздо понятнее.

Уилл был чем-то вроде мощного генератора идей. Помнится, я как-то пожаловался Фрэнку Харту, что его все время ставят в новые проекты. Дело было в том, что BBN производила много революционных продуктов, а Уилл как раз был мастером делать то, что еще никто не делал.

Он был не настолько хорош, чтобы выдавать на 100% идеальный готовый код. Но его код был на 75–80% вполне приличным и в большинстве случаев работал. Когда он ушел в другой проект – TIP, если не ошибаюсь, – мы с Дэйвом продолжали работать над системой IMP, и именно тогда я переписал алгоритм маршрутизации, поскольку не понимал, откуда взялись заложенные в нем константы. Это по-прежнему оставался алгоритм Уилла, но переписанный в моем стиле. И теперь я понимал, как и почему он работает, потому что сам заставил его работать.

Одним из наших главных расхождений с Уиллом – я мог часами работать под его скептическим взглядом – была его убежденность в том, что, переписывая программу, порождаешь больше новых ошибок, чем исправляешь старых. Поэтому его блокнот был полон заплаток. Он до последнего готов был исправлять имеющуюся программу, вместо того чтобы ее переписать. В итоге заплатки ставились на заплатки, и вся система настолько усложнялась, что сам Уилл не мог предсказать, как она будет работать. Тяжело было после этого все наладить, хотя, казалось бы, именно для этого и добавлялись заплатки.

Сейбел: Итак, у вас был листинг оригинального исходника, который нужно было скормить ассемблеру...

Козелл: Да. И запущенный двоичный образ. Затем мы с помощью перфоленты – а порой и вручную – вставляли переход на небольшую область, где три исходные строки кода заменялись на эти пять новых, после чего управление передавалось обратно и работа продолжалась как и раньше. То есть при исполнении этого кода машина обращалась к заплатке, выполняла нужные действия, а потом возвращалась обратно.

Сейбел: Значит, перфолента содержала двоичную версию заплатки?

Козелл: Да. Потом уже я написал маленький интерактивный отладчик, выполнявший функции проверки и помещения заплатки в нужное место. Это существенно облегчало работу: можно было нанести на ленту такую программу: «Перейти по адресу 12785, значение, значение, значение, значение. Пустая строка. Перейти по адресу 12832, значение, значение, значение, значение, значение». Если надо было загрузить программу с нуля, то сначала загружалась сама программа, потом – лента с заплатками.

Сейбел: Значит, на том этапе у вас не было никакого исходного кода, который ассемблировался бы в текущее состояние запускаемой двоичной программы?

Козелл: Совершенно верно. Одна из проблем состояла в том, что у нас было несколько вариантов текста программы. В одном из них мог быть помечен участок кода, где две строки нужно вычеркнуть, заменив таким-то кодом. Были ли такие пометки во всех вариантах? Хорошо, что Уилл все записывал в свой чудесный блокнот – именно он и был последней инстанцией. Таков был его подход.

Мой же подход состоял в том, что система всегда должна работать сразу. Я не хочу работать с множеством постоянно вносимых исправлений. Придя в проект, я первым делом проделал большую работу, добавив все необходимые заплатки. Мы работали над этим весь день, потом я всю ночь редактировал систему и реассемблировал ее, и утром у нас была готова новая перфолента, с которой можно было работать. После ночной редакции, как правило, нужно было внести буквально 2–3 изменения, и уже ясно было, каких. Так что сделать это было несложно.

То есть мы почти избавились от такой проблемы, как добавление новых ошибок при исправлении старых, разве что заплатка оказывалась неправильной.

Тут мы с Уиллом здорово спорили, потому что он действительно любил ставить заплатки, стремясь держаться подальше от ассемблера, – отчасти потому, что это занимало много времени, а так он мог, поставив заплатку, работать дальше, а отчасти потому, что не доверял циклу, считая редактирование слишком опасным.

Сейбел: Считаете ли вы работу над IMP одним из своих главных профессиональных достижений?

Козелл: Как ни странно, нет. Да, это был интересный, трудный проект. Но кроме него я написал DOCTOR, программировал на Лиспе, стал «царем больничной компьютерной системы». По-настоящему важной для меня работой была работа над той революционной системой разделения времени, в которой я так тщательно разобрался. А IMP был просто одним небольшим коммуникационным процессором. Там не было столько каналов прерываний, как на PDP-1. Не приходилось решать проблемы вроде такой: как быть, если у тебя всего 32 слота для своппинга, а в системе авторизовано 40 человек?

Работа с IMP принесла нам известность, это было весело и интересно. Написать и отладить некоторые места было действительно непросто. Но я бы не назвал это вершиной своей карьеры. Это была просто еще одна программа. К тому же система IMP была весьма ограничена в применении. Работа же PDP-1 была настоящим вызовом. Это была система разделения времени, которую требовалось постоянно развивать.

Самое примечательное в системе IMP – это скорость, с которой мы ее сделали. Ребята официально начали работать над ней в январе, я при-

соединился в феврале, а в сентябре она была уже готова. Относительно «готова», потому что мы продолжали вылавливать баги и вносить еще некоторые изменения. Но все же в сентябре программа официально была выпущена. Вскоре Уилл перешел в другой проект, а мы с Дэйвом и еще пара новых ребят продолжили дорабатывать систему.

Должен выразить самую искреннюю благодарность Фрэнку Харту. Не знаю, как ему удавалось руководить нами, при этом всегда позволяя оставаться самими собой. Не припоминаю ни одного собрания программистов. Не припоминаю, чтобы нас хоть раз загрузили бумажной работой, когда наши головы были полностью заняты программой и некогда было отвлекаться на все это. Это был такой уровень доверия и уверенности, когда нам троим просто дали эту работу и оставили в покое. Оглядываясь назад, я понимаю теперь, как это трудно быть менеджером проекта и какую потрясающую работу проделал Фрэнк. Никаких еженедельных собраний сотрудников, никакого рисования PERT-диаграмм на доске. Уилл, конечно, отслеживал текущие задачи, найденные ошибки и прочую проделанную работу, но отсутствие какого-либо систематического надзора над нами поражало. Собрать нас вместе, дать нам поручение и уйти в тень – думаю, для руководителя это был очень смелый поступок.

Другой прием, которым пользовался Фрэнк, – ревизии проекта. Тогда мне казалось, что у него они самые серьезные из всех возможных, и со временем мое впечатление не изменилось. У него на ревизии сотрудники тряслись от страха. Это было нечто вроде защиты диссертации. Он собирал в аудитории несколько человек, и ты должен был представлять свой проект. Тебя слушали со всей доброжелательностью, но при этом Фрэнк всегда сразу понимал, в каком месте ты блефуешь.

Думаю, каждый, кто прошел такую ревизию, знает: если ты проработал некоторые части недостаточно хорошо, то во время доклада стараешься побыстрее их проскочить. Тебе кажется, что все в порядке, но на самом деле ты недостаточно все проанализировал и в целом не очень понимаешь, что происходит. У Фрэнка на такие вещи был настоящий нюх, подкрепляемый подбором нужных людей, так что ему без труда удавалось засечь блеф и поймать тебя как раз на том, что ты недостаточно продумал.

Та часть выступления, в которой ты был уверен, не вызывала интереса. Тебе просто говорили: «О!» Но на той части, где ты «плавал», непременно концентрировали внимание. Я знаю тех, кто действительно боялся таких ревизий. Неуверенный в себе программист мог подумать, что его раскритиковали, выставив некомпетентным, и все плохо.

На самом же деле все было не так, и с другой стороны стола это было отлично видно. Ревизия проекта помогала сотруднику двигаться в вер-

ном направлении при работе над проектом. Там, где он чувствовал себя уверенно, он мог разобраться и сам. Но зато лучшие умы BBN могли подсказать ему что-то в тех местах, которые он недостаточно продумал. Почему так получилось? О чем он вообще думал? Что пошло не так? За 15 минут можно было получить реальную помощь.

Нужно быть достаточно уверенным в своих навыках профессионалом, чтобы сказать: «Отлично. Вот моя проблема. Я не знаю, что с ней делать, и когда готовил свой отчет, надеялся, что вы, ребята, этого не заметите и одобрите его». Естественным ответом на это будет: «Конечно, мы одобрим его, потому что он выглядит хорошо. Теперь давай разберемся с твоей проблемой – наши лучшие парни собрались здесь как раз для того, чтобы ты не бился над этим сам еще неделю или две».

На самом деле подобные ревизии нужны самому сотруднику, помогая ему убедиться в правильности решений в случаях, когда он знал, как это сделать, или получить совет там, где он сам понимает свою слабость. Когда я понял это – в мои 20 лет или, возможно, годом позже, – мне стало очевидно, что такая помощь старших товарищей чрезвычайно полезна.

Конечно, для клиента составлялся другой отчет, весь в радужных тонах. Но для сотрудника такие внутренние отчеты были реальной возможностью вырасти профессионально. И меня всегда удивляло, насколько многие люди их боятся. Умный парень может уверять себя и других в том, что его отчет будет порван на клочки. Хотя ясно, что не будет, если в нем есть хоть что-то хорошее, и что тут собрались не его враги. Но убеждать в этом такого парня было совершенно бесполезно. Он все равно пытался пустить всей BBN пыль в глаза, рассказывая о том, как все замечательно.

Не менее трудно было объяснить ему, что в другой раз столько хороших специалистов не соберутся вместе только для того, чтобы помочь ему справиться с его проблемой. После этого придется справляться уже самому, и это замечательный опыт.

Сейбел: Как часто проводились подобные ревизии? Один раз в самом начале проекта или периодически на протяжении всего срока работы над ним?

Козелл: Многократных ревизий не было. Обычно она проводилась один раз сразу после того, как ты разработал проект.

Сейбел: Получается, нужно было знать, как будет выглядеть результат, еще до того, как начнешь программировать?

Козелл: Да, точно. Именно так. Наверное, что-нибудь запрограммировать все-таки придется, потому что многим, включая меня, требуется собрать по кусочку кода, чтобы понять, как все в итоге будет работать.

Но обычно мы работаем в жестко заданном цикле: сначала предлагаем какой-то программный продукт, потом делаем его. То есть мы должны показать клиенту, что собираемся сделать, и показать хорошо, потому что после этого он даст нам деньги и время на работу и должен знать, за что он платит. Итак, на этом этапе мы определяемся с требованиями и пишем себе техническое задание. После этого разбираем его внутри компании, чтобы удостовериться, что нам все понятно. Не помню, чтобы Фрэнк вмешивался в уже начатые проекты. Во всех проектах, где я работал, Фрэнк не интересовался промежуточными результатами работы.

Сейбел: Вы упомянули DOCTOR. Что это за программа?

Козелл: Когда я работал над системой разделения времени для PDP-1, Дэн Мерфи с друзьями тоже работали с PDP-1, заканчивая систему на Лиспе. И я надумал выучить Лисп. Той весной Джо Вейзенбаум написал для журнала «Communications of the ACM» статью о своей программе ELIZA (Элиза). Я подумал, что это круто. И еще я был уверен – и пребываю в этой уверенности до сих пор, – что могу сделать на компьютере все, что сумею понять. Он описал, как работает ELIZA, и я сказал себе: «Спорим, я могу написать такую же». И начал писать программу на PDP-1, на системе Дэна Мерфи в BBN. Телетайп модели 33 в моем компьютерном зале с PDP-1 был подключен к PDP-1 Дэна Мерфи, так что я мог использовать его компьютер из моего компьютерного зала, будто это была моя система. Я писал эту программу, улучшал ее и одновременно работал. Постепенно втянулась практически вся BBN. Ребята оставляли мне комментарии: «Лучше сделай так-то и так-то» или «Я попробовал это, и оно не работает». Это помогло развить идею Вейзенбаума так, как он и не помышлял. Первоначально программа была написана на Лиспе для PDP-1. Но ребята уже программировали на Лиспе для PDP-6 или, может, даже PDP-10. Лисп получил широкое распространение через ARPANET. И, видимо, DOCTOR вместе с ним.

Я впервые был согрет лучиком славы, когда Дэнни Боброу написал мне: «A Turing Test Passed» (Тест Тьюринга пройден). Это было чуть ли не впервые, когда меня заметили из-за моих дурацких программ: мне пришлось прекратить работу над DOCTOR. Один из директоров BBN зашел в комнату с PDP-1, подумал, что Дэнни на связи, и начал переписываться с ним. Мы, кто был знаком с программами вроде Элизы, легко распознавали их ответы, не замечая, насколько они похожи на человеческие. Но тем, кто не имел дела с такими программами, их ответы казались вполне разумными. К сожалению, он и в самом деле подумал, что это был Дэнни. «Расскажите мне еще о...» – «Помнится, вы сказали, что хотите пройти в комнату для клиентов». Подобные фразы казались осмысленными в общем контексте, пока в какой-то момент

босс не забыл нажать кнопку и отправить очередное сообщение, так что программа не смогла ответить. И он подумал, что Дэнни отсоединился. И позвонил ему домой, наорав на него. А Дэнни даже не понял, что происходит. Но он знал о моем терминале. Так что Дэнни пришел на работу и вынул распечатку из телетайпа, чтобы сохранить ее.

То была очень продвинутая версия программы Вейзенбаума. Мы немного улучшили скрипты. Многие поколения хакеров работали над этим. Как я уже говорил, программа кочевала по Сети. Думаю, сейчас уже есть ее версия, написанная в макросах Emacs. Но тогда она стала моим «боевым крещением» в Лиспе.

Сейбел: Вот интересно – я знаю, что часто авторами самых сложных, запутанных программ становятся люди, способные держать в голове миллионы подробностей. Вы, без сомнения, это можете, но все же стараетесь сделать свой код как можно более простым и ясным.

Козелл: Признаюсь, у меня бывало по-разному. В целом, я стараюсь делать программы простыми. Но это не значит, что отдельные специфические аспекты их функционирования не могут быть сложными. Стремясь к цели, я могу написать очень сложный код, настолько запутанный, что другим его даже трогать не захочется. Но он всегда будет инкапсулирован.

Большая часть плохих программ, из которых я выкидывал неудачные места, переписывая их заново, не были такими. В них не было маленького островка сложности, который можно попытаться понять и исправить, – они были запутанными в целом.

Я выработал пару правил, которые всегда стремлюсь привить людям, особенно недавним выпускникам колледжей, думающим, что они знают о программировании все. Первое заключается в том, что есть совсем немного непреодолимо сложных программ. Если вы смотрите на участок кода и он кажется вам сложным – настолько, что вам даже не понять, для чего он предназначен, – то почти всегда это значит, что вы просто мало над ним думали. И не стоит, засучив рукава, пытаться прямо сейчас его исправить – лучше вернуться на шаг назад и попробовать понять еще раз. Когда вам это удастся, вы увидите, что на самом деле все гораздо проще.

Мы часто применяли это правило на практике. Как-то раз мы работали над одним большим проектом, и чем дальше, тем более запутанным он становился. Тогда мы начали разбираться вместе. Я сказал: «Это кажется слишком сложным». И тут вдруг выяснилось, что у нас есть блок-схема, изображающая, как все в итоге должно работать. Посмотрев на нее, все были потрясены: мы сразу поняли, как каждый блок должен выполнять свою задачу. Мы не стали утруждаться и расписы-

вать это все на бумаге, и так было понятно, как скоординировать свои усилия и добиться успеха. Я достаточно давно в этой профессии, чтобы понимать: в ней есть сложности. Но их очень немного. И чем напряженнее человек размышляет над проблемой, тем проще она становится, и в конце концов понимаешь, как на самом деле просто запрограммировать ее правильно.

Второе правило состоит в том, что программы должны быть читабельными. Хотя я, каюсь, в молодости писал страницы макросов ТЕСО, но довольно скоро – наверное, когда работал над системой разделения времени для PDP-1, и ее первоначальная сложность начала понемногу сходить на нет, – пришел к убеждению: исходный код программы предназначен для человека, а не для машины. Компьютеру все равно. Мне кажется, очень правильно, что в Perl есть и «если» (if), и «если не» (unless). Потому что если что-то должно быть сделано при невыполнении какого-то условия, по-английски правильнее сказать «unless», а не «if not».

Компьютеру нужен двоичный код, а текстовый файл нужен мне. Я брал в свои проекты умных людей, действительно хороших программистов, недавних выпускников, лучших на своем курсе. И давал им, этим молодым специалистам, фронт работ. После этого на планерках они спорили со мной: «Почему вы жалуетесь, что я прописал глобальные переменные здесь, что я делаю то-то и то-то, что вам не нравится моя структура подпрограмм? Программа же работает?»

Их удивлению не было предела, когда я отвечал: «Конечно, программа работает. Вас взяли сюда именно потому, что вы умеете писать работающие программы. Написание программ – чисто ремесленный навык, и у вас он есть. А теперь вам нужно научиться программировать». Некоторые из этих ребят, будучи очень хорошими программистами, в жизни не прочли ни строки чужого кода. Фактически некоторые даже свой код не читали и поэтому не понимали, как сильно он потом менялся за какие-то полгода.

Некоторые начинали бунтовать. Они были абсолютно уверены в своих силах, а я для них был просто замшелый старик. Я и сам когда-то удивлялся: если моя программа работает, какие к ней могут быть претензии? Теперь же я сам объяснял другим: «Работающая программа – не оправдание. Это необходимый минимум. Пора переходить на следующий уровень», – а они только охали в ответ. А потом, поговорив с коллегами, понимали, что в BBN это, по сути, стандарт. Ты не можешь создать что-то новое, не усвоив перед этим основы ремесла.

У меня есть свои предпочтения относительно того, как должны быть организованы мои глобальные переменные и подпрограммы. Как-то раз

я ввязался в многодневный спор с одним парнем, который говорил: «Ну посмотрите, все же нормально работает». Он был действительно хорошим программистом, настолько, что мне не хотелось пользоваться служебным положением. Я хотел, чтобы он сам понял, что я не стараюсь таким образом самоутвердиться, и увидел причину, по которой я хотел, чтобы он программировал так, а не иначе. Он просто не понимал, как трудна для понимания программа, в которой только подпрограмма на Си занимает 42 страницы кода.

Сейбел: Ого!

Козелл: Я спорил с ним, потому что сам решительно предпочитаю простые, вызываемые один раз подпрограммы. Единственная цель такой подпрограммы – абстрагировать одну небольшую часть родительской подпрограммы. По-моему, если родительская подпрограмма шокирует своим объемом и сложностью, это верный признак того, что все нужно переделывать. Допустим, у меня есть маленькая подпрограммка, в которой говорится: «Отсортируй таблицу и найди лучший путь», и она вызывается всего один раз. Кто-нибудь оптимизирующий код может сказать: «Это не должно быть подпрограммой. Просто добавь ее в код». Но эту маленькую подпрограммку я могу рассматривать изолированно. Сразу ясно, какие у нее входные данные. Ты видишь алгоритм и можешь быть спокоен, потому, что все понимаешь. Тот парень ненавидел, когда я говорил ему: «Твои подпрограммы слишком сложны. Они занимают много места». Он в таких случаях отвечал: «Все в порядке, потому что я могу сделать это все в одной подпрограмме».

Он сопротивлялся, но в конце концов поступал по-моему. Как-то раз ему нужно было взять большой кусок кода, написанный его предшественником, доработать и встроить в новую версию системы. Он потратил на это почти неделю. Программа того парня настолько вывела его из себя, что он пожаловался начальству: мол, в нашем отделе нет достаточно четких стандартов программирования. А тот его предшественник просто программировал так, как считал нужным, в своем стиле. Так мой коллега понял, что бывает, если хороший, серьезный программист уделяет недостаточно внимания ясности. В итоге получилась одна очень длинная программа – не то чтобы спагетти-код, просто слишком много уровней сложности в одном линейном куске. Парень почти достал меня, обратившись к начальству, как я уже говорил, через мою голову и сказав, что наш отдел работает не по стандартам.

Сейбел: Не понимая, что ранее написанные им программы также не удовлетворяли этим стандартам?

Козелл: Нет. Это он понял. Но уже изменил свои взгляды. Это тот тип людей, которые, бросив курить, начинают громче всех жаловаться на

других курильщиков. Он стал одним из лучших сотрудников в моем проекте. Он сам придирался ко мне, когда я был недостаточно последователен и допускал компромисс в работе. Мой проект стал для него первым проектом подобного рода. Передача данных, реальное время и все, что с этим связано, было ему в новинку. Но он оказался умницей, быстро во всем разобрался и стал очень хорошим программистом, как я и ожидал. Впоследствии я слышал о нем только самые лучшие отзывы. С ним все получалось. Остальные не любили работать со мной, считая меня слишком властным; не могу понять, почему.

Сейбел: Вы придерживаетесь каких-либо правил относительно количества комментариев?

Козелл: Я не добавляю в свой код слишком много комментариев, так как считаю, что он сам по себе должен быть читабельным и ясно выражать твои алгоритмы и мысли. Пишу комментарии к подпрограммам с кратким описанием того, как они действуют и как к ним обращаться – что делать, когда возникают исключения, какова последовательность аргументов и так далее. Но сам по себе код должен ясно выражать, что ты делаешь с его помощью.

Я добавляю комментарий в код, только если инстинкт говорит мне: «Этот конкретный участок кода, хотя и работает, не совсем ясно показывает, что я хочу сделать». Тогда я пишу: «Этот код сортирует таблицу», если по каким-то важным для меня причинам он не очень похож на стандартный предназначенный для этого код.

Я никогда не был фанатом структурированных программ, где каждая подпрограмма сопровождается 18-строчным комментарием и все аргументы идут в строгом порядке. Я не структурирую свои программы настолько тщательно. Какие-то из моих подпрограмм могут быть длинными, другие – короткими. Меня беспокоит общая структура программы, так сказать, расположение фигурных скобок.

Одна из причин состоит в том, что я смотрю на работу кода в целом, а не его отдельных частей. Так, встретив инструкцию `if`, я вижу условие. Я мысленно отвечаю на это условие «да» или «нет», и если хочу пропустить тело инструкции `if`, то код должен позволить мне перейти к тому, что идет после этой инструкции, не особо вникая в синтаксис. То есть я из тех старомодных парней, которые любят, чтобы открывающая и закрывающая скобки стояли одна под другой.

Если все делать по-моему, то код будет выглядеть так: «оператор, открывающая скобка, закрывающая скобка; оператор, открывающая скобка, закрывающая скобка», – это позволяет видеть последовательность операторов. Кроме того, как я уже сказал, если открывающая скобка находится слишком далеко от закрывающей, значит в нее заключено

слишком много всего и это надо оттуда убрать. Иногда я убираю даже не очень большие куски кода, заключенные в скобки, если не могу понять, что эта ветка программы делает и зачем в ней весь этот бардак.

Я всегда стараюсь задвинуть бардак подальше, поместить его в такое место программы, где он не мешал бы понимать структуру и работу кода. Мне тяжело воспринимать некоторые стили программирования, не предусматривающие ясную структуру. Интересно, что у автора языка Python мозги, похоже, устроены в точности как у меня. Он разом избавился от проблем с синтаксисом кода, потому что у него не требуется открывать и закрывать скобки. Если ты видишь инструкцию `if`, то открывающая фигурная скобка неявно будет рядом, как и закрывающая. И если хочешь пойти по коду дальше, то он будет выровнен по этому `if`. Я работаю с Си и Perl, предполагая, что в Python то же самое: можно нажать кнопку, и вся программа сворачивается так, что видишь только ее общую структуру.

Я не сторонник войн вокруг стилей с аргументацией «этот стиль уродливый». Хочется думать, что если я и ввязываюсь в такие войны, то только если это касается моего понимания структуры кода. Такое понимание всегда было моей сильной стороной. Пока вы не докажете, что понимаете код лучше меня, убедить меня в правильности вашего подхода будет чрезвычайно сложно.

Сейбел: Получается, беспристрастный анализ незнакомого кода и его отладка – особые навыки, присущие не всякому хорошему программисту?

Козелл: Конечно. И здесь есть два аспекта. Был еще один парень, Стив Баттерфилд. Он тоже был хорошим отладчиком, но представлял собой полную мою противоположность. В жизни не встречал другого человека, настолько хорошо исправлявшего программы и при этом совершенно в них не разбиравшегося. Он мог взять программу и исправить маленький кусочек где-то у нее в середине, изменив ее работу. В больших, сложных программах Стив мог исправлять какие-то мелочи, заставляя их работать лучше в некотором отношении, но, по-моему, только ухудшая общую картину.

Я же всегда стремился улучшить программу в целом и старался понять ее всю, даже если в исправлении нуждался лишь маленький кусочек. Я всегда разберу все с самого начала, вместо того чтобы сказать себе: «Вот это не работает. Исправь здесь». Так что порой я трачу слишком много времени на то, чтобы загрузить весь код себе в голову, хотя хватило бы быстрого локального вмешательства.

Но когда Стив уходил из проекта, как правило, выяснялось, что править код после него чрезвычайно трудно. Я старался поддерживать все

в хорошем состоянии, но если программа была огромной и ужасной, мне приходилось долго работать впустую, прежде чем приняться за реальную работу. Впрочем, такое случалось нечасто, потому что я, как правило, отлаживал программу иначе.

Как уже сказано, я зачастую не имел ни малейшего понятия где именно находится баг. В какой-то момент я говорил себе: «Этот участок кода должен делать то-то и то-то, но не похоже, что он действительно это делает. Зачем вообще было писать такой сложный код для такой простой операции?» Так что я просто выкидывал этот кусок и заменял его подпрограммой, которая делала то, что он должен был делать. И программа сразу, как по волшебству, начинала работать. Если разобраться, то дело, как правило, было в том, что программа развивалась и ее части должны были развиваться вместе с ней. А автор предпочитал исправить старую подпрограмму, вместо того чтобы заменить ее новой, и где-то ошибался.

Я такое никогда не отлаживал. День или два я набирал новый код, никто толком не понимал, что я делаю, но в итоге все начинало работать. Да он же гениальный наладчик! На самом деле, это очень опасно: Уилл, в общем, прав, и переписывая сто строк кода, рискуешь, исправив один баг, добавить шесть новых. Причем о первом ты хотя бы уже знал в то время, а новые шесть надо было еще найти. Но у меня все получалось: год за годом я переписывал код, и в итоге почти всегда все работало.

Сейбел: Значит, у вас должны быть какие-то стратегии чтения кода. Допустим, исправлять ничего не надо, просто перед вами большой кусок кода, который нужно разобрать. Как вы с этим справлялись?

Козелл: Как показывает опыт, не слишком хорошо. Одна из причин, по которой я скорее перепишу фрагмент кода, чем стану его исправлять, заключается в том, что после какого-то момента я уже не могу его разбирать. Я не читаю код, как книгу. Я пытаюсь понять, что программа делает, и затем ищу подсказки в коде, двигаясь от общего к частному.

Читая программу, я одновременно думаю, как бы я решил связанную с ней задачу. То есть ищу те места, в которых смогу сказать: «О, вот тут программа делает то-то». Нередко после этого я со свойственной мне самоуверенностью могу сказать, что автор написал эту часть неправильно. Или, по крайней мере, понимаю, что было сделано не так, как сделал бы я.

Итак, я двигаюсь от общего к частному. Но я знал и тех, кто поступал наоборот, добываясь отличных результатов. Они начинали читать мелкие подпрограммы и наконец находили среди них нужную. Но я, повторюсь, делал иначе. А именно, смотрел на программу и стремился понять, что ее авторы *хотели* сделать. Зачастую это позволяло исправлять

ошибки, даже не понимая, какие именно ошибки я исправляю. Я доходил до определенного места и говорил себе: «Этот участок кода – как я сейчас понимаю – должен делать то-то»; и после этого выяснялось, что либо он этого не делает, либо слишком запутан и дополнительно нагружен другими задачами.

В обоих случаях нормальная реакция – исправить этот участок кода в соответствии со своими представлениями о том, как он должен работать в программе. Очевидно, что это опасная затея, потому что всегда есть много способов организовать работу программы, и мой выбор может не совпадать с выбором автора. В итоге я рискую разрушить структуру программы и похоронить себя под лавиной багов. Но мне необычайно везло. Обычно, когда я говорил: «Тут ошибка и я ее исправлю», – мне действительно это удавалось. Причем с самого начала моей карьеры.

Работая над своей первой большой программой, системой разделения времени для PDP-1, я был рядовым программистом, решающим проблемы на уровне выпускника колледжа. После этого, в больничном проекте, я быстро прошел все стадии от написания приложений до уровня системного программирования. Имея полугодовой стаж работы, я уже мог уверенно сказать: «Вот этот кусочек своппинга удаленного процесса выглядит неправильно, и я должен его переписать».

Сейбел: Кроме опасности добавления новых ошибок есть еще одна – можно неправильно понять, что программа должна делать.

Козелл: Да, действительно. Надо сказать, что свой путь я избрал когда-то не из малодушия. Когда мне было 19, такое поведение казалось мне единственно возможным. У меня было два отлично служивших мне убеждения: одно говорило, что в программе должен быть смысл, другое – что по-настоящему сложных задач очень и очень мало. Если программа выглядит слишком сложной, то, скорее всего, это значит, что ее автор не вполне понял свою задачу и после долгих мучений получил результат, который только выглядит приемлемым.

Не знаю, с чего это я взял. Я пришел в BBN фактически неопытным новичком, не имея необходимых навыков, но почему-то твердо держался за эти две идеи. Я верил, что могу разобраться во всем и что для меня нет ничего невозможного. И выяснилось, что даже для программ такого уровня, как система разделения времени и система IMP, это работает. Обычно, как только я разбирался, для чего нужна программа, остальные кусочки головоломки сами вставали на место. Ненужная информация просто оставалась за бортом, как детали не того цвета в пазле.

Другой мой принцип состоял в том, что код всегда должен быть ясным и понятным. Я хотел, чтобы все было на своем месте. Исправляя баг в программе, я никогда не исправлял только ту ее часть, где он нахо-

дился. Правило такое: «Если бы ты знал раньше то, что знаешь сейчас об этом участке кода – а именно то, что тут ошибка, – как бы ты тогда написал его?» Почему ты тогда допустил ошибку? Исправь код так, чтобы это больше не повторилось. Когда ты закончишь, каждый переписанный тобой участок должен выглядеть как новый, только что написанный. Я не хочу видеть признаки исправлений или следы ошибок, или загадочные участки кода, говорящие: «Эта подпрограмма каждый раз выдавала неверные значения, и поэтому я должен был ее исправить». Я не хочу видеть ничего из этого, а хочу видеть код, написанный так, будто озарение снизошло на тебя с самого начала.

Плюс к этому у меня есть еще один трюк. Я научился ему, когда работал над проектами министерства обороны. Они никогда не финансируют новые проекты. И правительство, и BBN уже вложили слишком много в текущий проект, хотя, возможно, он ужасен и его надо исправлять. Назначение программы, требования к ней или что-то еще стало другим – и вот то, что ты на первых этапах сделал совершенно правильно, теперь, в новых условиях, лучше будет переделать. Ты к этому готов, но начальство спрашивает: а зачем нужны исправления? Там какая-то ошибка? Никакой ошибки, отвечаешь ты, просто это сделает программу лучше. Будь уверен, что разрешения ты не получишь.

Мне пришлось придумать довольно хитрый метод борьбы с такими ситуациями, который ни разу меня не подводил. Суть его в том, чтобы всегда продумывать заранее будущую версию программы. Через какое-то время понимаешь, что именно так программа и должна выглядеть – на уровне общей структуры, а не подпрограмм. Теперь, когда исправляешь ошибки, исправляй их так, чтобы приближать программу к этому будущему образу. Не иди кратчайшим путем. Вместо этого развивай программу так, чтобы несколько месяцев спустя она представляла собой не устаревшую основу, пестрящую заплатками, а реализацию нового подхода к ее написанию. Обычно в какой-то момент в программе остается всего несколько мест, работающих по старому принципу, так что ты можешь спокойно заменить их, не причиняя вред программе в целом.

Так что на вопрос «Сколько времени потребуется, чтобы внести изменения?» есть три ответа. Первый – это кратчайший путь, когда заменяешь одну строку кода. Вторым путем основан на моем простом правиле: перепиши кусок так, как будто ты с самого начала не собирался ошибаться. Третий же и самый длинный путь – это переписать кусок так, будто пишешь его для уже усовершенствованной программы. Ищешь разумную середину между вторым и третьим вариантами и в результате получаешь дополнительное время для совершенствования программы. Думаю, это очень важно. Так программа может развиваться посте-

пенно. Здорово, конечно, иметь дело с программой, которая всегда остается в одной и той же самой первой версии, но это как жить в каменном веке. Зато в моем варианте мы в итоге получаем новенькую с иголочки программу, причем без разрешения менеджера проекта на эти работы.

Сейбел: Вы когда-нибудь слышали о рефакторинге?

Козелл: Нет, а что это?

Сейбел: Как раз то, что вы сейчас описали. Думаю, сейчас такой подход получил большую популярность даже среди менеджеров проектов.

Козелл: О, это замечательно, потому что мне всегда нужен был баг в качестве предлога для изменения какой-либо части программы. Получить разрешение на доработку, если все в порядке, было нереально. Так что приходилось ждать, пока будет обнаружена ошибка или сверху придет запрос на внесение изменений, затрагивающих именно эту часть кода, и тогда можно было улучшить программу легально. Думаю, в рефакторинге главное – правильно понять, к какой цели нужно двигаться. Если люди, придерживающиеся разных точек зрения на этот вопрос, будут одновременно вносить в программу изменения, ничего хорошего не получится.

Я не придумывал своему методу никакого специального названия, а просто ставил перед собой две задачи: во-первых, разобраться со сложностями, во-вторых, получить в итоге такую программу, которая потом не вызовет желания переписать ее заново. К этому я пришел, работая с PDP-1. То был большой многолетний проект. Понадобилось три или четыре человека для написания первых двух версий системы, и эти версии нельзя было просто выбросить, но их требовалось улучшить.

Сейбел: Как вы нанимаете программистов? Как определяете, есть ли у них талант?

Козелл: Никогда не понимал стандартного подхода к собеседованиям. Говорят, им дают маленькие задания – кажется, этим особенно прославилась Microsoft. Мой подход можно назвать более интуитивным. Первым делом я просматриваю резюме, стараясь понять, мой ли это тип человека. Часто в резюме вообще нет никакого смысла, так как на работу приходит старшекурсник. Между строк легко прочесть, что такой-то красивый с виду проект на самом деле был учебным заданием по одному из курсов. Но можно поговорить с кандидатом и понять, присущ ли ему тот пытливый, любознательный, дотошный склад ума, который я так ценю в людях.

Какие у него есть интересы помимо профессиональных? Способен ли он схватывать на лету, любопытен ли? Такие вещи видны сразу. У меня сложился некий идеальный образ работника в стиле VBN, который включает в себя сообразительность, любопытство, обучаемость, заин-

тересованность и широту кругозора. И я всегда присматривался, не такого ли типа человек передо мной.

Сейбел: Как вы сказали, в Microsoft любят задавать кандидатам головоломные задачи. А вы любите головоломки? Как по-вашему, могут они дать представление о потенциале человека?

Козелл: Думаю, могут, если их правильно подобрать. Дело даже не в том, сможет ли человек их решить, просто сразу будет видно, как он умеет добиваться поставленных целей. Сам я никогда этот метод не применял. Не стал бы давать кандидату на должность игрушку-головоломку и смотреть, как он ее собирает. Проблема в том, что разные типы головоломок нужно решать по-разному, и обычно ты либо знаешь правильный способ, либо нет. Не брать же человека на работу только потому, что он виртуозно разгадывает головоломки.

BBN зачастую работала на самом переднем крае, делая то, что никто раньше не делал и в чем мы, ее сотрудники, сами не сразу разбирались. Чтобы стать первопроходцем, нужно было обладать как смелостью, так и умениями. Вот что я сам ищу в людях: умение не решать головоломки, а действовать в нестандартных ситуациях.

Хорошим примером может послужить история с кубиком Рубика. Когда эта головоломка появилась в Европе, до нас начали доходить слухи о ней. И вот один из наших сотрудников привез из командировки в Англию целую сумку этих кубиков. Никаких инструкций или готовых решений, и во всех Соединенных Штатах их тоже никто не знал. Просто необычная головоломка. И мы начали ее решать. Вскоре несколько наших сотрудников смогли собрать кубик, причем каждый, что интересно, сделал это по-своему. Теперь очевидно, что это были как раз самые талантливые люди в BBN. Таких людей я и ищу.

Не знаю, могут ли головоломки, которые дает Microsoft, – или тесты на сообразительность, которые, как я слышал, устраивает Google, или еще что-то такое, – показать, обладает ли человек нужными качествами. Но вот на что смотрю я. Готов ли человек работать в стиле BBN? Часто я вижу, что нет. Это может быть замечательный человек и гениальный программист, но в нем нет искры. Я всегда искал искру и едва ли смогу точно объяснить, как.

Сейбел: Думаете, программирование – это занятие для молодых?

Козелл: Возможно. Оглядываясь, я вижу, что люди, работавшие под моим руководством в конце моей карьеры в BBN, делали то, что я бы, наверное, уже не смог. Один из моих подчиненных решил, что для создания части интерфейса будет здорово использовать Tcl, и всего за полтора дня освоил этот язык настолько, что смог сделать всю необходи-

мую работу и сделать хорошо. Думаю, мне бы это было не по плечу. «Ух ты, а ведь и я так умел», – подумалось мне тогда.

Очевидно, написание кода – хорошего, рабочего, логичного кода – требует живости и гибкости ума, умения усваивать новое, что мне сейчас дается уже с трудом. Правда, с возрастом накапливаются знания. Сейчас я лучше знаю, как выполнить ту или иную задачу. Так что для меня лучше учить молодых и энергичных. Думаю, с программированием в общем та же история, что и с математикой: большинство великих математиков написали свои лучшие работы до тридцати. Та же энергия, та же концентрация, которые позволяют добиваться успеха в математике, нужны и для того безумно интенсивного программирования, которым я занимался в молодости.

Сейбел: Во многом энергичность – это просто физическая способность работать по много часов в день. Это необходимость или просто одно из последствий увлечения работой?

Козелл: Думаю, это индивидуально. Кто-то может отложить работу и вернуться к ней спустя некоторое время, кто-то нет. В BBN было много блестящих профессионалов, которым вполне хватало рабочего дня и которые не проявляли желания трудиться в выходные. Были, конечно, и обратные случаи – так, я одно время спал прямо в компьютерном зале, не желая тратить время на дорогу домой и обратно. Мне было все равно, насколько сумасшедшим меня из-за этого считают. Но не думаю, что такие подвиги необходимы, – скорее это побочный продукт нашей полной вовлеченности в работу.

Один из лучших сотрудников BBN умудрялся работать по совершенно нормальному графику и при этом закончить докторскую диссертацию просто за счет своей феноменальной организованности. Каждую субботу он целиком посвящал научной работе, дополнительно возвращаясь к ней в будни по вечерам. Думаю, организованность – важное слагаемое успеха. Гораздо проще делать что-то, имея четкий план, откладывая работу в сторону и вновь приниматься за нее в определенное время, зная, что в нерабочее время можешь о ней не думать. Я понял это недавно, поскольку сейчас моя жизнь стала более размеренной. Сам я программирую достаточно бессистемно, откладывая работу и снова принимаясь за нее позже. Я обнаружил, что если перерыв в работе над какой-то программой составляет больше двух недель, включиться в нее снова бывает очень трудно. Часто, когда я работаю над каким-то маленьким личным проектом и мне действительно хочется его закончить, я говорю сам себе: «Ладно, попробую составить график. Буду работать по два часа каждое утро». И это почти никогда не срабатывает. В какой-то момент мне надоедает такой темп, я сажусь и заканчиваю все за день-другой непрерывного труда.

Я еще могу концентрироваться на работе, но уже не так хорошо, как когда-то. Вот почему я думаю, что-то увлеченное, вдохновенное программирование, свойственное настоящим хакерам, – это в значительной степени занятие для молодых. Приходится признать, что почти все, кого я знал, кто сделал в молодости какие-то потрясающие вещи, работали над ними интенсивно. Мне трудно представить настоящий шедевр, который его автор делал бы по два часа каждое утро, как рутинную работу. Обычно такие вещи достигаются безумным напряжением сил. Это тяжело, это выматывает. И я в конце концов вымотался.

Сейбел: Вы бы назвали себя ученым, инженером, художником, ремесленником или кем-то еще?

Козелл: Скорее, я что-то среднее. Не считаю себя ученым, так как не считаю свою работу научной. Правильным ответом будет сочетание художника и ремесленника. То, как я программирую, – это комбинация искусства и ремесла.

Сейбел: Начнем с инженерной составляющей. Многие, как Уоттс Хамфри и сотрудники Института программной инженерии (SEI), считают, что программирование нужно преподавать как техническую дисциплину, вроде строительства мостов. Человека можно научить строить мосты, рассчитывать, сколько времени на это потребуется, и их мосты, как правило, не рушатся.

Козелл: Именно так. Хорошая аналогия, хотя и не безупречная. На самом деле, парень, который проектирует мост, так чтобы он не обрушился, сам не проверяет стальные канаты на прочность, не замешивает бетон и не делает много другой грязной работы.

Но в определенном смысле программирование действительно является инженерной дисциплиной. Ты должен знать, что делаешь. Трезво оценивать свои возможности. На своем уровне я должен был уметь охватить взглядом все детали, чтобы собрать их воедино. Должен был интуитивно понимать, что будет работать быстро, а что медленно; что будет легко сделать, а что – нет. И в результате, подобно инженеру, подготавливать проект.

Художественная составляющая заключается в том, что программа должна быть элегантной. В нашем деле это важно, так как искусностью написания программы определяется ее долговечность. Часть того, что я называю художественным программированием, – это искусство написать программу так, чтобы твои последователи могли легко изменять ее, не разрушая. Это не имеет ничего общего с функциональностью, но тем не менее определяет будущую жизнь программы.

Сейбел: Итак, для вас красота кода тесно связана с тем, что люди будут его изменять.

Козелл: Одна или две моих программы были эдакими «черными ящиками», от которых требовалось только одно – работать, пока компьютер включен. Но остальные представляли собой код, с которым поколения программистов могли потом возиться, не выбрасывая на помойку. Когда я говорю об искусстве и красоте, это значит вот что: приступая к написанию программы, вы имеете большую свободу действий. Как вы организуете ее функции, как расположите их, где добавите комментарии, как назовете переменные, будут ли одинаковыми вызывающие последовательности.

То есть вам приходится смотреть на программу глазами другого программиста, который будет работать с ней сколько-то лет спустя. Какова структура программы? Что она делает? Как она это делает? И почему? Работа художника – это когда тот, следующий парень читает программу и понимает, что вот эта подпрограмма предназначена для того-то. Когда он понимает, что лучше не выбросить вашу программу, а поработать с ней еще, оставив структуру как есть.

Сейбел: А как быть с противоречием между ясностью и эффективностью? Подчас простой, легко читаемый код не является самым быстрым.

Козелл: Программисты – худшие оптимизаторы в мире. Они всегда оптимизируют то, что им интересно оптимизировать, и почти никогда то, что действительно в этом нуждается. В результате получаются островки бесполезно сложного кода. Я всегда говорю тем, кто работает со мной: «Программируй так легко, прозрачно и просто для понимания, как можешь. Будь проще. Со скоростью разберемся потом, если понадобится. Если ты все сделаешь правильно, внести доработки будет несложно».

Когда-то давным-давно в исходном коде одной из версий Emacs была страница с большим черепом и скрещенными костями в комментариях, где говорилось примерно следующее: «Здесь очень запутанный код». То был кусок в самой глубине поискового кода, или чего-то такого, и он был заоптимизирован до смерти. Исключительно сложный для понимания код. Как черный ящик с надписью «Не влезай, если не уверен».

Но если поглядеть на программы, которые мы пишем сегодня, через призму моего опыта, они гораздо больше, уродливее и к тому же медленнее. И это нормально. Сейчас подобные вещи уже не имеют значения. Ребята, которые делают сегодня программы для сведения видео или для компьютерной анимации, просто не могут позволить себе такую роскошь. Для этого требуется много очень аккуратного программирования. Я уже не могу делать такие вещи. Но когда-то это было мне по силам. И я понимаю нынешних программистов. Большая часть программирования сегодня – рутина.

В одном колледже курс программирования длился два семестра, с сентября по май, и уже в самом начале студент должен был составить довольно трудную программу. Но он не подозревал, что в апреле ему придется делать это снова, причем на этот раз к его работе будут предъявлены гораздо более жесткие требования. Идея была в том, чтобы студент понял, насколько трудно вспомнить вещи, о которых полгода назад думал, что хорошо их понимаешь.

Сейбел: То есть все, что ты когда-то лихорадочно делал в последнюю ночь, возвращается к тебе во всей красе.

Козелл: Совершенно верно. По-моему, прекрасная схема. Отлично готовит к трудностям реальной жизни.

Сейбел: Беседуя с Кеном Томпсоном, я спросил его, есть ли у языка Си врожденные пороки, которые привели к проблемам с безопасностью. И он сказал, что на самом деле все было в порядке. Преподавая компьютерную безопасность, как вы прокомментируете это?

Козелл: Не хотел бы скрещивать с ним шпаги, но в своем курсе говорю, что самые большие проблемы с безопасностью у современных компьютеров бывают как раз из-за Си. Он был создан как язык системного программирования и оказался таким удачным, что его стали использовали во всех важных проектах. Мы пишем на нем операционные системы и системы реального времени.

Я помню, какие войны кипели в эпоху Паскаля. Говорилось, что компьютер должен сам помогать тебе; что Си слишком опасен как язык программирования. Помнится, эту позицию отстаивали два авторитета – Вирт и Дейкстра. По другую сторону баррикад находились все известные мне системные программисты, включая меня самого. Я писал все на Си. Этот язык, можно сказать, смел тогда всех конкурентов.

Правительство пыталось поддержать Аду, заключая контракты только на написание программ на Аде. Си пробил и эту стену. Он был просто великолепен. Но и сегодня я не перестаю удивляться тому, что в Си невозможно написать сколько-нибудь сложную программу, не столкнувшись при этом с проблемой безопасности. Этот язык требует от программиста много возни: необходимо каждый раз при копировании информации в буфер тщательно проверять, не выйдет ли она за его пределы; следить, чтобы ни в коем случае не стереть информацию из памяти не вовремя, чтобы указатель где-то в программе не стал неактуальным; следить за тем, чтобы не сохранить что-то, имеющее неправильный размер и способное затереть следующую переменную; эти проблемы нелегко бывает выявить впоследствии.

Но когда-то это было просто спасение для системных программистов. От мысли о том, что мы могли бы писать системы на ассемблере, а при-

ложения – на Паскале, у меня мурашки по спине бегут. Не думаю, что это было бы правильно. Но имея опыт написания как систем, так и приложений на Си, должен сказать, что этот язык не оправдал наших ожиданий. Работать в нем оказалось слишком сложно.

Пример тому – ошибки в программах с прерываниями. Вы можете сказать, что в написании программ с прерываниями ничего сложного. И это действительно так. Нужно всего лишь немного понимания и немного старания. Но я знаю случаи, когда по-настоящему хорошие программисты, обладающие и тем, и другим, делали ошибки в своих программах. Таким, как я, приходилось исправлять эти ошибки, и в конце концов я написал язык в стиле Никлауса Вирта, не дающий программистам ошибиться при работе с прерываниями.

Для системы IMP я написал сложный набор ассемблерных макросов, позволявших объявлять, что делаешь. Дойдя до прерывания, пишешь объявление вроде: «Я обрабатываю данные из модема» или «Я в высокоприоритетном или низкоприоритетном таймере». И когда программа ассемблируется, для каждой инструкции проставляется метка и на каком уровне прерывания она выполняется, а затем запускается постпроцессор, написанный мною, кажется, на макросах ТЕСО, и проверяет, нет ли проблем с разделением времени. Если он обнаруживает переменную, доступ к которой осуществляется из двух разных уровней, то пишет: «Обнаружен конфликт прерываний». И ошибка с разделением времени сама собой исправляется. Программист быстро привыкает к тому, что если он пишет правильные объявления, система сама удерживает его от ошибок. Я специально ездил в Венгрию, чтобы показать, как не разбирающиеся в проблемах разделения времени программисты могут с помощью этого метода писать соответствующие программы на хорошем уровне.

Вот, если вкратце, что я думаю о Си. Уверен, что есть хорошие программисты, возможно, и я в их числе, которые могут писать на этом языке хорошие программы. Но это дается нам труднее, чем должно было быть. В современном мире эта проблема усугубилась, потому что вокруг становится все больше и больше программ на Си, и у программистов все больше работы по преодолению их недостатков. Это одна из причин, по которым мне нравится программировать на Perl. Perl медленный. Думаю, это один из самых медленных языков программирования, но зато в нем нет тех проблем с безопасностью, которые есть в Си. Что происходит в Perl, если информация не влезает в массив? Массив увеличивается в размерах, только и всего.

Этот язык знает, куда указывают его указатели, так что вы никогда не сможете «сбиться с пути»: нужно просто сказать системе, чтобы проследила путь и узнала, куда он ведет. Я уверен в этом языке, потому

что на протяжении многих лет люди работают с его ядром, и оно очень стабильно. Не думаю, что в написанных на нем программах попадаетесь много ошибок выделения памяти или проблем с указателями; фактически их вообще трудно допустить в коде Perl. Не приходится надеяться только на программистов.

И даже после этого бывают случаи вроде того хрестоматийного, когда кто-то написал веб-страницу, искавшую имена в таблицах, а неизвестный хакер написал в строке ввода что-то вроде «Joe;drop all tables». Такое еще случается. Это, конечно, не вина языка Си, но видно, что программисты подчас бывают недостаточно осторожны. Они не продумывают все варианты, а в Си их слишком много. Все это кажется мне слишком опасным. Справедливости ради нужно добавить, что я программировал на Си на пять лет меньше Кена. Мы с ним в разных весовых категориях, но все же у меня большой опыт общения с Си, я знаю, как это непросто, и думаю, что дело тут, по большей части, в самом языке.

По мере того как приложения будут становиться все более и более сложными, построенными на все более и более сложных библиотеках, – и все окончательно перестанут понимать, где в них «дыры» из-за их чрезвычайной сложности, – наверное, нам придется перейти к программированию приложений на менее располагающем к ошибкам языке. Процессоры становятся потрясающе быстрыми, а память – удивительно дешевой. Не могу сказать, какой он – завтрашний язык программирования. Не думаю, что Си или его дальнейшие модификации, вроде C++, станут подходящей платформой, на которой программирование приложений – даже развитие систем – сможет двигаться дальше.

Java не кажется мне выходом. Мои старые рефлексy не подводят. Этот язык неприятно поразил меня своей авторитарностью. Это еще одна причина, по которой я хвалю Perl, – он безопасный, надежный и при этом чертовски разнообразный, так что художник во мне получает достаточное поле для самовыражения и выбора оптимальных путей. Я свободен.

Впервые попробовав Java – еще совсем молодой тогда язык, – я сказал себе: «А, еще один язык, призванный помочь посредственным программистам встать на путь истинный, ограничивая их». Но, возможно, сейчас это и правильно. Пожалуй, сегодня опасно полагаться на хороший, гибкий язык, на котором один или два процента программистов смогут писать высокохудожественные программы, потому что в мире уже 75 миллионов штампованных программистов, пишущих чрезвычайно сложные приложения и нуждающихся в помощи. Так что, вероятно, Java – это то, что нужно. Я не знаю.

Сейбел: До вас я брал интервью у Фрэн Аллен, сотрудницы IBM, писавшей компиляторы для Фортрана. Ее Си разочаровал совсем другим – тем, что для него невозможно написать хороший оптимизирующий компилятор, потому что язык слишком низкоуровневый.

Козелл: Ну, она из другого лагеря. Она создает компиляторы и видит Си как ужасный, неудобный этап, от которого никуда не денешься. Когда мы работали с ассемблерами, жонглируя битами, Си был просто глотком свежего воздуха. Понятно, что большинство лучших программистов того времени работали не с Бейсиком и не с Фортраном. Самыми крутыми были, конечно же, те, кто писал ассемблерный код. И мы сразу же перешли на Си – это было как небо и земля. Если вы думаете, что это у Си проблемы с границами массивов, попробуйте написать цикл по массиву на ассемблере. Так что здесь это был настоящий прорыв.

Я не хочу сказать, что Си выработал свой ресурс. Но мне кажется, что его использовало так много хороших программистов, что планка стала очень высокой, и их более посредственные коллеги, пытаясь сегодня писать приложения на Си, просто не справляются. Наверное, Си – идеальный язык для по-настоящему хороших системных программистов, но увы, его много используют и программисты похуже, а не стоило бы.

Сейбел: Не кажется ли вам, что суть программирования изменилась из-за того, что мы больше не знаем, как это на самом деле работает?

Козелл: О, да. Это еще одна причина, из-за которой я выгляжу живым динозавром. Ничто не появляется на пустом месте, а развивается из чего-то. Я помню, как на PDP-11 с седьмой версией UNIX мы делали кое-какую анимацию и графику. Это было по-настоящему трудно программировать. Мониторы были громоздкими. Не было библиотек.

Каждое поколение программистов уходит все дальше от того старого оборудования и получает в распоряжение все более совершенные инструменты. Это хорошо, потому что расширяет их возможности. Базовый уровень все выше, а следующий выглядит еще лучше – а через пару лет он сам становится базовым и так далее. Проблема в том, что сложность тоже возрастает. Набор инструкций PDP-1 покажется сущей ерундой по сравнению с тем, что происходит сейчас.

Я бы не хотел оказаться на месте тех ребят из Microsoft, которым приходится создавать операционную систему для четырехъядерного мультипроцессора. Видеокарты развились до такого уровня, что имеют множество мегабайт памяти, а их встроенные процессоры обрабатывают массивы и векторную графику на лету. Сегодняшняя видеокарта – фактически очень мощный обработчик данных. Мне трудно представить, как это программировать.

У нас была когда-то такая вещь, как IMLAC – одна из первых машин, имевших интегрированный векторный дисплей, подобный установленному на старом PDP-1, но в отличие от него это был мини-компьютер. Для него была игрушка: едешь в маленькой машинке по трехмерному лабиринту и видишь, как приближаются стены, заворачиваешь за углы. Помню, я был поражен тем, что она удаляла невидимые линии. Это было в ту эпоху, когда журнал «Communications of the ACM» публиковал статьи об алгоритмах. У меня была целая книга о том, как использовать симметричные координаты; и я видел написанный кем-то алгоритм, вычислявший, где пересекаются две линии, так что можно было узнать, где линия пересекает плоскость, то есть место, где ее надо уже прекратить рисовать, потому что дальше она не видна.

Работать со скрытыми линиями в те времена было очень трудно, а та программа делала это. Я был просто поражен. Это было что-то уникальное. Могу сказать, что сегодня видеокарты легко оперируют трехмерными координатами и удаляют невидимые линии. Восемь, девять лет назад наложение текстур и трассировка лучей были чрезвычайно сложными, трудно программируемыми задачами. Нужно было потратить часы, чтобы нарисовать блик на сфере.

Современные видеокарты, насколько мне известно, умеют делать трассировку лучей. Так что сегодня, с одной стороны, мы видим разработчиков NVIDIA и подобные чрезвычайно сложные программы обработки видео, с другой – современного программиста, которому мало нарисовать двумерную стену, а нужно создать полноценную 3D-среду, основанную на библиотеках, которые становятся все более и более сложными. Работать с ними проще, чем писать код самому, но я все равно не понимаю, как люди справляются. Для меня это слишком сложно.

Я ощутил это, поработав с Tk. Попробовал написать на Tk небольшую программку и был поражен сложностью этой библиотеки и количеством подводных камней, с которыми сталкиваешься, даже прописывая, например, размер кнопки или ее расположение. Это очень сложная работа. Разобраться в системе разделения времени для PDP-1, для сравнения, было гораздо проще.

Так что я не завидую современным программистам, им все труднее и труднее. Простые вещи упаковываются в библиотеки, остаются только сложные. Оборудование становится все совершеннее, а запросы людей – выше и выше. Недавно мне показали кое-что поразительное. Это одна из опций системы прокладки маршрута в Google Maps. Можно захватить мышью и перетащить участок маршрута в другую точку, так чтобы система поняла: путь должен быть проложен через нее. И Google переделывает маршрут соответствующим образом. Теперь я знаю, как

это работает: большой кусок кода в JavaScript отслеживает передвижения мыши. Когда отпускаешь кнопку мыши, он формирует запрос на Ajax XML, сообщая материнской системе, что выбран новый пункт маршрута. После этого вычисляется новый маршрут. Я просто не представляю, как это можно было так здорово запрограммировать! Люди жалуются, что им проложили маршрут через огороды и тому подобное, но проблема выбора оптимального маршрута – одна из старейших в компьютерной науке: как найти кратчайший маршрут на произвольном графе? Сногсшибательно.

С одной стороны, я восхищен. С другой, думаю про себя: «Боже, как здорово, что в мои времена такого не было». Я бы никогда не написал код для решения такой задачи. Как они это делают? Наверное, выросло целое поколение программистов, превосходящих меня. Я рад, что некогда заработал репутацию хорошего программиста и не должен больше ее подтверждать, потому что я бы просто не смог.

Сегодня мне приятно быть таким почетным старейшиной цеха программистов. Я заслужил это право своими предыдущими успехами и рад, что могу до сих пор пользоваться их плодами без необходимости что-то кому-то доказывать сегодня. Если вы недавно получили компьютерное образование, знаете, как программировать, и готовы сказать в этой профессии свое слово – вам и карты в руки.

15

Дональд Кнут

Из всех героев этой книги Дональд Кнут, пожалуй, меньше всех нуждается в представлении. На протяжении последних 40 лет он пишет свой многотомный шедевр «Искусство программирования» – библию фундаментальных алгоритмов и структур данных. Журнал «American Scientist» включил эту работу в список 12 самых важных естественнонаучных исследований века наряду с произведениями Рассела, Уайтхеда, Эйнштейна, Дирака, Фейнмана и фон Неймана. Кнут популяризировал применение асимптотической нотации (O большое) при анализе алгоритмов, изобрел LR-анализатор и защищал операторы перехода GOTO от критики Эдсгера Дейкстры.

Но он не просто теоретик. Закончив третий том «Искусства программирования» в 1976 году, Кнут взял перерыв на год, как он тогда думал, собираясь написать системы компьютерной верстки TeX и METAFONT, для того чтобы его книги были напечатаны так, как он хотел. Через десять лет он закончил этот проект, попутно изобрел новый стиль программирования – «литературное программирование» – и алгоритм для разбиения абзацев текста на строки, который и по сей день применяется в программах компьютерной верстки.

Среди его многочисленных наград – первая премия имени Грейс Мюррей Хоппер от Ассоциации вычислительной техники (1971), премия Тьюринга (1974) и Национальная научная медаль США (1979). В 1990 году он пере-

стал пользоваться электронной почтой, объясняя это тем, что его работа заключается не в том, чтобы быть «на самом верху», а в том, чтобы быть «в самом низу», – именно там можно глубже понимать и затем объяснять в книгах многие области компьютерных наук.

В ходе интервью мы поговорили о пристрастии Кнута к литературному программированию, о его двойственном отношении к «черным ящикам» и о том, что он понимает под прискорбным «излишним возвеличиванием ПО многократного использования».

Сейбел: Когда вы научились программировать?

Кнут: Во время учебы на первом курсе в Технологическом институте Кейса. Осенью 1956 года – в течение четверти или семестра – в институте появился компьютер.

Сейбел: IBM 650?

Кнут: Да, это была модель 650. Первая модель, которую IBM выпустила партией больше 100 штук. Наверное, количество проданных компьютеров тогда исчислялось тысячами, вряд ли десятками тысяч. Так или иначе, это был первый компьютер массового производства – и он появился даже в институте Кейса.

Я работал тогда в лаборатории статистики и занимался там сортировкой карт. Я сводил статистические данные в таблицы, зарабатывая этим хоть какую-то прибавку к стипендии. В окне первого этажа был виден стоящий в кабинете компьютер с мигающими лампочками. Зрелище завораживало.

Как-то раз один сотрудник лаборатории, встав к доске, объяснил мне и еще паре моих друзей-первокурсников, как работает эта машина. Я нашел руководство пользователя для этого компьютера – в нем были примеры программ строк по десять. Мне они показались глуповатыми – видимо, даже эти небольшие программки можно было улучшить.

Позже выяснилось, что ночью можно пойти и поработать на компьютере. Это было необычно. Думаю, только в университете Дартмута и Кейсовском институте первокурсникам позволялось дотрагиваться до компьютеров. В других университетах были профессиональные операторы, которым приносишь пачку перфокарт и на утро получаешь у них ответ. Но в Кейсе нас подпускали к компьютерам. Только предупреждали: «Смотри, осторожнее вот с этим; не нужно делать вот этого – в компьютере от этого произойдет сбой». В общем, нам выпал прекрасный шанс поработать с этим компьютером.

Так или иначе, я проверил, сработает ли одна из моих небольших поправок к одной из программ – и она сработала. Я подумал: «Боже мой, это поразительно. Я всего лишь первокурсник, а у меня уже получается лучше, чем написано в этой книге, – наверное, у меня к этому талант». В итоге оказалось, что у меня действительно к этому были способности, но не в том смысле, в каком я думал, ведь ту конкретную программу практически кто угодно мог написать лучше, чем в том конкретном руководстве.

Это была десятичная машина, поэтому работать было проще, ведь мне не пришлось учить двоичную арифметику, хотя в старших классах я в принципе немного занимался ею. Тем не менее из-за того что машина была десятичной, работать на ней было немного проще, удобнее. До сих пор помню тот машинный язык, например *sixty-five is reset-add-lower* (сейчас он помогает мне придумывать пароли и прочее).

Сейбел: Ого, кажется, вы только что выдали свой секрет.

Кнут: Да, точно. Затем я решил написать небольшую программу для разложения числа на простые множители. В программе было около 100 строк. Я приходил по ночам, когда машина была не занята, и занимался отладкой. В моей 100-строчной программе я нашел больше 100 ошибок. Но две недели спустя у меня была готова программа, с помощью которой можно было вычислить простые множители любого десятизначного числа, набранного с помощью переключателей консоли.

Так я и учился программировать – брал собственную программу и сидел за компьютером неделями, пытаясь сделать ее чуточку лучше.

Моя вторая программа переводила двоичный код в десятичный. А третьей была программа для игры в крестики-нолики – именно после нее я стал настоящим программистом.

Тогда мне пришлось использовать структуры данных. Я сделал три версии игры в крестики-нолики, одна из которых была основана на самообучении программы: она начинала играть, ничего не зная об игре, и затем запоминала после каждого проигрыша свои ходы как сомнительные, а ходы соперника как удачные, а после этого соответственно повышала рейтинг одних позиций и уменьшала рейтинг других. После 400 игр она была уже достаточно приемлемым игроком в крестики-нолики.

Сейбел: Похоже, многие из тех, с кем я беседовал, имели непосредственный доступ к компьютерам в самом начале своей карьеры. Тем не менее у Дейкстры есть работа – уверен, вы с ней знакомы, – в которой он прямо говорит, что студентов компьютерных специальностей первые несколько лет обучения не следует допускать к работе за компьютером – все это время они должны учиться работать с символами.

Кнут: Но он и сам так не учился. Он высказывал очень много правильных и прекрасных мыслей, но не всегда был прав. Как и я. Моя позиция по этому поводу такова. Допустим, есть некий ученый – в любой области науки. Становясь старше, этот ученый думает: «Да, что-то из того, что я делал, оказалось действительно полезно, а что-то я уже не использую. Моим студентам не стоит тратить время на какие-то неглобальные вещи. Я вообще не буду с ними обсуждать никакие частные и низкоуровневые задачи. Теоретические понятия – вот что действительно важно и необходимо. Да, и забудем о том, как я до всего этого дошел сам».

Думаю, это фундаментальная ошибка ученых в любой области науки. Они не понимают, что когда чему-то учишься, ты должен видеть это что-то на всех возможных уровнях. Нужно увидеть пол, прежде чем приступать к работе над потолком. Все это в голове переплетается и настолько смешивается, что ученые в возрасте забывают о том, что эти вещи им действительно нужны.

Сейбел: Всех, кого интервьюировал для этой книги, я спрашивал, насколько хорошо они знакомы с вашим трудом «Искусство программирования». Многие ответили, что используют ее в качестве справочника, и лишь некоторые прочли этот труд от корки до корки. Обязан ли каждый программист понимать ваши книги? Ведь их математическая составляющая достаточно сложна.

Кнут: Иногда я сам не уверен, что понимаю их. Я пытаюсь изложить множество умных мыслей по поводу темы того или иного раздела – я нахожу их в различных источниках, где они высказаны частично, и объединяю в некое целое, которое уже может быть использовано в дальнейшем, в котором история передана верно, где исправлены ошибки и двусмысленности, присутствовавшие в оригинале.

Взять, к примеру, главы, которые я пишу сейчас: работа начинается с обработки материала, взятого из математических журналов и написанного на собственном жаргоне, который большинство программистов, как мне кажется, даже и не подумают учить. Поэтому я пытаюсь адаптировать эти тексты до такой степени, чтобы хотя бы *самому* их понимать. Я пытаюсь излагать ключевые идеи так просто, как могу, но в итоге получается, что за каждыми пятью страницами моей книги – чья-то карьера.

Другими словами, любой вопрос, раскрытый в моей книге на пяти страницах, этими пятью страницами далеко не исчерпывается – его изучению можно посвятить всю жизнь, потому что компьютерные науки – действительно очень богатая область. Компьютерные науки во-

все не сводятся к ряду простых вещей. Если бы компьютерные науки действительно были очень просты и все свелось бы лишь к тому, чтобы сформулировать 50 правильных вещей и выучить их, тогда, да, тогда бы я сказал: «Да, каждый человек в мире должен знать эти 50 вещей – и должен знать их хорошенько».

Но это не так. В моих книгах – тысячи страниц и тысячи примеров, которые я записываю и включаю в книги, для того чтобы не держать их все в голове. Мне приходится постоянно возвращаться к ним и изучать их заново. В моих книгах есть ответы ко всем заданиям, так как я знаю, что десять лет спустя я не буду помнить ничего о том, как делать ту или иную вещь, и мне придется потратить много времени на то, чтобы восстановить утраченные знания. Поэтому я оставляю для себя хотя бы минимальные подсказки на будущее.

Я постоянно разрываюсь между мыслью «Ну, это слишком сложно – не стоит об этом говорить совсем» и ощущением, что те, кто прочтет книгу, скажут: «Все, о чем вы пишете, очевидно. От ваших книг никакой пользы». В любой отдельно взятый момент меня можно застать за спором с самим собой – выкинуть ли мне все или набрать еще материала, поскольку имеющегося недостаточно.

В итоге же все сводится к тому, что все по-настоящему интересное, что можно изложить на половине страницы моей книги, должно занимать полстраницы. Также должны быть включены все вещи, которые слишком хороши, чтобы их не включить. Я тут выяснил, что в раздел о двоичных диаграммах решений, который я только что написал, включено больше 260 заданий – по ходу дела у меня набиралось все больше и больше материала не для широкой аудитории. Не все из этих 260 заданий будут интересны абсолютно всем. Тем не менее я уверен, что многие оценят по достоинству каждое из этих заданий в отдельности.

Удивительно, что есть люди, которые читают мои книги от корки до корки. Насколько я знаю, обычно человек выбирает подходящие для себя разделы. Но знает, что если глубже копнет ту или иную тему, то увидит в ней узкий набор терминов, а не все возможные нотации и жаргоны, – и без моих книг людям было бы тяжелее разбираться во всех этих темах. Вот что меня по-настоящему вдохновляет.

Кроме того, я стараюсь вести исследования в наиболее актуальном для практикующего программиста ключе, а вовсе не пытаюсь добиться признания в академических кругах, публикуя вещи, интересные в теории, но вряд ли употребимые в настоящей программе.

Я не включаю в свои книги исследования, где, например, фигурирует структура данных, в которой множитель $\log \log n$ экономится только при n больше двух миллионов. Подобных исследований великое

множество. Их авторы фантазируют о том, что, в принципе, если бы компьютеры были богоподобны, то у нас были бы более быстрые алгоритмы. Но даже алгоритмы вроде сбалансированного дерева или AVL-дерева я не использую в своих программах, не будучи твердо уверен в том, что это будет действительно большое дерево.

Сейбел: А что вы используете?

Кнут: Обыкновенное дерево двоичного поиска, которое с недавнего времени стал особым образом рандомизировать.

Сейбел: Кстати, о практической работе – в разгар работы над «Искусством программирования» вы взяли десятилетний перерыв, для того чтобы написать систему компьютерной верстки TeX. Насколько я понимаю, первую версию TeX вы написали без применения компьютера.

Кнут: Когда в 1977–1978 годах я написал первую версию TeX, у меня, разумеется, еще не было литературного программирования, но уже было структурное программирование. Я писал в большом блокноте, без сокращений, карандашом.

Полгода спустя, завершив проект в целом, я начал набирать его на компьютере. Отладку я выполнил в марте 1978 года, а программу начал писать в октябре 1977-го. Код хранится в архивах Стэнфордского университета – он весь написан карандашом. И, конечно, я часто возвращался к написанному и по мере обнаружения ошибок менял подпрограммы.

Это была система первого поколения, поэтому можно было использовать разные архитектуры, которые следовало отбросить, пока я не пожил с ней достаточно долго и не понял что к чему. Это была своего рода вечная проблема – нельзя было набирать текст без шрифтов, но нельзя было создать шрифты, не имея возможности набирать текст.

Однако структурное программирование подсказало мне идею инвариантов и представление о том, как создавать «черные ящики», которые я смогу понимать. Поэтому я обрел уверенность, что код заработает, как только я наконец-то закончу с отладкой. Я думал, что смогу выиграть кучу времени, если подожду еще полгода, прежде чем приступить к какому-либо тестированию. Я не сомневался, что с кодом все было более или менее в порядке.

Сейбел: И на экономию времени вы рассчитывали, поскольку вам не пришлось бы писать вспомогательный код и модули для тестирования, чтобы протестировать незаконченный код?

Кнут: Именно.

Сейбел: Помимо того что ваши книги так хорошо набраны, как повашему, они бы сильно изменились, если бы вы не потратили десять лет на создание TeX?

Кнут: Хороший вопрос. Использование структурного программирования не в чисто академических целях – другими словами, я думаю об инвариантах не только в пробных, но и в реальных программах, – возможно, повлияло на то, как я сейчас описываю алгоритмы в своих новых книгах. А если не повлияло, то должно было.

Я бы ничего не узнал о кэшировании и о тенденциях изменений в компьютерной индустрии, и о других подобных вещах, если бы просто шел по накатанной дорожке – черпая знания для своих книг из уже написанных трудов. Пока я писал первые три тома, я не создавал программ, подобных TeX, написание которых более характерно для активной программистской деятельности. Я же писал пробные программы. Таким образом, я получал некое представление о числах и величинах.

На самом деле, это потрясающе – осознавать во время процесса написания книги те вещи, которые заставляют тебя использовать то или иное слово. Это поистине тайна – как это все происходит. Это наиболее важное влияние, которое оказал на меня опыт создания TeX, – я по-другому стал относиться к вещам, и я стал писать по-иному, по-иному составлять предложения – они стали менее громоздкими. У всей книги появилась какая-то новая интонация – уверенности или еще чего-то.

Сейбел: То есть к моменту завершения TeX вы были уже значительно лучшим программистом, чем когда начинали его создавать?

Кнут: Да, потому что я стал применять методы литературного программирования.

Сейбел: Понятно, у вас появились более качественные инструменты, но улучшились ли непосредственно ваши навыки?

Кнут: Я узнал очень много нового, пока работал над этим проектом. В частности узнал, как много ресурсов вашего мозга съедает разработка ПО. Это оказалось намного более сложным заданием, чем я ожидал. Я не мог одновременно преподавать на полную ставку и полноценно заниматься разработкой ПО. Но я мог преподавать на полную ставку и полноценно заниматься написанием книг; ПО же требовало невероятного внимания к мельчайшим деталям. Мой мозг был забит только программным обеспечением, так что я не мог думать ни о чем другом. Поэтому я стал с особым уважением относиться к тем, кто работает в крупных проектах по разработке ПО; я бы никогда не узнал, как они работают, если бы сам не занимался схожей работой.

Сейбел: То есть, по-вашему, программировать сложнее, чем писать книги; где-то я читал ваше мнение, согласно которому невозможно сказать, сколько времени займет написание той или иной книги. Значит ли это, что сказать, сколько времени займет написание одной программы, еще сложнее?

Кнут: Да, это так. Это очень верный вывод.

В этом году я написал три крупные программы с помощью методов литературного программирования, код каждой из которых занимает больше 100 страниц формата А4. Две из них взаимосвязаны, так что это скорее две с половиной программы. И еще около 150 небольших программ. Пожалуй, за предыдущий год мне удалось сделать меньше. То есть я очень плотно занимался в этом году созданием небольших программ, но некоторые из них я писал по месяцу или дольше.

Сейбел: И вы с самого начала могли сказать, что будете писать их месяц?

Кнут: Что касается одной из них – да, я ожидал, что на ее создание уйдет месяц. Я знал, что это будет непростое задание, но не мог предположить, насколько у нее богатый потенциал, – я даже добавил несколько дополнительных свойств по ходу ее использования. Я считаю, что это абсолютная истина, которую должен усвоить каждый руководитель отдела разработки: сроки написания программы нельзя спрогнозировать.

Сейбел: Помимо того что вы написали «Искусство программирования» и TeX, вы изобрели и всячески продвигали литературное программирование – способ программирования, при котором код гораздо легче прочитать неспециалисту. Вы написали WEB и CWEB, инструменты реализации языков литературного программирования на базе Паскаля и Си.

Кнут: Вы сказали *продвигал* – да, я люблю поговорить о достоинствах этого метода. Но все же не очень люблю проповедовать или пытаться обращать людей в свою веру. Мне кажется, что программирование очень похоже на религию – и там, и там у людей есть убеждения. Кто-то любит навязывать свои убеждения другим. А кто-то говорит: слушайте, вот что я думаю по этому поводу, я не могу вам доказать, что это самое лучшее, но мне это точно подходит. Потом остается только надеяться, что люди последуют вашему совету и придут к тому же выводу. Но мне не нравится выходить к людям и говорить им, во что они должны верить.

Сейбел: Так, может, объясните, почему вы так любите литературное программирование и чем оно отличается от нелитературного программирования?

Кнут: Первое правило пишущего человека – нужно понимать свою аудиторию: чем лучше знаешь своего читателя, тем лучше пишешь; это очевидно. Второе правило, касающееся технического писателя, – говорить все вещи дважды – так, чтобы у читателя была возможность усвоить информацию из нескольких дополняющих друг друга источников.

Поэтому в технических книгах так много избыточных конструкций. Все моменты объясняются и формально, и с помощью разговорного языка. Или даешь определение, а потом добавляешь: «Следовательно,

вот такое-то и такое-то утверждения – верны». И это добавление можно понять только в том случае, если понял определение.

Или можно сказать: «Допустим, что a , равное тому-то и тому-то, – это множество главных элементов». Таким образом, разговорный термин множество главных элементов дополняется математическим описанием того, как мы создали множество a .

То есть в основе литературного программирования лежит идея, что лучший способ передачи сообщения – совмещение формального и неформального способов. Такой подход обеспечивает естественную основу для переключения между естественным языком – английским – и формальным языком – Си, Лиспом или любым другим языком программирования – и их совмещения. Как мне кажется, такой метод облегчает работу с документацией.

И еще: когда я пишу программу, мне не нужно предоставлять ее в той форме, в какой ее хочет видеть компилятор. Я предоставляю ее в форме, по моему мнению, наиболее доступной для читателя.

Кто-то пишет код снизу вверх, создавая подпрограммы, дающие все более крупные и крупные объекты, и становясь все более уверенным в себе, поскольку теперь может сделать гораздо больше. Другие пишут сверху вниз; они начинают писать и думают: «Так, у меня есть задача, которую нужно решить, – сначала я сделаю вот это, а потом – вот это».

Если я пишу литературную программу, то могу выбирать между этими способами. И практически всегда в итоге моя программа создается в том порядке, в каком я ее сам продумал. То есть, начиная работу, я думаю: «Ага, у меня есть задача, которую нужно решить, то есть сначала мне нужно решить вот это, а потом я решу вон то».

Но потом я говорю: «А теперь давай-ка построим кое-какие инструменты снизу вверх». У нас есть в голове цель, но нам нужно построить несколько инструментов снизу вверх, после чего мы возвращаемся и делаем кое-какую работу сверху вниз. Но в каком порядке мы это делаем? Сначала мне нужно написать то, что я думал в первый день, когда мне пришлось столкнуться с данной задачей. А следующий этап будет посвящен тому, чем я решил заняться дальше.

И я начинаю заниматься тем, что в данный момент волнует меня больше всего, но и тем, что я готов решить в данный момент. Не откладывая этого дела в долгий ящик – если я готов выполнить его прямо сейчас, то прямо сейчас его и делаю. Но это уже совсем другой порядок – ни снизу вверх, ни сверху вниз. Это психологический момент: «Мне нужна задача, выполнение которой принесло бы мне сейчас наибольшее удовольствие и к выполнению которой я сейчас готов». В этом уравнении не так уж много неизвестных. Таким образом, мне очень важно то, что

я без всяких проблем могу создавать программу в подобном человечески понятном порядке.

Так почему же эта идея не получила широкого распространения по всему миру? Почему все не делают так? Я сейчас точно не вспомню, кто абсолютно верно сформулировал объяснение – кажется, это был Джон Бентли. В упрощенной форме мысль звучит примерно так: лишь два процента населения земного шара рождены, чтобы стать гениальными программистами. И лишь два процента населения земного шара рождены, чтобы стать гениальными писателями. А Кнут хочет, чтобы абсолютно все были и теми, и другими.

Мне не кажется, что нам удастся увеличить общее количество программистов в мире – оно не будет превышать двух процентов. Я имею в виду программистов, которые действительно понимают машину, которые были рождены для этого занятия, для кого это дело всей жизни. С другой стороны, сейчас, с появлением блогов, мне совершенно очевидно повышение общей способности выражать свои мысли. Таким образом, вторая часть мысли Бентли сейчас не так уж верна.

Я лишь немного занимался этим в Стэнфордском университете с группой студентов. Они писали программы в качестве летнего проекта, и я познакомил их с идеей литературного программирования. В то лето у меня была группа лишь из семи человек. Шестерым из них эта идея настолько понравилась, что они применяют ее до сих пор. Седьмой ее ненавидел. Его представление о литературной программе было следующим: он брал обычную программу, сверху создавал надстройку и говорил: «Это модуль номер один», – и так далее. Конечно, в Стэнфорд принимают тех, кто умеет хорошо писать, то есть это не вполне репрезентативный пример.

Сейбел: Вам когда-нибудь приходилось писать литературную программу, которую вы коренным образом перестраивали, чтобы объяснить ее? Просто мне трудно представить, что поток сознания *всегда* является наилучшим принципом организации материала.

Кнут: Такого практически никогда не было. Не помню, чтобы мне приходилось брать и действительно менять порядок частей. Просто у меня всегда было так, что я даже не задумывался над тем, какую задачу решать дальше. Не могу этого объяснить точно, но у меня такое ощущение, что одно просто плавно перетекало в другое.

Сейбел: Пишете ли вы литературный код для программ, которые никто кроме вас никогда не увидит?

Кнут: Конечно. Именно этим литературное программирование и хорошо – я могу разговаривать с самим собой. Я могу прочитать программу год спустя и точно понять, о чем я тогда думал.

Сейбел: Всегда ли этот метод работает?

Кнут: На самом деле, достаточно часто труднее понимать программу год спустя, чем до этого. Но это несравнимо с тем, что у меня было до того, как я придумал литературное программирование. Оно не делает сложную вещь очевидной – просто лучшего метода я не знаю.

Я недавно распечатал небольшой комплект больших коллекций подпрограмм, написанных на Си, которые в общем-то отражают текущее состояние дел в работе с булевыми схемами решений (BDD). Это полная противоположность CWEB; практически все во всем мире разрабатывают пакеты программ именно так. Делается это с помощью достаточно упорядоченных соглашений по комментированию, которые понимаются широким сообществом. И код не так уж труден для понимания, поскольку он логически разделен, в нем есть заголовочные файлы, и вы можете видеть структуры данных, и к каждой части структуры данных даны комментарии, поясняющие тот или иной момент. Это еще один вполне эффективный стиль программирования.

Тем не менее я уверен в том, что этот метод далеко не столь эффективен, как литературное программирование, в силу множества неосязаемых вещей, которые я не могу доказать. Наиболее убедительным аргументом для меня является моя уверенность в том, что некоторые из написанных мною программ я никогда бы не смог написать без методов литературного программирования. Например, создание эмулятора MMIX стало бы для меня такой чудовищной головоломкой, что если бы мне пришлось работать над ним с помощью обычного метода, то не думаю, что мне удалось бы его завершить. Обычного разделения его на подпрограммы было недостаточно для того, чтобы упростить его, сделав удобным для восприятия и работы с ним.

Этот эмулятор учитывает наиболее общим образом спецификацию компьютера: какими функциональными блоками он обладает, сколько инструкций может выполнять одновременно, каковы его стратегии кэширования, как работает шина и устройство вывода, каким образом производится прогнозирование ветвления и как работает конвейер.

Вы можете придумать компьютер с шестью блоками деления и конвейером, состоящим из определенного количества стадий, и смоделировать его. Будете ли вы быстрее вычислять простые числа с таким компьютером? Вам не нужно создавать такой компьютер.

Я не утверждаю, что невозможно взять эту программу и разбить ее на подпрограммы, но я бы никогда не смог ее выполнить подобным образом. Кроме того, код занимает всего 170 страниц, и он понятен – я не единственный в мире человек, который его понимает.

Сейбел: Я читал повторную реализацию игры Adventure, осуществленную вами с помощью методов литературного программирования; она мне показалась слегка монолитной. Это было похоже на стиль литературного программирования, поскольку тут вы можете интерполировать объекты и совсем не думать о том, чтобы разбить ее на подпрограммы.

Кнут: Верно. Вместо того чтобы вызывать подпрограмму, мы как будто имеем встроенные подпрограммы на протяжении всей программы. Идея подпрограммы по-прежнему используется, но в итоговом варианте вашей программы подпрограмм нет. Это больше походит на макросы. Но суть в том, что на уровне понятий механизм вызова подпрограмм не нужен, если у вас есть иной способ реализации этой функции на языке, который вы используете.

Что касается Adventure, то не думаю, что я на самом деле убрал подпрограммы из программы Дона Вудса, написанной на Фортране, – я взял его программу на Фортране и переписал ее на английском и на Си. Но абсолютно верно то, что если вы взглянете на мой код TeX, то увидите, что подпрограмм в стеке около 4–5, тогда как в программе, написанной кем-либо другим – без применения методов литературного программирования, – их может быть и 50, и 100.

Я пытаюсь работать с блоками, которые соотносятся с моим мыслительным процессом, а не подчиняюсь каким-то логическим принципам работы с формальной системой. Я хочу, чтобы мои программы соответствовали моему интуитивному процессу их создания, а не чьей-либо жестко закрепленной схеме.

Естественно, в конечном счете она должна быть перенесена на компьютер, у которого есть жесткие рамки, четкие правила понимания. Но в моем представлении хорошая и правильная программа должна наиболее точно передавать образ моих мыслей, а не принцип работы компьютера. Мне нужно найти способ перехода из одного состояния в другое, но я стремлюсь сохранять исходные тексты ближе к своей голове, чем к компьютеру.

Кроме того, я убежден, что литературное программирование – это очень эффективный способ ведения документации, обеспечивающий связь между группами людей. Многие люди понимали код TeX настолько хорошо, что могли создавать сценарии, при которых в программе происходил сбой. Мне кажется, что еще больше людей в какой-то из моментов своей жизни понимали именно эту программу, а не любую другую программу схожего размера.

Сейбел: И тем не менее сталкивались ли вы когда-нибудь с такими ситуациями, в которых люди читали ваш код и все равно задавали вам

вопросы, заставлявшие вас думать: «Неужели они действительно тут что-то не поняли?»

Кнут: Конечно. Такое происходит постоянно, но лишь из-за недостаточно высокого качества моих объяснений. Позвольте привести простой пример. В «Искусстве программирования» я пишу о первых применениях побитовых операций, и у меня есть такое предложение: «Компьютер Manchester Mark 1, созданный примерно в то же время, что и EDSAC, использовал не только побитовые операторы И, но также ИЛИ и исключающее ИЛИ. В его первом руководстве программиста, написанном в 1950 году, Алан Тьюринг отмечал, что побитовый оператор НЕ может быть получен с помощью исключающего ИЛИ или в сочетании с несколькими подобными операторами».

То есть во фразе «В его первом руководстве программиста Алан Тьюринг...» речь идет о первом руководстве программиста для компьютера Manchester Mark 1. Но четверо или пятеро человек, прочитавших это предложение, независимо друг от друга заявили, что поняли фразу в том смысле, что в 1950 году Алан Тьюринг написал свое первое руководство программиста.

На самом деле у него были и другие руководства по программированию, поэтому я написал все правильно, но фраза была неправильно истолкована людьми. Поэтому я исправил ее следующим образом: «В первом руководстве по программированию для компьютера Mark I, написанном в 1950 году, Алан Тьюринг отмечал...»

То же касается и чисто математических моментов – ко мне обращаются люди, которые их не понимают. Поэтому я скажу так: как правило, я все пишу верно, но знаю, что мне тем не менее нужно эти моменты исправлять и улучшать.

Сейбел: Как правило, публикуя литературную программу, вы публикуете ее финальную версию. Кроме того, вам часто приписывается авторство фразы «Преждевременная оптимизация – корень всех зол». Но на момент создания финальной версии программы уже нельзя говорить о преждевременности – возможно, некоторые части были оптимизированы весьма эффективно. Но не усложняет ли это чтение кода?

Кнут: Нет. Хорошая литературная программа всегда покажет свою историю. Хорошая литературная программа всегда скажет: «Вот очевидный способ выполнения данной задачи – почему бы нам им не воспользоваться?»

Когда вы вводите более тонкие детали в свою программу, литературное программирование показывает себя во всем блеске, потому что это не просто код, выполняющий поставленные задачи, это еще и докумен-

тация. Вы говорите: «Здесь был использован запрещенный прием, он работает, потому что...» – и расписываете в мельчайших подробностях причины и основные положения.

Я использую запрещенные приемы по двум причинам. Во-первых, если это действительно повысит производительность моей программы и если это повышение производительности действительно необходимо. Или иногда я думаю: «Это решение кажется не совсем чистым. Не могу сегодня ничего с собой поделать – хочется использовать этот запрещенный прием, потому что это так клево». То есть чисто для своего удовольствия. В любом случае я все это документирую, а не просто использую в программе и все.

Сейбел: То есть это чаще встречается не в коде, а в тексте?

Кнут: Да, речь о текстовой части. Я не показываю код, который исключил из программы. Хотя мог бы.

Сейбел: Есть ли в CWEB возможность включать код, не являющийся частью приложения? Тогда можно не документировать этот момент, а просто делать комментарий: «Это действительно очень простая версия данной функции».

Кнут: У вас просто есть код, который не используется. Он упоминается в документации с пометкой, что этот код нигде не используется.

Сейбел: То есть на этот фрагмент вы просто нигде не ссылаетесь?

Кнут: Да. Кроме того, у меня есть код, который я могу вызвать из отладчика. Я могу сказать: «Нужно вызвать то-то и то-то с такими-то и такими-то параметрами». Подпрограмма никогда не вызывается непосредственно из самой программы, но она всегда есть в документации. Поэтому я могу остановить выполнение программы посередине, вызвать эту подпрограмму – она осмотрится, как идут дела, как все обстоит на более масштабном уровне.

Сейбел: И точно таким же образом вы можете написать: «Раздел первый – это простейшая реализация данного алгоритма; раздел второй – слегка модифицированная версия первого раздела; раздел третий – вот его мы действительно используем, но вы его никогда не поймете, если не прочтете первые два раздела».

Кнут: Именно. У меня есть несколько программ в Интернете, которые решают головоломку «15». И я использую 3 разные версии. Я говорю: «Прочитайте версию номер один, иначе вы никогда не поймете версию номера два. И прочитайте версию номер два, иначе вы никогда не поймете версию номер три».

Мне приходилось писать самые разные программы. Иногда я работаю над программой, для которой производительность – последнее дело;

важно лишь получить ответ. В таком случае я применяю метод грубой силы – тогда мне совсем не нужно думать, поскольку не требуется придумывать никаких тонких решений, и я точно не переумдрю. При таких обстоятельствах ни о какой преждевременной оптимизации не может быть и речи.

Затем я могу внести кое-какие изменения и посмотреть, согласуется ли новая версия с моим методом грубой силы. После чего могу масштабировать программу и переходить к более крупным случаям. Работа с большинством программ заканчивается на этом этапе, потому что вы не будете запускать ее триллион раз. Делая иллюстрацию для «Искусства программирования», я могу менять ее несколько раз, и переводчикам моей книги также, может быть, придется переделывать эту программу, но абсолютно не важно то, что я ее рисую с помощью очень небыстрого метода, потому что мне нужно сгенерировать этот файл лишь один раз – после чего я передаю его своему издателю, и он попадает на страницы книги.

Но в данный момент я работаю над комбинаторными алгоритмами, которые по определению являются задачами гигантского размера. Поэтому, для того чтобы использовать в своей книге интересные примеры, мне приходится писать программы, решающие задачу таким образом, чтобы заставить читателя подумать: «О да, я бы не смог решить эту задачу с помощью обычных методов, поэтому мне нужно кое-что узнать об искусстве программирования, иначе у меня уйдет сотня лет на решение этой задачи с помощью метода грубой силы».

Комбинаторные алгоритмы – это потрясающая вещь, поскольку одна хорошая идея может в десятки раз сократить время работы программы. Но я вовсе без всякого высокомерия отношусь к идеям, с помощью которых можно сэкономить и 20%, если программа выполняется триллион раз. Потому что, сэкономив сотню наносекунд в цикле, который исполняется триллион раз, думаю, вы экономите целый день. Если код будет использоваться действительно часто, то будет совсем не лишним придумывать различные тонкие ходы, которые не так-то легко сразу понять.

Около года назад я прочитал обзор в журнале «Computing Reviews» – рецензент писал о книге под названием «Programming Tricks» (Уловки в программировании) или что-то вроде того. Суть рецензии сводилась к следующему: «Если бы я узнал, что кто-то из программистов, работающих на меня, применяет эти штучки, я бы их уволил». Естественно, я начал искать эту книгу, подумав: «Именно такая книга мне и нужна – в ней я могу узнать для себя много нового». К сожалению, уловки не были такими уж хорошими.

Сейбел: Они что, действительно причиняли вред?

Кнут: Вообще-то, это были очень слабые и неэффективные уловки. Они не были систематизированы и все такое, но самое главное – они были достаточно очевидны. Это была совершенно иная культура. Что же касается рецензента, который собирался увольнять своих сотрудников, – он хочет, чтобы в программировании все делалось неэффективно, но вписывалось бы в его идею аккуратности. Его абсолютно не интересует, хороша программа или нет – если брать в расчет скорость исполнения и производительность; он думает лишь о том, чтобы она соответствовала другим критериям, – например, чтобы ее поддержку мог осуществлять абсолютно любой человек. Что ж, у многих вообще очень странные взгляды на действительность.

Многие люди непостижимым для меня образом полагают, что нашей целью является создание программ как неких замкнутых миров, то есть чтобы человеку нужно было лишь установить пару дополнительных параметров, а дальше уже программа все сделает. Следовательно, в мире должно быть совсем немного программистов, которые будут писать библиотеки, какое-то количество тех, кто пишет руководства пользователя для этих библиотек, и те, кто будет применять эти библиотеки для своих нужд, – и всё.

Проблема в том, что программирование становится совсем не интересным и не привлекательным занятием, если можешь лишь вызывать что-то из библиотеки, но не можешь написать библиотеку сам. Если бы работа программиста заключалась в том, чтобы находить правильное сочетание параметров для выполнения достаточно очевидных вещей, то кто бы захотел выбрать себе такую профессию?

Существует некое излишнее возвеличивание ПО многократного использования, когда вовсе не нужно открывать ящик и смотреть, что же находится внутри. Всегда приятно иметь черные ящики, но – практически всегда – если можешь заглянуть внутрь ящика, то можешь улучшить его работу, как только узнаешь, что же находится внутри.

Люди же, наоборот, все засовывают в упаковку, которую не открыть, и преподносят эту закрытую упаковку программистам, и всем программистам во всем мире не разрешается вскрывать эту упаковку. Они только и могут что соединять одно с другим. И поэтому приходится помнить, что, вызывая вот эту подпрограмму, нужно ввести x_0 , y_0 , x_1 , y_1 , а при вызове другой подпрограммы нужно ввести x_0 , x_1 , y_0 , y_1 . Все выполняется строго по инструкции – и это ваша работа.

Сейбел: Многие с вами согласятся, что, да, интереснее писать код самому. Но помимо увлекательности есть еще и...

Кнут: Дело не только в том, что это интересно. Работа математика заключается в том, чтобы писать доказательства, но практически никог-

да не бывает так, что в процессе решения математической задачи вы можете найти теорему, чьи гипотезы абсолютно идеально подходят для вашего случая. Практически всегда вы работаете с тем, что *более или менее похоже* на классическую теорему из учебника. Поэтому, что вы делаете? Читаете доказательство этой теоремы и думаете: «Ага, вот как мне нужно изменить это доказательство, чтобы доказать гипотезу, которой я занимаюсь». То есть в математических книгах полно теорем, но вы никогда не берете всю теорему от начала до конца – вам нужно увидеть ее доказательство, потому что лишь в одном случае из ста вы сможете найти теорему, которая идеально подходит для вашей задачи. Думаю, примерно так же дело обстоит и с ПО.

Сейбел: И тем не менее, не является ли ПО – по-моему, вы сами же об этом и говорили – самой сложной вещью, когда-либо созданной человеком?

Кнут: По-моему, Дейкстра сказал это первым, но, вообще, да, все верно. Все дело в том, что сложность соединения разных вещей далеко не однородна. В чистой математике стремятся к тому, чтобы выработать небольшое количество правил, которые можно было бы применять ко всем явлениям – 3–4 аксиомы для описания целой системы. В компьютерной же программе множество частей: шаг 1 не похож на шаг 2, а тот, в свою очередь, не похож на шаг 3. Вам нужно соединить все эти вещи и организовать весьма нетривиальным способом.

Сейбел: Таким образом, учитывая сложность, получается, что в какой-то момент нужно взять ряд черных ящиков и сказать: «Ага, мы знаем, как эта штука работает, и можем ею пользоваться». Если бы нам пришлось заглядывать внутрь каждого черного ящика, мы бы никогда не смогли ничего довести до конца.

Кнут: Я не говорю, что черные ящики бесполезны. Я говорю лишь о том, что если мне не разрешается их открывать и если мне нужно выполнить определенное задание, располагая лишь библиотекой из чего-то подобного, то я получу гораздо более низкие результаты – и все будет происходить намного медленнее.

Сейбел: Медленнее – вы имеете в виду работу программы или ее разработку?

Кнут: И то, и другое. Ну хорошо, я могу достаточно быстро сделать работающую программу, поэтому этого утверждать не могу. Просто у меня уйдет больше времени в силу того, что мне придется потратить время на изучение большего количества справочников и на поиск нужных элементов, то есть эта проблема скорее связана с поиском, а не с процессом создания.

Сейбел: Какое-то время назад программист, писавший стандартные библиотеки Java, написал статью о том, как в их реализации двоичного поиска на протяжении девяти лет была ошибка. То есть они взяли минимум и максимум, сложили их и разделили на два. Но, естественно, если операция сложения переполняется, то это ошибка. В общем, конечно, плохо то, что в стандартной библиотеке была ошибка, но в итоге они нашли ее и устранили. Если бы все сами писали двоичный поиск, то ошибочных двоичных поисков, наверное, было бы больше.

Кнут: Это так. И это лишь один пример из огромного множества. Двоичный поиск – это конкретный пример того, с чего мы начинали наши занятия по программированию в 1970-х. В первый день учебы все писали программы двоичного поиска и сдавали нам, после чего ассистенты их изучали. В результате работали меньше 10% программ. И находилось около 4–6 различных ошибок. Но ни одна из них не имела никакого отношения к переполнению, которое вы упомянули; это новая ошибка – на моих занятиях мы не считали сложение этих двух чисел проблемой.

Вернемся к идее черного ящика. Вы можете спросить, почему же она мне так ненавистна? Мы говорим об арифметических операциях, поэтому предположим, что у нас есть программа для перемножения матриц. У нас есть черный ящик для перемножения матриц, затем мы меняем тип данных – с действительного на комплексный; и теперь у нас есть черный ящик для перемножения комплексных матриц, работа которого занимает $4n^3$ шагов, а не n^3 шагов. Если работа программы с действительным типом данных занимает определенное время t , то работа программы с комплексным типом данных занимает время $4t$. Но если вы можете заглянуть внутрь ящика, то вам понадобится лишь 3 операции перемножения матриц с действительным типом данных, а не 4, поскольку у нас есть тождество, которое описывает результат перемножения двух комплексных матриц. Это лишь небольшой пример; их список можно продолжать до бесконечности.

У меня есть очередь с приоритетами или структура вроде кучи; что бы там ни было – двоичный поиск – у меня будет хороший источник для алгоритма, но он будет не до конца соответствовать моим целям, поэтому я каждый раз буду его адаптировать. И мне кажется, что так лучше для меня. Я знаю, что со мной не согласятся многие, кто считает, что их работа – писать программы, которые все будут использовать; поэтому если в них находятся ошибки, они их исправляют, и программы всех других людей также станут работать лучше. Хорошо. Мне не нравится такой подход к делу. И я хочу увидеть их программы.

В то время, когда я писал первый том «Искусства программирования», люди не понимали, что они могли использовать связанные списки в своих программах, а также указатели для работы со структурами данных.

Столкнувшись с задачей, не связанной с массивами, вы обращались к имеющимся пакетам программ или к интерпретируемому языку, вроде IPL-V или Лиспа. Также были версии на Фортране, и вы могли взять эти подпрограммы, разобраться, как ими пользоваться, и создавать в нем собственную программу. Считалось совершенным абсурдом учить обыкновенного программиста использовать связанные списки в собственных программах. Считалось, что все должно делаться с помощью этих наработанных процедур.

Но тогда в обычных пакетах должны были быть представлены все дополнительные инструменты, необходимые лишь для решения каких-то узких задач, встававших перед небольшим количеством пользователей. Поэтому моя книга в общем-то открыла людям глаза: «Боже, я могу это понять и адаптировать таким образом, что смогу разместить элементы в двух списках одновременно. Я могу менять структуру данных». Эта практика стала повсеместно доступной, а не закрытой для употребления лишь в рамках этих пакетов.

То же самое я вижу и сейчас, работая над структурами BDD. На данный момент есть 3–4 пакета подпрограмм, работающих с BDD, но я пишу сейчас в «Искусстве программирования» – если вкратце, – что вы тоже можете писать простые версии BDD для множества приложений, которые будут весьма эффективны. Вы можете использовать их для самых разных задач, в которых вам не нужны все эти прибабасы других пакетов; и те вещи, которые вы теперь можете делать, легки для понимания и для использования в программах, которые вы пишете сами.

В этом году я закончил раздел, посвященный уловкам и приемам работы с побитовыми операциями, а эти вещи долго считались черной магией среди компьютерных специалистов. Я решил: пора сказать о том, что у этих приемов есть и теоретический аспект, благодаря которому можно понять эти идеи и то, как они используются. Вы можете абсолютно уверенно использовать их сами. Кроме того, вы можете создавать программы и делать удивительные вещи, о которых в прошлом году не имели особого представления. До нынешнего времени все это было в глубоком подполье, но этому мы также должны учить людей – это знание заслуживает того, чтобы быть доступным всем.

Я пишу много программ и не могу утверждать, что являюсь показательным примером, но точно знаю, что мне удастся создавать много программ, успешно выполняющих различные задачи, и мне было бы намного труднее, если бы пришлось тратить время на изучение работы с программами и процедурами, написанными кем-то другим. Мне гораздо проще разобраться с рядом базовых понятий, после чего повторно использовать код путем редактирования текста кода, который раньше работал.

Сейбел: Как изменились ваши взгляды на программирование с того времени, когда вы только начинали им заниматься?

Кнут: Мы уже обсудили с вами литературное программирование; это само по себе коренное изменение в понимании своей функции: не просто расстановка в нужном порядке инструкций, а объяснение того, что происходит. Дейкстра прошел ту же эволюцию. В конце концов, его программы были даже более литературными, чем мои, – в том смысле, что они даже не добрались до компьютера. Они были просто литературными.

Он был также одним из главных вдохновителей структурного программирования, где мы работали с шаблонами, которые можно было использовать для масштабирования наших программ, так чтобы мы могли создавать более крупные программы и тем не менее держать все под контролем. Вы пишете в десять раз более крупную программу, но вам не нужно из-за нее спать в десять раз меньше обычного, поскольку у вас есть инструменты, позволяющие надежно соединять элементы в более крупную систему. Это было очень непривычно.

Это очень важный момент – необходимость осознания абстрактных понятий, позволяющих нам работать с крупными системами, не теряя уверенности в том, что у нас все под контролем, что мы знаем, что делаем, несмотря на то, что эти понятия чертовски сложны.

Есть множество других вещей, которые могут считаться важными изменениями, но мне не кажется, что они что-либо значительным образом изменили. Это лишь оболочка, еще одна разновидность синтаксического сахара, различные диалекты уже существующих языков. У всех разные вкусы. Кто-то, например, обладает более логическим мышлением, чем я. Они используют в изобилии вводные слова, любят, чтобы все всему соответствовало, и всегда говорят: «Сейчас я начинаю делать вот это», – а в конце: «А сейчас я заканчиваю делать вот это». Мне это не очень-то по душе. Это не похоже на мой образ мышления. Но это образ мышления других людей, а какого-то одного – самого лучшего – образа мышления не существует.

Для меня одной из самых главных революций в языках программирования стало использование указателей в языке Си. Если у вас есть необычные структуры данных, то нередко появляется необходимость в том, чтобы одна часть структуры указывала на другую часть, и люди пробовали по-разному реализовать эту функцию в языках более высокого уровня. Например, Тони Хоар разработал ясную четкую систему, но язык Си привнес следующее (что сначала показалось мне большой ошибкой, но в конце концов я это полюбил): если x – это указатель, и вы пишете $x+1$, это означает не следующий байт после x , а следующий узел

после x в зависимости от того, на что указывает x : если он указывает на большой узел, то $x+1$ пройдет большой путь; если x указывает на что-то маленькое, то и $x+1$ продвинется немного. Для меня это одно из самых потрясающих усовершенствований нотации.

Сейбел: По сравнению с тем, что было прежде, безусловно, это шаг вперед. Но за время, прошедшее с появления указателей, многие пришли к мнению, что указатели в чистом виде весьма опасны и что лучше бы иметь ссылки, которые ведут себя как указатели, но при этом более безопасны.

Кнут: Указатели уже настолько вышли из моды, что мне приходится вступать по этому поводу в споры. Если говорить о моем 64-разрядном компьютере, то, если действительно заботиться о производительности моего компьютера, мне приходится признать, что лучше отказаться от использования указателей, поскольку на моей машине 64-битные регистры, но всего 2 гигабайта оперативной памяти. Поэтому у указателя никогда не бывает больше 32 значащих битов. Но каждый раз, когда я использую указатель, это стоит мне 64 бита, и это удваивает размер моей структуры данных. Более того, это еще идет и в кэш-память, и половины кэш-памяти как не бывало, а за это приходится платить – кэш-память дорогая.

Поэтому я на самом деле пытаюсь сейчас пробовать новые варианты, то есть мне приходится вместо указателей использовать массивы. Я создаю сложные макросы, то есть создаю видимость использования указателей, хотя на самом деле их не использую. Можно сказать, что это незначительное явление, и оно выходит из моды. Но для меня это было важным новшеством в нотации на низком уровне. Когда я пишу код, занимаюсь отладкой или еще чем-то в этом духе, то всегда испытываю огромную благодарность Томпсону и Ричи. Не знаю, кто именно первым это придумал.

Сейбел: Есть ли в вашем программистском арсенале другие важные инструменты?

Кнут: Файлы изменений – их я придумал после того, как уже стал использовать методы литературного программирования, и я не знаю их аналогов в инструментариях других программистов, поэтому позвольте объяснить их вам.

Когда я написал TeX и METAFONT, люди стали просить их у меня. У всех этих людей было 200–300 различных комбинаций языков программирования, операционных систем и компьютеров, поэтому я хотел сделать так, чтобы мой код с легкостью можно было адаптировать к любой системе. И мы разработали такое решение: я пишу главную программу, которая работает в Стэнфордском университете, после чего

создается дополнение – *файл изменений (change file)*, который может адаптировать эту главную программу для работы на чьем угодно компьютере.

Файл изменений очень прост. Он состоит из ряда небольших фрагментов изменений. Каждое изменение начинается с нескольких строк кода. Вы сравниваете до тех пор, пока не находите первую строку файла главной программы, которая совпадает с первой строкой вашего изменения. Когда вы добираетесь до конца той части изменения, которую нужно было соотнести с главным файлом, появляется часть, в которой написано: «Замените это вот этими строками».

Может быть, изменение будет состоять в следующем: «Замените эти шесть строк вот этими двенадцатью строками. Или ничем не заменяйте. Как только найдете совпадение, вставляйте те двенадцать строк, что вы изменили. Затем переходите к следующей части». Вам необходимо написать изменения по порядку – никаких интеллектуальных процессов, связанных со сравнением, нет; просто программа говорит: «Сравнивайте до тех пор, пока не найдете первую строку следующего изменения, которая должна совпасть с какой-либо строкой из главного файла».

Эту систему можно написать за час, и она достаточно хорошо справляется со своей задачей. После чего все инструменты, которые есть в нашем распоряжении для литературного программирования – программы «сплетания» и «спутывания», – будут работать с главным файлом и файлом изменений.

Поэтому мне время от времени приходится выпускать новую главную программу. У сотен людей по всему миру есть свои файлы изменений – возможно, их шесть строк, которые должны совпасть с моими, уже не совпадают, поэтому им нужно внести ряд изменений. Но им не приходится делать слишком много. Каждый раз, когда я исправляю ошибку, программа практически сразу заработает – исправление ошибки также отражается и на работе их программ. То есть проблема была решена очень просто, и все сработало. Любой может это понять и сделать.

Примером крайности в этой области может служить случай, имевший место, когда TeX адаптировался для работы с Unicode. У них был файл изменений раз в десять больше главной программы. Другими словами, из 8-битной программы они сделали 16-битную, но вместо того чтобы пройти и переделать мою главную программу, они были настолько увлечены файлами изменений, что просто написали свои файлы изменений и назвали это Omega, – миллион строк файлов изменений для 20 000 строк кода TeX. Это крайность.

Но сейчас я постоянно использую файлы изменений, потому что пишу для себя программы, которые использую в своей книге, – есть множество задач, в которых я бы хотел разобраться, и мне бы хотелось поэкспериментировать с различными версиями. Например, вчера я захотел выяснить, насколько большой является булева схема для перемножения n -битных чисел. То есть у меня есть программа, которая берет любую булевскую функцию и вычисляет ее BDD.

В моей исходной программе нужно ввести таблицу истинности функции в процессе работы; она говорит: «Введите таблицу истинности», – и я ввожу шестнадцатеричное число, потому что у меня есть множество небольших функций, которые я использую в качестве примеров. Но это работает только для небольших функций, которые я хочу ввести в таблицу истинности.

Есть и крупная функция, например «Перемножить все пары 8-битных чисел». Эта функция от 16 переменных – 8 бит в x и 8 бит в y . Поэтому я пишу небольшой файл изменений, который убирает этот интерактивный диалог и заменяет его программой, составляющей таблицу истинности для умножения.

Затем я заменил это фразами вроде «Прочтем биты справа налево, а не слева направо – получится другая BDD» или «Попробуем все булевы функции от шести переменных, посмотрим их все и выясним, у которой из них наибольшая BDD». Но это все лишь вариации моей исходной программы.

У меня наберется около 15 вариантов этой программы, и все они абсолютно доступны. Это было неожиданным ответвлением от литературного программирования, возникшим из-за того, что нам надо было посылать главные файлы множеству людей, которые изменяли их для своих систем. Сейчас я использую его совершенно по-другому.

Сейбел: В общем-то, нетрудно догадаться, почему этот инструмент может быть вам полезен, учитывая вашу текущую работу, в рамках которой приходится составлять множество вариаций на одну и ту же тему.

Кнут: Да, я пишу книгу.

Сейбел: Как по-вашему, этот механизм может получить более широкое применение?

Кнут: Не знаю. Не предполагаю, как бы все это происходило, работай я в команде из 50 человек. Но надеюсь, что программист-одиночка, пишущий программы, чтобы чему-то научиться, не вымирающий вид.

Сейбел: В начале карьеры вы занимались машинным кодом, затем перешли на структурное программирование, которое предоставило – логичным образом – структуру для ваших программ. Затем вы изобрели

литературное программирование, с помощью которого стали структурировать программы по-новому. Появилось ли с момента изобретения литературного программирования еще что-то, столь же кардинально изменившее ваши взгляды на программирование?

Кнут: У меня теперь есть более эффективные инструменты для отладки в рамках литературного программирования – вот, по большому счету, и все.

Сейбел: Хорошо, давайте поговорим об отладке. Что это за более эффективные инструменты, которые вы сейчас используете?

Кнут: Как оказалось, изобретатели отладчика GNU поняли, что препроцессоры можно использовать для написания программ. То есть можно устанавливать соотношение между низкоуровневыми объектами и высокоуровневыми источниками в совершенно разных языках. Другими словами, я могу писать на CWEB, но никогда не смотреть на низкоуровневые вещи, потому что они отображаются в моем CWEB-источнике по мере моего продвижения по программе.

Сейбел: То есть речь идет о средстве, встроенном в GDB, которое используется CWEB?

Кнут: И оно было встроено в GDB, потому что оно было встроено в Си, для того чтобы заполучить директивы `__LINE__`. Нам пришлось поработать, для того чтобы использовать директивы `__LINE__`, но теперь все работает превосходно. У компьютера есть только двоичная инструкция, но GDB знает, что это было получено из моего исходного WEB-файла, несмотря на то что WEB появился через 10 или 20 лет после Си. Следовательно, с их стороны было очень хорошим, дальновидным решением выполнить эту работу.

Сейбел: То есть вы используете GDB. Какие еще инструменты отладки вы применяете?

Кнут: Я добавляю много кода, проверяющего качество исполнения моих структур данных, со всеми их повторами. Эта проверка достоверности, когда она включена, может замедлить выполнение программы на два порядка.

Например, у меня была сложная структура данных, в которой использовался подсчет ссылок. Я пишу довольно сложные программы, и процесс подсчета ссылок усложняется. Время от времени мне приходится то увеличивать, то уменьшать счетчик ссылок. Но когда указатель находится в регистре или является параметром подпрограммы, считается это ссылкой в структуре данных или нет? Поэтому я пишу проверку, которая проходит через миллион счетчиков, проверяя, сколько ссылок действительно было сделано и все ли сходится. Затем я делаю небольшие

вычисления и проверяю все целиком. Таким образом, ошибки будут обнаружены за миллиард шагов до того, как они проявят себя и все рухнет.

Была одна программа, которая выполняла умножение с помощью нового способа, и я усиленно ее тестировал. Я составил ряд из 256 чисел и перемножил каждое с каждым, а после каждого шага делал проверку. Умножаю 2 на 3 – сбой! Исправляю. Потом еще что-нибудь. В конце концов я добрался до того момента, когда все 256 правильно умножались друг на друга.

Это очень важная техника отладки для меня. Наверное, около 10% кода посвящено тому, что мне нужно только во время отладки. Кроме того, код проверки также документирует структуру данных.

Кроме того, я обычно пишу программу, которая переводит структуру данных в удобную знаковую форму, чтобы избежать декодирования всего двоичного кода. Также при необходимости я могу распечатать структуру данных в хорошо организованной форме или скинуть ее в один файл и написать еще одну программу, которая проанализирует первую и найдет, что пошло не так.

Сейбел: Говоря об инвариантах и различных видах операторов утверждений, люди вроде Дейкстры говорят, что мы должны устанавливать очень формальные операторы утверждения на каждом этапе своих программ, чтобы в дальнейшем суметь доказать корректность этих программ. Я читал ваши рассуждения о том, что корректность программы нужно доказывать «неформально». Как вы думаете, стоит ли идти дальше и формально доказывать корректность программы?

Кнут: С одной стороны, невозможно что-либо доказать в принципе. Когда кто-то говорит, что его программа верифицирована, это значит, что она лишь соответствует неким критериям согласно результатам работы верификатора. Но в верификаторе может быть ошибка. Критерии могут быть неправильно сформулированы. Никогда нельзя с точностью утверждать, что программа корректна. У вас может быть достаточно причин, чтобы в это верить, но при этом вы никогда не добываетесь до конца цикла. Это теоретически невозможно.

Самая первая работа Тони Хоара про формальное доказательство называлась «Proof of a Program: FIND (Доказательство программы: FIND)». Это было прекрасное научное исследование, которое весьма продвинуло общее положение дел в этой области. Но в этом доказательстве было 2–3 ошибки. Тогда людям не приходило в голову проверять, находится ли список индексов внутри границ или еще что-нибудь в этом духе. Всегда есть опасность появления пробелов. Тем не менее он осуществил гораздо более качественную и тщательную проверку, чем кто-либо до него.

Или вот вчера я закончил программу и понятия не имею, как установить для нее все операторы утверждений. Мне это никогда не сделать, потому что я никогда не буду больше уверен в утверждениях, нежели в самой программе.

Или взять, к примеру, TeX – с формальной точки зрения там полный бардак. Задумывалось, что эту программу будут использовать люди, а не компьютеры. Было бы абсолютно невозможно сформулировать, что значит корректность для TeX. Некоторые методы формальной семантики настолько сложны, что никто не может внятно определить понятие *корректности*.

Сейбел: Работая над TeX, вы написали совершенно зубодробительный тест для программы.

Кнут: Верно.

Сейбел: Каким образом вам удается так относиться к своим программам? Программисты зачастую хотят защитить свое дитя, поэтому они, как правило, стараются не очень усердно тестировать свои программы.

Кнут: Я всю жизнь был придирчивым. И чтобы получить удовольствие от обнаружения ошибок, мне нужно просто забыть, что программу написал я сам. Я пытаюсь представить, что это чья-то чужая программа. В остальных же случаях мне достаточно легко переключиться в режим нападения. Не знаю почему.

Например, одно из моих лучших профессиональных достижений во время работы в Burroughs Corporation было связано с деятельностью по обнаружению ошибок в устройстве их аппаратного обеспечения. Их инженеры показывали мне спецификации для своих компьютеров, а мне нужно было сконструировать примеры, при которых они бы ошибались на единицу или что-то вроде того. Я нашел свыше 200 ошибок в их компьютерах серии В-5000, прежде чем они поступили в производство, хотя перед этим они прошли тест на эмуляторах.

Сейбел: То есть, по сути, вы изобретали программы, которые были верны с точки зрения семантики языка, но машина исполняла их некорректно?

Кнут: Да. Конечно же, если у них плавающая запятая неправильно вычисляла произведение двух чисел, я пытался найти примеры чисел, когда плавающая запятая не работала. Кроме того, иногда они реализовывали стек аппаратно, и у них были ситуации, когда регистры были пусты или не на верху стека, и я пытался придумать сценарии, в которых их логика не работала.

Сейбел: Вы делали это по какой-то системе? Как вы находили все эти вещи?

Кнут: Может быть, я просто излишне придирчив? Не знаю. Но если я пытаюсь что-то доказать, например теорему, если речь о математике, а не доказать правоту чего-либо, то мне, как правило, проще сказать: «Давайте найдем контрпример». Я могу просто завестись, отыскивая дырку во всем этом или пытаюсь объяснить, почему это не работает. Если же не могу найти никаких дырок, то вижу доказательство.

Думаю, все дело во мне – я люблю придираться и отыскивать ошибки. Мне гораздо интереснее, когда я выступаю против чего-либо, а не просто сижу и приговариваю: «Ага, да, это работает. Интересно, почему?»

Сейбел: Любопытно, что именно это движет вас вперед, хотя всю свою жизнь вы посвятили объяснению проблем. Не думаете ли вы, что такой подход подпитывает ваше стремление все объяснять?

Кнут: Единственное, что можно утверждать по поводу моих объяснений, – я пытаюсь совместить в естественном мыслительном процессе наблюдения за явлениями одновременно два разных способа, чтобы понять что-либо лучше. Думаю, чрезвычайно важно видеть вещи объемно, а не в одном измерении. Не знаю, как это связано с моей придирчивостью.

Но когда нападаешь, пытаешься кого-то победить, как в игре, это пробуждает соревновательный дух, что-то такое, что стимулирует мой мозг, так что я пытаюсь решить поставленную передо мной задачу более чем одним способом. С хорошим объяснением дело обстоит так же. В хорошем объяснении каким-то образом сочетаются разные точки зрения.

Сейбел: Еще один вывод, который вы вынесли из работы над TeX – вы об этом писали в «The Errors of TeX» (Ошибки TeX), – нужно фиксировать каждую ошибку, обнаруженную в программе. Ребята вроде сотрудников Института разработки ПО считают, что процесс разработки ПО можно назвать взвешенным и зрелым, если вы отслеживаете все ваши ошибки и пытаетесь понять, как предотвратить подобные ошибки в будущем. Но вы говорили о том, что ведение журнала ошибок никоим образом не помогает вам предотвращать появление ошибок в будущем.

Кнут: Верно. Хотя и не сказать, что без этого журнала ничего бы не изменилось в худшую сторону.

Сейбел: Но разве вы не думали: «Ага, теперь, когда я увидел эту ошибку, я ее больше не повторю»?

Кнут: Мне просто нужно узнать и признать свои грехи. Если использовать богословскую терминологию, то можно сказать, что все люди нуждаются в отпущении грехов.

Сейбел: То есть вы замечали ошибки в своих программах и думали про себя: «Ох, я снова сделал ту же самую ошибку».

Кнут: Да.

Сейбел: Почему же так? Может быть, дело в какой-то особой природе ошибок, что особенно затрудняет усвоение урока и недопущение подобных ошибок в будущем?

Кнут: Думаю, скорее всего, дело в том, что я пытаюсь заниматься все более сложными вещами. Я всегда пробую то, что требует от меня максимум усилий. Если бы мне нужно было сейчас вернуться и написать эти программы еще раз – те, которые полегче, – я бы не допустил так много ошибок. Но теперь, зная немного больше, я пытаюсь писать более сложные программы. То есть я совершаю ошибки, потому что всегда работаю на пределе своих способностей. Работать, никогда не покидая зоны комфорта, – это же скучно.

Сейбел: То есть теоретически вы могли бы продолжать писать системы для верстки текста до конца своей жизни?

Кнут: Да, они бы у меня выходили весьма неплохо. Но мы постоянно поднимаем для себя планку и неизбежно задеваем ее. Мы работаем с тем, как уже говорилось, что находится на пределе человеческих возможностей, и то, что мы делаем сейчас, еще сложнее того, с чем нам приходилось сталкиваться раньше.

Если же мы ограничиваем себя лишь тем, что действительно очень просто, то нам этого недостаточно, потому что наши аппетиты постоянно растут, мы всегда стремимся расширить границы своих возможностей, чтобы двигаться вперед, пока не доходим до того, что удастся нам с величайшим трудом. А добравшись до этого момента, мы снова хотим расширить границы своих возможностей и так далее.

Таким образом, нам никуда не деться от ошибок, разве что мы откажемся писать программы, расширяющие наши способности. Так как же нам делать это лучше? Каждые три года появляется новое модное словечко для обозначения новой панацеи – той, что решит все проблемы и с чем все будет работать без сбоев.

Последние два-три года новой панацеей считалось экстремальное программирование. До этого было еще что-то. Потом еще кто-нибудь придумает очередное решение всех проблем, и многие вспрыгнут на подножку этого поезда, но потом им придется задуматься: «Черт, а ведь по-прежнему не все так уж легко».

Сейбел: Изменился ли со временем тип человека, который может стать хорошим программистом?

Кнут: Судя по моему многолетнему опыту, практически всегда, когда я встречаю 100 человек из той или иной группы населения (не считая студентов специальности «компьютерные науки»), двое из них – программисты, в том смысле что действительно находят общий язык с ма-

пиной. В городе Василла (Аляска) проживает 10 000 человек, соответственно там должно быть 200 программистов.

Сейбел: Так изменилось ли программирование за это время настолько, что изменился тип людей, попадающих в эти два процента? Или тут все по-прежнему?

Кнут: Не знаю, ведь понятие «программирование» можно трактовать по-разному. Мы всегда пытаемся создавать инструменты, благодаря которым компьютерные процессы как можно больше походили бы на то, что происходит в мозгу человека. Я скорее говорю о таком способе работы машины, когда она пытается раздвинуть горизонты, а не просто решить задачу.

Сегодня машины настолько мощны, что люди, не очень-то смыслящие в программировании – в моем элитарном представлении, – способны решать с помощью этих машин задачи, которые на прежних машинах могли решать только очень сильные специалисты. А с новыми машинами эти люди, о которых я говорю, смогут решать даже те задачи, которые было невозможно решить с помощью старых компьютеров.

Помимо этой перемены есть и другое изменение, которое меня действительно беспокоит: сегодняшнее программирование, по большей части, совсем неинтересно, поскольку сводится к подбору волшебных заклинаний – нужно просто совмещать части чужих программ, делая из них новые программы. В этом не очень-то много творчества. Мне не нравится, что программирование становится скучным, предоставляя мало возможностей для создания чего-то нового. Удовольствие достигается за счет того, что на выходе, в компьютере, вы видите какие-то интересные результаты, но это не то удовольствие, которое я всегда испытывал, создавая что-то новое. Сейчас же удовольствие получается от того, что делаешь-делаешь скучную работу, а потом – откуда ни возьмись – прекрасный результат. Но раньше эта работа никогда не была скучной.

Сейбел: Но вам ведь все еще интересно программировать самому?

Кнут: Разумеется, да. Мне просто *необходимо* программировать. Я просыпаюсь утром, и у меня уже готово несколько предложений литературной программы. Перед завтраком – уверен, поэтам это знакомо – я должен сесть за компьютер и записать этот абзац, и только потом могу идти есть, и я счастлив. Без этого я не могу, должен это признать.

Ладно, давайте я расскажу о программе, которую написал вчера. Я умножаю большие целые числа, которые значительно больше Вселенной; это особые целые числа – для работы их можно ужать, поэтому я могу с ними работать, несмотря на то, что не могу выразить их с помощью обычной нотации. Я умножал эти невероятно огромные целые

числа, возводил их в квадрат и выяснял, как они выглядят после возведения в квадрат. Я пока мало понимаю, что там происходит, но мне это очень интересно.

Сейбел: Вы занимаетесь наукой, кроме того, участвовали в создании крупных систем и работали в промышленности. Какой вы видите связь между научными исследованиями в области компьютерных наук и их практическим применением в промышленности?

Кнут: Ситуация похожа на качели. В 1960-х наука сильно опережала промышленность, и программы, создаваемые в промышленных компаниях – кроме разве что системы заказа авиабилетов, – были абсолютно смехотворны для людей из университетов.

К 1980 году ситуация кардинальным образом изменилась: программы, написанные учеными, вызывали смех в промышленных компаниях, потому что наука ушла в теологию, и было запрещено использование операторов перехода GOTO. Я, конечно, утрирую – чтобы не усложнять, – но, по большому счету, в университетских программах были табу, которые связывали по рукам и ногам, а у тех, кто был занят в промышленности, ограничений не было.

Но затем люди в университетах придумали более эффективные идеи сетевых технологий и работы с крупными фрагментами данных и прочим – и вырвались вперед. То есть ситуация постоянно меняется. Но в области разработки алгоритмов и структур данных за последние годы наметилась тенденция, которая меня совсем не радует: создается множество структур данных, просто... *вычурных* – другого слова не могу подобрать. Они чрезвычайно сложны, умны и восхищают интеллектуального размахом, но мне они кажутся бесплодными. Они не имеют никакого отношения к жизни, они работают в каком-то своем мире. Это нормальный мир, у него есть своя структура; это милые и дружелюбные люди, но их программы лично мне не нравятся, и они совершенно несовместимы с практикой.

Не знаю, почему мне так важно – совместимо что-либо с практикой или нет. Есть математики, которые вообще никогда не думают о чем-либо конечном и практически никогда не работают со счетно бесконечными множествами; они публикуют потрясающие работы, в которых говорят о безумных видах бесконечности, про которые они все понимают, и это доставляет им удовольствие. То же самое можно обнаружить и в области изучения алгоритмов. Что касается меня, то мне намного больше нравится работать с идеями, которые я могу использовать на своем компьютере.

Сейбел: В 1974 году вы писали о том, что к 1984 году у нас будет «Утопия 84» – идеальный язык программирования, который вытеснит Ко-

бол и Фортран; кроме того, вы говорили о явных признаках того, что такой язык постепенно обретает реальные очертания. С 1984 года прошло уже два десятилетия – похоже, ваши прогнозы не сбылись.

Кнут: Нет, ничего такого не было.

Сейбел: Это в вас тогда говорил ваш юношеский оптимизм?

Кнут: Видимо, когда я писал об этом, то думал о Симуле и тенденциях в объектно-ориентированном программировании. Думаю, на самом деле с появлением каждого нового языка приводятся в порядок знания о старых языках и добавляется что-то новое, экспериментальное и так далее; но никогда не было так, что создается новый язык и все – дальше идти некуда, можно останавливаться на уже достигнутом. Всегда появляется желание развиваться дальше.

Может быть, когда-нибудь кто-то скажет: «Хватит, я не собираюсь быть новатором; я просто хочу быть внятным и понятным – этих принципов и буду придерживаться». Паскаль начинал с подобной философии, но не продолжил ее. Может, мы и доживем до того времени, когда кто-нибудь скажет: «Давайте-ка сузим наш угол зрения и постараемся сделать что-то стабильное». Возможно, это окажется неплохой идеей.

Сейбел: Не является ли частью проблемы тот момент, что помимо ошибок есть и недостающие детали – а если чего-то недостает, то нужно как-то заполнить эту пустоту.

Кнут: Да, верно. Каким-то образом языки должны быть расширяемыми. Java не стала расширяемой в хорошем смысле этого слова.

Сейбел: Вы и сами создали несколько языков, пожалуй, самый популярный из них – TeX.

Кнут: Ну да, TeX – это язык программирования, но все эти дополнительные свойства я добавил после долгих споров и препираний. Гаю Стилу, Терри Винограду, Лесли Лэмпорту и другим были нужны определенные вещи, когда они использовали TeX в качестве внешнего интерфейса при работе над своим материалом. Кажется, Терри Виноград писал книгу о синтаксисе естественных языков, поэтому ему были нужны по-настоящему мощные макросы, чтобы сделать диаграммы для своей книги. Именно это подтолкнуло TeX по направлению к тому, чтобы превратиться в язык программирования уже в начале своего существования.

Сейбел: Вам когда-нибудь хотелось плотнее заняться разработкой языка как такового?

Кнут: Не знаю. Наверное, да. Мне по-своему очень не нравится, что считается, будто каждый язык должен быть универсальным, потому что они будут универсальными – но по-разному. Как в UNIX есть 30 опреде-

лений регулярных выражений в одном флаконе – в зависимости от того, какую часть UNIX вы используете, вам предлагается немного отличное представление о регулярных выражениях. Если внутри каждого вашего инструмента находится машина Тьюринга – разве это нормально? Я на самом деле думал про TeX, что чем больше в нем языка программирования, тем меньше в нем того, что непосредственно связано с версткой.

Вводя расчет простых чисел в руководство по использованию TeX, я не считал это должным использованием TeX. Я представлял себе это так: «Да, кстати, глядите-ка: собаки могут стоять на задних лапах, а TeX может вычислять простые числа».

Сейбел: Но люди пользуются тем, что это тьюринг-полный язык программирования для вычислений, связанных с версткой. Если бы он не обладал полнотой по Тьюрингу, им бы не удалось делать все это.

Кнут: Да, это так. В 1960-х годах я написал язык программирования для эмуляции, на завершение которого потратил много сил, поскольку у него было множество пользователей, но когда появилась Симула, она понравилась мне больше, и я просил людей забыть о моем языке SOL. Вообще-то, мне не кажется, что у меня какой-то особенный талант к разработке языков.

Во время работы над TeX я обращался к сотням лет истории человечества, не желая отказываться от всего того, до чего дизайнеры книг дошли за многие века, и начинать заново, сказав: «В общем, забудьте тех ребят; знаете, мы будем рассуждать логически». В этом случае суть дела сводилась, в основном, к тому, чтобы выбрать чудовищно сложную задачу и найти сравнительно небольшой набор базисных элементов для ее поддержки. Вместо 1000 элементов у меня 100 базисных элементов или около того. Но пойти дальше, сведя все к 50 или 10 элементам – что нужно было бы сделать для математической ясности, – мне кажется, такой путь просто не сработает. Задача изготовления книг очень сильно взаимосвязана со сложностью реального мира, который просто противится любым упрощениям.

Сейбел: Я не проводил никаких специальных исследований, но у меня складывается такое ощущение, что подавляющее большинство математических и научных работ в наши дни сверстаны в TeX. Наверняка вы не раз видели какие-то исследования, сверстанные в TeX, и думали: «Ого, моя программа играет здесь определенную роль».

Кнут: Доказательство великой теоремы Ферма – один из таких случаев. Это одна из самых известных математических работ. И все равно постоянно встречаются книги, которые, я знаю, не были бы написаны, если бы их авторы шли по проторенным дорожкам. Опять же это своего рода проблема черных ящиков.

Как было раньше: что-то печатаешь, отдаешь это наборщику, затем получаешь гранки и так далее. Проходишь через всевозможных людей разных уровней, которые не являются математиками, но от которых зависит конечный результат. Поэтому не рискуешь сделать что-либо такое, что сможет смутить людей во всей этой цепочке.

Но если можешь сам наблюдать за тем, что из всего этого получается, и можешь сам разработать систему нотации, а не взять ее из чьего-то списка стилей, потому что точно знаешь, какой должна быть нотация для твоей работы, то у тебя больше шансов воодушевиться на создание гораздо более качественной работы.

Поэтому мне всегда очень приятно осознавать тот факт, что люди смогли пробиться через все это и что плоды их творчества поступают напрямую к их читателям.

Сейбел: Как по-вашему, программисты и ученые в области компьютерных наук хорошо знают историю нашей области знания? Ведь она, в общем-то, не столь продолжительна.

Кнут: Не так уж и много сейчас ученых. Даже когда я начинал писать книги – в 1963 году, – мне не казалось, что люди знали тогда, что происходило в 1959 году. На прошлой неделе я прочел в «American Scientist», что был заново открыт алгоритм, который Бойер и Мур открыли в 1980 году. Я на каждом шагу сталкиваюсь с тем, что люди не осознают все величие нашей истории. Многим молодым программистам кажется странной мысль о том, что в 1970-х люди тоже что-то знали, понимали и умели.

Неизбежно, что в такой сложной области знания люди чего-то не знают. Надеюсь, с такими вещами, как Википедия, достижения не будут так легко предаваться забвению, как это происходило раньше. Но надеюсь и на то, что мне удастся многим привить свою любовь к чтению первоисточников. Не просто знать, что такой-то известен тем, что сделал то-то, но обратиться к его работам и прочитать, что говорил этот человек, его собственные слова. Думаю, это превосходный способ улучшить свои навыки.

Очень важно уметь понимать образ мышления другого человека, декодировать его словарь, нотацию. Если вы поймете что-то о том, как они думали, и о том, как делали свои открытия, то это поможет вам делать собственные открытия. Я часто читаю первоисточники – что гении прошлого говорили о той или иной проблеме. Их мысли выражаются в непривычных для сегодняшнего человека формулировках, но они стоят того, чтобы разбираться в их нотации и пытаться как следует вникнуть в их идеи.

Например, я долго изучал вавилонские рукописи, чтобы узнать, как 4000 лет назад описывались алгоритмы и что они обо всем этом думали, были ли у них циклы и прочее и как их описывали. Для меня эта работа, безусловно, была не напрасной, поскольку я обогатил свое понимание работы человеческого мозга и понял, как тогда делались открытия.

Несколько лет назад я нашел старинную рукопись на санскрите, датированную XIII веком, которая была посвящена проблемам комбинаторной математики. Практически никто из тех, кого знал автор, не смог бы даже приблизительно понять, о чем он говорил. Но я нашел перевод этой рукописи – и она оказалось мне очень близка. Я думал сходным образом, когда только начал программировать. Чтение первоисточников очень сильно обогащает меня в личном плане и в том, что касается моего творческого потенциала.

Мне не удалось передать ничего этого своим студентам. В современной компьютерной науке есть те, кому это удастся (их не так много). Но я могу пересчитать по пальцам одной руки тех, кто разделяет мою любовь к чтению первоисточников.

У меня большая коллекция исходных кодов. Есть компиляторы, компиляторы Digitek 1960-х, созданные в очень интересной манере. В них использован собственный язык, идентификаторы длиной 30 символов, но они были очень описательными; эти компиляторы заметно превосходили своих тогдашних конкурентов – эта компания производила мейнстримовые компиляторы в 1963 или 1964 году.

У меня есть исходный код Дейкстры операционной системы TNE. Я его не читал, лишь пока проглядел, но я нашел его, потому что уверен, что мне интересно будет почитать его в свободное время. Однажды я сломал руку, катаясь на велосипеде, и практически ничего не мог делать целый месяц, поэтому читал исходный код, в котором, как я слышал, применялись некоторые интересные недокументированные идеи. Думаю, этот опыт был очень важен для меня.

Сейбел: Как вы читаете исходный код? Ведь непросто читать даже то, что написано на известном вам языке программирования.

Кнут: Но это действительно того стоит, если говорить о том, что выстраивается в вашей голове. Как я читаю код? Когда-то была машина под названием Bunker Ramo 300, и кто-то мне однажды сказал, что компилятор Фортрана для этой машины работает чрезвычайно быстро, но никто не понимает почему. Я заполучил копию его исходного кода. У меня не было руководства по этому компьютеру, поэтому я даже не был уверен, какой это был машинный язык.

Но я взялся за это, посчитав интересной задачей. Я нашел `BEGIN` и начал разбираться. В кодах операций есть ряд двухбуквенных мнемоник, поэтому я мог начать анализировать: «Возможно, это инструкция загрузки, а это, возможно, инструкция перехода». Кроме того, я знал, что это компилятор Фортрана, и иногда он обращался к седьмой колонке перфокарты – там он мог определить, комментарий это или нет.

Спустя три часа я кое-что понял об этом компьютере. Затем обнаружил огромные таблицы ветвлений. То есть это была своего рода головоломка, и я продолжал рисовать небольшие схемы, как разведчик, пытающийся разгадать секретный шифр. Но я знал, что программа работает, и знал, что это компилятор Фортрана – это не был шифр, в том смысле что программа не была написана с сознательной целью запутать. Все дело было в коде, поскольку у меня не было руководства по компьютеру.

В конце концов мне удалось выяснить, почему компилятор работал так быстро. К сожалению, дело было не в гениальных алгоритмах – просто там применялись методы неструктурированного программирования и код был максимально оптимизирован вручную.

По большому счету, именно так и должна решаться головоломка: составляются таблицы, схемы, информация извлекается по крупицам, выдвигается гипотеза. В общем, когда я читаю техническую работу, это такая же сложная задача. Я пытаюсь влезть в голову автора, понять, в чем состоял его замысел. Чем больше вы учитесь читать вещи, написанные другими, тем более способны изобретать что-то свое – так мне кажется.

Мы должны публиковать код. Доступны комментарии Джона Лайонса к 6-й версии UNIX, с исходным кодом. И программы Билла Аткинсона теперь находятся в открытом доступе благодаря Apple, и уже скоро мы сможем их прочитать. Это очень хорошо документированный код со множеством новаторских графических алгоритмов.

Сейбел: Конечно, благодаря политике открытого исходного кода у нас сейчас гораздо больше кода для чтения, чем раньше.

Кнут: Да, это так. И по-прежнему каждый может применять разные виды нотации – не стоит читать только тех программистов, которые пишут так же, как и вы.

Библиография

The Art of Computer Programming, Donald Knuth (Addison-Wesley, 1997).¹

Beautiful Code: Leading Programmers Explain How They Think, Andy Oram, Greg Wilson (eds.) (O'Reilly, 2007).²

Byte, Vol. 6, No. 8, "Smalltalk issue," August 1981.

Code Complete, Steve McConnell (Microsoft Press, 1993).³

Compiling with Continuations, Andrew W. Appel (Cambridge University Press, 1992).

The Design and Analysis of Computer Algorithms, Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley, 1974).

Design Patterns: Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides (Addison-Wesley Professional, 1994).⁴

A Discipline of Programming, Edsger W. Dijkstra (Prentice Hall, Inc., 1976).

Effective Java, Joshua Bloch (Prentice Hall, 2008).⁵

The Elements of Programming Style, Brian Kernighan and P.J. Plauger (Computing McGraw-Hill, 1978).⁶

Elements of Style, William Strunk and E.B. White (Longman, 1999).

Expert C Programming, Peter van der Linden (Prentice Hall PTR, 1994).

Founders at Work, Jessica Livingston (Apress, 2007).

¹ Дональд Э. Кнут «Искусство программирования». – Вильямс, 2008.

² Под ред. Энди Орама и Грегга Уилсона «Идеальный код». – СПб.: Питер, 2009.

³ Стивен Макконнелл «Совершенный код». – СПб.: Питер, 2007.

⁴ Эрик Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – СПб.: Питер, 2007.

⁵ Джошуа Блох «Java. Эффективное программирование». – Лори, 2002.

⁶ Керниган Б., Плджер Ф. «Элементы стиля программирования». – М.: Радио и связь, 1984.

Hacker's Delight, Hank Warren (Addison-Wesley, 2002).¹

Higher-Order Perl, Mark Jason Dominus (Morgan Kaufmann, 2005).

Java Concurrency in Practice, Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006).

Java Puzzlers: Traps, Pitfalls, and Corner Cases, Joshua Bloch and Neil Gafter (Addison-Wesley, 2005).

The Lisp 1.5 Programmer's Manual, John McCarthy (MIT Press, 1962).

Literate Programming, Donald Knuth (Center for the Study of Language and Information, 1992).

Machine Intelligence 1, N.L. Collins and Donald Michie (eds.) (Oliver and Boyd, 1967).

Machine Intelligence 2, Ella Dale and Donald Michie (eds.) (Oliver and Boyd, 1968).

Machine Intelligence 3, Donald Michie (ed.) (Edinburgh University Press, 1968).

Machine Intelligence 4, Bernard Meltzer and Donald Michie (eds.) (Edinburgh University Press, 1969).

Magic House of Numbers, Irving Adler (HarperCollins, 1974).

"META II a Syntax-Oriented Compiler Writing Language," D.V. Schorre in *Proceedings of the 1964 19th ACM national conference*, (ACM, 1964).

Mindstorms: Children, Computers, and Powerful Ideas, Seymour A. Papert (Basic Books, 1993).

The Mythical Man-Month: Essays on Software Engineering, Frederick P. Brooks (Addison-Wesley Professional, 1995).²

Principles of Compiler Design, Alfred Aho and Jeffrey Ullman (Addison-Wesley, 1977).³

"Proof of a Program: FIND", C.A.R. Hoare in *Communications of the ACM*, Vol. 14, Issue 1 (ACM, 1971).

Programming Pearls, Jon Bentley (ACM Press, 1999).

¹ Генри Уоррен мл. «Алгоритмические трюки для программистов». – М.: Вильямс, 2007.

² Ф. Брукс «Мифический человек-месяц или Как создаются программные системы». – СПб.: Символ-Плюс, 2000.

³ А. Ахо, Р. Сети, Д. Ульман «Компиляторы. Принципы, технологии, инструменты». – Вильямс, 2003.

Purely Functional Data Structures, Chris Okasaki (Cambridge University Press, 2008).

A Retargetable C Compiler: Design and Implementation, David Hanson and Christopher Fraser (Addison-Wesley Professional, 1995).

Smalltalk-80: The Interactive Programming Environment, Adele Goldberg (Addison-Wesley, 1983).

Smalltalk-80: The Language & Its Implementation, David Robson and Adele Goldberg (Addison-Wesley, 1983).

Structure and Interpretation of Computer Programs, Harold Abelson and Gerald Jay Sussman (MIT Press, 1996).¹

TeX: The Program, Donald Knuth (Addison-Wesley, 1986).

The Programming Language LISP: Its Operation and Applications, Edmund Berkeley and Daniel Bobrow, eds. (MIT Press, 1966).

The Psychology of Computer Programming: Silver Anniversary Edition, Gerald Weinberg (Dorset House, 1998).

The TeXbook, Donald Knuth (Addison-Wesley Professional, 1986).²

Writers at Work: The Paris Review Interviews, Malcolm Cowley (Penguin, 1977).

Zen and the Art of Motorcycle Maintenance: An Inquiry into Values, Robert Pirsig (Bantam, 1984).

¹ Х. Абельсон, Д. Сассман «Структура и интерпретация компьютерных программ». – Добросвет, 2006.

² Дональд Кнут «Все про TeX». – М.: «Вильямс», 2003.

Алфавитный указатель

Символы

++, оператор, 106

А

ACM, Association for Computing Machinery, 284, 289, 358

ActionScript, язык программирования, 138

Ada, язык программирования, 142

ADsafe, 101

Adventure, игра, 502

Agorics, 97

Ajax, технология создания веб-приложений, 93, 98, 102, 103, 111, 123, 138, 152

Altair, компьютер, 222

Android, платформа, 76

ANSI C, языковой стандарт, 364

AOL, компания, 59

Apache, веб-сервер, 63

APL, язык программирования, 16, 95, 283, 285, 286, 295, 298, 304, 305

App Engine, сервис хостинга сайтов, 69, 71

Apple II, компьютер, 56, 131

ARPANET, сеть, 451, 452, 460, 470

Atari 800, 96

Atlas, 103

awk, язык программирования, 140

В

Basic Four, 95, 120

BBN, компания, 451–458, 466, 469, 472, 477–481

BCPL, язык программирования, 222, 223, 230, 397

BDD, 501, 509, 513

Belle, шахматный компьютер, 391

Bell Labs, компания, 391, 398, 403, 404, 413

BigInteger, утилита, 167

BitBlt, 321, 327, 330–333, 339, 340, 352, 353

Bliss, язык программирования, 283, 305, 402

С

C, язык программирования, 133, 306, 437

C++, язык программирования, 47, 67–69, 89, 134, 135, 142, 143, 150, 155–157, 162, 174, 175, 179, 181, 185, 206, 224, 235, 283, 302, 305, 306, 325, 335, 350, 407, 413, 414, 437, 460, 486

C--, язык программирования, 238

C*, язык программирования, 283, 305

CGI, интерфейс, 59, 60, 61, 63

Chrome, браузер, 76

Chromix, операционная система, 224

Chronomancer, 147, 148

Cleanroom, процесс, 428, 429

Common Lisp, язык программирования, 17, 283, 293, 297, 303, 305, 314, 319

Connection Machine Lisp, язык программирования, 305

CWEB, инструмент, 498, 501, 504, 514

D

DEC, компания, 285, 288, 305

DOCTOR, виртуальный собеседник, 451, 467, 470

Е

E, язык программирования, 97, 379

ECMAScript, язык программирования, 283, 303

версия 3 (ES3), 93, 98, 101, 303

версия 3.1, 138, 141

версия 4 (ES4), 93, 94, 98, 100, 101, 130, 138, 139, 141, 151, 303
версия 5 (ES5), 93
Eiffel, язык программирования, 119, 237
EISCAT, научная ассоциация, 192, 195
Electric Communities, 93, 97, 112
Emacs Lisp, диалект Lisp, 25
Emacs, семейство текстовых редакторов, 45, 48, 88, 239, 284, 291, 298–300, 305, 363, 382, 471, 483
Encina, система обработки распределенных транзакций, 159
Energize, компания, 26, 40
Energize, среда разработки, 23
ENIAC, компьютер, 438, 440, 444
Ericsson, компания, 191, 192, 203, 204
Erlang, язык программирования, 191, 192, 196–198, 202–209, 213, 214, 216–218, 237
eToys, система, 329, 337
Expert Technologies, ETI, 18, 19

F

FastGCI, 63
Firefox, браузер, 129, 134
Fortress, язык программирования, 283, 294, 300, 302, 307, 308, 314
Free Software Foundation (FSF), компания, 24
FreeVote, сайт, 60
FX, язык программирования, 381

G

Gaim, сервис мгновенных сообщений, 75
GCC, набор компиляторов, 410
GDB, инструмент отладки, 147, 514
Genie, проект, 357, 361, 369
GHC (Glasgow Haskell Compiler), компилятор, 219, 235–241, 248–250
GHCI, цикл, 239
Ghostscript, программный комплект, 358, 362, 364, 371, 375, 376, 386, 387
GNAL, язык программирования, 283, 305
Google, компания, 61, 67–69, 76–78, 80–83, 85, 89, 90, 156, 159, 173–175,

255, 262, 266–269, 271, 274, 280, 282, 412–414, 418

Google, поисковая система, 255, 264

Google Web Toolkit (GWT), компилятор, 101

GOTO, оператор, 520

H

Habitat, игра, 96, 112

Haskell, язык программирования, 219, 227, 230, 233, 235, 239, 240, 242, 246–248, 250, 252, 262, 283, 305, 308, 309, 314, 379

Helgrind, динамический детектор гонок, 148

HyperCard, система, 140, 329

HyperTalk, язык программирования, 140

I

IBM, компания, 283–286, 295, 305, 312, 421–423, 426–428, 430, 434, 435, 440–442, 444, 445, 447–449

IMLAC, 488

Instant messenger (IM), сервис мгновенных сообщений, 31

Interface Message Processors (IMPs), компьютеры, 451, 452, 461, 463, 465–467, 477, 485

Interlisp, среда, 357, 358, 362, 383

J

Java, язык программирования, 24–26, 33, 42, 43, 50, 55, 58, 67–71, 86, 89, 130, 132, 137, 138, 140–143, 154, 155, 157, 159, 160, 162, 164–169, 173–175, 179–186, 283, 290, 294, 297, 302–308, 325, 326, 334, 335, 407, 410, 414, 437, 486, 508, 521

Java Collections Framework, набор классов и интерфейсов, 159

Java Community Process, 302

Javadoc, генератор документации, 307

JavaScript, язык программирования, 93, 95, 97–103, 105, 106, 108, 116–118, 120–123, 125–127, 129, 130, 132, 135–138, 144, 149, 152, 154, 158, 322, 326–328, 337, 346, 349, 351

JIT, just-in-time compilation, 357, 421
Joule, язык программирования, 97
JSLint, отладчик, 114, 117, 119
JSON, формат передачи данных, 93, 123
JsUnit, приложение для тестирования, 121

К

K&R, стиль оформления кода, 108

Л

«Lambda Papers», серия статей, 283
Linux, операционная система, 417
Lisp, язык программирования, 125, 467
LiveJournal, блог, 54, 55
Lively Kernel, среда разработки, 322, 326–329, 336, 339, 348
Lotus 1-2-3, электронная таблица, 342, 353
LR-анализатор, 491
Lucasfilm, студия, 93, 96, 97
Lucid Common Lisp, язык программирования, 20
Lucid, компания, 20–22, 24, 26, 27

М

MAC, проект, 453, 454, 457
MacLisp, язык программирования, 283, 288, 295, 296, 305, 313
Macsyma, система компьютерной алгебры и язык программирования, 287, 288, 295, 313, 415
MAD, система, 425, 426
Maple, язык программирования, 415
MapReduce, система, 412
Memcached, программа, 62, 64, 65, 69, 87
METAFONT, язык программирования, 317, 491, 511
Microsoft Research, подразделение Microsoft, 219, 220, 226, 229, 248
MicroUnity, компания, 129, 133
MMIX, эмулятор, 501
Mozilla, браузер, 129
Mozilla, компания, 36, 129, 134, 138, 145, 149–152
MS-DOS, операционная система, 363

MULTICS, операционная система, 125, 391, 398, 399, 403
MySQL, СУБД, 63, 66, 72

N

NCSA/Mosaic, браузер, 27
NELIAC, язык программирования, 394, 402
Netscape, браузер, 24, 25, 28, 41, 42
Netscape, компания, 27, 33, 43, 47, 129, 133, 134, 137, 138, 150, 151, 157, 158
NewtonScript, язык программирования, 140

O

OCaml, язык программирования, 155, 156, 205
OTP (Open Telecom Platform), платформа, 191, 196, 211

P

ParcPlace, компания, 358, 362, 377, 381
Perlbal, балансировщик нагрузки, 55, 64, 65, 67, 90
Perl, язык программирования, 24, 25, 60, 61, 65, 67, 70, 81, 83, 85, 90, 306, 384, 459, 472, 475, 485, 486
«ping of death», атака, 407
Plan 9, операционная система, 391, 417
PL/I, язык программирования, 325
PL/Z, язык программирования, 224
POPLmark, инструмент, 147
PostScript, язык, 362, 363, 364, 376
PowerPoint, программа, 255
PTRAN, проект, 421, 425, 430, 431, 434, 440, 441, 445
Purely Functional Data Structures, 247
Pycore, проект, 382
Python, язык программирования, 152, 227, 232, 308, 378, 381–386, 389, 475

Q

QuickCheck, программа, 240, 242

R

REPL, цикл, 239
Replay, 148

Research Staff Member, RSM, 432
Ruby, язык программирования, 142, 152

S

S-1 Lisp, язык программирования, 305
Scala, язык программирования, 181, 185
Scheme, язык программирования, 97,
126, 185, 283, 303, 305, 306
SEI (Институт программной инженерии), 482
Self, язык программирования, 139, 140
SGI (Silicon Graphics), компания, 129,
132, 133, 136, 146
Sibelius, программа для написания музыки, 388
Six Apart, компания, 80
SKIM (SKI Machine), компьютер, 231
Smalltalk, язык программирования, 113,
125, 126, 139, 142, 154, 321, 322, 324,
325, 329, 331–337, 340, 341, 343–346,
348, 349, 351–353, 357, 362, 378, 381,
382
SML/NJ, 144
Spice Lisp, язык программирования, 17
Squeak, версия языка Smalltalk, 321,
326, 327, 333, 337, 339, 344, 346, 350
SRI, 95
SSA-представление, 421, 431
STRETCH-HARVEST, компьютер, 421,
424–427, 433, 434, 441, 443, 448
Sun Microsystems, компания, 159, 162,
358
Symbolics, 279

T

Tcl, язык программирования, 459, 480
TECO, язык программирования, 305
TECO-макросы, 461, 472, 485
Tektronix, компания, 58, 59
TENEX, операционная система, 452
TeX, система компьютерной верстки,
115, 270, 283, 284, 292, 305, 317–319,
330, 344, 491, 496–498, 502, 511, 512,
516, 517, 521, 522
Tk, библиотека, 488
TI Explorer Lisp, компьютеры, 19
TraceMonkey, виртуальная машина, 129
Transarc, компания, 159, 176, 178

U

UML, язык моделирования, 171, 271
UML-схемы, 236
UNIVAC, компьютер, 368
UNIX, группа операционных систем,
23, 27–29, 33, 59, 60, 63, 130, 131, 142,
146, 150, 154, 198, 211, 258, 357, 369,
391, 397, 399, 402–404, 416, 420, 456,
487
UTF-8, кодировка, 391

V

Valgrind, компания, 145
VisiCalc, электронная таблица, 342

W

WEB, инструмент, 498, 514

X

Xenix, операционная система, 132
Xerox PARC, компания, 321, 323, 357,
365
XLISP, язык программирования, 17
XP, экстремальное программирование,
374, 376
XScreenSaver, 25, 26, 35, 37
X Windows, программа, 196, 206

Y

уасс, программа, 415
Yahoo!, поисковая система, 93, 103, 111

A

Абельсон, Хэл, 248, 265
абстракция, 34
Ада, язык программирования, 302, 484
аксиома проектирования API, 171
акторы, 97, 185, 379
Алгол, язык программирования, 220,
224
Аллен, Фрэн, 487
Адрессен, Марк, 27, 30
антифишинг, 99
Армстронг, Джо, 144

B

багтрекер, 240
Баттерфилд, Стив, 475

Бейсик, язык программирования, 16,
56, 67, 160, 161, 162, 216, 256, 257,
321, 324, 329, 332
Беркли, университет, 357, 393, 394, 396
бинарный поиск, 215
Бина, Эрик, 29
битовые поля, 257
Би, язык программирования, 397
Боброу, Дэн, 373, 402
боты, 59
Британская робототехническая ассоциа-
ция, 192, 194
Брукс, Фред, 428, 429
булевы схемы решений (BDD), 501
Бьёрклунд, Мартин, 211

В

важнейшие навыки, 51, 90
Вайсман, Терри, 31
вариативность, 180
вейвлет-преобразования, 206
Вейзенбаум, Джо, 451, 470, 471
Вейссман, Терри, 30, 31, 32, 33
Вейсс, Стив, 456
Виноград, Терри, 521
Вирдинг, Роберт, 204, 210, 211, 215
виртуальная машина, 204
владение кодом, 78, 188
вывод типов, 235
Хиндли-Милнера, 143
выражения, 380
выращивание языка, 139
выявление талантов, 47

Г

Гал, Андреас, 145
Гамильтон, Грэм, 180
Гарвардский университет, 283, 286–289,
322, 357, 368
Гейтс, Билл, 216
Гетц, Брайан, 164, 176
Гилдер, Джордж, 133
Гиффорд, Дэйв, 381
Голдберг, Адель, 154
Голдин, Дон, 274
гонки, 120
Гослинг, Джеймс, 180
Госпер, Билл, 454

Гринблатт, Ричард, 454
Гровер, Джордж, 427
группа пользователей Apple, 17
Гуибас, Лео, 365
Гэбриел, Ричард, 305, 403

Д

Де Беллис, Томас, 161
Дейкстра, Эдсгер, 122, 173, 214, 248,
263, 491, 493, 507, 510, 515, 524
декомпозиция, 370
Джой, Билл, 283
Диас, Джон, 238
динамическая компиляция, 357
динамическая типизация, 207
динамические языки, 135
Дойч, Питер, 321, 333
доказательства, 284, 313–316
документация, 21, 169, 171, 184, 212,
213, 401, 402, 404

Ж

женщины-программисты, 422, 426, 427,
441–448
Живой журнал, 54–56, 60–63, 65, 73,
78–81, 84, 86

З

Завински, Джейми, 129, 134, 138, 150,
158
Закон Мура, 214, 251
Закон отладки Джо, 209
замыкания, 97, 303, 308, 332
запросы к X11, 37, 38
Зигмонд, Дэн, 17

И

Иборра, Пепе, 240
Иллинойский университет в Урбана-
Шампейне, 132
инварианты, 84, 145, 276, 314, 349, 408,
496, 515
Ингаллс, Дэн, 214
интенсивная работа, 29
информатика, 191
информационный поиск, 372
исключение, 106
искусственный интеллект, 192, 266

«Искусство программирования», 247, 491, 494, 498, 503, 505, 508

Исследовательская лаборатория электроники, 453

исследовательский центр IBM, 175

история развития программирования, 494, 503, 522, 523

Й

Йегг, Стив, 89, 261

Йост, Дэйв, 133

К

качество кода, 30

Кембриджский университет, 219, 221–225, 230

Кембриджский ускоритель электронов, 357–360

Керниган, Брайан, 153

Кларк, Джим, 132

Кларк, Стивен, 228

Кларк, Томас, 222, 231

кластер базы данных, 64

Клуб технического моделирования железной дороги, 453

ключевые слова в контексте, 285

книги, 50, 52, 85, 153, 154, 166, 265, 284, 285, 287, 289, 345, 416, 458

Кнут, Дональд, 114–116, 247, 270

Кобол, язык программирования, 283, 305, 323, 343, 520

кодер и программист, 373

код на Си, 24

Кок, Джон, 448

Колумбийский университет, 159

команда разработчиков, 46, 47

комбинаторные алгоритмы, 505

комбинаторы S и K, 230

комментарии, 45, 470, 474, 483

коммит, 151

компилятор GNU, 68

компьютерные науки, 49, 71, 73, 127, 131, 133, 145, 154, 194, 201, 214, 222, 224–226, 256, 258, 262, 263, 274, 288, 290, 300, 326, 335, 338, 345, 353, 374, 394, 422, 423, 437, 440, 444–447, 492–494, 520, 523

конвейерный параллелизм, 217

контрактное программирование, 237, 314

красота кода, 215, 482

Крокфорд, Дуглас, 137, 139, 141, 143

Кроутер, Уилл, 457, 460, 462, 464–468, 476

Кэглер, Тед, 334

Кэй, Алан, 96, 321, 324, 325, 329, 335, 337, 347

Кэнеди, Радд, 404

Л

Лайтхилл, Джеймс, 194

Ланетт, Марк, 29

латентность памяти, 426

ленивые вычисления, 219, 231, 232, 234, 235

Лирсон, Винсент, 284

Лисп-машины, 403

Лисп, язык программирования, 17, 18, 20, 22, 40, 283, 286–289, 292, 296, 298, 303–305, 322, 325, 333, 357, 359, 361, 362, 368, 372, 383–387, 451, 460, 470, 471

литературное программирование, 109, 114, 115, 278, 279, 293, 351, 415, 491, 496–498, 500, 503, 510–513, 519

Лондонский университетский колледж, 193, 225

Лэмпорт, Лесли, 521

любопытство в программировании, 51

лямбда-выражения, 101, 103, 230, 308

М

Макилрой, Дуглас, 150, 151

Маккроски, Майк, 167

Мартин, Билл, 286, 287

Массачусетский технологический институт, 283, 285–288, 290, 292, 296, 298, 314, 357, 359, 361, 368, 371, 373, 381, 403, 451, 453, 454, 457, 458

математика и программирование, 52, 81, 85, 122, 167, 173, 263, 289, 290, 300, 301, 314, 318, 327, 341, 345, 392

мгновенные сообщения, 31

метод грубой силы, 397

Миранда, Элиот, 383

Миттельхаузер, Джон, 29

Мичи, Дональд, 191, 192, 194
«модель водопада», 429
модульное тестирование, 314
модульность, 233
модульные тесты, 43
Мозес, Джоэль, 287
монады, 234, 308, 379
Монтулли, Лу, 29
Морган, Боб, 456
Морнингстар, Чип, 96, 107
Мор, Тренчард, 297
Мур, Джей Стротер II, 386
Муэрс, Кэлвин, 372
мьютексы, 142, 177
мэшапы, 100, 126

Н

навыки, 136, 145, 206, 214, 216, 257, 259, 261, 264, 273, 290, 371, 373, 432, 469, 475, 477, 480
написание кода, 203
НАСА, 255, 272, 273, 274, 275, 276
наставничество, 48, 435
Натт, Рой, 424
низкоуровневое программирование на ассемблере, 163
Норвиг, Питер, 19, 143, 149
Норман, Артур, 230, 247
Нью-Йоркский университет, 431, 434, 445

О

обертки, 205
обобщенные типы, 180, 181, 182
О большое, нотация, 491
обратная разработка, 21
объектно-ориентированное программирование (ООП), 162, 224, 377, 521
объектно-ориентированные языки, 162, 197
объектно-ориентированный дизайн, 334
Овицки, Сьюзен, 315
Одерски, Мартин, 185
О'Каллагэн, Роберт, 147
О'Лири, Уилфрид, 286
операторы
 continue, 107
 перехода GOTO, 491

 утверждений, 41, 42, 119
оптимизация, 84, 403, 409, 410
оптимистический параллелизм, 245
организация кода, 46
отладка, 22, 40–42, 51, 83, 90, 119, 120, 147, 172, 176, 178, 179, 193, 208, 239–241, 261, 265, 274, 276, 277, 291, 292, 296, 311–313, 315, 316, 333, 340, 342, 344, 348, 349, 394, 400, 401, 405–409, 418, 424, 425, 448, 452, 456, 463, 465, 467, 475, 476, 496, 511, 514
отладчик GNU, 22, 27, 41
отслеживание кода, 291, 294

П

Палач, Кшиштоф, 339
параллелизм, 217, 430–432, 439
 данных, 217
параллельное программирование, 217
парное программирование, 77, 210, 211, 261, 340, 418
парсер, 259, 329, 425
парсинг, 332, 385
Паскаль, язык программирования, 58, 130, 131, 291, 293, 302, 484, 485
перенос данных, 64, 80
переписывание кода, 44
переработки, 481
петафлопсы, 437
ПЛ/1, язык программирования, 285, 302, 433
почтовая программа, 30, 31, 33
предварительное объявление переменных, 109
приложение, 354
приятельское программирование, 187
проблема второй системы, 110
проверка типов, 380
программирование и сочинительство, 37
программисты-самоучки, 50, 74, 126
программное обеспечение, 74
проектирование ПО, 168, 169, 171, 237, 267, 284, 295
Пролог, язык программирования, 191, 198, 204, 206, 207, 214
прототипы, 268
профилировщик, 323
«пыльные колоды», 430

Р

разделение времени, 453, 455–457, 463, 467, 470, 472, 477, 488
разработка через тестирование, 266, 268
ревизии, 272
 кода, 78
 проекта, 468
рекурсия, 310
рефакторинг кода, 107, 110, 117, 170, 248, 479
Ричи, Деннис, 391, 402, 410
Россум, Гвидо ван, 272
Рочестер, Нэт, 440
ручное управление памятью против автоматического, 69
Рэмзи, Норман, 238

С

Сан-Франциско, 94
Сассмен, Джеральд, 248, 265, 283
сближение функциональных возможностей, 103
сборка мусора, 348, 363, 407, 410, 437
сверхурочная работа, 79
сертификат удобочитаемости, 78
сигнатуры типов, 236
сигнатуры функций, 364
символы подстановки, 180
Симула-67, язык программирования, 378
синдром второй системы, 28, 32
синтаксический сахар, 138
Си, язык программирования, 24, 40, 77, 130, 139, 142, 174, 179, 184, 198, 204, 205, 283, 290, 305, 334, 350, 363, 386, 397, 400, 406, 436, 484, 487, 510, 514
«Словарь хакера», 284
Снобол, язык программирования, 415
создание программ, 399
сортировка слиянием, 286
списочные ячейки, 44, 152
Спольски, Джоэл, 150
средства обобщенного программирования, 181
стандарт оформления кода, 77
Стеллмен, Ричард, 23, 137, 299, 300
Стил, Гай, 139, 243, 521
стиль оформления K&R, 108

Стой, Уильям, 231
Страуструп, Бьёрн, 306, 413, 414
Стрейчи, Кристофер, 256
структура кода, 149
Стэнфордский университет, 130, 322–324, 344, 496, 500, 511
суперкомпьютер ACS-1, 421, 442, 443
сценарии использования, 169

Т

талант программиста, 411
тестирование, 42, 43, 121, 228, 240, 265, 269, 271, 296, 316, 330, 338, 339, 350, 408
тесты, 268
Технологический институт Кейса, 492
Томпсон, Кен, 272, 357, 484
Тотич, Алекс, 29
транзакционная память, 185, 186, 187, 243, 246
Тьюринг, Алан, 194

У

Уайт, Джордж, 324
Уильямс, Майк, 204
Уиттакер, Брэд, 61
Унгар, Дэвид, 139
университеты
 Беркли, 19, 357, 393, 394, 396
 Гарвардский, 283, 286–289, 322, 357, 368
 Иллинойский в Урбана-Шампейне, 132
 Карнеги–Меллона, 17, 18, 159, 179, 305
 Кембриджский, 219, 221–225, 230
 Колумбийский, 159
 Санта-Клары, 130
 Стэнфордский, 130, 322–324, 344, 496, 500, 511
 штата Мичиган, 422, 423
Уолден, Дэйв, 460, 461, 466, 468
Уолл, Ларри, 145
Уошбрук, Джон, 225
утверждения, 119, 145, 173, 276, 307, 313, 314, 349, 408, 515, 516
Утопия 84, язык программирования, 520
Уэллингс, Ральф, 287

Ф

файлы изменений, 511
Фальман, Скотт, 17, 18
Фармер, Рэнди, 97
Фаулер, Мартин, 171
фигурные скобки, 107
Фицпатрик, Брэд, 54
Фокал, язык программирования, 305
Фортран, язык программирования 17,
93, 94, 106, 130, 160, 162, 163, 192,
195, 206, 208, 283–287, 304, 305, 308,
321–325, 341, 343, 344, 393, 397, 421–
424, 426, 427, 430, 436, 453, 464, 487,
502, 509, 521, 524, 525
фотохостинг, 62
Фрёберг, Магнус, 211
фреймворки, 103, 119, 196
функциональное программирование,
219, 220, 226, 227, 229–231, 234, 239,
246, 248, 250
функциональное сегментирование, 368
фэн-шуй, 215

Х

Харрис, Тим, 246
Харт, Фрэнк, 461, 466, 468
Хаук, Крис, 29
Херлихи, Морис, 246
Херман, Дэвид, 139
Хиндли, Милнер, 143
Хоар, Тони, 184, 207, 248, 253, 510, 515
Холи, Skef, 48
Хомский, Ноам, 310
Хорват, Вальдемар, 138
Хоули, Skef, 17
хрупкий код, 406
Хьюз, Джон, 232, 236
Хэмминг, Ричард, 215

Ч

Чарльз, Филипп, 425
Чёрч, Алонзо, 141
чтение кода, 94, 103–105, 107, 115, 126,
151, 152, 172, 291–294, 351, 417, 424,
474, 476, 524

Ш

Шведская космическая корпорация,
192, 195, 203, 209
Шенберг, Эдит, 447
Шорр, Вэл, 343

Э

Эйк, Брендан, 100
экстремальное программирование, 374
элементы нижнего уровня, 46
Элиза (ELIZA), виртуальный собесед-
ник, 451, 470
Элкинд, Джерри, 373
эстетика, 253, 317, 318, 319, 424
эффект второй системы, 112

Я

ядро, 327
Якобсон, Ван, 148

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 978-5-93286-188-2, название «Кодеры за работой. Размышления о ремесле программиста». Идеальная фотография со вспышкой» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.