The Peopleware Papers

Notes on the Human Side of Software

Larry L. Constantine

PRENTICE HALL

ПРОФЕССИОНАЛЬНО

Человеческий фактор

в программировании

Ларри Л. Константин

СИМВОЛ®

Санкт-Петербург — Москва
2004

Серия «Профессионально»

Ларри Л. Константин

Человеческий фактор в программировании

Перевод Ю. Асотова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Михайлов*
Редактор	*Ю. Куниевер*
Верстка	*Н. Макарова*
Корректор	*С. Беляева*

Константин Л.
Человеческий фактор в программировании. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 384 с.
ISBN 5-93286-044-8

Хорошее программное обеспечение создается людьми. Так же как и плохое. Именно поэтому основная тема этой книги – не аппаратное и не программное обеспечение, а человеческий фактор в программировании (peopleware). Первое издание «Constantine on Peopleware» признано классическим трудом в области информационных технологий. Новая книга Ларри Константина включает все 52 легендарные статьи из предыдущей книги и 25 новых эссе.

Peopleware охватывает все аспекты, связанные с ролью людей в разработке программного обеспечения. Это качество и продуктивность, модели и методы, динамика поведения коллектива, руководство проектами, разработка интерфейсов и взаимодействие между человеком и компьютером, психология и процессы мышления. В данное издание включены два новых раздела, посвященных организационной культуре и юзабилити программных продуктов.

ISBN 5-93286-044-8
ISBN 0-13-060123-3 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 04.11.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 24 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Посвящение

Памяти Денниса Дэви и Кена МакКензи –
и всех, кто открыл путь

Благодарности

Ни одна книга не является результатом труда одного человека, а эта, с ее упором на сотрудничество и совместную инициативу, с самого начала создавалась совместными усилиями. В ее основе лежат написанные мною статьи, которые появились благодаря настойчивости и дальновидности моих редакторов, Ларри О'Брайэна (Larry O'Brien) и Мэри Лензи (Marie Lenzie). Ларри заманил меня в мир постоянных крайних сроков, предложив лучшее место на страницах журнала *Computer Language Magazine* для публикации статей обо всем, что меня интересует. Мэри привела меня в журнал *Object Magazine*, где среди знатоков процессов и технологий я был чудаком, ориентированным на человеческий фактор. И Ларри, и Мэри были не только хорошими редакторами, но и моими хорошими друзьями в течение многих лет. Усилия Ларри поддержала группа талантливых и очень терпеливых людей из Miller Freeman Publications, включая Гретхен Бэй (Gretchen Bay), Николь Фримэн (Nicole Freeman), Мишель Гэхи (Michele Gehee) и Николь Кларо (Nicole Claro), которые справились с причудами моего стиля и смогли снести мои порой невыносимые лекции по электронной почте.

Выражаю свою благодарность Полу Петралии (Paul Petralia), редактору издательства Prentice Hall, за энтузиазм, с которым он вдохнул новую жизнь в старый проект. За последние штрихи профессионализма в рукописи я благодарен Мэри Судул (Mary Sudul), которая была усердным выпускающим редактором этой книги, и Хэриет Телем (Harriet Tellem), внесшей существенный вклад в первое издание.

В список редакторов, которые помогали мне в подготовке книги, я добавил бы имя моего лучшего критика и советчика, Люси Локвуд (Lucy Lockwood). Она мой партнер в жизни и в работе и обычно первая читает то, что я пишу, и первая говорит о моих писательских взлетах и падениях. Спасибо, Люси. Спасибо всем.

Предисловие

Другая сторона программного обеспечения

Эта книга о другой стороне программного обеспечения – той, что смотрит во внешний мир. Эта сторона компьютеров касается людей – технарей, как вы и я, и обычных людей, как вы и я. В собранных здесь заметках исследуются многие разнообразные аспекты человеческого фактора в программировании (peopleware), которые обеспечивают интерфейс между программным обеспечением и его разработчиками, а также между программным обеспечением и его пользователями.

Мои редакторы, как в журналах, в которых этот материал появился впервые, так и в издательстве Prentice Hall, разрешили мне затронуть большой круг вопросов. Тема человеческого фактора в программировании необъятна, и это позволило мне писать почти обо всем, о чем я хотел написать – начиная от организационной культуры и организации проектов, хаоса и дисциплины в кодировании, инструментов и методов программирования и заканчивая пользователями, юзабилити и пользовательскими интерфейсами. Эта широкая область охватывает особый промежуточный мир, в котором сливаются границы между техническими и социальными вопросами. Здесь психология встречается с кибернетикой, а теория и практика смешиваются друг с другом. Все это отражает мой давний личный и профессиональный интерес как к людям, так и к программному обеспечению для компьютеров.

Эта книга является пересмотренным, расширенным и обновленным переизданием книги «Constantine on Peopleware», Prentice Hall, 1995 [26]. Книга слишком радикально пересмотрена, чтобы называться вторым изданием, но в то же время она тесно связана со своей предшественницей. Читатели смогут найти здесь много новых материалов, освещающих данную тематику. К главам из первоначального издания добавлены 25 новых статей, которые впервые публикуются в виде книги. Статьи включают в себя все 52 заметки, которые изначально публиковались в *Computer Language Magazine* и *Software Development* под рубрикой «Peopleware»,

включая «потерянную заметку», появившуюся в самом конце этой серии (см. приложение). Кроме того, для удобства читателя я добавил еще семь близких по содержанию статей из журнала *Object Magazine*. Они особенно важны с точки зрения общего представления о том, что такое дизайн, ориентированный на использование. Описание этого подхода было улучшено и расширено в книге Люси Локвуд (Lucy Lockwood) «Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design», Addison-Wesley, 1999 [30], которая получила всеобщее признание.

В том, чтобы писать о людях, имеет одно большое преимущество – люди изменяются намного медленнее, чем технологии. При перечитывании и редактировании материалов для этой книги я часто поражался тому, как мало изменилось за все эти годы, в течение которых я пишу о человеческой стороне технологии. Проекты все еще превышают установленные для них бюджеты, продукты все еще создаются с бессовестным запаздыванием, ресурсы все еще безнадежно недостаточны для решения задач, а руководители проектов все еще бьются над тем, как лучше всего раскрыть в подчиненных творческий потенциал и направить его в нужное русло. Разработчики, со своей стороны, все еще раздражаются из-за ограничений, которые создают схемы проекта, инструменты моделирования и жесткая дисциплина при создании программного обеспечения. В свою очередь, пользователи все еще вынуждены стараться изо всех сил, чтобы понять программное обеспечение, которое хорошо говорит с компьютерами, но плохо – с обычными человеческими существами.

Тем не менее, если люди меняются мало, то в технологии произошли настолько радикальные изменения, что некоторые примеры и ссылки из ранних статей могут показаться чуть ли не архаичными. Например, статья о переходе от монохромного монитора к полноцветному выглядит возвратом к предыстории разработки программного обеспечения – описанием, предназначенным для новичков. Однако центральная тема этой статьи – правильное и неправильное применение цвета – до сих пор остается актуальной для Интернета. Поэтому с учетом изначальной идеи и цели книги я обновил содержание статей там, где это понадобилось с точки зрения текущей ситуации и корректности.

Данный расширенный сборник разбит на девять тематических разделов. Каждая заметка помещена в соответствующий раздел с тем, чтобы в какой-то мере сохранить последовательность в изложении. Были добавлены новые разделы, посвященные организационной культуре, а также юзабилити объектов программного обеспечения. Читатель быстро поймет, что каждая статья, будучи взаимосвязанной с другими, является, тем не менее, самостоятельным материалом. Думаю, что отчасти именно это вызывало интерес к моим журнальным заметкам и к первому сборни-

ку, поскольку каждая глава может быть прочитана независимо от других в такси или в перерыве между встречами.

Кроме того, новые главы размещены блоками для того, чтобы читатели первой книги смогли их быстро найти. Новые главы: 22–25, 31–32, 40–41, 43–49, 53–61, а также приложение.

Данный материал собран из разных мест. Я надеюсь, что эта книга еще долгое время будет оставаться ценным источником и продолжит раздражать, вдохновлять и просвещать этих удивительных людей, которые работают в индустрии программного обеспечения. В первую очередь я писал и продолжаю писать именно для них – дизайнеров, разработчиков и руководителей, которые изобретают и внедряют программное обеспечение.

Предисловие к первому изданию

Аппаратное и программное обеспечение и человеческий фактор в программировании

Хорошее программное обеспечение берет начало не в применении CASE[1]-инструментов, методов визуального программирования, ускоренной разработки программ или объектно-ориентированной технологии. Хорошее программное обеспечение создается людьми. Так же, как и плохое. В 1992 году я начал вести колонку в журнале, исходя из простого принципа: поскольку программное обеспечение создается и применяется людьми, то наилучшее понимание людей – как они работают, каким образом выполняют свою работу, как взаимодействуют между собой – является основой для усовершенствования программного обеспечения и процесса его разработки. Таким образом, основным предметом заметок в этой колонке было не аппаратное (hardware) и не программное (software) обеспечение, а человеческий фактор в программировании (peopleware).

В области, которая изобилует неологизмами, термин «peopleware» – один из немногих, которые действительно стоило выдумать. По-видимому, в печатном издании этот термин первым применил Питер Г. Ньюман (Peter G. Newmann), который, наверное, более всего известен по своим регулярным статьям о риске потребителей и опасностях компьютеров и компьютерных программ для человека. В 1976 году его статья «Peopleware in Systems» была опубликована в книге, получившей свое название от этой статьи. Судя по всему, этот термин придумал Меилир Пейдж-Джонс (Meilir Page-Jones), который использовал его в 1980 году в своей книге «Practical Guide to Structured Systems Design» (Практическое введение в проектирование структурированных систем) – книге, в конце концов сделавшей мою работу по структурному проектированию более понятной для обыч-

[1] CASE (computer-aided software engineering) – автоматизированное проектирование и создание программ. – *Примеч. науч. ред.*

ного программиста. Но постоянное место в лексиконе нашей отрасли этот термин обрел, скорее всего, после того как в 1987 году под таким заголовком вышла небольшая, но великолепная книга Тома ДеМарко (Tom DeMarco) и Тима Листера (Tim Lister) [33].[1] Таким образом, можно сказать, что название для своей колонки, «Peopleware», я позаимствовал из самых лучших источников.

На самом деле «человеческий фактор в программировании» (peopleware) является третьей волной компьютерной революции. Сначала произошел кризис, связанный с аппаратным обеспечением. Одно время мы думали, что наши проблемы возникают из-за аппаратного обеспечения. Нам казалось, что если бы только у нас были более быстрые и мощные компьютеры с большими объемами памяти и более совершенными периферийными устройствами, мы смогли бы создавать более совершенные системы и решать наши задачи. И вот мы получили лучшие компьютеры. Год от года аппаратное обеспечение становилось все быстрее, память – больше, а периферийные устройства – более разнообразными и эргономичными. Однако наши проблемы не исчезли. Мы все еще продолжали создавать системы, трудные для применения. Мы все также опаздывали с завершением своих проектов и превышали запланированный для них бюджет. Поэтому мы решили, что на самом деле проблема состоит в программном обеспечении, после чего линия фронта в компьютерной революции переместилась к рубежу, который многие стали называть «кризисом программного обеспечения». Если бы только у нас были более совершенные инструменты, языки более высокого уровня, более мощные библиотеки компонентов и программы для создания программ, мы смогли бы решать наши задачи и создавать хорошие системы вовремя и в пределах бюджета. Языки третьего поколения стали еще более сложными и породили языки четвертого поколения (4GL). Компиляторы становились все быстрее и эффективнее. Библиотеки компонентов многократного использования расширялись, редакторы стали контекстно-зависимыми, а разнообразные инструменты автоматизированной разработки программного обеспечения появлялись, как грибы после дождя. Вслед за структурной революцией, давшей нам структурное проектирование и анализ, стало развиваться и набирать популярность объектно-ориентированное направление. Однако графики работ по-прежнему не выполнялись, бюджеты превышались, а количество ошибок в программах упрямо не желало становиться меньше.

В конце концов, подобно Пого и его легендарным друзьям из Окифиноки, на своем опыте мы выяснили, в чем тут дело. Как мудро сказал этот ма-

[1] Т. ДеМарко и Т. Листер «Человеческий фактор: эффективные проекты и команды». – Пер. с англ. – СПб: Символ-Плюс, I кв. 2004 г.

ленький опоссум: «Мы встретили врага – это мы сами». И это действительно так. Все сводится к человеческому фактору в программировании. Мы являемся проблемой, и мы же являемся ее решением. Как удобно.

Человеческий фактор в программировании охватывает довольно широкую область. Сюда входит все, что связано с ролью людей в процессе разработки программного обеспечения и приложений. В заметках и в книге затрагиваются разнообразные темы: качество и продуктивность, работа в команде, динамика поведения коллектива, личность и программирование, руководство проектом и организационные вопросы, разработка интерфейса и взаимодействие между человеком и машиной, познавательная деятельность, психология, процессы мышления.

Все эти предметы интересуют и увлекают меня. Я получил ученую степень по менеджменту отчасти потому, что это позволило мне соединить компьютеры и теорию систем с психологией. Моя диссертация была посвящена психологии программирования на компьютере. В течение нескольких лет я знакомил тысячи студентов и десятки коллег с работами психолога Джорджа Миллера (George Miller) и его магическим числом (конечно, я имею в виду 7 ± 2 [1]). Структура схем была тщательно продумана таким образом, чтобы облегчить формирование визуальных понятий и решение задач. Связывание и сцепление, эти устоявшиеся метрики, лежащие в основе структурного проектирования, в действительности имеют отношение к восприятию людьми компьютерных программ. Сложными или простыми программы делает именно то, что кажется простым или сложным для программистов, которые эти программы пишут, сопровождают и модифицируют.

В известном смысле я могу избегать тему человека не больше, чем тему компьютеров. Казалось, мне удалось уйти от этих вопросов, когда в июле 1976 года я попрощался с компьютерной областью и объявил о своей независимости (Америка как раз отмечала двухсотлетие своего суверенитета). Будучи по образованию семейным врачом, я больше десяти лет потратил на частную практику и работу в медицинской организации, помогая супружеским парам, семьям или просто больным людям. Но силам вселенной было угодно снова вернуть меня в пределы технологии.

Человеческий фактор является перекрестком в этих пределах, пересечением, где сходятся магистрали из моих разных миров. Управление, организационное развитие, личность, моделирование, инструменты, методы, процессы, взаимодействие между компьютером и человеком и т. д. Во

[1] Число, выведенное в 1956 году психологом Джорджем Миллером, относится к возможности человека обрабатывать в кратковременной памяти только от пяти до девяти элементов. – *Примеч. науч. ред.*

всех этих областях в то или иное время я либо писал, либо работал, либо преподавал. Колонка в журнале дала мне повод еще раз окинуть взглядом панораму отрасли. Подобно Чарльзу Куралту (Charles Kuralt), я мог останавливаться для исследования интересной идеи, решать возникавшие задачи и проходить по главным дорогам и магистралям в области разработки программного обеспечения и приложений.

Эта книга является сборником путевых заметок, начатых еще в *Computer Language Magazine* и продолженных в переименованном *Software Development*. Мои заметки выходили под рубрикой с простым названием «Peopleware». В этой книге собраны заметки из этой колонки, а также несколько других, тесно связанных с этой темой и опубликованных в других местах. Эти статьи и эссе были отредактированы в интересах последовательности изложения. Восстановлены некоторые материалы, которые были сокращены при публикации статей в журналах. Таким образом, данная подборка является «редакцией режиссера», в которой статьи размещены в квазилогических разделах, чтобы создать определенную иллюзию организации материала. Однако это не энциклопедия или учебник, и даже не карта огромной территории под названием «человеческий фактор в программировании», а только путевой журнал одного странника.

Путешествие продолжается.

I

Групповая разработка

Введение

Микеланджело, этот гений-индивидуалист, не работал в одиночку. «Сикстинская часовня» не является творением одного художника. Это произведение, созданное группой мастеров под руководством Микеланджело. Очевидно, что его талант проявлялся не только в искусстве как таковом, но также и в искусстве управления командой. Своих художников и подмастерьев он называл по именам и прозвищам. Он поощрял инициативу и взаимовыручку, вселял чувство гордости за мастерство. На собрании, посвященном повышению качества программного обеспечения, он чувствовал бы себя как дома.

Разработка программного обеспечения – это групповой процесс. Независимо от того, идет ли речь об искусстве или о технике, большинство программных продуктов рождается в результате коллективных усилий, когда разработчики трудятся в согласии или разобщенно. Качество программ определяется продуктивностью обсуждений в группе, принимаемыми решениями и отклонениями от них. Создать хорошее программное обеспечение легче, когда разработчики трудятся слаженно, когда группа успешно справляется с конфликтами и использует свои ресурсы рационально и эффективно.

Групповая разработка связана и с развитием самой группы. Процессы, происходящие в группе, и правила, устанавливаемые в ней, могут превратить простое сборище людей в высокопроизводительную команду. К групповой разработке и построению команды многие программисты относятся с долей скептицизма. Им не нравятся лидеры, они не любят петь хором и принимать участие в групповых играх. Им кажется, что все это – лишь видимость командного духа. Эта искренность и открытость не для них. Они были наняты за искусность в программировании, а не за социальные навыки – в большей степени для кодирования, чем для сотрудничества.

Понимание языка, на котором говорят в группах, и процессов, протекающих в них, может быть настолько же важным, как и понимание языка программирования. Вот почему эта книга начинается с исследования того, как работают группы и как можно повысить качество создаваемого ими программного обеспечения.

1

Решения, решения

Путей есть два и больше. Путей всегда есть два и больше. Всю мою профессиональную жизнь этот простой принцип служил практическим ориентиром, который побуждал меня искать различные альтернативы при разработке программного обеспечения и организации трудового процесса. Однако признание существования различных вариантов влечет за собой необходимость выбора. Создание более совершенного программного обеспечения подразумевает выбор между различными вариантами. Более того, это творческий подход, вбирающий в себя самое лучшее из нескольких методов и потому превосходящий каждый из них. Хорошо организованные команды, в которых принятие решений и преодоление трудностей осуществляется на основе консенсуса, в наибольшей степени способны к принятию качественных решений и выработке такого творческого подхода. Тем не менее таким командам следует знать, как избегать некоторых ловушек, в которые обычно попадают группы. Для этого полезно исследовать секреты командной работы, основанной на согласии.

Я всегда считал, что способность принимать решения является одним из самых важных жизненных навыков. Этому можно научиться только опытным путем. Семьи и компании, достигшие успеха, стараются обеспечить достаточные возможности для такого рода практики. К середине карьеры типичный профессиональный программист уже решил бесчисленное множество задач и, вероятно, принял не одну тысячу решений. Естественно, от профессионалов мы вправе ожидать, что как раз в этом они обладают хорошими способностями. Однако большинство таких решений программист принимает самостоятельно, тогда как принятие решений и преодоление трудностей в группе – дело совсем другое.

Опасность посредственности

В давние времена, когда я только начал изучать менеджмент и теорию поведения групп в школе управления Sloan при Массачусетском технологическом институте, исследователи уделяли много внимания возможным слабым местам, типичным при преодолении трудностей и принятии решений в группах, особенно так называемому «поднятию порога риска«, а также обратной тенденции скатывания к посредственному среднему. В ту эпоху консерватизма даже демократически настроенные менеджеры больше беспокоились о том, чтобы не допустить поднятия порога риска, чем о прогрессирующей посредственности. Перевороты 70-х были еще впереди, а «групповое мышление» соответствовало духу времени.

Согласно исследованиям, коллективные решения зачастую сводились к более рискованным вариантам по сравнению с теми, которые сотрудники группы могли бы принять самостоятельно. Если применить эту модель к программированию, то можно было бы ожидать, что группы будут создавать программы, использующие экзотические структуры данных, или необычные алгоритмы, или неявные возможности языка. Однако другие исследования групповой динамики показали, что при выполнении задач и принятии решений в группах имеет место эффект нивелирования, который сводит результаты к своего рода наименьшему общему знаменателю индивидуальных способностей и вкладов. Казалось, что один человек принимает более верные решения..

Очевидно, что такие эффекты зависят главным образом от организации и методов управления в группе. В России, где я работал с консультантами и руководителями новых предприятий, «посредственность» доминировала благодаря старой советской системе. Для многих руководителей, учившихся еще при старом режиме, командная работа означала переход на уровень общей некомпетентности. В советской системе управления работа в команде часто позволяла избежать ответственности. Иногда коллективы умышленно снижали свою производительность. Отстаивание своей точки зрения или выдвижение новой, спорной идеи грозило не только недовольством коллег, но и возложением ответственности и ожиданием подобных инициатив в будущем. Зачем беспокоиться, если производительность никак не вознаграждается, а потерять работу в типичном советском коллективе трудно даже при большом желании?

Социальный и организационный климат, в котором работает группа, – это как раз то, что в действительности определяет возможность раскрытия потенциала. Для достижения лучших результатов корпоративная культура и само руководство группы должны активно поощрять и поддерживать новаторство и сотрудничество. В известном смысле, команды советского периода работали действительно хорошо, удовлетворяя *реальные* ожида-

ния начальников и политических руководителей, хотя их усилия больше сводились к прикрытию тылов, нежели к достижению результатов. Советские менеджеры говорили мне, что они научились «никогда не оказываться крайними».

Слегка управляя

Если процесс разработки и принятия решений основан на консенсусе, то роль группового лидера становится решающей – не только в создании общего климата сотрудничества, но также и с точки зрения нюансов осуществления руководства. Совместная разработка и принятие решений дает наилучшие результаты, если решение рождается из талантов всех участников команды и отражает их опыт, творческие способности и умение мыслить критически. Речь идет не об отражении «в среднем», а об истинном синтезе, в котором соединяются лучшие качества каждого сотрудника. Какими бы талантливыми и выдающимися ни были лидеры группы, но если они начинают проталкивать собственные решения, качество работы всей команды снижается.

Процесс руководства группой может быть едва заметным. Даже простое выражение своего мнения в неподходящий момент может отклонить группу от правильного пути и привести к худшему результату. Исследования показали, что если руководитель откладывает изложение собственных идей до тех пор, пока все участники группы или их большинство не представят свои, группа приходит к более совершенным решениям. Это означает, что лидер, который всего лишь высказывается слишком рано, возможно, снижает качество работы группы. Самоуверенные руководители, убежденные в том, что они всегда правы и знают все на свете, могут создать наибольшие трудности в работе.

Большинство лидеров проектов и руководителей среднего звена в сфере разработки программного обеспечения являются довольно сильными специалистами. Почти каждый из них поднялся из отделов программирования, системного анализа и проектирования. Они достигли своего положения потому, что разбираются в вопросах разработки программ. Многим из них трудно оставить клавиатуру и позволить кому-либо другому выполнять эту работу и решать технические вопросы.

Теперь мы знаем, что одним из самых важных факторов, способствующих наилучшему решению задач на основе консенсуса, является нейтральность руководства. Позиция ведущего обсуждение довольно сильна, поэтому тот, кто ведет или помогает вести собрание и обсуждение, должен быть нейтральным. Лишь в этом случае проявится наилучшее из того, что может предложить группа.

Такой лидер является другом для каждого и, в то же время, не является чьим-то адвокатом. Такой лидер стремится получить вклад от всех участников группы, никого из них не выделяя. Такой лидер помогает группе прийти к техническому консенсусу, но не оказывает влияния на исход обсуждения и не навязывает свой вариант решения.

Звучит странно, но это означает, что менеджеры проектов и официальные командные лидеры, вероятно, меньше всего подходят для ведения обсуждений или собраний, посвященных преодолению технических трудностей и принятию технических решений. Для них слишком многое стоит на кону. В известном смысле они, наверное, и знают слишком много. Чем сильнее они как лидеры, тем вероятнее то, что они будут сдерживать свободное исследование различных альтернатив и выработку единодушного технического решения, ведущего к наилучшему результату.

Некоторые руководители избирают подход полного невмешательства и стараются совсем не вдаваться в технические вопросы. Однако это не является полезным ни для команд, которые остаются без опыта и мастерства своих руководителей, ни для самих руководителей, которые упускают много интересного. Лучшие из них передадут ведение собраний или обсуждений нейтральному ассистенту, а сами, оставаясь в заднем ряду, станут учиться тому, как участвовать в работе, не доминируя. Некоторые так никогда и не смогут научиться этому, но многие из тех, с кем мне довелось работать, действительно получают удовольствие от того, что они снова могут быть «в одной связке» и на равных со всеми обсуждать технические вопросы.

Жизнь продолжается. Даже после того, как вас повысили в должности.

Из журнала *Computer Language Magazine*, том 9, № 3, март 1992 г.

2

Консенсус и компромисс

Возможность получить максимальную отдачу от команды, разрабатывающей программное обеспечение, зависит от способности профессионалов, вовлеченных в проект, приходить к техническому консенсусу. Но почему же так важно, согласны ли вы и ваш коллега в том, как должна выглядеть форма для заполнения или как следует выдавать сообщения об ошибках? Технический консенсус не касается того, насколько хорошо вы ладите с собратьями-программистами или насколько вы близки с ними по духу (естественно, нет ничего плохого в том, чтобы ладить между собой или хорошо относиться друг к другу). Технический консенсус подразумевает максимальное использование способностей и опыта каждого участника команды. Речь идет о создании более совершенного программного обеспечения.

Разработчики-профессионалы могут понимать, что такое хорошее программное обеспечение, или, по крайней мере, увидев его, заявить об этом. Но о техническом консенсусе у них смутное представление. Вероятно, у большинства разработчиков программного обеспечения накоплен негативный опыт в отношении того, что, по их мнению, было «разработкой на основе консенсуса». Они расскажут вам о том, какие блестящие идеи были загублены в дискуссиях, как была задета их творческая натура, как на шестимесячные проекты уходили годы и как группы довольствовались малым, далеким от лучшего. Послушайте эти истории внимательно, и вы поймете, что речь в них идет вовсе не о консенсусе, а о компромиссе. В чем же разница?

Пустой компромисс

Компромисс – это ни то ни се, нечто среднее, зачастую находящееся на полпути в никуда. Рассмотрим классический пример. Ваша команда занимается разработкой графического пользовательского интерфейса. Одна группа настаивает, что кнопки управления следует разместить в нижней части экрана, другая группа считает, что на левой стороне экрана нужно предусмотреть специальную панель. Между этими вертикальным и горизонтальным вариантами есть совершенно реальный компромисс, который может ошарашить: просто разместить эти кнопки по диагонали, проходящей по середине экрана!

Подобный компромисс зачастую хуже любого из исходных вариантов, но решение на основе консенсуса может быть лучше, чем все варианты, вместе взятые. Часто технические компромиссы не могут учесть достоинства каждой альтернативы. Их преимущества теряются, поскольку выбирается нечто среднее. Настоящий консенсус основан не на компромиссе, в котором каждый человек и каждый вариант что-то теряет, а на синтезе, когда выигрывает каждый. Несомненно, программное обеспечение от этого только выигрывает.

Синтез – это нечто оригинальное, вбирающее в себя важнейшие черты каждой идеи или предложения. В приведенном выше примере можно легко увидеть вариант творческого синтеза, в котором положение панели с кнопками выбирает сам пользователь. Консенсус, основанный на синтезе, включает в себя лучшее из альтернатив. Более того, такая комбинация обычно привносит новые свойства и возможности. Из синтеза горизонтальной и вертикальной панелей управления может появиться возможность пользовательской настройки. Таким образом, продукт вбирает в себя лучшее из двух идей, а не худшее.

Приход к настоящему консенсусу – это непростой процесс, о чем отлично известно всем политикам и посредникам в трудовых спорах. Достижение технического консенсуса немного отличается от прихода к политическому консенсусу, но у этих процессов есть общие черты. В обоих случаях необходимо стремление к успешному разрешению дела. В обоих случаях нужна вера в происходящий процесс.

Истинно верующие

Участники команды должны верить в достижимость варианта, в котором будет воплощено лучшее из всех предложенных идей. Веря в это, они будут упорно искать лучшие варианты, а не успокаиваться на компромиссе или упрямо стоять на своем. Сохраняя такую настойчивость, команда формирует свое понимание задачи, а также выявляет слабые и сильные

стороны каждого подхода. Поэтому шансы найти творческий вариант, превосходящий все предложенные подходы, увеличиваются.

Кроме того, согласие более достижимо, если каждый из нас верит, что подготовка более совершенного программного продукта важнее, чем принятие наших собственных идей в заранее предопределенной форме. Такое отношение к качеству конечного продукта дает возможность увидеть достоинства всех идей, рождающихся в групповом обсуждении.

Всегда полезно, когда работа в команде ценится больше, чем индивидуальное самовыражение. Компании, где индивидуальная эффективность поощряется больше, чем успех всей группы, а программистам-одиночкам отдается большее предпочтение, чем командным игрокам, обычно превращаются в сборище бескомпромиссных одиночек, которые, вероятно, не будут (да и не смогут) играть в командные игры. В таких компаниях приходят к справедливому выводу, что их лучшее программное обеспечение создается гениями, стремящимися к обособленности. Однако, руководители таких компаний не понимают, что сами же и создали эту ситуацию. Хотя, конечно, все может сложиться и по-другому.

Важнейшее правило достижения технического компромисса таково: никаких махинаций! Торговля голосами, поддержкой или влиянием – это одна из классических тактик для достижения политического успеха, но она может снизить эффективность технических решений. Например, при разработке интерфейса можно заключить сделку. Я соглашусь с вашей глупой идеей поместить панель с кнопками внизу экрана, если вы согласитесь с моим отличным предложением сделать пиктограммы без надписей. В результате мы получаем интерфейс, качество которого оставляет желать лучшего уже по двум признакам. Махинации – это тот же компромисс, только в другом, еще худшем виде, поскольку в этом случае одни решения портят другие. Для достижения настоящего согласия каждый технический вопрос должен рассматриваться отдельно, по существу, а не как часть некоторой системы подсчета очков, в которой уступки в одной области можно выторговать упрямством в другой.

Только факты

Принято считать, что технические решения принимаются в ходе обсуждения технических материй – фактов, количественных оценок, практического опыта. Правда же состоит в том, что чувства, мнения, интуиция и обыкновенные человеческие пристрастия играют роль во всех процессах принятия решений и преодоления трудностей. Это вполне объяснимо, поскольку люди остаются людьми. И хотя некоторые люди пытаются отрицать, сдерживать или подавлять такие иррациональные проявления, это никогда не удается в полной мере.

Важнейшим умением для любой команды, надеющейся достичь технического консенсуса, является умение отделять факты от мнений. Если группе необходимо найти наилучшие варианты и подойти к решению задач творчески, то для этого сотрудники группы должны получить доступ к самым надежным данным и понимать, какого рода информация у них есть. Сами по себе мнения не являются плохими, и члены команды должны иметь возможность свободно их выражать. Мнения могут быть даже полезными, особенно когда они подкреплены ценным опытом, но их нельзя смешивать с фактами, данными или анализом. К слову сказать, факты тоже имеют свои ограничения. В области эстетики или рыночной привлекательности фактов может недоставать. К сожалению, некоторые сотрудники рабочей группы, однажды придя к какому-то решению, перестают обращать внимание на факты.

Ничто не может стать фактом просто потому, что вы это так назвали. Поэтому группам следует учиться пресекать пустословие и договориться не злоупотреблять средствами языка. В первые годы нашей совместной жизни моя первая жена научилась с подозрением относиться к любым утверждениям, которые я начинал с фразы наподобие: «Факты ясно свидетельствуют, что...». Это было сигналом, что далее, вероятно, последует глубоко личное мнение, не подкрепленное ни данными, ни доказательствами. После того как этот гамбит перестал действовать, за мной стали иногда замечать приверженность к другой тактике, которую мы назвали ходом «95% всех ученых». Некоторые из вас, наверное, поняли, о чем идет речь. «Вы знаете, этот метод предпочитает большинство, уж точно более 95% профессиональных программистов-разработчиков». Естественно, для того чтобы продолжать пользоваться этой уловкой, вам нужно варьировать проценты. «Почти 78% пользователей WordPerfect знают, что лучшим способом является...» или «Если провести опрос, то более двух третей программистов, пишущих на C, согласятся, что...». Иногда кажется, что стоит только присмотреться, как увидишь легионы ученых, программистов-разработчиков или конечных пользователей, выстроившихся в ряд позади говорящего, чтобы лично поддержать его (или ее) точку зрения.

Но это только мое мнение.

Из журнала *Computer Language Magazine*, том 9, № 4, апрель 1992 г.

3

Достижение консенсуса

Консенсуса нельзя достичь до тех пор, пока вы сами не признаете его наличие. Это значит, что в группах по разработке программного обеспечения, стремящихся к коллективным решениям, разумно заранее договориться о критериях, относительно которых будут решаться технические вопросы. Что является важным? Что имеет значение? Что «хорошо», а что «плохо» в рамках данного проекта?

Часто, когда какая-нибудь группа застревает, проводя анализ или принимая проектное решение, а участники группы говорят: «Мы не знаем, как поступить», я спрашиваю их: «Как вы определяете, какой подход лучше?». Инжиниринг связан с поиском компромиссов – согласия на уступки в одном в обмен на уступки в другом. В любом проекте компромисс требует понимания ценности того, что приобретается или теряется при взаимных уступках.

Почти во всех областях инжиниринга проектные решения принимаются по конкурирующим критериям. Отвечать всем этим критериям в равной мере и по каждому пункту невозможно. В большинстве проектов ставятся технические и экономические цели. Обычно требуется завершение проектов в срок, эффективность продукта, простота и удобство в использовании, пригодность для продажи, масштабируемость, удобство сопровождения и множество других достоинств, приближающих продукт к идеальному. Как же все это оценивать?

Непосредственные приоритеты

Всегда полезно, когда приоритеты – непосредственные. Знатоки метрик, возможно, станут убеждать вас свести критерии, по которым принимаются решения, к математическим формулам, учитывающим каждый фактор с помощью весовых коэффициентов и степеней. Однако в общем случае это не является ни необходимым, ни особенно полезным. Достаточно простого выстраивания критериев в определенном порядке. Во время анализа и проектирования, когда должно быть найдено большинство компромиссных решений, у нас редко (если вообще когда-либо) есть все данные для того, чтобы дать нашим представлениям о степени важности тех или иных критериев хотя бы приблизительную количественную оценку. Неверный рецепт, составленный из интуитивных «оценок-догадок», может создать опасное своей обманчивостью впечатление объективности. Это может даже стать спасательным кругом, с помощью которого команда разработчиков может уйти от ответственности. «Ну вот, мы все сделали по рецепту, и не мы виноваты, что на обновление экрана уходит 17 секунд».

Ответственность появляется, если команда разработчиков участвует в определении главных целей и ранжировании критериев, по которым будут оцениваться варианты решений. После того как критерии и их приоритетность согласованы, критерии закрыты для обсуждения. Большую часть времени они даже не будут затрагиваться в дискуссиях по техническим вопросам. Не каждый компромисс обязательно анализировать исходя из семи или восьми критериев. Список согласованных критериев достается с полки только в тех случаях, когда ситуация неясна или решение принимается слишком долго.

Спор и диалог

Отсутствие ясности или согласия по поводу критериев – это не единственное, что может затруднить достижение технического консенсуса. Свободное обсуждение – это не только основа командных усилий по достижению согласия. Такое обсуждение интересно и само по себе. Однако будучи интенсивным, оно может перейти в злопыхательский спор. Ни зал суда, ни политическая трибуна не подходят для командной работы, основанной на достижении согласия. Как бы ни зарекомендовал себя состязательный подход в судебной системе, важно, чтобы группы по разработке и проектированию не разбились на противоборствующие сообщества.

В одном из учебных классов по объектному ориентированию участник студенческой команды разработчиков пожаловался мне, что группа стоит на месте. Они постоянно увязали в бесконечных спорах. Небольшой

прогресс, которого они достигли, был несопоставим с успехами других команд из этого класса.

Понаблюдав за их работой (или попыткой работать), я понял, что в обсуждениях доминировал один человек, ярый спорщик, однако его идеи не были под стать его умению спорить. Другие члены команды видели недостатки в его суждениях, но, будучи задавленными его аргументацией, отступали со словами «мне так не кажется», «такое ощущение, что это неверно».

Пожаловавшийся студент имел мотивацию к тому, чтобы работа шла лучше, поэтому я попросил его стать диспетчером группы. Его обязанности состояли из двух частей: следить, чтобы никто из участников или сторон не доминировал в обсуждениях, а также помогать менее активным или менее настойчивым членам группы выражать суть и логику своих идей.

Если вы хотите создавать самые лучшие системы, ваши технические решения не должны опираться лишь на мнения тех, кто может переспорить остальных, обладает большей властью или может кричать громче всех. Чтобы избежать этого, возможности логики и аргументации должны быть доступны всей группе, а не только отдельным представителям или группировкам. Необходимо выровнять игровое поле таким образом, чтобы проявились достоинства идей и анализа, а не изощренность аргументации или сила голоса и многоречивость.

Когда людям, отстаивающим свои позиции, не удается найти общий язык, поможет такой метод. Стороны меняются ролями и начинают защищать позиции, предложенные другими. Или же активным спорщикам можно поручить защиту технически интересных, но слабо отстаиваемых идей. «Послушай, Мэвис, твоя идея хороша, но сможешь ли ты убедить нас, что в идее Грега есть преимущества?» Еще один прием – предложить следующее: «Давайте применим те же аргументы в рассмотрении другого предложения».

К техническому консенсусу лучше приходить в диалоге и переговорах, чем в споре и препирательствах. Очень полезными могут быть сведения о переговорах, почерпнутые в других областях. Прежде всего, следует порекомендовать две великолепные книги, изданные в рамках «гарвардской программы по ведению переговоров»: Фишер (Fisher) и Юри (Ury) «Getting to Yes» (Как добиться согласия), 1981 [37] и Фишер и Браун (Brown) «Getting Together» (Как добиться единства), 1988 [38].

Вечной проблемой в переговорах является то, что участвующие в них стороны садятся за стол, зачастую придерживаясь какой-то позиции, размышления над которой уже отняли много сил. Вместо того чтобы быть искренне открытым, каждый приходит с заранее определенным решени-

ем. Подобные переговоры трудны сами по себе. В лучшем случае они могут привести к компромиссу, но не к консенсусу. Кроме того, они легко могут перейти в перепалку между сторонниками разных подходов.

На помощь могут прийти простейшие методы. «Гарвардская программа по ведению переговоров» помогла понять, что переговоры идут лучше, если несогласные между собой участники садятся рядом, лицом к «их общей проблеме», а не рассматривают друг друга, сидя по разные стороны стола. Я обнаружил, что если при спорах по техническим вопросам группировки садятся вместе перед белой доской или экраном монитора, то это способствует более продуктивному обсуждению и скорейшему разрешению проблемы.

Собирая все вместе

Бывает так, что избежать применения заранее подготовленных предложений или уже выработанных решений не удается. Например, две группы в одной и той же компании, возможно, уже провели какую-то работу, которую необходимо принять во внимание. В некоторых компаниях даже поощряется конкуренция среди разработчиков на внутреннем свободном рынке идей. Когда приходит время создавать какую-то систему, авторы или соперники выставляют «на продажу» и описывают свои подходы. Достичь консенсуса будет легче, если перед началом обсуждения все альтернативы представит сотрудник, который менее пристрастен, чем те, кто эти варианты предложил. Выбор верного тона для обсуждения будет способствовать проектным решениям на основе согласия. Участников обсуждения следует поощрять к поиску сильных сторон других предложений, перед тем как переходить к критике. Стоит поощрять и реализм в оценке исходных позиций: «Поскольку для нас важнее знать о технических слабостях наших систем, чем делать вид, будто они идеальны, пусть каждый из вас расскажет о недостатках своего подхода».

Если в подготовке предложенных решений участвовали отдельные подгруппы, то после первоначальных обсуждений каждой подгруппе может быть предложено вернуться назад и улучшить свое предложение, используя то, что по их мнению является лучшим в подходах, предложенных другими. Тогда в начале следующей встречи противостоящие точки зрения окажутся ближе друг к другу.

В общем, технический консенсус достигается на основе комбинирования лучших черт всех вариантов и даже генерации новых. Вместо того чтобы начинать с конкретных технических предложений, зачастую бывает более разумно и эффективно начать с самой задачи. Первым делом команде нужно выяснить основные технические аспекты, присутствующие в раз-

личных вариантах, а также базовые предпосылки и техническое обоснование исходных позиций и предлагаемых решений.

Процесс творческого синтеза начинается еще до первой встречи. Вместо того чтобы обдумывать, скажем, структуру файлов, члены команды могут очертить круг вопросов, связанных с разработкой эффективной файловой структуры. Они могут составить список конкретных критериев для принятия решений и определить их приоритетность. Их можно даже попросить пока не думать об идеях и предложениях. С большинством тех разработчиков программного обеспечения, которые в большей мере являются одиночками, проблема состоит, скорее, не в том, чтобы побудить их к работе, сколько в том, чтобы сдержать их пыл перед тем, как они сорвутся с мест.

Из журнала *Computer Language Magazine*, том 9, № 5, май 1992 г.

4

Скромный и высокопоставленный писарь

Помните, как Боб Крэтчит (Bob Cratchit) трудился над книгами в солидной фирме Скруджа (Scrooge) и Марли (Marley), надев на руки перчатки без пальцев, чтобы они не замерзали, пока Боб перелистывал страницы? Я очень люблю «Рождественский гимн» (A Christmas Carol). Недавно мне подарили видеокассету с изумительной черно-белой экранизацией, где главную роль играет Алистэр Сим (Alistair Sim). Посмотрев этот фильм, я задумался о старом Бобе и других «клерках», которые на протяжении столетий вели учетные книги для множества предприятий. Эти писари были настоящими компьютерами своего времени. Без них предприятия пришли бы к банкротству, а целые отрасли были бы ввергнуты в хаос. Их реальная власть и влияние намного превосходили их скудные жалования или невысокий статус. Вообще говоря, продолжительный успех Скруджа и Марли был в большей мере связан с работой старого доброго Боба и его соотечественников, чем с тем, что привнес Эбенезер.

Сегодня вряд ли что-то изменилось. Сотрудники, которые ведут учетные книги, по-прежнему ценятся невысоко. Но в их карандашах, маркерах и клавиатурах может скрываться сила, предопределяющая успех или провал разработки программного обеспечения.

Если группы по разработке программного обеспечения ведут какие-то записи, то в архивах и заметках отражаются только результаты и выводы, рабочие продукты или готовые компоненты. Программисты особенно не любят записывать что-нибудь кроме самого кода, если только перед ними не стоит угроза штрафа или тюремного заключения. Заставить их нари-

совать диаграммы – это все равно, что заставить слона сделать наброски карандашом. В конце концов, разве не является хороший код самодокументируемым?

Жизненно важная статистика

Такое отношение приводит к потере жизненно важной информации. Вообще говоря, когда сохраняется только конечный продукт, нам известен результат, но мы не знаем, как его получили. Как создавалось программное обеспечение, какие решения были найдены в процессе работы – все это является важным. Можем ли мы полагаться на свою память? Беспокоимся ли мы только об ошибках или вдобавок хотим извлечь из них пользу?

Особые трудности в работе групп, сохраняющих только конечный программный продукт, вызывает отсутствие записей о решениях, от которых отказались. Знать о том, какие методы были отклонены и по каким причинам, часто бывает так же важно, как и о том, какие методы были выбраны. Это жизненно важно в тех случаях, когда готовятся новые версии или системы или когда разработка текущей системы заходит в тупик.

Наверное, вы когда-нибудь рассматривали свой код, написанный несколько месяцев или лет назад. Случалось ли так, что вы находили такие места, которые казались неверными, и вы удивлялись, как же программа вообще могла работать? Если вы поддавались соблазну «исправить» эту скрытую ошибку, как это иногда делал и я, то могли обнаружить, что «исправление» загоняло систему «в угол». Естественно, код только *казался* неправильным, но сам по себе он не объяснял, почему здесь все в порядке. Не помогут и такие комментарии в коде: «Ничего здесь не меняйте. Это место кажется неверным, но здесь все правильно». Если программист знал, что «здесь все правильно» и почему другой вариант будет неверным, то почему же тогда эта логика не отражена в комментарии? Если мы хотим поддерживать систему не один год или иметь возможность выпустить следующую версию через пять лет после того, как все разработчики первой версии системы будут далеко, нам нужно знать, какие варианты рассматривались, какие были отклонены и почему.

Сегодня бизнес-консультанты говорят об организационном обучении как о ключе к долговременному успеху предприятия. Организации, как и люди, могут учиться на своем опыте, накапливая знания и улучшая свою производительность. Если в организационное обучение вовлечены только отдельные сотрудники, то такая организация становится уязвимой. Работники болеют, уходят в отпуск, меняют работу. В конце концов, они забывают.

В действительности знания организации содержатся в ее архивах, в ее стратегии, в ее правилах и процессах, а не только в ее работниках. Чем больше мы документируем и ведем записи о происходящем, тем более вероятно, что эти знания переживут группу или команду, создавшую их.

Каракули

Входит скромный писарь. Звучат фанфары и приветствия!

В команде, разрабатывающей программное обеспечение, писарь или протоколист отвечает за коллективную память команды, в которой хранятся как рабочие продукты, так и описание процессов, давших результаты. Писарь ведет учетные книги. В модели командной работы с открытой структурой (Structured Open teamwork), которую независимо друг от друга разработали Роб Томсет (Rob Thomsett, 1990 [62]) и я (Constantine, 1989 [11], 1991 [13]), такая роль была обозначена как «информационный менеджер». Этот термин был введен для того, чтобы повысить статус этой роли, подобно тому как в объявлениях о вакансиях вместо «водитель мусоровоза» пишут «инженер санитарно-гигиенического транспорта».

Писаря можно часто увидеть на собраниях в качестве простого должностного лица, едва ли более значительного, чем диктофон. Однако на самом деле это занятие требует большого умения. Вы не можете сделать правильные записи о том, чего не понимаете, поэтому в группах, разрабатывающих программное обеспечение, писарями становятся опытные разработчики. Они должны знать, как вести записи и как обращаться с данными в реальном проекте. Качество записей определяет качество «постоянной коллективной памяти» о том, что происходило и как. Писарь, ведущий полный, без лишних деталей, хорошо организованный и понятный протокол о том, что делала группа, является командным игроком, ценность которого так же высока, как и его навыки в программировании на C++.

Хорошая групповая память должна вмещать много информации. Одна часть этой памяти является непостоянной или временной памятью, другая часть – постоянной. Часть групповой памяти является активной и может оперативно использоваться в обсуждениях и решении повседневных задач. К другой части памяти команда обращается реже. Для удобства функции управления информацией можно разделить на функции сессионной памяти и функции проектной памяти. Сессионная память состоит из записей, созданных и задействованных во время групповых сессий, когда участники команды встречаются и вырабатывают групповые решения. В проектной памяти хранятся постоянные записи и документация, написанная и применяемая группой. Она включает в себя рабочий продукт, под которым понимается не только код, но и все проектные и анали-

тические модели и документы, созданные при написании кода. Кроме того, в проектной памяти хранятся входные данные, такие как требования, спецификации и другие основополагающие документы. Для управления проектной памятью нужен библиотекарь, и часто эта роль известна именно под этим названием.

Модульная память

Важнейшая функция сессионной памяти – хранение отчета о процессах, который отражает проведенные обсуждения и решения, принятые в группе (Дойл (Doyle) и Страус (Strauss), 1982 [35]). Наверное, идея о ведении записей в ходе процесса нова для большинства групп, разрабатывающих программное обеспечение, однако на различных собраниях это практиковалось десятилетиями. Ведя записи, начинающие писари часто допускают одну из двух ошибок. Они либо пытаются записывать все подряд, словно под диктовку, либо ждут, пока группа не придет к какому-то заключению, и просто записывают итог. Для технической работы, выполняемой командой, записи вида «он(а) сказал(а)» не особенно подходят и не являются необходимыми. В хорошем протоколе отмечены ключевые события на пути к конечному результату, особенно рассмотренные альтернативы, принятые решения и представленные аргументы. Такие записи вносят наибольший вклад в групповое приобретение опыта. Они могут стать бесценными, когда необходимо оценить проект или когда проект вступает в фазу «post mortem»[1].

Для разработки программного обеспечения сплошной неструктурированный протокол не подходит. Некоторые виды данных так часто анализируются в ходе сессий, что требуют ведения отдельных записей для последующего особого рассмотрения. Полезно также вести записи под заголовком «нужно сделать», где отмечать те идеи, которые возникают в обсуждениях, но сразу не воплощаются. Уже одно это может оправдать утомительное ведение сессионных записей, поскольку позволяет избежать нелепых оплошностей, обычно выявляемых во время системной интеграции или уже после отправки продукта заказчикам: «О, да мы совсем забыли о проблеме с висячим указателем!»

Хорошая сессионная память также включает отложенные решения, которые лучше всего хранить отдельно от уймы записей в списке «нужно сделать». Наличие специального места для отложенных решений может также ускорить их принятие. Вместо того чтобы тратить время на бесконечные споры, вызванные недостаточным пониманием или отсутствием данных, группа может занести этот вопрос в список отложенных реше-

[1] После смерти (лат.). – *Примеч. перев.*

ний. Зачастую к тому времени, когда группа возвращается к этому вопросу, уже многое известно для принятия быстрого решения. Третий специальный раздел записей, который тоже полезен команде разработчиков, – это раздел «запасных частей». Сюда можно временно откладывать фрагменты хороших технических идей или частичные решения, чтобы не нарушать основной ход обсуждения. «Мусорная корзина» играет противоположную роль. Здесь можно отмечать все неиспользованные идеи и пути наряду с вескими причинами для их отклонения.

Все четыре специальных списка («нужно сделать», «отложенные решения», «запасные части» и «мусорная корзина») служат для записи того, что могло быть потеряно или забыто. Помимо всего прочего, такие записи помогают группе эффективно использовать время. Отмечая в одном из специальных списков факты отхода от основной темы, группа не теряет главной нити своего обсуждения и не упускает полезную информацию. Все это также может помочь группе избежать никчемных споров. Для того чтобы не пробуксовывать, тот или иной вопрос можно поместить в одну из «корзин» для более позднего рассмотрения. Кроме того, сами корзины служат в качестве механизмов обеспечения качества (QA, quality assurance). В конце проекта все содержимое корзин должно быть уничтожено или каким-то образом учтено.

Как же определить того счастливчика, который станет выполнять роль писаря? Согласно некоторым методикам, таким как «Joint Application Design» (Совместная разработка приложений) Вуда (Wood) и Силвера (Silver), 1989 [69], специально подготовленные помощники и писари привлекаются со стороны. Это позволяет разгрузить участников проекта для того, чтобы они смогли сосредоточиться на создании программного обеспечения. В некоторых группах эти обязанности вменяются одному из членов команды (обычно младшему сотруднику или стажеру). Для большинства команд, разрабатывающих программное обеспечение, более эффективным является совмещение этих подходов. Информацией заведует вся команда в целом, а реальная ответственность может по очереди возлагаться на всех участников проектной команды. При таком подходе никто не освобождается от обязанности играть роль писаря, однако никто не играет ее слишком долго. В течение одного собрания обязанности сессионного протоколиста могут переходить от одного человека к другому или же такая работа может оставаться в одних умелых руках в течение всей сессии. Однако для сохранения рассудка и поддержания хорошей рабочей атмосферы обязанности писаря, возможно, должны передаваться по крайней мере на каждом новом собрании. Если собрания проходят довольно долго, то такая смена должна происходить не менее одного раза в час. Дело в том, что для хорошего выполнения обязанностей писаря тре-

буется чрезвычайная концентрация внимания, и очень мало людей на планете действительно получает удовольствие от этой роли.

Заботы об архивах, автономной или проектной памяти могут перекладываться реже. Если передавать «эстафетную палочку» каждый день, это приведет лишь к беспорядку и неисполнительности. В течение годичного проекта обязанности, закрепленные за должностью так называемого «менеджера проектной информации», или «проектного писаря», могут передаваться из одних рук в другие не более одного-двух раз.

Так что, пригласите писаря на ланч. Возможно, на следующей неделе писарем станете вы.

Из журнала *Computer Language Magazine*, том 9, № 6, июнь 1992 г.

5

Официальное пространство

Ваш коллега в офисе жует жевательную резинку, проигрывает на персональном компьютере «Where-in-the-World-is-Carmen-San-Diego?», охает и задает глупые вопросы. А в это время вы пытаетесь разобраться с каким-то непонятным багом. В этой компании вы работаете несколько лет и чувствуете, что пришла пора получить отдельный кабинет. Вы идете к начальнику и говорите, что вам нужно большее пространство и защита от отвлекающего шума и помех. Вы говорите, что это повысит эффективность работы. Дескать, если у вас появится больше пространства и вас будут меньше отвлекать, вы станете работать более продуктивно. Вы ссылаетесь на научные исследования, подтверждающие это. Вы считаете, что вам нужн, по крайней мере сто квадратных футов выделенного пространства и тридцать квадратных футов для рабочего стола. Окно вам тоже не помешает. В Дании даже есть закон, по которому вы должны сообщить об этом начальнику. В Дании начальник обязан предоставить вам окно.

Народная мудрость гласит, что чем больше свободного пространства, чем тише в офисе, чем меньше помех, тем выше продуктивность и меньше ошибок в программном обеспечении. Сто квадратных футов свободного пространства и тридцать квадратных футов рабочего места просят программисты во всем мире. Эти представления вошли в отраслевой фольклор благодаря Тому ДеМарко (Tom DeMarco), Тиму Листеру (Tim Lister) и их классическому произведению «Peopleware: Productive Projects and Teams», 1987 [33].[1] Исследовав несколько источников (главным образом их

[1] Т. ДеМарко и Т. Листер «Человеческий фактор: эффективные проекты и команды». – Пер. с англ. – СПб: Символ-Плюс, I кв. 2004 г.

собственный ежегодник о «военных играх по кодированию»), они пришли к следующему выводу: программисты, работающие в отдельной комнате и имеющие больше места для раскладывания своих диаграмм и распечаток, обладают большей производительностью.

Форма процесса

На работе, дома, в ресторане или в учебной аудитории физическое пространство определяет то, что происходит и может происходить между людьми. За длинным банкетным столом вы можете беседовать с сидящими по обе стороны от вас и еще с несколькими людьми, сидящими напротив. Однако вряд ли вы сможете пообщаться с теми, кто сидит на другом конце стола. Король Артур сделал свой стол круглым в знак равенства тех, кто за ним сидит. Кроме того, форма стола позволяла всем рыцарям легко говорить друг с другом.

Планировка и меблировка зданий или офисов может оказывать сильное влияние на общение людей и их совместную работу. Когда-то я два года прожил в квартирном доме и за все это время встретил лишь нескольких соседей. Вход в дом представлял собой маленький проем, едва достаточный для одного человека с сумкой продуктов. Коридоры были узкими, темными и пустынными. Там просто не было места для того, чтобы люди могли случайно встретиться друг с другом и поговорить.

Обычно офисы не страдают наличием узких проходов или темных коридоров, однако в офисах современных высокотехнологичных компаний есть другие архитектурные «изыски». В схемах компоновки современных офисов преобладает так называемая открытая планировка с гибким разделением помещений. Несмотря на такое привлекательное название эта система обычно не является ни «гибкой», ни «открытой».

Одна компьютерная компания размещается в зданиях с многомильными коридорами и кабинками крест-накрест, разделенных звуконепроницаемыми перегородками песочного цвета. Для того чтобы оглядеться, приходится вытягивать шею. Такие перегородки лишь сводят шум от разговоров к тому отвлекающему гулу, который слышен в оперном зале перед поднятием занавеса. Нет ни уединения, ни пресловутой открытости. Люди в офисах, отделенные в этом бежевом кроличьем садке более чем двадцатью футами, могут в течение многих лет не встречаться друг с другом. Посетителей в таких помещениях нужно сопровождать – не ради их безопасности, а просто для того, чтобы они не заблудились. Почтовые адреса там обозначаются наподобие KK14-HDQ:117N\BB.R3 (тайные коды, сообщающие об этаже, колонке и ряде). Гибкость этой гибкой планировки офисов весьма иллюзорна. Перемещение любой перегородки означает необходимость выкорчевывания и прокладывания заново телекоммуника-

ционных и сетевых кабелей, не говоря уже об исправлении почтовых адресов. Стены могут быть гибкими, но персонал отдела почтовой доставки – уж точно нет.

С точки зрения использования офисного пространства для повышения производительности разработчиков, эти негибкие «гибкие» системы обычно расставляются так плохо, как только можно себе представить. Они не дают возможности работать одному и работать вместе. Они не позволяют уединиться или встречаться мимоходом.

Влияние планировки офиса на производительность разработчиков определяется не только объемом свободного пространства и возможностью уединиться. В первую очередь можно упомянуть парадокс причины и следствия – старое пугало всех социологических исследований. Работники, проявившие высокую производительность в военных играх по кодированию, имели более просторные помещения и испытывали меньше помех *именно потому*, что они работали с высокой производительностью, а не наоборот. Либо их производительность способствовала тому, что компания смогла предоставить им более спокойные условия и более просторные помещения.

Еще более важно то, что результаты исследований, приведенные ДеМарко и Листером, касались кодирования и тестирования и не относились к процессу разработки программного обеспечения в целом. По условиям проводимых ими ежегодных соревнований каждый участник работал самостоятельно, а не в составе команды. Поэтому, как представляется, свободное пространство и тишина помогают обособленным кодерам. А как насчет командной работы?

Совместная работа и общение

Для эффективного сотрудничества командам действительно необходимо пространство: пространство для команды в целом и служебное пространство, необходимое для выполнения заданий. Исследования подтверждают, что для хорошей совместной работы нужны как открытые, так и изолированные помещения. Офис должен предоставлять возможности для интенсивной работы группам из двух-трех человек в течение коротких или длительных периодов. Кроме того, в офисе должно быть по крайней мере одно место, где все сотрудники, работающие в проекте, могут собираться вместе. Если в офисе нет места, где вся команда может спокойно собраться, то превратить команду в сплоченный рабочий коллектив намного труднее.

Один руководитель в области разработки программного обеспечения горячо поддерживал метод командной работы, но был разочарован резуль-

татами совместного труда своих подчиненных. Этаж, на котором они работали, представлял собой сумасшедший лабиринт из узких коридоров и крохотных офисов со стеклянными стенами – отдельных, но не уединенных помещений. Лишь в некоторых офисах могли поместиться два человека, а в тех помещениях, которые были достаточно большими, часто размещались сотрудники, привлеченные к разным проектам. Однажды я наблюдал, как одна команда проводила собрание в самом большом зале на этом этаже. Одиннадцать разработчиков втиснулись в комнату, в которой стояли стол и шесть стульев, находящихся друг от друга на расстоянии семи дюймов. Между краем стола и небольшой доской на стене едва хватало места для того, чтобы руководитель собрания мог сделать пару шагов и повернуться. Нечего и говорить, что собрание было коротким.

Для командной работы крайне необходимо место для общих собраний в течение всего периода работы. Требуется место, которое может служить штаб-квартирой, «ситуационной комнатой», где может собраться вся команда. Нужна защищенная территория, где члены команды могут укрыться от шума и сосредоточиться. Плох вариант, когда зал для собраний используется совместно с другими командами.

Отдельная «ситуационная комната» достаточных размеров особенно важна для команд, которые ведут «архивирование системных документов» (Zahniser 1990 [71], 1993 [72]) и применяют другие методы группового анализа и разработки. Настенные доски командной штаб-квартиры становятся архивом групповой работы, ее видимой, внешней «групповой памятью», в которой хранятся важные элементы рабочего продукта и информация о процессе его создания. На стены командной штаб-квартиры можно поместить все, что угодно, начиная от командного флага и изложения целей проекта и заканчивая основными документами, связанными с разработкой. Само помещение и его оформление становятся частью командной культуры и способствуют тому, что каждая личность ощущает себя частью целого. Это помогает членам команды успешно и эффективно работать вместе.

Если члены команды находятся в разных зданиях, офисах или рассеяны по всему миру, то сотрудничать – и даже просто общаться – становится труднее и дороже. При прочих равных условиях размещение участников проекта по разным местам, даже в разных зданиях или на разных этажах, может привести к увеличению расходов на 50–100%. Поскольку команда, которая рассредоточена в пространстве, почти всегда будет менее эффективна, чем группа, работающая в одном месте, такой команде нужны компенсационные механизмы. На помощь приходит электронная почта и телеконференции, но ничто не может заменить возможность хотя бы раз собраться вместе и увидеть друг друга.

В давние времена, когда разработка программного обеспечения только зарождалась, в Массачусетском технологическом институте были проведены исследования инженерных групп и групп НИОКР.[1] Выяснилось, что производительность возрастает при улучшении взаимодействия, а взаимодействие, в свою очередь, улучшается, когда группа работает в хороших условиях. Самые лучшие варианты компоновки предусматривали открытое пространство в центре помещения, что способствовало или даже вынуждало инженеров сталкиваться друг с другом. Здания с фиксированными стенами и руководство с ограниченным мышлением являются сдерживающими факторами, однако творческие компромиссы все же возможны. Райза Химэн (Risa Hyman) рассказывает об одной группе, которая проявила находчивость в использовании традиционного ряда офисов, выстроенных вдоль коридора (Hyman, 1993 [42]). Коридор был увешан настенными досками и стал главным местом для проведения общих собраний!

Правильная планировка способствует улучшению взаимодействия. Однако станут ли стены барьерами или мостами к более эффективной командной работе, зависит от членов группы и ее руководства.

Из журнала *Software Development*, том 1, № 12, декабрь 1993 г.

[1] Научно-исследовательские и опытно-конструкторские работы. – *Примеч. ред.*

6

Раздражающие прерывания

Очень раздражает, когда вас отвлекают. С другой стороны, в коде может быть ошибка, которую вы не замечаете, а идея, о которой вы не стали слушать, могла бы помочь найти решение. Эффективность взаимодействия людей в офисе или в команде может оказывать существенное влияние на производительность. Совместное использование офиса облегчает совместное использование информации. Иногда это является плюсом, иногда нет. Для интенсивных и сосредоточенных размышлений, необходимых для написания хорошего кода или для выискивания скрытых ошибок, требуется продолжительное и пристальное внимание. Когда слова статьи или строки кода просто вылетают из-под пальцев, то больше всего не хочется, чтобы кто-нибудь между прочим вставил свой комментарий по поводу того, что написал Роберт К. Крингли (Robert X. Cringely). Однако бывает и так, что чья-либо случайная ремарка может помочь по-новому увидеть свою задачу. Иногда обсуждение проекта с внимательным слушателем может помочь увидеть недостатки или скрытые ошибки.

Для хорошей командной работы в офисе требуется возможность свободного, не вызывающего раздражение доступа друг к другу. Люди, которые работают вместе или делят одно рабочее пространство, имеют много разумных поводов для того, чтобы прерывать друг друга: просьбы о помощи, обсуждение идей, вопрос о состоянии дел и координирование текущей работы в целом. С другой стороны, очень важно, чтобы оставалось время для размышления и творчества, когда вам никто не помешает. Если вас кто-то не вовремя прервет, вы можете забыть ценную идею, потерять мимолетную мысль или полностью запутаться в сложном вопросе.

Эта проблема не является новой и не относится лишь к группам по разработке программного обеспечения. Однако разработчики могут получить пользу от более совершенных способов управления взаимодействием в группе. Естественно, компьютерщики любят решать социальные и организационные задачи с помощью компьютеров, и, вероятно, первое, что здесь приходит в голову, – это создание быстрой сети с какими-нибудь хитрыми почтовыми средствами. Но иногда даже очень простая система может сослужить хорошую службу. Может быть, все, что нужно, – это более удобный лексикон.

Вооружившись словом

Техническая терминология и обычная речь зачастую тесно взаимодействуют друг с другом. В компьютерный лексикон вошло много повседневных слов в качестве строгих технических терминов. Такие слова, как «объект» или «сущность», заимствованы для обозначения понятий, которые никак не связаны с окружающим нас физическим миром. Мы присвоили «метод» и «сообщение», «протокол» и «папку» (file[1]). Конечно, происходит и обратный процесс. Технические термины входят в обиходную речь в такой степени, что компьютерный жаргон теперь можно услышать даже в разговорах на улице.

Специалисты по компьютерам особенно любят переносить свой жаргон в обычный разговор, расширяя значение терминов таким образом, чтобы их можно было применять при общении. Они стараются произвести «парсинг» (parsing) неразборчивого сообщения на автоответчике или маниакально «мультиплицируют» два разговора. Привычные поведенческие шаблоны таких специалистов «жестко запрограммированы в ПЗУ». Программисты «портируют» автомагнитолу с одной машины на другую, а не покупают новую деку. Для того чтобы написать черновик отчета о вчерашнем совещании, они осуществляют «дамп оперативной памяти». Все это может стать весьма утомительным и звучать ужасно глупо, однако порой техноболтовня обогащает обычный язык полезными и интересными новшествами.

Я помню свой первый опыт применения программной терминологии в устной речи. Наша группа занималась экспериментами, связанными с коммерческим использованием языка Lisp. Мы настолько увлеклись этим языком, что стали говорить списками и точечными парами. В на-

[1] В отличие от английского языка, в котором слово «file» существует давно, в русский язык слово «файл» вошло сравнительно недавно (с развитием информационных технологий). Слово «папка» вошло в компьютерный лексикон из обычной речи. – *Примеч. ред.*

ших разговорах то и дело встречались довольно своеобразные высказывания. Мы «cons» две идеи вместе, а также «car» и «cdr» темы разговора. Такая беседа могла быть настолько «многопоточной», что в конце концов кто-нибудь жаловался на «переполнение стека» и просил о сборе мусора. Для тех, кто никогда не учился «шепелявить на ЛИШПе» (thpeak with a lithp)[1], скажу, что «car» означает «первый элемент списка», или «левая часть», а под «cdr» (произносится «куд-эр») понимается «остальная часть списка», или «правая часть». Такие странные названия закрепились в Lisp несмотря на то, что они относятся к аппаратным регистрам давно исчезнувшей IBM 709/7090/7094 и буквально означают «содержимое адресного регистра» и «содержимое регистра декремента».

Наша вычурность продолжалась недолго. Вероятно, она казалась глупой и не была особенно полезной. Ну, разве что помогла нам изучить жаргон и лучше запомнить понятия языка Lisp. Уже много лет я не «cadadr»'ю какую-нибудь идею. Хотя если сегодня побродить по нашим офисам, то можно услышать не менее странные, а порой и более полезные идиоматические выражения.

Офисный протокол

Обычные способы прервать кого-либо, принятые в приличном обществе, могут быть слишком медленными и неудобными для эффективного взаимодействия. «Извините, вы не заняты? Надеюсь, не помешаю. У меня только один небольшой вопрос, на одну секунду.» На одну секунду? Это уже заняло 6,5 секунд! К этому моменту прерывание стало *fait accompli*[2]. К тому времени как ваш мозг разобрался и обработал весь этот шум и принял решение о том, что с этим делать, вы уже забыли, на какую строку кода смотрели и какой метод какого подкласса собирались задействовать.

Для рабочих групп нужен словарь прерываний, который был бы кратким, приятным и простым. То, что подходит для аппаратного обеспечения, подойдет и для людей, поэтому в наших офисах мы тоже IRQ, ACK и NAK.

IRQ – это сокращение от «interrupt request» (запрос прерывания, произносится «ирк»). Например, «Можно мне «ирк» вас?». Впрочем, достаточно и одного слова «Ирк?». Восходящая интонация делает высказывание более вежливым, а взрывной согласный звук в конце слова привносит оттенок

---

[1] Слово lisp в переводе с англ. означает «шепелявить». Именно это и обыгрывает автор (thpeak with a lithp нужно понимать как speak with lisp). Из-за особенностей синтаксиса LISP часто расшифровывают (в шутку, конечно) как Lots of Idiotic exceSsive Parentheses или Lots of Irritating Superfluous Parentheses, подразумевая, вероятно, что нормально «говорить» на подобном языке нельзя.

[2] Делом свершенным (фр.). – *Примеч. науч. ред.*

настойчивости. Это слово звучит достаточно отчетливо, чтобы его можно было услышать среди шума вентиляторов и визга лазерных принтеров. В то же время, оно достаточно коротко, чтобы не сильно отвлекать от ментальных процессов. Возможные ответы: ACK[1] и NAK[2] (первое произносится «эк», второе – «эн-эк» или «нэк»), что означает «Ок, давайте!» и «Нет, не сейчас!» соответственно. Не нужно отрываться от ваших дел, чтобы пробурчать либо «эк», либо «нэк». Оба варианта ответа имеют характерное фонетическое звучание, что делает их различимыми при прерывании.

Протокол прерываний чрезвычайно прост. Тот, кто хочет прервать кого-нибудь, говорит «Ирк!» и ждет ответа. Тот, кого прерывают, может еще некоторое время продолжать свою работу, пока не подтверждено установление связи. Например, он может пометить место в тексте, заполнить поле заголовка или оставить себе короткую записку-напоминание. Когда сотрудник готов обслужить прерывание, он отвечает: «Эк». Ответ «Нэк» означает: «Нет, не отвлекайте меня сейчас». Это можно расценить как вежливый вариант высказывания «Уйди отсюда и не стой над душой». Все эти слова кажутся очень глупыми, но такая простая система может поразительным образом способствовать более гладкому взаимодействию в рабочей группе. Иногда в словарь может входить высказывание NMI[3] (произносится «ними»), что означает «немаскируемое прерывание». По правилам этикета такое высказывание должно применяться лишь в критических случаях, требующих наибольшего приоритета обработки со стороны центральной нервной системы вашего бедного коллеги. Прежде чем начать говорить, лучше сделать короткую паузу, хотя дожидаться ACK или NAK не требуется.

Людей, которые слишком часто применяют IRQ, называют надоIRQливыми.[4] С ними можно поступать на манер кота Билла, громко крича «Эк, нэк. Нэк! Эк!» и вырывая на себе волосы. Если в нужный момент вы сможете превратиться в пушистый клубок, то это будет еще более эффективно.

Конец прерывания.

Из журнала *Software Development*, том 2, № 6, июнь 1994 г.

---

1 ACK (сокр. от англ. acknowledgement) – подтверждение приема. – *Примеч. перев.*

2 NAK (сокр. от англ. negative acknowledge) – отсутствие подтверждения приема. – *Примеч. перев.*

3 NMI (сокр. от англ. nonmaskable interrupt). – *Примеч. перев.*

4 IRQsome, здесь обыгрывается англ. irksome – надоедливый, докучливый. – *Примеч. перев.*

II

Ковбои и ковгерлы

Введение

У автора, ведущего колонку в компьютерном журнале, есть одно замечательное преимущество. Оно заключается в том, что читатели отвечают на заметки по электронной почте, внося тем самым живую струю в отношения между читателем и автором. В течение многих лет колонка под названием «Peopleware» находила отклик у понимающих и заинтересованных читателей. Зачастую они были настолько взволнованы, что посылали комментарии или задавали вопросы. В конце концов, я стал рассматривать чтение регулярно приходящей электронной почты (и нерегулярно приходящей обычной почты) как часть работы по ведению этой колонки.

Однако я был совсем не готов к тому потоку сообщений, которые стали поступать после выхода моей первой заметки, посвященной «кодирующим ковбоям». На самом деле эта заметка и та, которая за ней последовала, побили все рекорды по читательским откликам. Внезапно я оказался на «Диком западе» в период его освоения. Для большей остроты ощущений можно представить, что я обсуждал с охотниками закон об оружии, а с фермерами – наказание за незаконный выпас скота на государственных землях. Один флейм[1], пронесшийся по электронной прерии, был слишком сильным для дословного воспроизведения в таком благопристойном издании, как *Software Development*. По существу, речь шла об анархии, изначально царящей в среде разработчиков программного обеспечения. Мой словарный запас расширялся по мере получения в свой адрес модных оскорблений, характерных для электронного мира. Я удивился, узнав, что некоторые программисты все еще применяют «commie pinko»[2] как ругательство. *Plus c'est change, plus c'est mкme chose*.[3]

---

[1] Флейм (от англ. flame – пламя, пыл) здесь – оживленное обсуждение какого-либо вопроса в сети. – *Примеч. перев.*

[2] Умеренный приверженец коммунизма. – *Примеч. перев.*

[3] Чем больше это меняется, тем больше это остается все тем же (фр.). – *Примеч. науч. ред.*

Эта тема оказалась довольно богатой залежью, запасов которой хватило на несколько заметок и целую статью. По пути мне удалось приобрести нескольких друзей и, вероятно, нажить нескольких врагов. Скорее всего, я никогда не получу приглашение на ланч от Билла из Редмонда[1].

[1] В Редмонде находится штаб-квартира Microsoft. – *Примеч. ред.*

7

Кодеры-ковбои

Наступило новое тысячелетие, а вы даже не знаете об этом! Программное обеспечение наконец-то стало надежным. Как же был достигнут этот прорыв в области проектирования программного обеспечения? Вот цитата из 16-страничного рекламного буклета корпорации Nanomush Inc., который был разослан миллионам отсталых пользователей и разработчиков: «Одним из самых мощных дополнений к Blerbbleflox 3.1 является «проверка параметров» (parameter validation). Когда информация поступает из какого-либо приложения в операционную систему Blerbbleflox, система проверяет правильность полученных данных». Вот такая новая идея! Почему же вы об этом не думали, а? (Естественно, все поняли, что речь идет о Microsoft и Windows 3.1.)

Эта попытка самовосхваления открыла, что корпорация Nanomush, являющаяся одним из мировых лидеров в области разработки языков и операционных сред, в конечном счете стала применять самые элементарные приемы надежного проектирования программного обеспечения. Это настолько основополагающие приемы, что каждый человек, достойный имени программиста, знает и применяет их вскоре после того, как приобретает способность кодировать. Может ли это пролить хоть какой-то свет на недостатки в предыдущих версиях программного обеспечения? Однако нужно не придираться, а радоваться и хвалить всех начинающих разработчиков программного обеспечения за их усилия, а также поощрять их стремление к дальнейшему профессиональному развитию. Возможно, таким специалистам следует посоветовать книги о связывании и сцеплении или о том, как скрывать информацию от пользователя.

О том, почему компьютерный мир пришел к такому плачевному состоянию дел в области системного программного обеспечения, можно говорить долго. Но лучше пристальнее взглянуть на характер разработчиков, создавших некоторые из тех продуктов, от которых мы так сильно зависим. Мои коллеги, занимающиеся организационной динамикой, иногда называют их «ковбоями». Ковбоев, этих последних из грубых и диких индивидуалистов, можно встретить в разных местах, но в настоящее время многие из них пасут коров на силиконовых полях, причем эти коровы говорят на ассемблере. Кстати, обратите внимание на то, что прозвище «ковбои» заканчивается словом «мальчики» (boys), а не «мужчины» (men).

Весной 1992 года я участвовал в конференции по разработке программного обеспечения (Software Development Conference), организованной компанией Miller Freeman. Один из семинаров был посвящен такой скользкой теме, как «структурированное» и «неструктурированное» управление процессом разработки программного обеспечения. Я сидел рядом с одним из представителей компании Nanomush, отвечающим за разработки. Он был всецело на стороне ковбоев. Его позиция заключалась в том, что полному раскрытию потенциала разработчиков препятствуют руководители, которые стремятся ввести стандарты, ограничения или во всеуслышание заявляют об ожидаемых результатах. По его мнению, нужно просто отойти в сторону и не мешать программистам делать свое дело. Структурированные методы, регламентированная разработка, моделирование на бумаге и оценка программного обеспечения – все это неоправданно ограничивает возможности свободного творческого самовыражения наших блестящих ковбоев-программистов. Неудивительно, что продукты, поставляемые такими компаниями, так непредсказуемы в работе.

Почему нужно выпустить четыре релиза и задействовать 12000 сайтов бета-тестинга (это не шутка!) для того, чтобы выражение «необратимая ошибка в работе приложения, ok?» заменить на более понятное? Но ковбои не любят, когда их обуздывают с помощью особых критериев качества. Ковбоям не нравится заранее думать о том, какие функции продукта действительно необходимы. Нет, давайте лучше просто заскочим в старый добрый загон для разработки и настрочим какой-нибудь код – мы ковбои! Можно придумать симпатичный графический интерфейс и применить искусные методы кодирования – как-никак мы творческие и гениальные личности. Но к черту юзабилити и надежность – для этого может потребоваться планирование или дисциплина (боже упаси).

Не стоит выделять какую-либо из компаний, разрабатывающих программное обеспечение. На рынке имеется бесчисленное множество примеров, подходящих для иллюстрации. Тем не менее очень редко в соответствующей литературе или на семинарах специалистов можно встре-

тить беспристрастную оценку зрелости разработчиков и качества методов программирования.

Зрелость индивидуалистов

Зрелость является центральным вопросом в разработке программного обеспечения. Методисты хотят знать, сколько времени должно пройти для того, чтобы проектирование программного обеспечения созрело как дисциплина. Менеджеры беспокоятся об уровне «зрелости процесса» в тех подходах к разработке, которые применяются в их организациях. Наконец, руководителей проектов интересует зрелость тех индивидуумов, руководить которыми их пригласили.

Одна большая корпорация провела исследования в своих группах, разрабатывающих программное обеспечение. Исследовалось, как часто и на каком уровне сложности эти группы применяют общепринятые систематические или строгие подходы к проектированию программного обеспечения. Оказалось, что наиболее эффективно методы проектирования использовались отделом администрирования информационных систем и отделом деловой информации. Средние результаты показали группы обеспечения проектирования. Как вы догадались, группы, разрабатывающие операционные системы, компиляторы и сервисное программное обеспечение, замыкали таблицу. Исследования о распространенности инструментов CASE дают ту же картину. Там, где дисциплине в проектировании и зрелости процесса не придается значения, разработка программного обеспечения напоминает шоу о Диком Западе, в котором главные роли играют кодирующие ковбои.

Наша культура превозносит гения-одиночку, который от начала и до конца разрабатывает какую-нибудь блестящую теорию, машину или программу. Однако на самом деле никто не делает это, полагаясь только на свои силы. Даже хакер-подросток, укрывшийся в своей спальне и взламывающий программу с помощью собранной им же самим машины, зависит от целой армии инженеров, которые спроектировали и создали чипы, и от целых легионов программистов, которые создали соответствующие инструменты. Для тех, кто следит за моим «показателем скрытой половой дискриминации» (LSI, Latent Sexism Index), скажу, что здесь я не случайно использую местоимение мужского рода. Большинство юных хакеров – мужского пола, а особый склад ума, связанный с желанием взламывать онлайновые компьютерные системы или запускать новых изощренных червей для того, чтобы полностью нарушить работу сетей, почти исключительно является предметом мужской психопатологии. Большинство актов вандализма также совершается подростками, и давайте признаем это. Сочинение вирусов, уничтожение файлов компаний или

взламывание правительственных компьютеров является просто-напросто вандализмом и ничем иным. К сожалению, не за горами то время, когда такой компьютерный вандализм начнет приводить к потерям в людях, тем более, что уже несколько раз мы были близки к этому.

Совместное обучение

Итак, как же мы всего за несколько абзацев перешли от проектирования программного обеспечения и методологической зрелости к вопросам, связанным с полом? Суть состоит в том, что незрелое ковбойское мышление, отвергающее как дисциплину, так и сотрудничество, в значительной степени свойственно мужчинам.

На одной встрече группы планирования, проходящей в компании по разработке программного обеспечения, у собравшейся «команды» ушло 40 минут на конкурентные игры мужчин. Они спорили по поводу определений, старались занять выигрышную позицию, рисовались своими знаниями и эрудицией. В целом, каждый старался одержать верх над другими. Постепенно, один за другим, эти мужчины уходили с совещания под тем или иным предлогом. Чаще всего это было нечто «более важное», имеющее отношение к «реальной работе». Когда же остались только женщины, одна из них сказала: «Ну вот, сейчас мы можем что-то сделать». Они быстро и эффективно справились с задачей, получая удовольствие от самого процесса общения.

Конечно, это стереотип, но нам, парни, нужно признать, что женщины лучше понимают, что такое сотрудничество. Какие бы причины ни лежали в основе этого, маленькие мальчики стремятся состязаться, а маленькие девочки стремятся сотрудничать. Женщинам, как правило, намного лучше удается выстраивать отношения, поддерживать и поощрять друг друга. (Можно спросить, почему же в нашей отрасли они так редко бывают менеджерами и руководителями проектов. Задумайтесь об этом, руководители среднего и высшего ранга: при прочих равных условиях женщина может иметь явное преимущество над мужчиной в качестве лидера команды, даже если команда состоит из тех программистов-неандертальцев, которые утверждают, что не могут работать на женщину.)

В те годы, когда я был семейным терапевтом, я с неохотой пришел к выводу, что большинство современных мужчин знает очень мало о том, что такое воспитание детей или семейные отношения в целом. Это сказано не для того, чтобы принизить мужской пол, это всего лишь статистический факт. Мне посчастливилось встретить необыкновенных мужчин, которые хорошо разбирались в том, как *устанавливать отношения*. Также я встречал женщин, не способных к общению. Ни тот, ни другой пол не удерживает монополию ни на чувствительность, ни на способность уста-

навливать отношения. Но говоря об умении ладить с людьми, нужно отметить, что по крайней мере в большинстве культур у женщин есть преимущество.

Думаю, сейчас я услышу, как кодирующие ковбои станут долго и настойчиво уверять в том, что суровый индивидуализм и независимость в сфере программирования есть единственная надежда американского бизнеса (или человечества – в зависимости от наивности мировоззрения). Именно они утверждают, что совместная работа ограничивает их в средствах, что приход к согласию низводит их до уровня наименьшего общего знаменателя, что необходимость проектировать до начала кодирования затормаживает их работу. Странно, что многие из них создают такие посредственные программы или даже системы, наделенные существенными недостатками.

По правде говоря, встречаются отдельные люди, как мужчины, так и женщины, которые настолько одарены, дальновидны, талантливы и способны к творчеству, что могут выполнять работу самостоятельно и создавать надежные, достойные похвалы продукты. Однако для большинства из нас возможность проявления нашего потенциала основана на умении соединять идеи каждого в творческом синтезе, когда результат превосходит то, что каждый из нас мог бы сделать в одиночку. *Наша* работа, скорее всего, будет лучше, чем *моя* работа либо *ваша*.

Поэтому растите, ковбои! Учитесь, как становиться на плечи друг друга, а не наступать на ноги. И пусть нам подадут руку помощи женщины, ведь нам нужно так многому научиться.

Из журнала *Computer Language Magazine*, том 9, № 8, август 1992 г.

8

Возвращение блудного ковбоя

Когда я впервые обратился к теме программного обеспечения, чтобы написать о проблеме «кодеров-ковбоев», я даже не представлял себе, какое осиное гнездо потревожу. Кодеры-ковбои – это как раз те обитатели индустрии, которые чернят дисциплину любого рода и с равным презрением отвергают методы, модели и менеджмент. Они скорее уйдут, чем станут сотрудничать. Мысль о необходимости проектирования до начала кодирования вполне может привести их к настоящей клаустрофобии. Моя идея о том, что эти ковбои (в основном, мужчины) могли бы поучиться сотрудничеству у коллег женского пола, вызвала целую бурю протестов. Почти все протестующие были мужчинами. Они обвиняли меня во всем, начиная от дискриминации по половому признаку и заканчивая коммунизмом, поэтому неудивительно, что некоторое время я воздерживался от возврата к этой теме. Но обстоятельства, похоже, вынуждают меня вновь поразмышлять о диких местах в дебрях программирования.

Для начала моя младшая дочь закончила колледж. Церемония выпуска сопровождалась бесчисленными фотовспышками. Я особенно помню, как меня распирало от отцовской гордости, когда ей вручали диплом о получении степени по психологии, а также осознание облегчения от того, что в колледж отправлен последний из целого множества мучительно больших чеков. И я помню речь на церемонии выпуска.

Рассказы о шимпанзе

Речи на выпускных церемониях редко бывают запоминающимися. Обычно они либо ханжеские, либо банальные, либо политические; некоторые

особенно ужасные варианты могут обладать всеми тремя характеристиками. Однако новый президент колледжа Уитон (Wheaton) уговорил свою предшественницу Корнел (Cornell) принять его помощь в проводах ее первого выпускного класса. Замечания Карла Сагана (Carl Sagan) были немногословны и полны остроумия и прозорливости. Вдохновленный выпускным классом, лишь вторым по счету, прошедшим четыре года совместного обучения в Уитоне, Саган начал с обсуждения взаимосвязи между полом и поведением. Он сосредоточился на женщинах, мужчинах и том мире, который они создают для себя. Сперва Саган обратил всеобщее внимание на шимпанзе. Безусловно, шимпанзе являются нашими генетическими братьями и сестрами – 99,6% активных генов шимпанзе и человека совпадают. Естественно, поведение шимпанзе и человека определяется не только генетикой, но в то же время не все решает и обучение. Подобно тому как мы способны, взглянув на своих собственных родителей, рассказать о себе, мы можем узнать что-то о *homo sapiense sapiense*, если посмотрим на отражение своих родственников-приматов.

Оказавшись в стрессовых условиях густонаселенности, особи шимпанзе мужского пола становятся более агрессивными. Они все больше стремятся к соперничеству, собирая камни для бросания и не подпуская к себе других самцов. Саган рассказал об одном шимпанзе, который, держа в руках камни, столкнулся на своем пути с тихой самкой. Медленно и мягко она раскрывает пальцы его сжатых кулаков, камни падают на землю, и она уходит. Для части самцов этого достаточно, чтобы они могли заняться другими делами, но некоторые этого просто не понимают. Такие самцы медленнее обучаются, поэтому их нужно несколько раз мягко разоружать для того, чтобы им все стало ясно. Это напомнило мне некоторых из знакомых мужчин.

От шимпанзе Саган перешел к людям. Он говорил о широко известных различиях между тем, как играют мальчики и девочки, и о том, что женщины стремятся к сотрудничеству, тогда как мужчины склонны к соперничеству. Он также задавал вопрос о том, стал бы мир более единым, если бы женщины действительно имели равные возможности для получения власти и влияния. Не в виде нескольких сенаторских мест или символических административных должностей, а по-честному – 50% всего управленческого пирога.

И я подумал, где же был Карл Саган в прошлом году, когда меня осаждали банды программистов-шимпанзе мужского пола за то, что я говорил то же самое. Наверное, им было тесновато от разговоров о совместном кодировании.

Если бы все кодеры-ковбои были героями-одиночками, то они не создавали бы такого количества проблем. У многих из них есть положительные

черты. В уединении они могут проявлять головокружительную производительность в работе над отдельными приложениями. Даже один или двое таких программистов, включенных в команду, могут влить в проект живую струю. Если присматривать за их работой над интерфейсами и личными контактами в команде, то ковбои могут привнести разнообразные идеи и подходы, не нарушая целостности всей системы.

К сожалению, ковбои известны тем, что собираются не только на родео и в загонах, но и в больших фирмах, разрабатывающих программные продукты. В них ковбои создают сложное системное программное обеспечение – с акцентом на слове «сложное». Вполне возможно, что срывы рабочих графиков и неработоспособные программы, заполонившие нашу отрасль, связаны с дикими методами программирования, которые практикуют недисциплинированные кодеры-ковбои и те руководители-одиночки, которые их поощряют.

Несомненно, кодеры-ковбои могут обладать творческими способностями. Но это не всегда является ценным. От системного программного обеспечения зависит все остальное, поэтому мы ждем от него безупречной производительности, основывающейся на максимальной надежности. Иногда производительность может быть достигнута с помощью ковбойской тактики, однако надежность программного обеспечения обычно достигается за счет дисциплины, к которой ковбои относятся с пренебрежением. Для создания кода, который работает не только быстро, но и безошибочно, необходимо в строгой последовательности применять самые лучшие методы разработки программного обеспечения.

Большое количество ковбоев означает большое количество строк кода, независимо от того, нужны они или нет. Это означает большое количество неструктурированного кода, который был придуман на лету и написан в одиночку. Он разный, и каждый кусок несет на себе отпечаток уникального ковбойского стиля.

Ковбои над ковбоями (trail bosses[1])

Представьте, что вы пытаетесь создать совершенно новую операционную систему, но не с помощью хорошо скоординированной команды дисциплинированных профессионалов, а с помощью пары сотен кодирующих ковбоев. Довольно трудно руководить небольшой компанией ковбоев. Нечего и говорить о целом ранчо, где ковбоев пруд пруди. Нужно сильно постараться, чтобы привлечь их внимание.

---

[1] Trail boss (амер., устар. от *trail* – тропа и *boss* – начальник). На Диком Западе это старший группы ковбоев, ответственный за скот и погонщиков при перегоне табуна. – *Примеч. науч. ред.*

Наверное, после этого мы станем благожелательнее относиться к руководителю проекта по разработке Windows NT, который постоянно осаживал своих пастухов. Рассказывают, что для этого он даже пробил дыру в стене офиса. Возможно, как бывший программист с хорошей репутацией он понимал склад ума одиночек.

Увы, понимания бывает недостаточно. Как писала газета *Wall Street Journal* (от 26 мая 1993 г.), весь проект, казалось, был обречен на то, чтобы произвести колоссальный, чрезвычайно сложный и насыщенный ошибками программный продукт. Около 200 программистов неистово занимались написанием кода в соответствии с определяемыми на лету требованиями, отчего вся разработка проекта превратилась во всеобщую драку.

Программисты и тестеры, как скотоводы и овчары в давние времена, были разбиты на два конкурирующих лагеря. Эти команды натравливались друг на друга, а их участники сходились в ожесточенных схватках. Такое разделение труда может быть эффективным, если говорить о сокращении ошибок в программном обеспечении. Но в «табели о рангах» кодеры (в основном мужчины, самоуверенные мужчины) стояли первыми. Когда кодеры жаловались на то, что тестеры копают слишком глубоко, результаты тестеров признавались недействительными. Вероятно, для того, чтобы завершить проект вовремя или повысить производительность.

С нарушением всех графиков и растущим количеством слабых мест коллектив разработчиков NT перешел в «режим дополнений»... и так и оставался в нем почти два года. Обычно частота ошибок возрастает, когда люди испытывают усталость или находятся в стрессовых условиях. Многие часы интенсивной работы только увеличивают количество ошибок, вносимых в программный код. Несколько недель разработчики занимались поиском и исправлением примерно тысячи багов. Но даже самые лучшие методы регрессивного тестирования могут выявить лишь часть внесенных ошибок. Оставшиеся ошибки, порожденные в те долгие рабочие и выходные дни, еще ждут будущих пользователей.

Однако подход может быть иным. Дисциплина бывает полезной. Вот что было сказано в докладе Эла Пиитресанта (Al Pietresanta) на конференции по разработке программного обеспечения, проходившей в 1989 году в Бостоне: если применять «чистые» («clean room») методы кодирования и непрерывного технологического улучшения, то даже в очень больших системах может быть меньше одной ошибки на 10 000 строк кода.

Зрелость окупается. Замечено, что если большие проекты осуществляются зрелыми организациями с помощью устоявшихся технологий, то расходы на разработку сокращаются в 20–30 раз по сравнению с применением более свободных методов «на скорую руку».

«Ковбоизм» в кодировании может иметь много бастионов, но случай с ранчо Редмонд можно считать уникальным. Издание *Journal* приводит слова Митчела Дункана (Mitchell Duncan), главного конструктора проекта NT: «Все наши разработчики-ковбои строчат код просто как сумасшедшие».

И настрочили они 4,3 миллиона строк. Только смотрите внимательно, куда наступаете.

Из журнала *Software Development*, том 1, № 10, октябрь 1993 г.

9

Единство в разнообразии

Что требуется для того, чтобы быть лидером? Какой вид лидерства ведет к эффективной командной работе и успешному решению задач? В конце 70-х годов английский колледж подготовки административных работников (Administrative Staff College) попросил британского консультанта по менеджменту Р. Мередита Белбина (R. Meredith Belbin, 1976 [3]) найти ответы на эти вопросы с помощью проведения формальных наблюдений и экспериментов.

Белбин организовал команды из опытных руководителей среднего и высшего ранга. Эти команды приняли участие в соревнованиях, которые можно было объективно оценить с помощью новейших научных методик и подходов. Начав с команды «А», объединившей самых лучших и самых ярких, действительно выдающихся личностей, Белбин распределял участников по разным командам. Нераспределенными остались те, кто не подходил ни для одной из тщательно собранных команд высококлассных участников. Из них он сформировал последнюю команду, которая представляла собой разношерстную группу ничем не примечательных специалистов.

После проведения довольно сложных тестов и упражнений команды были распределены согласно объективным показателям производительности. Ко всеобщему удивлению, команда «звезд» заняла последнее место, тогда как разношерстная группа оказалась первой.

Необходимые роли

Тщательное исследование результатов показало, что ключевым фактором было разнообразие. Команды, члены которых проявляли большее

разнообразие в стиле руководства, достигали больших результатов, чем команды, применяющие однообразные методы руководства и взаимодействия внутри группы. На основании своих наблюдений и данных об участниках команд Белбин выделил восемь различных «лидерских ролей» (или командных функций), которые, по всей видимости, играли специалисты самых разношерстных, самых успешных групп.

Лидерские роли, выявленные Белбином, представляют собой командные функции, выполнение которых необходимо для достижения наивысшей производительности. Кроме того, они представляют стили, в рамках которых эти лидерские функции могут осуществляться. Четыре функции соответствуют типичным представлениям о том, каким должно быть поведение настоящих лидеров. *Бригадир* – это член команды, который обычно определяет правила и руководит мыслительной деятельностью команды и происходящими в ней обсуждениями. Он ведет команду к конкретной цели или результату, навязывая подходы и схемы работы. В этом смысле мы все знакомы с бригадирами. Зачастую такие люди занимают главенствующее положение в командах. Однако существуют и другие важные формы командного лидерства. *Зачинатель* – это генератор идей, рационализатор и изобретатель, который выдвигает новые идеи и подходы. *Координатор* – это лидер с точки зрения процесса. Он координирует решение задач, рассматривая всю команду в качестве «человеческого ресурса». *Наставник* занимается критической оценкой и анализом всей групповой работы. По существу, это лидер в обеспечении качества.

Другие четыре роли, которые Белбин рассматривает как формы допустимого лидерства, чаще всего считаются вспомогательными. Одна из них именно так и называется – *помощник*. Помощник является эмоциональным лидером. Он пробуждает командный дух и заботится о каждом члене команды в отдельности. *Конструктор* руководит переходом от концепций к практическим системам и решениям в соответствии с утвержденным планом. *Доводчик* контролирует завершение работы группы, поддерживает сосредоточенность внимания группы на критических участках и напоминает о сроках. Наконец, *исследователь* управляет взаимодействием с другими группами и доступом к ресурсам, а также организует и проводит исследования.

Для наибольшей производительности руководство командой должно быть представлено в нескольких лицах. Обычно функции лидера распределяются между членами команды, а не аккумулируются в одном «суперлидере», играющем все эти роли. Такая модель «совместного лидерства» является отличительным признаком успешных групп.

Для эффективной командной работы существенное значение имеют два различных толкования разнообразия. Высокопроизводительным коман-

дам необходимы разнообразные технические навыки и квалификация. Для большинства людей эта часть представляется простой. Команда, участники которой обладают разнообразным опытом и способностями, может охватить больший технический ландшафт, чем это мог бы сделать один человек. Безусловно, всегда существуют «суперзвезды», которые стремятся играть все роли одновременно и быть экспертами во всем, но мало кто из них добивается настоящего успеха. Слишком большой диапазон интересов наносит ущерб глубине и строгости знаний, а дилетант рискует стать объектом критики. (Некоторые люди считают, что эклектичный и разносторонний автор этой книги демонстрирует лишь «интеллектуальную беспорядочность».)

Работа Белбина (а также множество других исследований, проведенных после нее) показывает, что разнообразие в *стиле* по меньшей мере так же важно, как и разнообразие в содержании. Под стилем я подразумеваю не только личностные качества, но и навыки межличностного общения. Если каждый старается раньше других протиснуться в одну дверь, то в этой двери застревает вся команда. Если каждый начинает что-то выторговывать для себя перед тем, как оказать необходимую помощь, то вся команда может так и остаться стоять у трапа самолета, который улетает без нее.

Предпочитаем похожих

Полученные данные ставят разработчиков программ в непростые условия. Одна из трудностей состоит в том, что почти каждый предпочитает работать с людьми, похожими на него, в то время как производительность команды повышается, когда ее составляют люди, непохожие друг на друга. Разнообразие навыков и стилей работы, в которых эти навыки проявляются, придает команде гибкость.

Я думаю, что тема разнообразия в команде очень близка к теме этнического разнообразия. Особенно стоит отметить вклад этнических групп в богатство культуры и поваренное искусство. Мир был бы намного беднее без chiles rellenos, tom kah gai, szekely goulash, pesto sauce, lamb vindalo, coq au vin, strange-flavored beef, pasta primavera, paella и feijoada.[1] Если

[1] Chiles rellenos – фаршированные перцы, Мексика; tom kah gai – суп из молока кокоса, Таиланд; szekely goulash – гуляш, Трансильвания; pesto sauce – соус из листьев базилика, Италия; lamb vindalo – баранина, приготовленная с острой приправой карри, Индия; coq au vin – курица в красном вине, Франция; strange-flavored beef – мясо с овощами, грибами и молодой кукурузой, Китай; pasta primavera – блюдо из макарон со свежими овощами, Италия; paella – блюдо из моллюсков и морепродуктов, Испания; feijoada – тушеное мясо с черными бобами, Бразилия. – *Примеч. науч. ред.*

мы хотим узнать, как можно эффективно работать в разношерстных командах, нам нужно научиться ценить разнообразие в навыках, знаниях и стиле так же, как мы ценим разнообразие в питании.

Конечно, я также знал одного выдающегося человека, который работал системным аналитиком на почте и каждый день ел на завтрак одно и то же в течение 17 лет. Есть люди, которые как раз предпочитают есть одну и ту же пищу все время, и я не знаю, как им это удается. Хорошая модель командной работы найдет для них место, но я не уверен, что они действительно захотят работать в команде с разными людьми или просто пообедать у меня дома.

Не всегда легко определить, когда разнообразие переходит в несовместимость. «Странная парочка» (*The Odd Couple*[1]) часто приводится как классический пример принципиальной несовместимости, однако нужно отметить, что Феликс Ангер (Felix Unger) и Оскар Мэдисон (Oscar Madison) работали как раз очень слаженно, хотя и постоянно донимали друг друга. Наверное, в командной работе чье-то личное недовольство не так важно, как коллективная производительность.

Должно быть, именно на этом фоне важно отметить поступление волнующих свидетельств, подтверждающих, что программисты и другие профессионалы в области компьютеров в основном принадлежат к определенному типу личности. Австралийский консультант Роб Томсет (Rob Thomsett) обнаружил, что более 60% людей, работающих в компьютерной индустрии, имеют одинаковый базовый тип личности. Эти люди практичны, склонны к логическому, реалистичному мышлению, основанному на фактах. Они способны сосредотачиваться и удерживать внимание на полезных предметах и практических вопросах. Они не очень общительны. Такому типу личности соответствует менее 20% от общей численности людей. Типичные компьютерщики, принадлежащие этой группе, склонны играть лишь некоторые из описанных выше лидерских командных ролей. Чаще всего они становятся конструкторами, доводчиками, наставниками или бригадирами. Только немногие из них (если вообще кто-нибудь) выбирают роль координатора или помощника, незаменимых для четкого и эффективного общения в команде, или роль зачинателя, которая важна для творческого решения задач.

Томсет считает, что причина кроется в отборе курсов профессиональной подготовки и высших учебных заведений, однако здесь могут быть и другие факторы. Мои российские коллеги по консультированию в области управления утверждают, что компьютерное программирование является такой мощной субкультурой, что она может оказывать большее влияние

[1] Художественный фильм, США, 1968 г. – *Примеч. ред.*

и требовать к себе большей преданности, чем даже национальная культура. По их мнению, программисты из Москвы больше похожи на программистов из Миннеаполиса, чем любой из них похож на своих соотечественников-непрограммистов из тех городов, в которых они живут и работают. Какими бы ни были причины, такая тенденция к единообразию может стать помехой в нашем деле, особенно в командной работе.

Джошуа Якобсон (Joshua Jacobson), дирижер бостонского хора «Zamir», а также один из моих любимых философов, на одной из репетиций сказал об этом так: «Если каждый станет петь одну и ту же ноту, вы не получите гармонии. Для достижения гармонии нужно, чтобы люди пели разные ноты, которые сочетаются друг с другом». Здесь он вторит другому моему любимому философу, Гераклиту Темному. «Темным» он был назван не потому, что его труднее понять, чем последующих греческих философов, а потому, что о нем почти ничего неизвестно кроме косвенных сведений и того, что ему приписывали другие. Возможно, это объясняет, почему он славится таким большим количеством мудрых и замечательных высказываний в поддержку столь разных позиций. Предположительно, он сказал следующее: «Εχ τωυ διαφεροντων», что в переводе с греческого приблизительно означает: «Из разных песен рождается высшая гармония».

Аминь, Гераклит! Селах[1], Якобсон!

Из журнала *Computer Language Magazine*, том 9, № 9, сентябрь 1992 г.

[1] Selah (древнеевр.) – в книге Псалмов обозначает паузу, разъединяя и в то же время связывая соседние отрывки текста, играя роль, аналогичную слову «аминь». Этимология и точное значение неизвестны.

10

Кодеры-ковбои и программисты-мудрецы

Качество – вот что стало лозунгом в разработке программного обеспечения. Отчасти озабоченность качеством является искренней, а отчасти это всего лишь панические настроения руководства, которое старается догнать уходящий поезд. Одни «программы обеспечения качества» дают результат, другие служат только для того, чтобы создавать благоприятное впечатление у потребителей. Когда вопросы качества программного обеспечения выходят на первый план, проблема зрелости приобретает большее значение, поскольку для эффективного обеспечения качества требуется зрелость как организации в целом, так и ее отдельных работников.

По мере того как программные системы становятся больше и сложнее, перед руководителями в области разработки программного обеспечения встают новые задачи. Организации стремятся к большей «зрелости процесса», применяя более строгие и сложные модели разработки программ. В умах многих менеджеров возникает вопрос о том, смогут ли профессионалы-разработчики оправдать свое звание профессионала и достичь зрелости, используя эти методы. Программисты, аналитики и дизайнеры – это особая порода людей, и руководство целой толпой таких одиночек может отнять все силы даже у самых лучших руководителей. Проблемы взаимодействия с твердолобыми индивидуалистами, которые скорее уйдут, чем станут сотрудничать, сегодня являются самыми острыми для групп по разработке программного обеспечения.

Несмотря на избыток одиночек в данной области, большинство программ производится группами людей, работающих вместе. Часть программного

обеспечения создается группами людей, работающих раздельно. Некоторые программы производятся группами людей, работающих даже друг против друга. И только очень небольшая часть программного обеспечения разрабатывается индивидуумами, работающими в одиночку.

Эти очевидные, простые и, как может показаться, нелогичные факты, может быть, и не стоило здесь упоминать, если бы не мифы о гигантах, имеющие хождение в сфере разработки программного обеспечения. Эта мифология прославляет блестящего гения, который один придумывает и не покладая рук кодирует совершенно новые умные системы, не спя ночами и работая по выходным. Наше счастье и беда в том, что у нас есть эти прометеевские образы необщительных первопроходцев, которые дают нам новые языки, новые инструменты и новые парадигмы компьютерных приложений. И наше счастье и беда в том, что есть армия менее богоподобных и менее талантливых программистов, хотя и не менее упорных. Они также могут быть решительными индивидуалистами, способными стоять на своем, работать в одиночку без вмешательства со стороны, без помощи, без постороннего контроля, методологии или обсуждений.

Консультанты в области управления и организации часто называют таких людей «ковбоями». Ковбоев, этих последних из грубых и диких индивидуалистов, можно найти в разных областях, но в настоящее время многие из них пасут на силиконовых полях коров, говорящих на ассемблере. За стремление к уединению и странные привычки ковбоев также называют «людьми-пумами». Это одиночки, которые либо делают все посвоему, либо не делают ничего.

На случай каких-либо сомнений важно отметить, что я и сам являюсь одиночкой (по крайней мере, мне так говорили). На самом деле я был официально отнесен к типу одиночек Полом Вордом (Paul Ward) – методистом, ставшим историком, – в его истории структурного анализа (Ward, 1992 [64]). Он лично заверил меня, что это нужно воспринимать как комплимент. Конечно, большую часть моей профессиональной жизни я не был конформистом. Современная разработка программного обеспечения может охватывать многие основные принципы структурного проектирования. В качестве ортодоксальной технической иконографии могут применяться инструменты CASE, среды комплексной разработки, схемы потоков данных и блок-схемы, однако так было не всегда. В это трудно поверить, но все эти схемы когда-то считались отступлением от традиций и даже радикальным бредом индивидуалистов.

Я верю в индивидуалистов; многие из моих друзей – одиночки. И я верю в индивидуалистское воображение, в индивидуальное творчество как источник почти всех подлинных новшеств. Я также считаю, что индивидуализм – это истинная причина большинства глупых ошибок. На каждого

Эйнштейна находится целая толпа Великовских[1]. Иногда нововведения и идиотизм даже исходят из одного и того же источника, как от Теслы (Tesla) или Вильгельма Рейха (Wilhelm Reich)[2]. Так или иначе, индивидуалисты-одиночки обогатили нашу жизнь – если не всегда чем-то полезным, то хотя бы развлечениями.

Быть ковбоем – это не то же самое, что быть независимым или быть индивидуалистом. Это определение не связано с тем, использует или нет кто-либо какую-то конкретную методологию разработки программ, если вообще использует какую-то методологию. Ковбой – это некий тип мышления или стиль жизни. Программисты-ковбои – это всего лишь те оппозиционные разработчики, которые ненавидят ограничения в виде стандартов, условий или дисциплины. Это специалисты, которые сопротивляются всем попыткам руководить ими с помощью контроля или сотрудничества с другими и ставят свою уникальную оригинальность выше понятий юзабилити и надежности.

Естественно, не каждый программист, который предпочитает работать самостоятельно, является кодером-ковбоем. Некоторые люди являются одиночками по своему темпераменту. Некоторых, наверное, можно охарактеризовать как отшельников, другие всего лишь любят составлять компанию самим себе, а многим просто кажется, что компания других людей отвлекает их или даже подавляет. Некоторые из моих лучших работ были сделаны, когда в комнате находились только я и мой верный компьютер.

Как руководить индивидуалистами

Почему это важно – руководить индивидуалистами? Почему бы просто не дать им волю, чтобы они могли объединиться с такими же ковбоями в качестве программистов-контрактников и независимых консультантов? Во-первых, их довольно много. Во-вторых, многие их них являются неплохими профессионалами и потенциально могли бы стать еще большими доками, если бы их творческое упрямство можно было хотя бы немного умерить с помощью сотрудничества. Индивидуалисты и кодеры-ковбои способны внести существенный вклад в процесс разработки программного

---

[1] Иммануил Великовский (1895–1979) – американский историк русского происхождения, успешно издавший ряд лженаучных работ по истории и астрономии. – *Примеч. науч. ред.*

[2] Никола Тесла (1856–1943) – сербско-американский изобретатель, Вильгельм Рейх (1897–1957) – немецко-американский психолог; оба известны как своими научными достижениями, так и весьма спорными тезисами. – *Примеч. науч. ред.*

обеспечения. Хорошее руководство создает условия для того, чтобы принять и использовать этот вклад с пользой для организации и для самих ковбоев.

Некоторые руководители являются сторонниками либерального стиля управления кодерами-ковбоями. По их мнению, дисциплина или навязывание стандартов только удерживает программистов от полного раскрытия их потенциала блестящих кодеров-ковбоев.

Я думаю, что у руководителей есть варианты получше – по крайней мере, я надеюсь, что это так. Такой свободный подход, возможно, объясняет ненадежность и плохую работу продуктов, поставляемых многими крупными компаниями-разработчиками. Этот подход также проявляется в пользовательских интерфейсах большинства программ, которые страдают прогрессирующим функционизмом и начинены мешаниной из несогласованных между собой кнопок и переключателей, добавляемых чуть ли не каждым членом компании разработчиков.

По мере того как размер и сложность программных продуктов возрастают, для руководителей проектов становится все более важно знать, как можно с умом задействовать изолированных разработчиков. Даже в проектах, в которых доминирующей моделью является работа в команде, лучший руководитель найдет способы учесть потребности одиночек. Обеспечить для них полную изоляцию бывает трудно, но, с другой стороны, не обязательно заставлять их посещать каждое собрание и каждый критический разбор программ.

Одиночки могут делать хорошую работу и могут делать плохую. Именно руководитель должен найти способ использования одиночек таким образом, чтобы они могли делать хорошую работу. Отчасти секрет заключается в том, как работа разбита и распределена между членами команды. Тем сотрудникам, которые лучше работают в уединении, следует поручать задачи, которые в той или иной мере могут быть автономными, то есть те части системы, которые можно определить как черные ящики с простыми интерфейсами и четкими внешними спецификациями. Таким разработчикам нужно давать задачи по созданию компонентов, которые не сильно связаны с другими частями системы и не требуют тесной координации или постоянного общения с другими разработчиками.

Другой частью секрета является тщательный контроль и наблюдение. Уменьшение взаимозависимости между индивидуалистами и остальной частью команды разработчиков не означает, что индивидуалистов нужно игнорировать. На самом деле, когда часть системы разрабатывается отдельно, еще более важно следить за остальными интерфейсами и взаимосвязями. Руководители часто понимают это, когда имеют дело со сторонними исполнителями или удаленными сотрудниками, работающими до-

ма. Однако они склонны забывать эту истину, когда имеют дело с индивидуалистом, который сидит в соседней комнате.

Когда работа выполняется в открытых группах, повышенная видимость помогает сократить ошибки и повысить качество (см. главы 26 и 33). Поэтому работа, выполненная в некоторой степени изолированности от остальной группы, требует большего внимания к соответствию внешним спецификациям и более пристального наблюдения за качеством продукта. Если приемочные испытания показывают, что код работает, то этого недостаточно; для обеспечения качества следует также изучить его с точки зрения соответствия стандартам ясности и надежности.

Индивидуалисты и методы

Самостоятельная работа не подразумевает работу без дисциплины или без хорошей методологии, а работа в группе не гарантирует получение результата. Работа в группе гарантирует только то, что в нее будет вовлечено большее количество людей. Однако с точки зрения руководителя использование системной методологии в разработке приобретает большее значение, когда разработчики трудятся отдельно друг от друга. И здесь для обеспечения качества особенно полезны регрессивные испытания и критический разбор внедрения программы. Настоящее качество не может быть достигнуто после факта разработки – его нужно продумывать и встраивать в процесс с самого начала. Системные и формальные методы разработки программного обеспечения являются проверенными способами обеспечения качества. Работа даже самого независимого кодирующего ковбоя может быть улучшена и, конечно, может стать более надежной, если в процессе применяется системная методология.

Для того чтобы заставить кодирующих ковбоев применять системные методы разработки, руководству, возможно, придется искать компромисс в отношении того, *какой* именно метод будет выбран. Если независимые исполнители будут применять собственные уникальные методы, то это лучше, чем не использовать никаких.

Модели проектирования могут не только ускорить разработку и улучшить качество, но и способствовать трассируемости. Это достигается с помощью частичного протоколирования мыслительного процесса разработчика, то есть с помощью ведения журнала контроля, в котором отражаются источники проблем и их решения. Необходимость соответствующих проектных документов (таких как схемы потоков данных, блок-схемы, структурные схемы потоков и т. п.) должна быть отражена в спецификациях для подсистем, которые предполагается разрабатывать независимо.

Для руководителей проектов, стремящихся удержать в голове набор подходов к управлению программистами-одиночками, может быть полезной

аналогия с выпасом скота. Старшим ковбоям приходилось перегонять стада на пастбища, следя за тем, чтобы животные не отбивались от стада, и периодически собирая их вместе. Они внимательно следили за своими работниками и скотом. Они также не забывали про то, чтобы ковбои могли выпустить пар в субботние вечера.

Ковбойские коллективы

Меилир Пейдж-Джонс (Meilir Page-Jones) придумал очень практичную схему для определения эпохи зрелости, в которой пребывает процесс разработки программного обеспечения. Некоторые группы работают в *эпоху анархии* и разрабатывают программное обеспечение без помощи каких-либо системных подходов и даже не применяют систематизированные знания. Все опирается на индивидуальные умения. *Эпоха фольклора* характеризуется культурой коллективной мудрости, накопленного знания, которое часто воплощено в историях об успехе или неудаче или в эмпирических правилах, приобретенных в прошлом. *Эпоха методов* основана на использовании системных, не всегда формальных подходов к разработке программного обеспечения, которые выходят за пределы фольклора. *Эпоха метрики* базируется на проведении измерений для оценки качества и производительности и на организации обратной связи для улучшения процесса разработки, основанного на измерениях. И наконец, *эпоха инжиниринга*, в которую разработка программного обеспечения становится настоящей инженерной дисциплиной. В процессе непрерывного усовершенствования применяются методы, основанные не на фольклоре или догадках, а на теории, проверенной в исследованиях. Принимаемые решения и компромиссы являются системными и вырабатываются на основе моделей и измерений, вовлекая данные из растущего объема знаний.

Инжиниринг – это то, что вы получаете, когда зрелые индивидуумы в зрелых организациях используют зрелые методы. Анархия возникает, когда вы просто загоняете группу кодирующих ковбоев в загон и указываете им на проблему.

К сожалению, когда индивидуалисты включаются в группы, у них проявляется склонность к противостоянию. Нередко они начинают стремиться к противоположным целям. Без творческого лидерства группа кодирующих ковбоев может привести к неуправляемому хаосу.

Находясь в команде с другими, менее упрямыми людьми, кодирующие ковбои могут склоняться к нарушению командной работы. Одни ковбои воздерживаются от участия в групповом решении задачи, другие могут отнимать много времени, предлагая свои любимые варианты и подходы. Зачастую они весьма критически относятся к любым идеям, которые были предложены не ими, и вступают в конфликт со всей остальной группой.

В худших случаях ковбои будут загнаны в угол и подвергнутся презрению за свой негативный настрой. Со своей стороны, кодирующие ковбои могут начать смотреть на всю команду как на сборище поборников группового мышления, которые не смогут закодировать даже выход из картонной коробки и просто не способны оценить истинную гениальность.

Руководители должны знать, как предотвратить такую поляризацию в коллективе ради сохранения команды и получения конечного продукта. Прямая конфронтация является нежелательным стилем руководства в отношении кодирующих ковбоев, подавление которых может быть довольно рискованным шагом. Они заработали свою репутацию за свои твердые мнения, чувствительное самолюбие и зудящие указательные пальцы. Руководители вряд ли смогут победить в таком столкновении. Оно может только побудить ковбоев уйти в еще более глубокую оппозицию.

Главный вопрос для руководителей – что делать с индивидуалистами. Некоторые руководители настаивают на том, что индивидуалистов следует отделять от стада и уводить на выпас, но этот вариант не всегда доступен и может не соответствовать интересам организации. Мой начальник говорит, что работа руководителя заключается в устранении препятствий, которые мешают подчиненным делать работу хорошо и раскрывать свой потенциал. Как мне повезло. Часть потенциала индивидуалистов, которые работают под началом хорошего руководителя, состоит в том, чтобы принести в организацию свой заряд творческой энергии и недюжинные способности.

Некоторые команды выигрывают от оппозиции, тогда как другие не могут перенести разногласий. Это зависит от основной модели, по которой организована команда (Constantine, 1993 [21]; Larson и LaFasto, 1989 [47]). Проектные команды, которые основаны на иерархическом типе руководства и назначении заданий или опираются на традиционные приказы и инструкции, или возглавляются властными лидерами, будут испытывать большие сложности с индивидуалистами и кодирующими ковбоями. Работа таких традиционных тактических команд (см. главу 11) может быть затруднена оппозицией, а подобный стиль управления может подтолкнуть индивидуалистов к противодействию.

С другой стороны, ничем не стесненная команда «прорыва» (глава 12) может только укрепиться от присутствия в ней индивидуалистов, чей независимый дух и творческая индивидуальность так нужны группе. Весь вопрос состоит в том, как создать обстановку творчества, а не хаос. Ключом к получению полезного результата является поддержание высокого уровня взаимоуважения и признание профессионализма друг друга. В командах, где сотрудники смотрят на своих коллег как на опытных про-

фессионалов, способных генерировать хорошие идеи, здоровое соперничество не переходит в драку.

Команды, стремящиеся к техническому консенсусу (см. часть I), могут также быть хорошим местом для индивидуалистов. Технический консенсус достигается тогда, когда вклад каждого члена команды учитывается в общем решении, которое все участники команды признают правильным, хотя и не обязательно могут быть согласны с каждой его деталью.

Оппозицию и критицизм никогда не следует путать с нелояльностью. В действительности критическая оценка часто является самой ценной информацией, услышать которую важнее всего. Здоровый скептицизм и критический взгляд нужно поощрять, а не отвергать. Исследования выявили, что качество группового решения задач напрямую зависит от этой «негативной обратной связи». Группы оказываются более эффективными, если в них присутствует «резидентный критик», или «адвокат дьявола», или если в них лучше используются диалектические процессы противопоставления идей и активная критика (Constantine, 1989 [11]; Priem и Price, 1991 [59]).

Руководители должны помнить, что оппозиция сопротивляется контролю и власти в той мере, в какой власть старается контролировать оппозицию. Поэтому, вместо того чтобы контролировать, сотрудничайте! Один из способов эффективного сотрудничества с оппозицией – наделить ее законным статусом. В одном из подходов (Constantine, 1989 [11]; 1991 [13]) резидентный критик или «адвокат дьявола» рассматривается как ценный сотрудник, роль которого очень важна для успеха группы. Так же, как и «песчинка в устрице», критическое наблюдение за работой команды является необходимым раздражителем, формирующим базис для появления жемчужин в области программирования (Thomsett, 1990 [62]). Как член команды индивидуалист формально обязан критиковать решения, придерживаться альтернативных точек зрения, отмечать исключения, сложности и ограничения и следить за тем, чтобы группа рассматривала альтернативы и исследовала трудности, связанные с предложенными решениями.

Зачастую индивидуалисты хороши как критики, а не как критикуемые. Однако когда они с головой уходят в работу, их результаты следует контролировать и проверять. Это может вызвать возмущение индивидуалистов, поскольку воспринимается ими как назойливость и подозрительность. Критические отзывы о работе кодирующих ковбоев лучше всего выражать в максимально безличной форме, обращаясь прежде всего к аналитическим и критическим способностям индивидуалистов и призывая к творческому подходу в решении любых возникающих задач.

Как это ни удивительно, но наибольшие ограничения на индивидуалистов накладываются ими самими. Они могут упорствовать в своем желании быть уникальными, сделать все по-другому, даже в ущерб практичности. Работа в одиночку также ограничивает список того, за что они могут взяться. В конце концов, индивидуалисты могут вызывать у коллег отторжение.

Разногласия и разнообразие

Теперь мы знаем, что разнообразие в командах способствует лучшему преодолению трудностей и принятию решений. Исследования по групповой динамике, которые проводились в течение нескольких десятилетий, показали, что разнородность почти всегда дает больше преимуществ, чем однородность. Команды, состоящие из мужчин и женщин, лучше справляются с решением проблем, чем однополые команды. Группы, которые состоят из людей разных специальностей и с разным образованием, достигают лучших результатов. Разнообразие приносит успех независимо от того, как оно проявляется – в личностях, в стиле общения, в культуре или в этническом происхождении.

Таким образом, не следует избавляться от всех индивидуалистов. Нужно, чтобы в командах проявлялось все разнообразие стилей лидерства и откликов на это лидерство, в том числе и сопротивление ему. Командам нужны движущие силы, новаторы, способные генерировать новые идеи, и чемпионы, способные их продвигать. Но также нужны критики и скептики, которые своим холодным сомнением противостоят необузданному энтузиазму и не пропускают идеи, не имеющие хорошего обоснования. Оппозиция лидерству может быть полезной даже для самой команды, так как позволяет удерживать руководство в определенных рамках.

Трудности возникают, когда мы рассматриваем оппозицию с точки зрения упрощенных методов командной работы, основанных на идеалах полной гармонии и фантазиях о сотрудничестве без конфликтов. Спор может способствовать наивысшей коллективной производительности и быть самым важным элементом в творческом процессе. В результате появляется качественный продукт.

Кто же такой программист-мудрец? Это тот, кто рассматривает оппозицию как возможность и способен видеть гибкость в споре. Программист-мудрец – это старый, умудренный опытом работник фермы, который прошел через фермерские войны, побывал в пустынных прериях и вернулся обратно на родную ферму. Программист-мудрец может быть даже исправившимся ковбоем – тем, кто никогда не терял уважения к индивидуализму, но который научился ценить сотрудничество.

По материалу из журнала *American Programmer*, июль 1993 г.

III

Организация работы

Введение

Хотя я приступил к работе в компьютерной отрасли, получив степень по менеджменту в Массачусетском технологическом институте, сначала мне была более интересна техническая сторона программирования. Впервые организацией командной работы и вопросами организационного развития я серьезно занялся во время моей работы с семейными системами. Несмотря на то что моя деятельность в области вычислительной техники и программного обеспечения продолжается уже более тридцати лет, случилось так, что в середине карьеры я отклонился от нее лет на двенадцать. В те годы мне посчастливилось быть теоретиком в области прикладных систем. Мой переход к деятельности, связанной с семьей, произошел в то время, когда семейная социология и семейная терапия двигались по направлению к теории систем (Whitchurch и Constantine, 1992 [67]). В исследованиях супружества и семьи, в социологии и социальной психологии появилось понимание того, что группы людей функционируют как системы. В результате к человеческим системам стали применяться общая теория систем и системный подход.

Мне посчастливилось учиться и работать под руководством психолога Дэвида Кантора (David Kantor), одного из самых талантливых исследователей и самых творческих мыслителей в данной области. Кантор изучал семьи с помощью непосредственного наблюдения и документирования, описывая непростой танец повседневной жизни, в котором семьи согласовывают и поддерживают свою деятельность и внутренние отношения. Его необычайно богатые наблюдения привели к пониманию того, что даже беспроблемные семьи не похожи друг на друга. Кантор разработал схему, позволяющую понять разнообразные и отличные друг от друга модели функционирования семьи. Его книга «Inside the Family» (Семья изнутри) (Kantor и Lehr, 1975 [45]) является одной из самых примечательных работ в области современных социологических исследований и считается классическим трудом.

Дэвид – хороший исследователь, однако я, как мне кажется, хороший теоретик, и поэтому я задался целью выяснить и восполнить некоторые пробелы в его теоретическом подходе. Результатом этого стало расшире-

ние упомянутой структуры с целью охватить все множество человеческих систем – от семей до неформальных групп, от проектных команд до международных организаций. В конце концов, я написал свою собственную книгу о «семейных парадигмах» (Constantine, 1986 [10]), и, к моему удивлению и удовольствию, ее стали использовать в учебных курсах по менеджменту и организационному развитию. В итоге возникло свободное международное сообщество, в котором объединились консультанты по вопросам управления, интересующиеся применением работы Дэвида и моих идей на практике. Некоторые из этих людей помогли мне понять, что в компьютерной области происходят удивительные вещи, и поэтому я должен туда вернуться, захватив с собой багаж знаний, накопленный при изучении семей и работе с ними.

11

Традиционная тактика

Ок, ребята, теперь давайте организовываться! Вопрос только – как? Работая самостоятельно, вы можете работать как угодно. Вам не нужно координировать свою работу с тем, что делают другие, и вам не придется подлаживаться к кому-либо. Вы можете в навязчиво-маниакальной манере аккуратно класть все вещи на свои места и в точности выполнять инструкции в соответствии с моделью жизненного цикла, принятой в разработке стандартного программного обеспечения. Или вы можете оставить все разбросанным по всему офису в полном беспорядке и кодировать по вдохновению или просто когда вздумается. Однако если вместе работают два человека или более, то ключевым словом здесь является слово «вместе». Работа, которую они выполняют, и способ ее выполнения требуют координации.

В этой главе начинается исследование вопросов организации и управления работой людей: как работа организована и как люди, работающие вместе, координируют свои действия (Constantine, 1990 [12], 1991 [15], 1993 [21]). Рассмотрим организацию как человеческий эквивалент архитектуры программного обеспечения, соотнося руководство с динамическим управлением программными компонентами. И снова мы придем к знакомым понятиям структуры и динамики систем управления. Когда вы хотите организовать новую компанию по разработке программного обеспечения или очередной программный проект, возникают все те же вопросы и проблемы.

Каким же образом их решать? Как можно наладить совместную работу группы людей – будь это проектная команда или целая корпорация? Как организовать группу, структурировать работу и управлять процессом?

Какой способ является правильным? Самое время вспомнить один из стандартных «ответов консультантов»: «Это как посмотреть!» Поиск *правильного* способа организации проекта похож на поиск *правильного* способа кодирования подпрограммы. Нужно исходить из того, что вы собираетесь получить в результате (Constantine, 1993 [21]).

Обычная работа по созданию еще одного набора драйверов для принтера или еще одного варианта традиционного генератора растровой развертки может потребовать совсем иной модели организации и руководства, чем работа над новейшим CASE-инструментом, поддерживающим процесс параллельного проектирования программного обеспечения и проектирование объектно-ориентированного ПО на основе консенсуса. В своей небольшой, но превосходной книге, посвященной командной работе, Ларсон и ЛаФасто (Larson и LaFasto, 1989 [47]) определили несколько основных вариантов такой работы, каждый из которых имеет свои преимущества. Здесь мы рассмотрим четыре отдельных и очень различных модели организации командной работы, их преимущества и недостатки.

Создание организации

Итак, давайте организовываться! Самый простой и надежный вариант основан на проверенной временем стандартной процедуре. Координирование работы более чем одного человека заключается в наделении кого-либо полномочиями руководителя. Функция руководителя состоит в том, чтобы управлять и следить за работой других. Эта структура может быть расширена с помощью рекурсии. В результате появится иерархия руководителей, где высшие чины руководят низшими. Это простая, устойчивая и знакомая всем форма организации: традиционная пирамида, основанная на иерархии власти. В проектах по разработке программного обеспечения, ведущихся согласно этой модели, в иерархии может не быть много уровней, но это все равно иерархия. В принципе, такая структура может быть очень эффективной, однако на практике она может разрастись до огромной бюрократической машины, не способной ни к чему, кроме сохранения собственной бюрократической неэффективности.

Такая модель является не просто способом работы. Она может быть образом жизни. Недавно я занимался реорганизацией своей коллекции записей, чтобы освободить место для новых компакт-дисков, и обнаружил настоящее сокровище – старинную запись «Paean» из корпоративного альбома компании IBM.[1] Слушая, как Ассоциация британских секретарей в

[1] Этот альбом был выпущен в качестве шутки по случаю большой компьютерной конференции, однако он дает верное представление о корпоративной культуре и политике IBM.

Америке (на самом деле!) энергично исполняет «Песню деревенского клуба IBM», я стал думать о корпоративной культуре и о том, как способ нашей работы определяет наше видение мира, и наоборот.

Традиционный взгляд с вершины пирамиды рассматривает организацию как основу устойчивого функционирования, а контроль – как ключ к поддержанию целостности. Лидерство опирается на власть. Руководители принимают решения, а подчиненные должны выполнять их, демонстрировать лояльность и следовать инструкциям начальства. Предсказуемое и надежное исполнение достигается за счет стандартов, процедур и правил работы. У каждого есть своя работа, своя роль, свои обязанности и свое место в иерархии. Корпоративные интересы, или интересы отдела, или интересы проекта стоят превыше всего, а работники получают вознаграждение за добросовестное исполнение своей части в большой работе.

В общем виде эта модель довольно проста. Ответственность возлагается на кого-либо, кто может принимать решения и отдавать приказы (желательно, чтобы этот человек разбирался в деталях и тонкостях дела). В традиционных иерархиях подразумевается, что решения принимаются строго сверху вниз. На то, чтобы необходимые данные поступили к ответственному лицу, может потребоваться время, но как только это произошло, процесс принятия решения может быть довольно эффективным. Решение принимается, когда руководитель готов его принять. Не требуется никаких консультаций. Не нужны ни обсуждения, ни дискуссии, ни исследования. Очевидно, что это может быть как силой, так и слабостью, поскольку многое начинает зависеть от того, кто находится на вершине пирамиды. Правильные решения могут приниматься так же быстро, как и неправильные, а если руководитель обдумывает решение слишком медленно или с ним трудно выйти на связь, то это может привести к краху всей пирамиды.

Властная пирамида

Когда речь идет о проектах по разработке программного обеспечения и командной работе, традиционная модель лучше всего подходит для того, что называется тактической работой. В тактических проектах территория знакома, а параметры известны. Самое главное – это выполнить работу. Я заточил свои «программистские зубы» на тактических проектах, в которых разрабатывались стандартные бизнес-приложения, например платежные ведомости, исчисление себестоимости и кадровые архивы. После того как вы сделаете пару систем ведения платежных ведомостей, они все начинают казаться одинаковыми. Даже если это не так, такой взгляд позволяет упростить работу и дает возможность применять стан-

дартные подходы. В итоге вы знаете каждый свой шаг и сможете сделать их даже во сне.

Ясность имеет решающее значение для тактической командной работы – ясность требований, ясность инструкций, ясность ролей. В этом мире методы разработки становятся более эффективными, если они хорошо определены и опираются на ясные стандарты и нормативы. При разработке программного обеспечения необходимо тщательно продумать жизненный цикл проекта и последовательность его этапов. Работа должна быть выполнена аккуратно и эффективно. В такой команде внимание группы акцентировано на текущей задаче. Точка. Члены команды больше всего нуждаются в ясных инструкциях от лидеров проекта и руководства. Каждому члену команды назначается конкретная часть хорошо понимаемой работы, и все выполняют то, что должны делать.

Традиционная иерархия работает. Многие компании в нашей отрасли и бесчисленные проекты по разработке программ организованы по этому принципу, в том числе и некоторые из наиболее успешных. Корпоративные песенники, восхваляющие бесстрашных лидеров компаний и их верных последователей, – лучшие тому свидетельства.

Такая ясно определенная и предсказуемая среда может быть комфортной, однако не каждый может чувствовать себя уютно в таком мире. Те, кому это удается, чаще всего лояльны, преданны и исполнительны. Они придают большее значение работе, чем самим себе, и всегда готовы к выполнению указаний лидеров. Такие люди предпочитают обращать большее внимание на программы, чем на людей. Тактические команды – это рай для конформистов и одержимых, для тех программистов, которые занимаются базовым программированием и о которых можно сказать, что они «соль земли». Тактическая командная работа подходит для тех, кто хочет знать, что делать, и делает это.

Обратной стороной медали является то, что любая группа, организованная по принципу традиционной иерархии, может очень легко застрять на месте. Сопротивляясь полезным инновациям в интересах устойчивости власти, такая группа тратит силы на строгое следование стандартам и процедурам, а не на решение задач и производство продукта. В лучшем случае такие группы эффективны и продуктивны, однако они не проявляют должной творческой активности. Их новаторство может выражаться не более чем в частичных улучшениях уже существующих технологий, то есть в эволюционных изменениях, а не в революционных. Если посмотреть на лидеров отрасли, полагающихся на традиционную иерархию, то зачастую можно увидеть, что их главные новшества были либо заимствованы извне, либо приобретены через внутренний обмен опытом.

Компании или группы разработчиков, организованные по этому принципу, вероятнее всего будут иметь особые трудности с кодирующими ковбоями (см. главы 7, 8 и 10). Независимые индивидуалисты и традиционная иерархия – это несовместимые понятия. Разработчики, которые сопротивляются власти или подвергают ее сомнению, а также стремятся идти своей дорогой или предпочитают отходить от стандартов, могут остаться без надбавки к зарплате или обнаружить себя в списке сокращения расходов.

С вершины пирамиды или изнутри может показаться, что другого способа организации и управления не существует. В конце концов, кто-то ведь должен быть ответственным, правильно? Либо вы руководите, либо выполняете указания, либо уходите с дороги. Некоторые люди до сих пор не видят никакой альтернативы иерархии. Но мы также когда-то думали, что вложенные подпрограммы, вызываемые из управляющей программы, – это единственный способ компоновки программ. Теперь мы используем множества связанных объектов и одноранговые сети с распределенными базами данных.

Что касается людей, то они всегда намного более гибки, чем программы. По крайней мере, *некоторые* люди.

Из журнала *Software Development*, том 1, № 3, март 1993 г.

12

Методы хаоса

Для всех кодирующих ковбоев у меня есть хорошие новости. Существует такой тип организации, который не только выносит настоящих индивидуалистов, но и опирается на их таланты.

Возможно, некоторые из программистов старой школы и бескомпромиссная молодежь будут шокированы, если узнают, что ни иерархия, ни власть не являются необходимыми для эффективного сотрудничества людей, работающих в одной группе. Действительно, сейчас в области управления происходит мировая революция. Все больше и больше компаний во многих отраслях открывают для себя преимущества самостоятельных команд, автономных рабочих групп и «управления без управляющих». Одна из главных проблем традиционной пирамиды управления заключается в том, что она очень плотна в середине (во многих смыслах этого выражения). Управление среднего уровня привносит дополнительные издержки – не только в денежном смысле, но и с точки зрения оперативности. Чем больше вы стремитесь быть на переднем крае в любой технологии, тем более неповоротливой становится «властная пирамида». Истинно новаторский подход требует такой стремительности, которая недостижима с помощью традиционной тактики вертикального управления.

Прорывы

Проектные команды скорее добьются настоящего успеха в развитии технологий, если обратятся к гибкости, которую дает независимость. Необходимо использовать всю силу самостоятельного творчества, не стесненного инструкциями и контролем. Суть состоит в том, чтобы извлечь все

лучшее из изобретательской энергии вольнодумцев, поощряя дух свободных исследований и личной инициативы. Нужно поддерживать некоторый творческий беспорядок, граничащий с хаосом, – своего рода контролируемое безумие, которое позволяет отойти от шаблонов, ограничений и расширить представления о собственных возможностях. Все это противоречит закрытым шаблонам корпоративных пирамид, поскольку подрывает основы власти и выбрасывает традиционный подход на ветер. Это серьезно угрожает традиционным методам управления, однако это как раз то, что нужно для освоения новых земель.

По существу, ключевые ингредиенты для инициативы прорыва несовместимы с традиционной иерархической моделью. Вместо стабильности, превалирующей над изменениями, поощряется нестабильность как движущая сила для преодоления слепо принимаемых правил и не подвергаемых сомнению понятий. Вместо корпоративных и коллективных интересов, главенствующих над индивидуальными, на первое место ставится индивидуальная свобода самовыражения и действия. Если традиционная пирамида старается приструнить строптивых кодирующих ковбоев, то «команды прорыва» охотно их принимают и дают им возможность скакать свободно. Такая атмосфера общей свободы стимулирует творчество и способствует «наивысшей личной» производительности, приводящей к крупным достижениям.

Деятельность «команд прорыва» зависит от индивидуальной инициативы. Решения принимаются не централизованно, а самостоятельно, на рабочем месте – теми, кто обнаружил проблему и знает, как ее решить. Держать курс такой группе помогает своего рода дружеская соревновательность. Именно общий интерес и любовь к игре, а также взаимное уважение игроков друг к другу удерживает их от того, чтобы разбежаться в разные стороны.

Чаще всего эту модель можно обнаружить в небольших высокотехнологичных компаниях, недавно возникших предприятиях и в проектно-исследовательских отделах больших организаций. Все они могут быть довольно успешными. Действительно, многие мускулистые корпоративные бегемоты полагаются на «отделы скунсов».[1] Именно оттуда исходят новые идеи и продукты, позволяющие сохранять мотор коллектива в работоспособном состоянии.

[1] От англ. «skunk works» – маленький, часто изолированный исследовательский отдел какого-либо предприятия, функционирующий полусамостоятельно, практически без контроля начальства. – *Примеч. перев.*

Работа и игра

Представьте такую сцену, действительно имевшую место в одной из крупных компаний западного побережья. Вы входите в вестибюль и видите, как бородатый человек средних лет, который потом оказывается вице-президентом научно-исследовательского отдела, отвечает на телефонные звонки. Вы говорите ему, что вы консультант по графическим пользовательским интерфейсам, и в ответ он машет вам рукой – «по коридору налево...». Едва уклонившись от «летающей тарелки», пронесшейся на околозвуковой скорости, вы отправляетесь на поиски какого-нибудь начальника. Безуспешно. Увидев мерцающий экран монитора в одной из комнат, вы подходите к сотруднику, занятому просматриванием библиотеки классов. Этот человек оказывается офис-менеджером. Но разговор с ним становится все более трудным, так как игра в «летающую тарелку», происходящая в коридоре, набирает обороты. Два программиста, занятые отладкой кода с целью устранения ошибки в операционной системе, орут, чтобы было потише, но, в конце концов, сами втягиваются в состязание. Через некоторое время весь отдел переходит на соседнюю парковку и устраивает там целый турнир, в котором участвуют пять «летающих тарелок».

Для случайного наблюдателя все это может показаться бедламом или детским садом, однако игра «в тарелку», в которой множество «летающих тарелок» не сталкивались друг с другом, натолкнула пару программистов на неожиданные идеи. Они придумали новое решение одной сложной проблемы в многопользовательском программном продукте, разработкой которого сейчас занимается их отдел. Была ли это простая случайность? Было ли это простым валянием дурака на работе? Трудно сказать. Работа – это игра, а игра – это работа. Был ли встреченный офис-менеджер инженером по разработке программного обеспечения? Вы никогда не догадаетесь.

А как насчет вице-президента научно-исследовательского отдела? Он что, входит в состав службы поддержки? Наверное, это так, раз он с этим так хорошо справляется. Руководители, которые эффективно управляют таким творческим хаосом, знают, что их задача заключается в том, чтобы обеспечивать ресурсы и поддержку, ограждать группу от внешних помех, а самим оставаться в стороне. Самые лучшие из таких менеджеров чаще всего сами являются технарями и пользуются особым уважением за свою удаль в программировании или другие технические таланты.

Конечно, такая модель не всегда работает, и даже когда она работает, оригинальность может иметь и обратную сторону. Чрезвычайно трудно вести собрание по планированию или разбор кода, если сотрудники постоянно заходят и выходят, а те, кто сидят в комнате, заняты написанием писем на лэптопе или играют в шахматы на карманной доске. В лучшем

случае коммуникации внутри «команды прорыва» могут быть случайными, даже если все находятся в одной комнате. Дело не в том, что эти творческие индивидуалисты необщительны – просто в таком хаосе информация очень легко пропадает. («Да-да, этот клиент на прошлой неделе писал нам об изменении в интерфейсном протоколе. Это письмо где-то сохранили. Наверное.»)

Блестящий и совершенно беспристрастный психолог Дэвид Кантор (David Kantor) одним из первых заметил, что в человеческих группах под внешней случайностью скрывается лежащая в ее основе сложно организованная логика (Kantor и Lehr, 1975 [45]). В этом отношении его работы предвосхитили последние исследования по применению теории хаоса в группах. Кантор выделил два варианта кажущейся случайности: одну он назвал творческой, а другую – хаотичной (в старом и более традиционном понимании слова «хаос»). Разница между «случайным творческим» и «случайным хаотичным» имеет решающее значение – это разница между успехом и провалом в достижении прорыва при разработке программного обеспечения.

В крайних случаях «команда прорыва» может стать «командой *провала*». Без наличия правильных ингредиентов дружеская соревновательность может стать недружеской, непродуктивной или даже бесперспективной. Специалисты, которые должны были сотрудничать, стремятся к противоположным целям и борются за ресурсы. В конце концов, люди могут полностью выйти из-под контроля, оставив организацию и проект в полной неразберихе.

Первый необходимый элемент, позволяющий избежать хаотического провала, – это хорошие люди с хорошей подготовкой, владеющие хорошими приемами. Члены команды должны обладать способностями и навыками, которые отвечают уровню сложности проекта и признаются другими членами команды. Взаимное уважение к уровню технической компетентности и полная вера в способность других членов команды внести свой вклад в общее дело являются важнейшими связующими звеньями, которые сохраняют единство «команды прорыва».

Вторым необходимым элементом, ведущим к успеху, являются достаточные и даже избыточные ресурсы. «Броуновское движение» в команде новаторов может способствовать появлению блестящих решений. Однако такие решения не всегда являются самыми эффективными. Новые алгоритмы следует испытывать, нестандартные структуры данных нужно применять, а оригинальные варианты разбивки экрана должны быть макетированы. Процесс свободного создания и проверки новых идей имеет важнейшее значение. Даже тупиковые идеи, от которых следует отказаться, часто являются важной частью процесса группового обучения. Творчество имеет свою цену и требует определенного риска.

Против управления

Как можно управлять кучей индивидуалистов-изобретателей? Не нужно заниматься руководством. Управление «командой прорыва» – это особая роль, которая принципиально отличается от руководства традиционной тактической командой. Самые эффективные лидеры команд и проектов считаются равными среди равных и пользуются большим уважением как программисты и специалисты. Они является харизматическими лидерами, на которых равняются остальные. Чаще всего такие лидеры руководят своим личным примером, а не приказом, которому почти всегда будут сопротивляться. Они способны создать атмосферу глубокого взаимного уважения членов команды друг к другу. Они знают, как обеспечить команду необходимыми ресурсами – рабочими станциями, программным обеспечением, учебными курсами, временем. Они поддерживают заряженность команды и предотвращают неконструктивную конкуренцию за ограниченные ресурсы.

Для достижения успеха такие команды также нуждаются в автономности. Должна быть обеспечена свобода от вмешательства извне, свобода исследовать неожиданные повороты и подходы. Хорошие лидеры таких команд говорят: «Итак, игра начинается!», а затем отходят в сторонку. А самые лучшие лидеры еще и следят, чтобы никто другой не смог этой игре помешать.

Проектная команда, организованная и направляемая на достижение прорыва, может стать веселой и интересной компанией для работы, однако не каждый сможет хорошо работать в таких условиях. Люди, которые нуждаются в ясных указаниях, четко поставленных целях, а также в ясном понимании того, что от них ожидают, скорее всего, будут лучше себя чувствовать в традиционной иерархии. Разработчики, которые продуктивнее всего работают в «командах прорыва», могут быть артистичны либо быть интеллектуалами, но прежде всего они проявляют себя как свободомыслящие люди. Они активны, они не ожидают инструкций. Более того, самые производительные из них могут быть не в меру упрямы. Хорошими членами тактических команд часто становятся люди, которые особенно чутко реагируют на указания лидеров, но хорошие новаторы в большей степени являются индивидуалистами или даже могут сопротивляться власти и инструкциям.

Вряд ли есть смысл спускать с цепи целую команду творчески мыслящих ковбоев для работы над обычной базой данных, необходимой для бухгалтерского отдела. Полученный продукт может привести к бегству бедных бухгалтеров. С другой стороны, масштабные проекты со множеством компонентов и сложными взаимосвязями – например, программное обеспечение для научно-исследовательской космической станции – также не

подходят для «команд прорыва», даже когда ясно, что новые подходы необходимы. По очевидным причинам команды, состоящие из творческих индивидуалистов, больше подходят для менее масштабных и сложных проектов, которые не требуют согласованности множества частей, в сильной степени зависящих друг от друга.

Во многих сложных современных проектах по разработке программного обеспечения следует применять другие подходы, которые будут рассмотрены далее. Что касается компаний типа Nanomush Corporation и International Behemoth Management, Inc., то им придется плутать со своими хаотичными ковбоями среди мегапирамид.

Из журнала *Software Development*, том 1, № 5, май 1993 г.

13

Открытая архитектура

Многие думают, что есть только два типа людей: мы, знающие все лучше других, и остальные, отличные от нас. То же самое можно сказать о работе в организациях. Некоторые люди думают, что есть только два выбора: приказная власть или необузданная анархия. Другими словами, свобода или подавление – и все! Остальные понимают, что не все так просто. Что бы ни предпринимали менеджеры и консультанты по управлению, реальные люди в реальных организациях не поддаются ограничениям примитивной дихотомии и выпрыгивают из одномерных концептуальных коробок, в которые мы стараемся их втиснуть.

С одной стороны, есть иерархическая модель организации работы, обсужденная в главе 11, а с другой стороны – ее полная противоположность, творческая анархия, рассмотренная в главе 12. Между этими двумя крайностями в виде упорядоченного традиционализма и свободного новаторства размещается столько вариантов, сколько существует проектов и людей, в них участвующих.

Впрочем, не всякой рабочей группе найдется место в данном ряду. Некоторые группы, находящиеся далеко в левой части диапазона, не рассматривают основные вопросы организации работы как компромиссы. Это те люди, которые мыслят в терминах «и то, и это», а не в терминах «или-или». Они говорят: «Мы можем это». Как в традиционном, так и в «свободном» подходе есть тенденция обязательно противопоставлять «я» и «мы». В традиционной иерархии интересы индивидуумов подчинены интересам команды, или группы, или организации. В новаторском индивидуализме верховодят индивидуумы, а коллективные интересы отодвинуты на задний план.

Однако другие исследователи считают, что между целым и его частями нет принципиального конфликта – так же как нет необходимости выбирать между изменением и стабильностью. Такой подход к совместной работе, так называемая «открытая парадигма», является гибкой моделью, основанной на равноправном сотрудничестве и взаимодействии. Для краткости мы можем назвать эту модель «адаптивным сотрудничеством», открытой архитектурой построения человеческих организаций.

Работаем спокойно, работаем вместе

Адаптивное сотрудничество идеально подходит для решения технических задач. В этом подходе сами по себе не ценятся ни традиция и стабильность, ни новаторство и изменение. При таком взгляде на проекты и прогресс важным становится адаптивное соответствие между тем, как команда работает, и тем, над чем она работает. Сегодня мы разрабатываем независимые друг от друга программы, поэтому мы работаем по отдельности. Завтра мы будем думать над общим протоколом обмена данными, поэтому мы соберемся все вместе. У каждого есть свои мысли об архитектуре базы данных, так давайте обсудим все эти идеи.

Целью в такой группе является спокойствие и обсуждение вопросов таким образом, чтобы конкурирующие идеи можно было объединить, а различные подходы синтезировать. В некотором смысле такие группы непрерывно перестраиваются, изменяя способы работы в соответствии с текущими потребностями и долгосрочными задачами. Кто становится «ответственным» и что это подразумевает, зависит от того, чем занята группа в данный момент.

Группы разработчиков, организованные в таком духе, напоминают скорее плоский круг, чем пирамиду. Члены команды работают как коллеги, меняясь ролями. В отличие от свободных радикалов в «командах прорыва», которые могут работать независимо и даже соревнуются между собой, члены такой команды тщательно согласовывают свои действия. Решения принимаются коллективно в процессе обсуждений, переговоров и построения консенсуса. Это не значит, что все во всем соглашаются, однако они приходят к техническому консенсусу. При техническом консенсусе все члены команды поддерживают действия и решения, выработанные группой по всем главным вопросам. Для этого требуется, чтобы каждый мог внести свой вклад в важные решения. Такой подход позволяет каждому сотруднику «вложиться» в совместное усилие и гордиться своим личным участием в коллективном продукте.

Обсуждая вопросы совместно, группы могут блестяще решать сложные задачи, особенно в тех случаях, когда нужно получить и проверить значительный объем информации (звучит так, как будто требуется большой

объем разработок). На самом деле, чем больше в задаче факторов, граней и деталей, тем больше относительное преимущество открытых команд в сравнении с тактическими командами, где информация может контролироваться слишком жестко, или с «командами прорыва», в которых информация может быть потеряна в конкуренции и хаосе.

«Команды прорыва», отдающие предпочтение оригинальности свободного мышления, могут также приходить к блестящим решениям, которые, тем не менее, не вполне практичны. Невозмутимые представители тактических команд могут производить надежные рутинные программы, но с трудом находят творческие решения для структур данных, алгоритмов, или пользовательских интерфейсов. Команды, в которых предпочтение отдается совместному решению задач, лучше всех способны объединять разные идеи, предлагаемые всеми членами команды, и принимать такие решения, которые одновременно и практичны, и прогрессивны.

Для настоящего сотрудничества требуются правильные методы руководства, которые поощряли бы свободное обсуждение и помогали достигать технического консенсуса. Самые лучшие менеджеры в проектах открытого типа являются коллегами, компетентными профессионалами, которые играют активную техническую роль в процессе анализа, проектирования и разработки. Обычно они не ведут собрания или обсуждения технических вопросов, поскольку не хотят оказывать влияния на результат. Вместо этого они предлагают другим участвовать в обсуждении, чтобы получить от каждого члена команды наибольшую отдачу. Это требует определенной веры, убежденности в том, что группа как целое знает больше и может достичь лучшего решения, чем кто-либо в отдельности (в том числе и начальник!). Руководители, которые уверены в том, что они самые умные, самые способные и могут разметить программу лучше, чем кто-либо другой в проекте, скорее всего, не смогут достигать такого же успеха, как сотрудничающие друг с другом коллеги. Будет лучше, если они станут работать в одиночку или руководить традиционной тактической командой.

Естественно, упор на полное согласие при адаптивном сотрудничестве может быть и препятствием. Эти люди способны уговорить даже ураган. Если обсуждение, которое продлилось час, не помогло решить вопрос, они будут готовы потратить на это еще пару часов. Они пререкаются не только по поводу технических проблем, но и по поводу философии, которая за ними стоит; они подвергают сомнению не только методы разработки, но и те предположения, на которых они основаны. Может обсуждаться даже работа самой группы, когда она рассматривает свои методы и структуры и пытается адаптировать их к решению задачи. («Послушайте, может быть, сейчас нам нужно разбиться на небольшие группы и поработать так

несколько дней, а потом вновь соединиться и попытаться собрать все это вместе?»)

Есть способы уменьшить пробуксовку в работе такого рода команд. Дискуссии могут быть ограничены по времени. Если технический консенсус не достигнут по истечении какого-то установленного времени, то вопрос может быть отложен или сведен к некоторому стандартному решению, или передан руководителю для арбитражного вердикта. Каждый из этих вариантов в некоторой степени нарушает правила совместного принятия решений. Однако если такие варианты являются нечастыми исключениями из правил, то эффективность может перекрыть некоторую потерю в виде чувства принадлежности к конечному результату.

Однако здесь не может быть простой формулы. Для того чтобы оставаться открытой, каждая группа должна искать свой собственный путь и договариваться о собственных процедурах и исключениях.

Оставляя дверь открытой

Одна из лучших команд, построенных на сотрудничестве, которые я когда-либо знал, была известна как Цех построения теорий (Theory Construction Workshop). Это независимая рабочая группа, слабо связанная с профессиональным сообществом. Ее члены общались друг с другом как коллеги, стараясь продвигать создание теорий в атмосфере свободного обсуждения и взаимопомощи. Эти собрания были открыты для всякого, кто разделял интерес к построению более правильной теории. Содержание и формат собраний обсуждался в соответствии с традицией тесного сотрудничества и взаимодействия. Почти на каждом ежегодном деловом собрании обязательно находился новичок, который в порыве радости от присутствия в такой замечательной и приятной компании предлагал ввести какой-нибудь формальный порядок в виде регистрации, членских билетов, устава, публикации протоколов и т. п. Каждый год мы исправно рассматривали и обсуждали эти предложения и каждый год их отклоняли, по общему согласию оставляя наше собрание открытым, неформальным и гибким.

Однажды на одном из ежегодных собраний я, не подумав, выдвинул радикальное предложение: «Отныне, для того чтобы избежать траты времени на обсуждение этих бесконечных предложений, давайте запретим все попытки установить жесткие и формальные правила или структуры». Еще не успев закончить своей фразы, я рассмеялся над своим внутренне противоречивым предложением. Наше искреннее и серьезное рассмотрение даже самых необдуманных предложений, вносимых рядовыми новичками, было именно тем, что заставляло групповой процесс работать. Все заулыбались, я сел, и мы вернулись к обсуждению очередного пред-

ложения по публикации протоколов наших собраний. Для того чтобы оставаться открытым, любой открытый процесс должен быть открыт для альтернативных вариантов.

Свою блестящую пьесу «Sunday in the Park with George» (В парке с Джорджем в воскресенье) Стивен Сондхейм (Steven Sondheim) заканчивает чудесными словами, приписываемыми Джорджу Сера (Georges Seurat): «Белая, чистая страница или холст – вот что он любит; в них есть так много возможностей».

Могу сказать то же самое.

Из журнала *Software Development*, том 1, № 6, июнь 1993 г.

14

Синхронное плавание

Хотите увидеть, как по-настоящему быстро разрабатывать приложение? Возьмите фильм Гаррисона Форда (Harrison Ford) «Свидетель» («Witness») и найдите в нем сцену, когда общество Амиш строит сарай за один день. Самое интересное в этой проектной команде – не ее поразительная эффективность, а отсутствие руководителя и даже необходимости в руководителях или руководстве. Они почти не разговаривают и им уж точно не приходится спорить по поводу конструкции сарая или договариваться о том, кто и что должен делать для того, чтобы его построить. Они просто взялись за работу и стали ее выполнять быстро и эффективно, почти в полной гармонии, пока не закончили.

Конечно, это мечта менеджера, своего рода управленческая утопия. Слово «утопия» буквально означает «место, которого нет», а сцена из фильма «Свидетель» исправно происходит в округе Ланкастер, штат Пенсильвания. Более подходящим, наверное, является слово «эутопия», означающее «хорошее место». Таким образом, эутопия – это то идеальное место, где вы хотите быть. Поэтому мы можем назвать такую модель организации *эутопической командной работой*.

Быть руководителем эутопической группы, разрабатывающей программное обеспечение, очень просто – настолько просто, что для осуществления руководства вам вряд ли придется что-либо делать. Люди вокруг вас делают все, что должно быть сделано. Не нужно стоять над душой у каждого или разъяснять что-либо на пальцах. Все идет гладко потому, что люди, которые работают для вас и вместе с вами, разделяют ваши представления о работе команды – о том, что нужно делать и каким образом. Это люди, которые вам подходят; они думают так же, как и вы, и во многом смотрят

на мир вашими глазами. Едва вы успеваете сказать слово, как они уже принимаются разрабатывать следующий набор модулей или кодировать расширения для графического пользовательского интерфейса, или делать еще что-нибудь нужное. Все – люди, программное обеспечение, программирование – подходит друг другу так точно, как будто вы все придумали сами, не прилагая к этому никаких усилий. В команде не существует конфликтов и не нарушается дисциплина. Подобно команде по синхронному плаванию, это само воплощение идеального совершенства и гармонии.

Такая эутопическая фантазия является последней из наших четырех основных моделей организации проектов – *синхронная парадигма*, модель эутопической командной работы. Традиционные тактические команды координируются иерархической властью, «команды прорыва» – индивидуальной инициативой, а команды совместного решения задач – взаимодействием сотрудников. Эутопические команды зависят от выбора направления. Члены такой команды следуют направлению, заданному общим пониманием и общими ценностями. Поскольку все члены команды понимают друг друга и разделяют общее представление о том, куда движется команда и каким образом она будет достигать своих целей, они могут работать совместно без какой-либо заметной координации. Подобная направленность обычно не становится доминирующим объединяющим принципом для всей организации в течение длительного периода времени. Однако многие группы, большие и маленькие, зависят от некоторой степени направленности или функционируют синхронно в течение более коротких периодов.

Где находится место, которого нет

Случай, который произошел с моим коллегой несколько лет назад, поможет проиллюстрировать этот режим работы. Консультант по организационному развитию встречалась с клиентами в конференц-зале госпиталя. Во время этой встречи одна из ножек массивного дубового стола подломилась. Стол придавил ее ногу. Нога оказалась вывернута, а движения скованы. В то же мгновение все вскочили со своих мест, чтобы помочь. Двое стали приподнимать тяжелый стол с ее ноги, а третий взял за руку, чтобы успокоить. Кто-то побежал в рентгенологию. Еще один выскочил в холл, чтобы принести носилки. Между членами этой медицинской команды не проскочило ни единого слова. Они все точно знали, что нужно делать, и каждый просто стал выполнять какую-то часть. Никто не назначался руководителем, не требовались ни обсуждения, ни переговоры, и, тем не менее, все занимались одним делом, не противореча друг другу.

В подобном синхронном взаимодействии нет ничего магического. Если посмотреть на этих людей внимательно, то можно заметить, что взаимодействие между ними главным образом невербальное. В существенной степени оно основано на молчаливом согласии и предварительном зна-

нии. Так как каждый член команды видит, что делают другие, он может подстраивать свои действия под общее дело. Поэтому не все сразу бросаются к телефону, а носилки появляются в нужный момент.

При условии, что все члены эутопической команды хорошо понимают стоящую задачу, такая команда может достигать высокой эффективности в сложных или критических условиях (подобно ситуации в конференц-зале госпиталя) или при выполнении предсказуемых и ясных задач в течение более длительных периодов (как, например, построение сарая). Понимание задачи и наличие представления о том, как она может быть выполнена лучше всего, имеет важнейшее значение. Каждый работник в обществе Амиш видел, как строят сараи, и, вероятно, когда-нибудь участвовал в этом. Каждый, кто находился в конференц-зале госпиталя, имел более чем достаточный опыт в оказании медицинской помощи. Почти наверняка никому из них не приходилось иметь дело с падающими столами переговоров, но это не имело значения; их профессиональный медицинский опыт и знания позволили им быстро понять новую ситуацию и действовать как обычно.

Ключом к эффективности эутопических команд является полная приверженность всех членов команды довольно сложной, но четко определенной концепции того, что является задачей и методами группы. Если эта направленность является слабой или частичной или концепция является неадекватной, группа не сможет выполнять свою работу без консультаций или без конфликтов или же не сможет отвечать требованиям изменяющихся условий; в этом случае либо потребуется более жесткое управление, либо трудности команды станут возрастать.

Люди, изучающие менеджмент и организации, хорошо понимают первые три из наших основных моделей команд и лежащих в их основе механизмов. В то же время о проектных командах, основанных на направленности, известно меньше. Хотя полностью синхронные команды на практике встречаются нечасто, сочетание высокой синхронизации с традиционной иерархией является менее редким. Такое сочетание можно встретить в крепких компаниях, которые работают в давно существующих отраслях, имеющих давние традиции. Консультант Роб Томсет (Rob Thomsett) и я обнаружили ряд довольно синхронных групп по разработке программного обеспечения в австралийском банковском доме, функционирующем по британской модели. В Японии, где корпоративный мир свято чтит традиции преемственности и единообразия, даже высокотехнологичные фирмы могут широко применять синхронную направленность.

Тихие воды

Эутопическая командная работа является полной противоположностью модели, основанной на сотрудничестве. Вместо обсуждений и перегово-

ров нормой здесь является их отсутствие. Зачем нужны переговоры, если люди думают настолько одинаково, что почти с самого начала согласны друг с другом? Обратной стороной такой эутопической гармонии является то, что в обычной, повседневной работе все это безоблачное и спокойное сотрудничество может стать несколько скучным. Поскольку члены такой команды привыкли работать без особых дискуссий или вообще без обсуждений, они могут не общаться даже тогда, когда это им нужно. Если условия на рынке или базовая технология неожиданно или радикально изменяются, команда может оказаться неспособной среагировать или адаптироваться к этому так же хорошо, как группы, построенные на индивидуальной инициативе или на совместном взаимодействии. В худшем случае члены команды могут продолжить действовать привычным способом и не обращать внимание на изменяющийся вокруг них мир. Если что-то не соответствует их общим представлениям, они могут посчитать это не заслуживающим внимания или ответа.

Согласно общепринятым представлениям, современные руководители должны учиться выживать в бурлящей воде, плавать в компании с акулами и при этом добиваться успеха. Выжить, плавая в окружении акул, конечно, возможно, но большинство пловцов, наверное, предпочтет теплые, тихие воды. Естественно, команде по синхронному плаванию лучше держаться в стороне от бурлящих вод и подальше от водоемов, населенных акулами. Точно так же и эутопические команды лучше приспособлены к стабильным, постоянным и невраждебным условиям.

Однако небольшая степень синхронности может быть полезной даже для команд, построенных на других базовых моделях. Более узкая направленность и более четкое взаимопонимание уменьшают потребность в жестком контроле и расширяют границы, в которых усилия одних членов команды могут подкрепить или поддержать усилия других, а не свести их на нет или помешать им.

Эффективные руководители эутопических команд являются харизматическими гуру, способными формировать и применять сложные концепции, а также вызывать к ним доверие других людей. Для длительных проектов особую важность имеет их способность изменять и расширять эти концепции для того, чтобы учесть новые требования и изменить направленность в соответствии с ними.

Естественно, вы понимаете, о чем идет речь. Мы хорошо понимаем друг друга, верно? Поэтому здесь нечего больше говорить (даже то, что эта эутопическая направленность просто отлична!).

Из журнала *Software Development*, том 1, № 17, июль 1993 г.

15

Командная политика

Этот проект по разработке программного обеспечения был чрезвычайно успешным. Команда разработчиков приобрела известность созданием потрясающей системы с усовершенствованными сервисами и отличным графическим интерфейсом. По каким-то причинам руководство отказалось от этого продукта.

Команда – не остров. Группы, которые хорошо работают вместе в комнате для обсуждений, могут не иметь успеха при выходе на уровень корпоративной политики. Неудача есть неудача. Знать интерфейс программирования приложений и библиотеку классов так же недостаточно, как недостаточно знать способы прихода к консенсусу или процессы параллельного проектирования. Если вы не знаете правила игры, вы проигрываете. Название этой игры – «внешняя среда».

Команды по разработке программного обеспечения должны сторожить свои границы, защищая свою территорию. Однако нужно и наводить мосты. Новый компилятор является частью набора инструментов. Система поддержки принятия решений должна согласовываться с системой бухгалтерского учета, но также она должна быть пригодной и полезной для руководителей. Репутация компании неуправляемых приверед может способствовать попаданию команды в список на сокращение. С другой стороны, репутация супергероев C++ может привести к необоснованным ожиданиям при создании очередной объектно-ориентированной чепухи, начатой зятем президента компании.

Мы рассматриваем командную деятельность по созданию программного обеспечения с точки зрения моделей, определяющих стиль внутренней ра-

боты. В свою очередь, Дебора Анкона (Deborah Ancona) из школы управления Sloan при Массачусетском технологическом институте исследует функционирование команды в реальной корпоративной среде, а также то, как внешние стратегии и стили команд влияют на производительность (Ancona и Caldwell, 1992 [1]). Анкона изучала консалтинговые команды, команды разработки новых продуктов, а также команды по сбыту товаров и управленческие команды. Данные, полученные ею в течение многих лет, совпадают с моим опытом изучения команд по программированию.

Сквозь измерения

Внешние стратегии команд – это сложная тема. Лидер команды или менеджер проекта может играть ключевую роль в урегулировании внешних отношений, однако эта тема сводится не только к тому, как ваш начальник ладит с ее начальником. Команды должны взаимодействовать и сотрудничать со множеством других подразделений организации. Эти взаимодействия происходят в нескольких измерениях: во властной структуре, в структуре задач и в информационной структуре.

Представим «властное» измерение как вертикаль. Важные внешние связи в этом измерении направлены вверх. Немногие команды могут достичь действительно высокой производительности, если они не умеют «ладить с руководством». Командам нужны «послы», политики, которые знают, как играть в политические игры в организации и как работать со властной структурой, чтобы команда и проект оказались в выгодном положении. Они способствуют хорошей репутации группы, представляя саму группу и ее преимущества. Связи с общественностью имеют большое значение для успеха команды, поскольку репутация команды может стать самоисполняющимся пророчеством. «Хорошие» команды получают самые лучшие проекты и первоочередной доступ к новому программному обеспечению и оборудованию. Часть успеха может быть связана с тем, что команда берется только за те задачи, которые ей по силам, а скучные или невыполнимые оставляет в стороне.

Вероятно, самой важной политической проблемой для команд является необходимость обрести и сохранить эффективную поддержку со стороны высшего руководства. Высокопоставленный руководитель или покровитель может сделать для команды больше, чем все CASE-инструменты и рабочие станции в Силиконовой долине. Энергичные политики с хорошими связями могут также служить буфером, ограждая команду от переменных ветров влияния и интересов, когда какие-то части компании продаются или поглощаются или когда руководители средней руки идут на повышение, или непрерывно меняются начальники отдела информатизации.

В основном координация задач проходит в горизонтальной плоскости с учетом боковых соединений, которые определяют взаимосвязи данной команды с другими организационными единицами. Хорошие координаторы договариваются с другими группами, выторговывая услуги или важные ресурсы. Они получают информацию о прогрессе других участников проекта, чьи продукты будут применяться совместно с продуктом, разрабатываемым командой. Они обеспечивают поступление работы и ее сдачу, отвечают за наличие соответствующих библиотек компонентов, направляют спецификации людям, разрабатывающим тестовые комплекты, или исправляют графические интерфейсы совместно с группой изучения человеческого фактора.

Информационное измерение тоже, главным образом, горизонтальное. Взаимодействие здесь подразумевает исследования, сбор информации, необходимой для успеха проекта, а также обмен данными между отделами. Командные исследователи могут эффективно служить в качестве информационных привратников, отбирая информацию таким образом, чтобы другим разработчикам не приходилось перекапывать горы документации, и отыскивая недостающие и необходимые данные.

В таких командах развиваются свои стили внутренней работы и специализация в разных областях. И точно так же в них разрабатываются характерные методы управления собственными границами и взаимодействия с остальной частью организации. Среди команд, исследованных Анконой, выделялось четыре варианта, которые мы назовем «политиками», «исследователями», «изоляционистами» и «универсалами».

«Политики» специализировались на «работе по вертикали», сосредотачивая свои усилия на создании хороших отношений с вышестоящим начальством. «Исследователи» занимались поиском и сбором информации. «Изоляционисты», в свою очередь, старались не выходить за свои плотно закрытые границы, защищая и охраняя их. Они не имели хороших связей с начальством, не полностью представляли задачи или владели не всей информацией.

«Универсалы» специализировались на всем. Они проявляли себя как команды, хорошо интегрированные в свою организацию благодаря эффективной управленческой деятельности. Они подключались к информационной структуре в ходе «исследовательской» и «поисковой» деятельности. Они сотрудничали с другими командами и ответственными лицами через инфраструктуру. У них были политические связи и защита со стороны властной структуры.

Как эти команды разных типов преуспевали на практике? Эффективность команд можно рассматривать как изнутри, так и снаружи, как с точки зрения членов команды, так и с точки зрения всей организации.

Члены «политических» и «изоляционистских» команд считали, что они самые лучшие, хотя для высшего руководства картина выглядела несколько иначе – при первоначальных оценках оно было склонно считать, что «политики» и «универсалы» эффективны больше всех, так как лучше других связаны с властной структурой.

Окончательный счет

Спустя время проявилось следующее. Согласно проведенному через полтора года исследованию, «политики» растеряли свое преимущество, получив самые низкие оценки производительности. Как оказалось, эти команды говорили разумно, но не достигали результата (что, впрочем, похоже на политиков вообще, не правда ли?).

Команды «исследователей», которые порой не занимались ничем, кроме сбора информации, зачастую оказывались расформированными по решению руководства. Команды «изоляционистов» проявляли себя неодинаково. Большинство из этих замкнутых групп приходили к полному провалу, однако некоторые из них достигали ощутимого успеха. Изоляция вашей команды первоклассных кракеров от остальной части компании может оказаться неплохим способом концентрации усилий на конечном продукте. Это один из тех шагов, которые связаны с высоким риском, но могут принести большую пользу.

Команды «универсалов», использующие гармоничные и разнообразные внешние стратегии, оказались корпоративными победителями. Такие команды способны уравновешивать внутреннюю производительность и внешние требования. Они получают нужную им информацию, но не застревают в бесконечных исследованиях. Для достижения своих целей они работают с системой через структуру власти и инфраструктуру. Другими словами, высокопроизводительная командная работа – это больше, чем умение хорошо работать вместе. Это также умение хорошо работать с другими.

Поэтому спрашивайте не о том, по ком звучит гонг. Если вам приходится спрашивать, ваша команда имеет неэффективную внешнюю стратегию. Значит, гонг звучит по вам.

Из журнала *Software Development*, том 1, № 8, август 1993 г.

16

Все сразу

Ни одна команда не может подходить для всех проектов. Одни группы больше подходят для рутинных разработок, другие превосходно разрабатывают самые сложные приложения, а третьи лучше всего осваивают новые области. Отчасти это зависит от того, как команда организована и скоординирована. Закрепите за членами команды постоянные обязанности и начните руководить «сверху вниз», применяя методы жесткого контроля и постоянного надзора, и, скорее всего, вы вряд ли получите какие-нибудь новые идеи. Свободно действующие команды, в которых поощряется индивидуальная инициатива, в большей степени способны нанести на карту новые территории. Традиционные команды с закрепленными ролями больше подходят для разработки обычных приложений. Команды, которые применяют открытое обсуждение и приходят к консенсусу, лучше справляются с решением сложных задач. В зависимости от сути задачи, с которой вы столкнулись, и технических целей проекта, тот или иной вид организации командной работы повышает шансы на достижение успеха.

Но все это вы и так знали. К сожалению, работа, которой вы занимаетесь, не очень хорошо вписывается в эти стандартные варианты. Ваши программные проекты не являются беспрецедентными, но, в то же время, и не совсем обычны. Задачи, которыми вы занимаетесь, – сложные и многоплановые, однако для того, чтобы решить их вовремя и в соответствии с поставленными условиями, высокий уровень производительности и надежности должен сочетаться с некоторой долей новаторства. Возможно, вы даже задумываетесь о применении элементов творческого сотрудничества, но начальник не понимает этих чувствительных и легкоранимых работников и поэтому собирается назначить ответственным вас и только вас.

Так обычно и происходит в мире, по крайней мере, в мире разработки программного обеспечения. Ни одна из моделей командной работы, представленных в главах 11–14, не отвечает всем требованиям типичных проектов. Необходима та или иная комбинация этих моделей, подходящая под непростые требования реальности.

Модели управления

В действительности реальные группы программистов прибегают к огромному множеству смешанных моделей. Однако, хотя некоторые из них могут быть более удачными, чем другие, большинство из них либо является мешаниной, собранной по методу проб и ошибок, либо основано на моделях управления, заимствованных из других отраслей. Мы хотим, чтобы модель командной работы над проектом подходила для разработки программного обеспечения. В такой модели предсказуемость принятой методологии и простота отчетности, достигаемые при назначении ответственным одного лица, должны сочетаться с высоким уровнем взаимопонимания между членами команды и возможностью достижения творческого консенсуса в процессе свободного сотрудничества.

Применяя разные подходы и работая независимо друг от друга на разных континентах, австралийский консультант Роб Томсет (Rob Thomsett) и я занимались этой проблемой и нашли одинаковые решения (Thomsett, 1990 [62]; Constantine, 1989 [11], 1991 [13]). Команды, разрабатывающие программное обеспечение, могут достичь наибольшей эффективности, если они хорошо структурированы. Тогда открытое сотрудничество будет более результативным, а достижением консенсуса будет легче управлять. Такой «структурно-открытый» подход сочетает в себе элементы традиционной «закрытой» модели командной работы и «открытой» модели коллективного принятия решений. Со стороны такая команда выглядит как традиционная иерархия (за проект отвечает руководитель), но внутри она функционирует как группа равноправных коллег, сотрудничающих друг с другом. Внутри команды существуют четкие роли, однако члены команды играют их поочередно. Существуют правила и формальные процедуры, однако они предназначены для того, чтобы способствовать свободным исследованиям и достижению консенсуса. Каждый аспект этого подхода продуман так, чтобы компенсировать недостатки одной модели с помощью элементов, взятых из другой. В модели Константина-Томсета нет ничего принципиально нового, но сама по себе такая комбинация является довольно интересной.

Например, в рамках этой модели руководитель проекта, который в конечном итоге несет ответственность за полученный результат, является равноправным активным участником обсуждений и работы, происходя-

щей в команде. В частности, руководитель проекта никогда не ведет рабочие собрания – вместо него это делает нейтральный помощник. Открытый процесс достижения консенсуса может быть значительно эффективнее, если обсуждения будут проводиться не руководителем проекта, а с помощью нейтрального ведущего (см. главы 1–3). С другой стороны, процесс совместного принятия решений может легко застрять на второстепенных вопросах или в бесплодных спорах. В таких случаях в действие может вступить ответственный лидер проекта, который может отложить обсуждение данной темы или, в редких случаях, сыграть роль третейского судьи, если группа зашла в тупик.

Постоянное ведение записей о всех изгибах и поворотах в групповой разработке также позволяет сделать работу более надежной и управляемой. В структурированной «групповой памяти» могут фиксироваться решения, аргументы, рабочие продукты, отложенные решения, списки того, что нужно сделать или найти, и даже те подходы, которые были отклонены. Групповая память расширяет возможности контроля над рабочим процессом и делает обсуждения более эффективными. Ключевая информация и выводы находятся под рукой, а аргументы будут не так часто забываться и повторяться. Кроме того, групповая память может упростить или ускорить обсуждения, так как служит удобным местом для сохранения того, что сейчас отвлекает или не может быть решено сразу.

Как ведущие, так и секретари должны держаться в стороне от технических обсуждений для того, чтобы группа могла достичь наилучшего результата. Поскольку типичные группы разработки не могут приглашать специально подготовленных ведущих или секретарей, эти обязанности могут передаваться друг другу, а не входить в должностные инструкции. Человек, ведущий обсуждение, может сменяться при переходе к другой теме, а пополнение структурированной групповой памяти возлагается на всю команду, когда роль «информационного менеджера» (того самого «скромного и высокопоставленного писаря», упомянутого в главе 4) поочередно играют все сотрудники.

Управление обсуждениями

Команды с открытой структурой большую часть своей работы выполняют «лицом к лицу». Это не значит, что они тратят время на всякие встречи. Это означает участие в рабочих обсуждениях, совместное распределение функций и определение требований, анализ задач, планирование системной архитектуры, пересмотр дизайна и даже кодирование критических участков. Суть состоит в том, чтобы за счет открытости работы (см. главу 26) и применения разнообразных навыков и идей, привносимых членами команды, добиться сокращения ошибок и повышения качества работы.

Члены команды могут поочередно исполнять и другие функциональные роли. Это может быть обеспечение доступа к специальным знаниям по разработке приложений (что особенно важно в объектно-ориентированных методах разработки и для проектирования пользовательских интерфейсов), а также обеспечение связи с остальной частью организации (см. предыдущую главу о «командной политике»). Поскольку критические отзывы имеют важное значение для улучшения качества программ, роль «резидентного критика» формально считается неотъемлемой частью командной работы. «Резидентный критик» должен указывать на проблемы и предлагать альтернативные варианты, а также удерживать группу от соблазна остановиться на скороспелом, но неудовлетворительном решении. Однако ни один член команды не должен быть постоянным придирой; это временная роль, которая должна исполняться по очереди. На некоторое время вы становитесь скептически настроенным критиком, а потом моя очередь.

Основные особенности «структурно-открытой» модели – ведение рабочих обсуждений с помощью помощников, структурированная групповая память, поочередно исполняемые роли – оптимальны для разработки программного обеспечения и приложений. Поддерживая творческое равновесие между структурой и гибкостью, эта модель делает технический консенсус более эффективным, а техническую производительность более гибкой.

Разумеется, читатели, которые помнят классический роман Роберта Э. Хайнлайна «Луна – суровая хозяйка» (The Moon is a Harsh Mistress), могут подумать: «Tanstaafl»[1]. Да, в этой модели тоже есть недостатки. Для ее применения необходимо дополнительное обучение и приемлемые темпы работы. Кроме того, не все руководители с удовольствием расстаются с частью своей власти, чтобы работать наравне со всей командой. Формальное назначение ролей и строгая поочередность их исполнения может быть неудобной для небольших команд. Глупо, если один человек проводит обсуждение, другой ведет записи, а третий (и последний) программист произносит живой монолог по поводу дизайна иконок. Маленьким командам придется идти на слишком большие компромиссы или выбирать другую модель.

В любом случае, согласно духу этой модели, я открыт для всех идей, относящихся к улучшению структуры.

Из журнала *Software Development*, том 1, № 9, сентябрь 1993 г.

[1] Аббревиатура от английской фразы из этого романа: «There ain't no such thing as a free lunch!» (В природе не существует таких вещей, как бесплатные завтраки). – *Примеч. перев.*

17

Заговор упрямцев

Новая Англия известна своими зимами и дорогами. В каждом регионе есть свои традиции разметки улиц и дорог. Например, дорожные знаки в Новой Англии обычно устанавливаются только на перекрестках, а не на главных улицах. По мнению самих янки, каждый должен и так знать, где находится Мэйн-стрит или Массачусетс-авеню. Какой смысл их обозначать? Знак на дороге, связывающей два штата, может предупреждать вас о том, что через одну милю будет поворот на Мидлборо, но на самом повороте будет знак с надписью: «Выезд на Шервуд и Бинвайл». Далее внизу, на выезде с магистрали, вам будет предложен выбор: либо повернуть налево в Мертон, либо направо в Честер! Пожалуй, если вы не знаете, куда поворачивать, вам и не нужно туда ехать. Уверен, что такой логике следуют и некоторые разработчики программного обеспечения. Они используют один термин в документации, другой в онлайновой помощи и непонятную иконку на панели. Ориентироваться в сделанных ими меню или диалоговых окнах – это все равно что ехать из Вест Роксбери в Фрипорт через Провиденс. Как говорят в штате Мэн: «Попасть отсюда туда вы не можете».

Некоторые из наших автомагистральных развязок – просто произведения искусства. Интенсивность движения на дорогах не такая, как, например, в Лос-Анджелесе, но это компенсируется самыми запутанными авторазвязками в мире. Эти асфальтовые крендельки способны привести к «пробкам» даже при движении небольшого количества транспорта. Стоит только появиться нескольким машинам с номерами других штатов или застрять какому-нибудь грузовику в будний день где-то после трех, как весь город превращается в сплошную автостоянку.

Как я убедился, самые ошеломляющие из запутанных развязок в Большом Бостоне были спроектированы подрядчиками с узкой специализацией. Такие монументы творческой комплексификации требуют особого инженерного таланта и склонности к упрямству. Например, автомобилю, направляющемуся на запад и съезжающему с платной дороги, для выезда на северную дорогу нужно сделать поворот налево, проехать под эстакадой с движением на восток, затем выехать на кольцо, проехать три четверти круга, держась правой стороны, остановиться для оплаты, потом опять проехать под восточной и западной дорогами, потом над дорогой с уличным движением, потом выехать на северный подъем, переходящий в другой подъем на протяжении одной мили, и затем, наконец, слева влиться в общий поток. Понятно? Не напоминает ли вам это какую-нибудь любимую программу для Windows?

Самое лучшее из Массачусетских дорожных чудовищ никогда не могло быть спроектировано командой в привычном понимании этого слова. Команды традиционного типа неспособны на такие высоты многоуровневой маниакальности. Нет, для построения таких конструкций требуется особая модель организации, которая обязательно должна встречаться и в области разработки программного обеспечения. Вероятно, они были спроектированы и построены группой инженеров, организованной на основе совершенно иной парадигмы, а именно «Заговор Упрямцев!».

«Заговор Упрямцев» – это международное тайное сообщество инженеров, технических специалистов и руководителей из различных отраслей. Их тайной эмблемой является гордиев узел. Их цель – окончательная комплексификация всего на свете. Их кредо: «Или по-другому, или никак». Важно не то, чтобы система была удобной или хотя бы приемлемой, а только то, чтобы она была другой, чтобы в ней было побольше «безделушек». Выглядите и чувствуйте себя *ьber alles*[1].

Пользуйтесь или выбросьте

Дух Сирила Норткота Паркинсона[2] (Cyril Northcote Parkinson) является божком этого заговора. Их самый священный рабочий принцип – никакие ресурсы не должны остаться неиспользованными. На каждый выезд с автомагистрали должна найтись своя эстакада. Каждый неясный вызов API должен найти свое применение, а действительно хорошая программа использует все вызовы. Если архивированная система не поставляется на

[1] Над всем (нем.) – *Примеч. науч. ред.*

[2] Сирил Норткот Паркинсон (1909–1993) – английский писатель, публицист и историк, прославившийся сборником трактатов под заглавием «Закон Паркинсона». – *Примеч. ред.*

десяти гибких дисках и более или, еще лучше, на CD, то она вряд ли может стоить тех денег, за которые вы ее купили. В инсталлированном состоянии программа должна занимать не менее 25 мегабайт. В процессе инсталляции должно быть создано множество каталогов. По крайней мере, некоторые из них должны быть подкаталогами в каталоге \WINDOWS, в который будут скинуты различные файлы с непонятными именами вместе с собственными .INI файлами нового продукта. И, естественно, инсталляцию вряд ли можно считать надежной, если основательно не переделать содержание WIN.INI, CONFIG.SYS, AUTOEXEC.BAT и даже SYSTEM.INI. Иначе какой-нибудь несчастный пользователь сможет деинсталлировать эту программу, просто удалив несколько файлов или каталогов.

«Заговор Упрямцев» имеет корни в области гражданского проектирования и городских подрядов, где суть игры заключается в том, чтобы применить как можно больше кирпича и строительного раствора, потому что дядюшка Берт владеет кирпичной фабрикой, а племянник Финис имеет цементный завод. Что касается программирования, то и здесь кому-то приходится искать способ, как использовать все эти мегабайты оперативной памяти и гигабайты дискового пространства. Мы терпеть не можем, когда мощности Pentium остаются незадействованными.

В любом случае конечные пользователи могут испытывать неудобства от слишком больших компьютерных мощностей. Их ослепляет ослепительная скорость. Ведь на самом деле пользователи хотят видеть события: сверкающие изображения, прыгающие значки, мигающие вспышки. Образ действия более важен, чем его выполнение. Именно здесь «Заговор Упрямцев» как раз и старается «уважить» закон Паркинсона с помощью поиска изощренных и излишне сложных решений. Проектные команды специально организуются таким образом, чтобы их участники работали с противоположными целями, так как это обеспечивает увеличение количества элементов в системе и их внутреннюю несовместимость, приводя к высокой степени бесполезной сложности.

Дорогой читатель, не относитесь с сомнением к моим словам. Не забывайте их как параноидальную болтовню сумасшедшего методиста! Доказательство этой дьявольщины как раз перед вами. В окнах вашей чудесной машины, в значках сворачивания и разворачивания, в панелях управления, в мельницах состояний ожидания можно увидеть тайные символы их программной извращенности.

Какого черта

Вы просите доказательств – вы их получите. Мой коллега по офису и я проверяли Персональные Информационные Менеджеры (Personal Information Managers), функционирующие в среде Windows. Один из них был

снабжен вычурным имитатором времени DayTimer™ и очень расточительно использовал свободное пространство экрана. Он был просто чертовски умным, настолько, что было ясно: только «Заговор Упрямцев» мог создать такое.

Все становится понятно уже по одной операции удаления: для удаления чего-либо из блокнота вам нужно перенести этот элемент к иконке с изображением проволочной (да-да, именно *проволочной*!) мусорной корзины. После чего корзина воспламеняется! Пиктографическое жертвоприношение! На создание модуля этой корзины были затрачены немалые программистские ресурсы. Но она такая *замечательная*! Она обязательно произведет впечатление на начальников, агентов по материально-техническому снабжению и других людей с умственными недостатками. Однако мало кто заметил, что на машине с быстрым 486 процессором, ускоренной видеокартой и большим монитором эти языки пламени очень напоминали шоу Лоренса Уэлка (Lawrence Welk[1])! Что еще здесь нужно говорить?

Методы разработки, которые применяет «Заговор», отражают его цели и служат им. Лучшие профессионалы объявляют о создании продукта, потом придумывают к нему упаковку и затем начинают кодировать. Составление схем разработки – это довольно трудное дело. Намного проще их составлять на основе уже существующего кода, поэтому упрямые команды, разрабатывающие программное обеспечение, действуют так. Они начинают с кодирования модулей низкого уровня, например драйверов экрана или пылающих корзин, а на их основе строят все остальное и составляют схему разработки в целом. При проведении бета-тестирования могут быть написаны и системные требования – в качестве предисловия к руководству пользователя.

Будьте внимательны, поскольку «Заговор упрямцев» может маскироваться под обычные научно-исследовательские группы или проектные команды, однако на тайных совещаниях, проводимых вечером после десяти, они замышляют противоестественные сценарии разработки. Вы когда-нибудь задумывались о том, почему некоторые из ваших приятелей допоздна задерживаются на работе? Очень может быть, что они оказались втянуты в тайные церемонии, проводимые 1 апреля 1993 года[2] в ознаменование 666-й годовщины создания «Заговора упрямцев».

Из журнала *Software Development*, том 1, № 4, апрель 1993 г.

---

[1] Lawrence Welk (1903–1992) – создатель и ведущий музыкальных программ на американском телевидении (Lawrence Welk Show). – *Примеч. ред.*

[2] Из писем, пришедших по электронной почте, стало ясно, что не все читатели этой заметки поняли, что в день выхода заметки из печати отмечался День дураков.

IV

Инструменты, модели и методы

Введение

Наши древние предки были названы *homo habilis*[1], «человек – изготовитель орудий». Человеческие существа – не единственные изготовители инструментов в природе, но такое определение описывает одну из важнейших черт в наших отношениях с окружающим миром. С помощью инструментов мы расширяем сферу нашего влияния. Они позволяют нам видеть невидимое, двигать недвижимое, манипулировать микроскопическим и строить колоссальное. Мы пишем программы, которые являются одновременно и невидимыми, и огромными. Многие известные виды деятельности были бы совершенно невозможны в их современном виде, если бы не наличие хороших инструментов. Столярничество, механика, гражданское строительство, аэронавтика, электроника, приготовление пищи – во всех этих областях используются свои наборы основных инструментов и свои методы их применения.

Инструменты, применяемые во многих ремеслах, включают в себя не только те, которые можно держать в руке, но также и те, которые можно держать в голове. Это способы мышления и изложения мыслей, которые определяют профессию и профессионала. Модели и методы их построения – это концептуальные инструменты, которые служат для разработчиков программного обеспечения интеллектуальными рычагами так же, как лом является рычагом для бригад строителей. Аналогично тому как столяр должен знать, как и когда можно использовать поперечную пилу, инженер по разработке программного обеспечения должен понимать, какие способы применения подходят для концептуальных и других инструментов разработки программ.

Инструменты и ремесла идут в ногу со временем и с изменениями в практической деятельности и материалах. Было время, когда вы могли узнать инженера по его неизменной логарифмической линейке, висящей на ремне. Когда я только начал заниматься программированием, стандартными инструментами были бланки для записи программ, пластиковая панель

[1] Человек умелый. – *Примеч. перев.*

для составления блок-схем и дамп памяти. В то время мы писали на Фортране или ассемблере и никогда не причиняли вреда компьютерам. Большая программа представляла собой целую стопку карточек и состояла из 4 000 строк кода. Вы могли запомнить ее, если у вас были такие способности. Сложные инструменты для моделирования, разработки и отладки не применялись, да в них и не было необходимости.

По мере увеличения требований к приложениям развивались инструменты и методы, которыми пользовались разработчики. Типичный современный разработчик применяет целый набор любимых инструментов, методов и моделей на каждом этапе работы – от замысла до отладки и инсталляции. В данном разделе будут рассмотрены некоторые аспекты взаимосвязей между разработчиками и их инструментами, а также между инструментами программирования и методами их применения. В главе 22 перепечатано первое опубликованное описание сущностных пользовательских ситуаций. Значение этого описания как концептуального инструмента в современной практике разработки программ постоянно возрастает. Более подробно этот инструмент обсуждается в разделе VII.

18

CASE и познание

Автоматизированное проектирование и создание программ (CASE, Computer-Aided Software Engineering) больше не является актуальной темой в области разработки программного обеспечения и приложений. Даже производители CASE-инструментов стараются переименовывать свои продукты, называя их «средами комплексной разработки» или просто «наборами инструментов». Как бы они ни назывались, средства разработки, которые мы применяем (успешно или неуспешно), в значительной степени связаны с тем, к чему мы стремимся как разработчики.

Иногда я бываю ярым сторонником инструментов. Как известно, многие из современных инструментов являются относительно примитивными. Зачастую заложенные в них идеи неверны. Такие системы созданы заблуждающимися производителями, которые не понимают и не применяют те методологии разработки программ, которые поддерживаются их инструментами. И все же, эти инструменты могут быть эффективными в той же мере, в какой каменный топор может быть более эффективен для рубки леса, чем голые руки.

Неудивительно, что зачастую можно услышать что-нибудь этакое: «У нас нет времени на использование CASE-инструментов, нас поджимают сроки». Это могут произносить те программисты (теперь уже полысевшие или поседевшие), которые когда-то возражали против языков высокого уровня. Наверное, они никогда не составляют блок-схем или схем потоков данных и настаивают на том, что в отличие от нас, простых смертных, они могут все удерживать в своей голове. С другой стороны, многие критики CASE-инструментов действительно пытаются применять определенные методы разумного проектирования и разработки. К сожале-

нию, многие CASE-инструменты вместо того, чтобы способствовать процессу методичного решения задач и творческого проектирования, на самом деле препятствуют ему.

Что неправильно в этой картине? Вы видите высокооплачиваемого разработчика программного обеспечения, сидящего в офисе за рабочей станцией стоимостью шесть тысяч долларов и пользующегося CASE-инструментом стоимостью двенадцать тысяч долларов, который чертит в своем блокноте и делает заметки на желтых клочках бумаги. В конце концов, после множества исправлений, зачеркиваний и перерисовок он берет мышь и начинает вводить то, что разработал. Таким образом, сложный набор инструментов, установленный на его рабочей станции, превращается в замысловатую электронную доску для черчения.

Вот что здесь неправильно: вместо того чтобы взаимодействовать с разработчиком согласно его представлениям, инструмент работает против него. Вместо того чтобы поддерживать врожденные способности и привычки, укоренившиеся при обучении, CASE-инструмент становится помехой. Для глубокого понимания таких трудностей нам следует рассмотреть, как люди, и в особенности люди инженерного склада ума, решают задачи.

Создание эскизов

Например, мы знаем, что многие из хороших разработчиков, аналитиков и проектировщиков в самых сложных проектах делают приблизительные зарисовки того, что они собираются создать. Далее такие зарисовки дополняются деталями или в них вносятся уточнения. Загляните через плечо такой специалистки, когда она занята решением задачи, и понаблюдайте за ее действиями. Сначала она может нарисовать целый набор символов, означающих компоненты. Затем она отображает взаимосвязи между некоторыми из этих пустых рамок, проводя между ними линии и стрелки. И наконец, она описывает компоненты и уточняет некоторые детали во взаимосвязях между ними.

Что можно сказать о типичных CASE-инструментах? Во многих из них с помощью мыши вы выбираете из набора пиктограмм нужный символ, устанавливаете курсор (опять же с помощью мыши) в том месте внутри создаваемой диаграммы, где вы хотите расположить этот символ, и затем выполняете щелчок мышью, чтобы поместить символ на это место. В этот момент появляется диалоговое окно, в котором запрашивается имя для нового элемента. Это имя должно быть назначено в соответствии с общими и корпоративными стандартами, которые установлены для таких символов. Потом вы должны описать его, назначить для него интерфейсы и, возможно, задать другие параметры. И только после того, как все это выполнено в соответствии с принятыми правилами синтаксиса, вы сможете

продолжить составление диаграммы. Однако к этому времени вы, наверное, уже забыли, что собирались делать дальше. Более того, общее представление о содержании и структуре задачи, которое казалось таким ясным, когда вы только протянули руку к мыши, теперь стерлось из вашей ментальной карты благодаря отвлекающим деталям CASE-инструмента.

Альтернативы и альтернативные точки зрения

Все проектирование основано на компромиссах. Исследование, проведенное более тридцати лет назад, показало, что эффективные инженеры-проектировщики обычно сравнивают два или более альтернативных подхода к каждой существенной задаче. В разработке программного обеспечения эта стратегия может применяться так же успешно, как и в сферах деятельности с более богатой историей. (Моя диссертация в Массачусетском технологическом институте была как раз посвящена этому вопросу.) Сравнение альтернативных подходов может происходить быстро и главным образом в уме, или же оно может включать в себя составление сложных описаний и моделирование каждого варианта с тщательным анализом и оценкой всех результатов. В итоге может появиться ясная выигрышная стратегия, но иногда предпочтение отдается творческому синтезу на основе более чем одной альтернативы или компромиссу. Важнейшей частью процесса является взвешивание альтернативных вариантов с помощью прямого сопоставления двух схем или интерпретаций. Большинство современных CASE-инструментов не позволяют оставлять активными и доступными две версии одной системы, диаграммы или модели и уж точно не дают возможности их непосредственного сравнения.

Я видел несколько довольно хитрых способов, предназначенных для преодоления этого ограничения. В одной фирме у системного аналитика были в распоряжении две рабочие станции. На одной из них он вносил изменения и таким образом мог анализировать преимущества и недостатки сразу двух полных версий одной системы. Однако чаще всего альтернативный вариант содержится на бумаге, а другой – в хранилище CASE-инструмента.

Вот сцена, которую вы, возможно, видели или даже принимали в ней участие. Разработчик программного обеспечения, использующий CASE-инструмент, отдает команду распечатать или вывести на плоттер модель проектируемой системы, например схему потоков данных. Далее он бежит по коридору к серверу печати и получает результат. Затем он возвращается в офис, чтобы вывести другую модель той же системы, например ее структурную схему. После этого разработчик разрывается между моделью на экране и моделью на бумаге.

Даже так называемые «интегрированные» наборы CASE-инструментов обычно не дают возможности просто и быстро переходить от одной модели системы к другой. Такие переходы должны осуществляться одним нажатием клавиши. Еще лучше, если будет предусмотрена возможность наглядного сопоставления. Окна не очень подходят для этого. К тому времени, как вы откроете два окна в CASE-инструменте, работающем в оконном режиме или среде, на экране уже не останется места, чтобы хорошо рассмотреть что-либо. Или будет виден только небольшой кусочек каждой схемы, или на экране будут отображены маленькие, нечитаемые символы и текст. Вряд ли компьютер может помочь[1] в разработке программного обеспечения!

Создание и оценивание

Среди самых отъявленных преступников, нарушающих мыслительный процесс разработчиков программного обеспечения, есть некоторые из современных инструментов. К их числу относятся контекстно-зависимые программные редакторы, выполняющие синтаксическую проверку при вводе, или CASE-инструменты, которые поддерживают и навязывают определенную «методологию» разработки. Такая «методология» принуждает пользователя вводить только правильные схемы и описания, причем именно в том порядке, который был определен в «авторитетной» инструкции, написанной гуру в области методологии.

Когда компьютеры только начали применять для обработки текстов, системы проверки орфографии были отдельными программами – настолько медленными и неэффективными, что вы проверяли документ только в случаях крайней необходимости. Зачастую вы просто «забывали» это делать. Однако и компьютеры, и методы поиска стали быстрее. Системы проверки орфографии были интегрированы в текстовые редакторы. Довольно скоро один программист, у которого было свободное время, придумал орфографическую проверку «на лету», выполняемую непосредственно при вводе слов. Как-никак, во время ввода текста процессор большую часть времени все равно ничем не занят, а просмотр слов может вестись побуквенно между нажатиями клавиш. Хорошая идея, правда? Нет, неправда!

Если вы когда-нибудь пользовались текстовым редактором или электронной печатной машинкой, снабженной системой проверки орфографии в реальном времени, вы сами знаете, почему. Этот вредный гном постоянно встревает, чтобы сказать вам, что, возможно, вы сделали ошибку. При

[1] Автор с юмором намекает на аббревиатуру CASE (Computer-Aided Software Engineering). «Aid» (англ.) – помощь, поддержка. – *Примеч. ред.*

этом он подпрыгивает, словно резвящийся щенок, или пищит, как сканер штрих-кода в магазинной кассе. Даже если он прав, а вы не правы, вас это не волнует. Вы хотите просто записать свои мысли, и чтобы при этом вас не прерывал никакой сумасшедший умник, знающий орфографию.

Одно из правил простого, но мощного метода «мозгового штурма» заключается в том, что никто не может критиковать или комментировать какую-либо идею до тех пор, пока весь процесс не закончен и все идеи не изложены. Отделение творческого процесса от процесса оценки позволяет улучшить процесс решения задач.

CASE-инструмент, который соответствует способу мышления человека, не выступает в роли критика, пока вы создаете что-либо. По существу, такой инструмент позволят нарисовать и описать все виды элементов, которые «неправильны», поскольку эти «отступления от правил» часто имеют решающее значение для нахождения удачных решений. Такой инструмент позволит вам отклониться от предписываемого порядка введения элементов, так как «методологии», описанные в книгах, не обязательно являются последним словом в разработке программного обеспечения. (На самом деле многие методологии являются неверными с точки зрения решения проблем человеком, но это уже другая тема.)

Следует ли нам оставить надежды и поставить CASE-инструменты на полку? Нет, надежда все же остается. Современные CASE-инструменты – это примитивные предшественники тех инструментов, которые нам действительно нужны. Они еще появятся.

Все это немного напоминает первые программы обработки текстов, такие как Electric Pencil или первые версии WordStar. Согласно сегодняшним стандартам функциональности и удобства, они не выдерживают критики. Пользователю приходилось ждать минуты, чтобы перейти с одного конца документа на другой. Для выполнения элементарных действий требовались непонятные и сложные нажатия клавиш, а средства форматирования были ограниченными. Однако применять эти редакторы было намного удобнее, чем писать от руки или печатать на машинке, а потом перепечатывать и перепечатывать.

Впрочем, кто-нибудь из тех, кто создает CASE-инструменты, возможно, прислушается к сказанному.

Из журнала *Computer Language Magazine*, том 9, № 1, январь 1992 г.

19

Вопросы моделирования

«Хороший инструмент для разработки – это тот инструмент, который не замедляет мою работу». Программист осторожно посмотрел на коробку весом почти 40 кг, в которой находилась новейшая и лучшая среда разработки на C++. «Мне нужен такой инструмент, который даст мне возможность программировать так, как я хочу. После этого на основе кода такой инструмент должен составить эти дурацкие схемы, которые требует от меня начальник». Думаю, что сейчас, наверное, самое время поговорить об этих дурацких схемах.

Многие разработчики – особенно те, которые ломают головы над созданием микрокомпьютеров и рабочих станций, – имеют очень смутное представление о структурных схемах, диаграммах объектных коммуникаций, схемах информационных потоков и блок-схемах. Довольно многие из них никогда не рисовали функциональных иерархий и растерялись бы, обнаружив такие схемы на своем столе. Увидев Booch-грамму[1], они подумали бы, что это какая-нибудь плохая новость, доставленная на желтой бумаге курьером из компании Booch Telecommunications.

Многим сегодняшним волшебникам в области программного обеспечения эти прямоугольники и овалы, окружности и стрелки кажутся иероглифами, которые оставили исчезнувшие служители культа. Такие значки рассматриваются ими как наследие отживающих свой век методов, таких как структурный анализ и проектирование, которые уступили дорогу ус-

[1] Grady Booch (Гради Буч) – признанный эксперт в области объектно-ориентированной методологии разработки программного обеспечения. – *Примеч. ред.*

коренной объектно-ориентированной разработке программ на основе создания прототипов. Схемы и диаграммы не находят своего места в быстро меняющемся мире скороспелых микроприложений, разрабатываемых с помощью методов визуального программирования и быстрого создания приложений. «У нас нет времени на рисование картинок. Нас поджимают жесткие сроки выпуска новой версии!» «Зачем вообще рисовать схемы, если можно просто писать код?» Это хороший вопрос. Для чего нужны эти рисунки?

Конечно, структурные схемы и другие классические модели проектирования и анализа были разработаны не для того, чтобы замедлить процесс программирования, так же, как они не были созданы для того, чтобы удовлетворить нужды суетливых потребителей или осчастливить во все вмешивающихся руководителей. Большинство этих методов разработчики придумали для собственных нужд – чтобы упростить и ускорить свою работу. Время, затраченное на обдумывание программ с помощью моделей, – это время, которое экономится при программировании и отладке. Модели проектирования и анализа сокращают время разработки, так как позволяют моделировать программы без необходимости их кодирования и облегчают принятие замысловатых решений для сложных задач. Все это хорошо известно. Хороший план сокращает время разработки; хорошие модели проектирования упрощают планирование.

На практике не все графические модели работают таким образом. Некоторые, например HIPO-схемы[1] и их вспомогательные методы от компании IBM, вполне заслуженно канули в лету. Другие страдают от громоздкости и сложности обозначений или механизмов их рисования. Первоначально CASE-инструменты появились как специальные инструменты рисования, облегчающие создание схем. Со временем они превратились в более комплексные средства, облегчающие разработку программного обеспечения.

Изобразите это

Разработчики программного обеспечения рисуют картинки перед созданием самих программ по тем же причинам, что и архитекторы, рисующие поэтажные планы и вертикальные профили перед тем, как приступить к строительству дома. Тем не менее здания не всегда строились с помощью планов и рисунков. Если схема здания достаточно проста и знакома, строители могут работать без помощи моделей, ведя проектирование

[1] HIPO (hierarchical-input-processing-output) – технология разработки программного обеспечения на базе идеи структурного программирования. – *Примеч. перев.*

в процессе строительства. Сельским общинам на рубеже веков не требовались чертежи, чтобы построить сарай. В те времена сараи имели простую конструкцию, а в их строительстве применялись несложные методы. Каждый знал, как выглядит сарай, как он строится и что для этого требуется. Большинство членов общины уже занималось этим, а новички могли научиться, внимательно наблюдая за тем, что делают другие, и делая то же самое. Но если перейти от хижин и сараев к домам в колониальном стиле с четырьмя спальнями или высотным квартирным комплексам, то все становится сложнее.

То же самое можно сказать и о компьютерных программах. В те времена, когда 64 Кбайт оперативной памяти было пределом, а в качестве операционной среды использовалась CP/M, было вполне возможно и разумно удерживать всю программу в своей голове. Но тот, кто говорит, будто сможет запомнить все детали в сотнях тысяч строк программы на C, говорит неправду – если не вам, то себе. Именно здесь возникает необходимость в моделях проектирования.

Обратите внимание, что мы говорим о моделях, а не о схемах, о проектировании, а не о документировании. Для многих разработчиков эти странные картинки – все равно что китайская грамота. В лучшем случае это еще одна часть документации, которую нужно держать в папочке потому, что это требуется по контракту. Действительно, проектные модели могут быть полезными в качестве документации. Чертежи вашего дома не только нужны для строителей, но и могут помочь определить, где пробить стену, чтобы добраться до трубы с горячей водой (и не задеть ее). Структурные схемы и диаграммы взаимодействия между блоками могут показать, где нужно искать те или иные процедуры, или помочь отследить потоки движения информации, или понять, как организована система в целом. Однако основное назначение моделей проектирования – помогать проектировать.

Управление сложностью

Модель, по крайней мере любая разумная модель, проще того, что она моделирует. С помощью хорошей системы обозначений основные элементы и характеристики очень сложной системы можно отобразить в виде сравнительно небольшой и простой схемы.

Моделирование – это интеллектуальный инструмент. Если вы знаете, что означают символы и применяемый язык, модель может стать способом описания и осмысления сложных задач, особенно с точки зрения взаимосвязей между ее компонентами. В коде или тексте такие взаимосвязи неочевидны; слово или метка здесь могут относиться к тому, что находится совсем в другом месте. В большинстве графических моделей взаимосвязи

наглядно представлены в виде линий и стрелок, соединяющих разные части схемы. Хорошие графические модели дают ясное представление о системе, которое невозможно получить при прочтении (или написании) страниц кода.

Конечно, многие люди создают только мысленные модели тех задач, над которыми они работают. Некоторые опытные разработчики могут даже отчетливо представлять системы с помощью структурных схем, которые они держат в своей голове. Однако схемы на бумаге или на экране монитора все же имеют преимущество, поскольку позволяют облечь внутренние мысленные модели в конкретную внешнюю форму. Во внешнем представлении модели становятся устойчивыми, и их можно обдумать или сравнить с другими моделями. Такую модель можно отложить в сторону или рассмотреть позже, на свежую голову. Зачастую, если вы видите что-либо, представленное в виде прямоугольников и линий, это изменяет способ вашего мышления об этом. Появляются другие варианты или возможность понять ошибки или упущения. Кроме того, никто не может увидеть модель, которую вы держите в голове. Но если модель вывешена на стене или имеется в хранилище CASE-инструмента, то вся команда, работающая над проектом, сможет изучать ее и развивать.

Хорошие инструменты моделирования также позволяют делать своего рода эскизы, которые трудно выполнить в коде. Языки программирования точны и детальны, а компиляторы упрямо требуют наличия точного синтаксиса и законченных конструкций. Чистая доска не имеет таких ограничений. Разработчик или вся группа могут подойти к ней и «поиграть» с картинками. Они могут передвигать элементы по схеме, отслеживая сложные взаимосвязи между распределенными областями системы. Словом, модели проектирования позволяют разработчикам программного обеспечения действовать подобно настоящим инженерам, проверяя и сравнивая альтернативные варианты организации программ без необходимости переложения идей в код.

Почти любой аспект программного обеспечения можно моделировать графически: алгоритм, структуру данных, взаимосвязи, композицию, динамику. В каждой из этих и других областей могут применяться десятки конкурирующих систем обозначений и представлений. Однако одна модель не обязательно так же хороша, как и другая. Системы обозначения могут существенно различаться по своим возможностям передачи смысла. Важна форма фигур и то, куда показывают стрелки.

Как же отличить хорошую модель от плохой, а плохую от безобразной? Оставайтесь с нами.

Из журнала *Software Development*, том 2, № 2, февраль 1994 г.

20

Свет мой, зеркальце

«Свет мой, зеркальце, скажи, да всю правду доложи: я ль на свете всех милее»? Злой королеве из «Белоснежки» достаточно было посмотреть в зеркало, чтобы получить точную картину. Разработчики программного обеспечения тоже должны иметь такую возможность. Им нужны хорошие зеркала, которые просто и точно отражают создаваемое программное обеспечение. Злая королева могла быть недовольна тем, что видела в зеркале; тем не менее ее зеркало давало достоверную и понятную картину.

Именно это предлагает хорошая система моделирующих обозначений – ясное, однозначное и легко понимаемое отображение программного обеспечения. В отличие от зеркала, хорошая система моделирующих обозначений не может дать детальную картину, «один к одному» соответствующую ее коду. Хорошая модель – это точное, но все же избирательное изображение программы, то есть ее намеренно упрощенное описание. Эффективность системы моделирующих обозначений для изображения задач и их программных решений зависит от того, каким образом достигается это упрощение. Сама суть преобразований между средой программирования и средой моделирующих обозначений должна быть простой, непосредственной, логичной и легко запоминаемой.

К сожалению, многие из существующих систем обозначений для программного анализа и проектирования не отвечают этим фундаментальным требованиям. Они чрезмерно сложны, непоследовательны и неясны. Их трудно запомнить и интерпретировать. А еще их слишком много.

Полное моделирование программного обеспечения – это сложный процесс. Он включает в себя рассмотрение множества статических и динами-

ческих аспектов: структура и композиция информации, суть алгоритмов и их реализация в виде процедур, разбиение на составные части и взаимосвязи между частями. Если модель данных, процедурная модель, модель коммуникаций, модель состояния и функциональная структура программы слиты в единую схему с одной всеобъемлющей системой обозначений, то результат принимает причудливую форму, становится непонятным и визуально перегруженным.

Получение картины

Модели разработки программного обеспечения во многом служат той же цели, что и зеркало злой королевы. Системы обозначений должны делать ясным различие между прекраснейшими решениями и теми, которые всего-навсего прекрасны. Рассматривая модель, разработчики хотят знать, является ли их замысел здравым или глупым. Модель проектирования программного обеспечения – это не просто место хранения пока еще нереализованных идей. Такая модель позволяет разработчикам найти упущения и ошибки в замысле, а также сравнить разные подходы и выяснить, какой из них лучше. Вот почему хорошие разработчики рисуют картинки перед тем, как приступить к кодированию, – создать «бумажную» модель дешевле, чем создать программное обеспечение. Кроме того, хорошие модели помогают увидеть самый подходящий способ решения задачи.

Хорошая система обозначений предполагает простой, непосредственный и понятный способ преобразований модель-код, то есть позволяет писать код на основе модели или составить модель на основе существующего кода. Небрежная, неясная и неточная система обозначений приводит к небрежному, неясному и неточному преобразованию. Каждый визуальный элемент в системе обозначений должен соответствовать конкретному и существенному аспекту моделируемого программного обеспечения. Каждый важный элемент кода должен иметь возможность быть выраженным в системе обозначений.

Хорошая система обозначений позволяет создавать картины, которые можно интерпретировать аналитически, с помощью тщательного изучения деталей, и понять интуитивно как гештальт[1], то есть как целое, представляющее общий характер системы. Сложные схемы должны выглядеть сложнее, чем простые. Хорошая архитектура должна быть визуально привлекательнее, чем плохая. Другими словами, хорошая система

[1] От нем. *gestalt* – структура, модель некоего явления, интегрированная таким способом, что воспринимается как единое целое, которое не может быть достигнуто как простая сумма его частей. – *Примеч. науч. ред.*

обозначений позволяет разработчикам использовать оба полушария мозга; она помогает думать о проектируемой системе как с позиций логики, так и на основе интуиции.

Для точного и достаточно простого отображения программ хорошая система обозначений выделяет важные элементы и скрывает несущественные. Главные части и основные компоненты выделяются крупно, в то время как менее значимые детали становятся примечаниями или комментариями или совсем не обозначаются на схеме.

Картина сохраняет простоту, так как не отображает внутренних деталей. Компонент программы, который по сути является «черным ящиком», становится простым прямоугольником, нарисованным на экране или бумаге без отображения внутренних деталей. Здесь речь идет о целых программных блоках, а не об их внутреннем содержимом.

В идеале мы хотели бы получить такой инструмент контроля отображения, который достижим только с помощью компьютерных средств. Например, мы хотели бы просматривать схему коммуникаций между объектами, пробегая глазами мелкие элементы, а затем увеличивать масштаб, чтобы изучить взаимосвязи в какой-то одной части схемы. Мы могли бы увеличить какой-нибудь объект, чтобы узнать, какие методы он поддерживает. Делаем двойной щелчок мышью и видим код C++, определяющий данный метод. Или же мы могли бы переключиться в режим, в котором сценарий взаимодействия с пользователем совмещен со схемой коммуникаций между объектами, причем объекты, участвующие в данном сценарии, выделены или подсвечены.

Система обозначений и юзабилити

Систему обозначений можно легко разработать самому; очень многие так и поступают. Однако хорошую систему обозначений придумать трудно. Многие системы, которые встречаются в наших журналах, не в полной мере подходят для моделирования.

Разработка хорошей системы обозначений подобна разработке хорошего графического интерфейса. Целью здесь является сокращение нагрузки на человеческую память. Великий закон юзабилити утверждает, что система должна быть доступной для использования человеком, который знает о ее назначении, но не знаком с ее программным обеспечением. При этом не должно возникать необходимости в обучении, помощи или руководствах (Constantine, 1991 [14]). Таким образом, по-настоящему хорошая система обозначений – это такая система, которую опытный разработчик, знающий, как проектировать и писать программы, сможет интуитивно понять без шпаргалок и недельных курсов обучения. Вы не должны запоминать разные условные обозначения (например, то, что

прямоугольник с двойной рамкой означает динамический объект, а прямоугольник с флажком в углу – компонент библиотеки).

Все должно выглядеть так, как оно есть. Формы символов или стили линий не могут быть какими угодно. Они не должны противоречить интуиции. Например, базовую основу, на которой с помощью наследования и повторного использования выстраиваются другие компоненты, необходимо изображать с помощью сплошных, строго очерченных элементов, а не эфемерными облаками.

Сильные связи между частями должны выглядеть, как сильные связи; слабые связи должны выглядеть, как более слабые. Устойчивые объекты должны быть показаны сплошными фигурами, а динамические объекты должны выражать смысл активности или изменяемости. Например, наследование одним классом свойств и характеристик другого класса означает, что подкласс в большой степени зависит от родительского класса – подобно тому как генетические особенности ребенка сильно зависят от особенностей обоих родителей. Для точного отражения характеристик программного обеспечения, использующего наследование, этот механизм должен быть показан более четко, чем передача сообщения объекту или ссылка на него как на атрибут (Page-Jones, Constantine и Weiss, 1990 [57]).

Сделать систему обозначений интуитивно понятной и простой в изучении помогут небольшие детали. В системе обозначений для объектно-ориентированных программ, разработанной Джекобсоном (Jacobson и др., 1992 [44]), те объекты, которые взаимодействуют с внешним окружением, на одной стороне имеют символ ├ в качестве визуального ключа, обозначающего это взаимодействие. Динамические объекты, которые управляют последовательностью действий, снабжены стрелкой, входящей в границу, что означает цикл или итерацию. В нескольких системах обозначений внутренние элементы компонентов, которые доступны извне, показаны в виде расширений границы компонента.

Кроме того, хорошая система обозначений основана на том, что разработчики уже знают. Главным образом в ней применяются обозначения, знакомые разработчикам. Другими словами, не следует использовать новые символы для давно известных понятий, а старые обозначения – для новых и несовместимых идей. На самом деле чаще всего нам и не нужны новые обозначения. Наши усилия следует направить в основном на стандартизацию и упорядочивание того, что уже имеется, с применением разумных принципов моделирования и человеческого мышления.

Подумайте об этом.

Из журнала *Software Development*, том 2, № 3, март 1994 г.

21

Методичное сумасшествие

Каждая модель привносит свой собственный метод, как будто всех этих бессловесных схем недостаточно, чтобы похоронить под собой загруженного программиста. Структурные методы уже не вызывают особого интереса, однако в желающих написать книги, статьи и поучаствовать в конференциях, недостатка не ощущается. Эти новейшие, наилучшие методы позволяют упростить разработку и повторно использовать объекты для прототипов, но даже в самых радикальных методах разработки многое кажется знакомым.

В методах, применяемых при создании программ, нет никакой тайны. Вы тоже можете стать методистом в области программного обеспечения, даже не учась на заочных курсах. Какие бы названия методов не использовались – структурный анализ и проектирование, информационный инжиниринг, круговой гештальт, объектно-ориентированное или простое процедурное программирование – во всех книгах и разговорах, посвященных этому, все сводится к системному решению задач. Предоставляя возможности для упорядоченного применения моделей и инструментов, методы помогают сделать процесс разработки типовым, а значит, более простым в изучении и доступным для детализирования и усовершенствования.

В основе всех основных методов анализа и проектирования программного обеспечения лежит очень небольшой набор базовых принципов, которые постоянно открываются заново и описываются новыми словами, но речь по-прежнему идет о все том же старом гуталине. Суть дела заключается в том, как люди решают сложные задачи. Все методы разработки программного обеспечения базируются на пяти принципах: (1) упорядочен-

ное продвижение, (2) решение с помощью разбиения, (3) независимость компонентов, (4) целостность компонентов, (5) структурное соответствие.

Шаг первый

При выполнении большой работы важно начать где-то, сделать что-то, а затем сделать что-то другое. Методы дают возможность определить, где нужно начать и куда нужно перейти на следующем шаге. Каждый этап в этом процессе, выполняемый с надеждой в сердце и кодом в голове, позволяет вам придвинуться еще на один шаг к получению законченного программного решения.

Все методы предлагают некоторый способ упорядоченного описания того, что потребуется сделать для выполнения работы и получения программного продукта. Они служат своего рода напоминаниями о различных аспектах, которые следует рассмотреть и понять, чтобы разработать систему. Хорошие методы ставят первоочередные этапы первыми и откладывают то, что можно отложить. В этом и заключается принцип упорядоченного продвижения: сделать то, потом другое, затем следующее.

Разбиение

Все *реальные* задачи в разработке программного обеспечения, то есть те задачи, с которыми профессионалы в области программирования сталкиваются каждый день, являются слишком большими и сложными для решения. Единственные полностью решаемые программные задачи – это те игрушечные приложения и академические упражнения, которые встречаются в учебниках и преподаются в учебных аудиториях. Это одна из причин, по которым университетские научные работники, специализирующиеся в компьютерной области, могут сделать карьеру на элегантной математике и методах формального доказательства, которые безнадежно неадекватны повседневным задачам, возникающим при программировании. Они никогда не имели дела с реальными задачами.

Мы имеем. Если мы сталкиваемся с неподатливой сложностью, то что мы делаем? То же, что делали наши предки, когда имели дело с огромной ногой мастодонта, которую нельзя проглотить: мы откусываем часть и начинаем ее пережевывать. Для того чтобы справиться с большой задачей, мы разбиваем ее на более мелкие. Здесь есть некоторая магия ума, махание руками, которое, вообще говоря, нисколько не уменьшает сложность задач, возникающих при программировании. Но зачастую это так или иначе помогает, создавая у перегруженного разработчика иллюзию, что большие, сложные системы можно построить из множества маленьких, простых кусочков кода.

Иногда мы, словно неудачливые начинающие волшебники, обнаруживаем, что разбиение большой задачи на части лишь все перемешало. Большие задачи становятся меньше, если разбиваются только до той степени, когда отдельные ее части перестают быть тесно связанными друг с другом и могут быть рассмотрены более или менее независимо. При этом каждая часть разрабатывается и программируется как самостоятельная задача меньшего масштаба, решить которую вполне по силам. Все эффективные методы разработки в том или ином виде содержат некоторые правила или указания, в которых говорится о необходимости «делать края ровными», когда вы выделяете из задачи ее части. Целое разделяется на элементы, каждый из которых может быть осмыслен как самостоятельная подзадача.

На самом деле в этом принципе решения задач содержатся два других. Принцип независимости компонентов говорит, что задачу следует разбивать на самостоятельные подзадачи, а принцип целостности компонентов утверждает, что каждая часть сама по себе должна восприниматься как нечто целое.

В традиционном структурном проектировании эти принципы воплощены в общепризнанных понятиях связывания и сцепления. Связывание – это мера взаимосвязи и взаимозависимости между компонентами программного обеспечения; сцепление определяет, в какой мере компонент является четко определенным функциональным целым. Эти давно известные понятия были заново открыты в новом мире объектной технологии. Хорошие объектно-ориентированные методы напоминают программистам об уменьшении объектного связывания и об увеличении сцепления методов с помощью инкапсуляции всего необходимого в отдельный объект (Henderson-Sellers и Constantine, 1996 [39]). В противном случае вы получите запутанный клубок переплетенных между собой объектов, не поддающихся анализу и повторному использованию.

Структурное соответствие

Все три первых принципа решения сложных задач касаются отдельных частей программного обеспечения, но они мало что говорят о том, как эти части лучше всего объединить в рабочее целое. Большинство методов в той или иной форме говорят о необходимости ориентироваться на реальное окружение задачи и повторять структуру задачи в структуре решения. Другими словами, программное обеспечение необходимо планировать в соответствии с той практической задачей, для решения которой оно предназначено. В этом состоит принцип структурного соответствия, своего рода вариант высказывания Баухауса (Bauhaus), перенесенного в область разработки программного обеспечения: форма должна следовать из функции.

Свою самую радикальную реализацию этот принцип находит в более упрощенных методах объектной технологии. Согласно этим методам, все, что вам требуется делать, – это смотреть на реальное окружение задачи и создавать объектные классы для всего, что вы там видите. Есть стул – вот вам, пожалуйста, стул! Мы создаем класс «стулья» в родительском классе «мебель». Слепое следование этому принципу приводит к топорным отображениям физических систем в неуклюжих программах. Еще одна трудность: ваше представление о реальности может не совпадать с моим.

Тем не менее такой подход является верным. Отражение «естественной» структуры задачи в программном обеспечении позволяет избежать решения тех задач, которые не требуется решать. Вместо изобретения новых структур мы применяем уже существующие. Придерживаясь структуры задачи, мы упрощаем процессы обслуживания, расширения и повторного использования, поскольку программное обеспечение будет структурировано согласно тому, как люди представляют эту задачу.

Вот так. Методы служат каркасом для применения моделей в решении задач, возникающих при разработке программного обеспечения.

Естественно, для того чтобы создать хороший метод и оправдать звание настоящего методиста, требуется намного больше. Например, вы должны обладать способностью выдумывать новые слова. Вам нужно применять как можно больше новых терминов, иначе люди подумают, что они уже знают то, о чем вы им рассказываете. Кроме того, вы должны уметь растягивать несколько хороших идей на 700 страниц текста. И не забудьте про систему обозначений, про те красивые фигурки, которые отличают ваш метод от множества других. Однако будьте осторожны и не сделайте картинки слишком простыми, а принципы – слишком прозрачными, иначе вы не сможете обосновать свои гонорары за консультантские и преподавательские услуги!

Из журнала *Software Development*, том 2, № 5, май 1994 г.

22

Говоря по существу

Если для того чтобы добраться до работы или дома, нужно преодолеть 11 000 миль, то поневоле задумаешься о том, что является предметами первой необходимости. Когда вы упаковываете вещи в чемоданы и коробки, рассчитывая прожить год за границей, это делает еще более отчетливой разницу между тем, что «нужно», и тем, что «хочется», поскольку тарифы на лишнюю кладь составляют более 90 долларов за сумку. В разработке программ и приложений также важно прежде всего рассмотреть самое главное, то есть отделить основную часть того, что вам нужно программировать, от несущественных требований и ненужных предположений.

Сущностное моделирование – это концептуальный инструмент, предназначенный для акцентирования внимания разработчика на главном. Сущностная модель описывает ядро приложения, извлекая из задачи самое необходимое и отделяя все ненужные или ограничивающие предположения.

Понятие сущностного моделирования в разработке программ происходит, по крайней мере, от идей структурного проектирования. Схемы потоков данных были придуманы в качестве непроцедурной модели вычислений, которая отделяла суть того, что должен был выполнить компьютер, от того, как это должно быть исполнено посредством алгоритма или процедуры. Модульная конструкция, соответствующая сути задачи, позволяет создавать более надежное программное обеспечение, основа которого сохраняется при изменениях в деталях процесса. По крайней мере, такова была идея. На практике схемы потоков данных зачастую превращались чуть ли не в обыкновенные блок-схемы с забавными символами. Последующие дополнения, такие как сохранение данных и управляю-

щая логика, неявным образом способствовали процедурному мышлению. В конце концов сущностные модели получили признание с появлением книги «Essential Systems Analysis» (Основы системного анализа) (McMenamin и Palmer, 1984). Эта книга, признанная сегодня классической, сделала сущностные модели краеугольным камнем перестроенного и обновленного метода структурного анализа.

В более широком смысле сущностные модели отображают суть системы. Они представляют собой идеализированные и абстрактные описания, свободные от технологий и соответствующие намерениям пользователей и фундаментальному назначению системы. Лучшими сущностными моделями становятся те, которые подвергаются упрощению и обобщению в процессе многократных уточнений. Они отражают дух приложения, а не его физическое воплощение в виде реального кода, работающего на реальной технике.

Сущностная модель основывается на идеальной технологии: бесконечно быстрые компьютеры, сколь угодно большие мониторы, бесклавиатурные системы ввода данных. Есть все, что может понадобиться для наиболее быстрой реализации необходимых функций. Такой полет технической фантазии – это не дань воображению, а обеспечение максимальной независимости модели от современного уровня технологии. Технология меняется намного быстрее, чем практика деловых отношений или люди; решения, которые слишком тесно привязаны к какой-либо технологии, безусловно, менее живучи и гибки.

Сущностные интерфейсы

Разработка пользовательских интерфейсов – это одна из областей, где сущностные модели могут найти хорошее применение. Моделирование сущностных пользовательских ситуаций при проектировании интерфейсов является подходом, расширяющим разработанный Джекобсоном (Jacobson и др., 1992 [44]) метод объектно-ориентированных пользовательских ситуаций. Сущностная пользовательская ситуация – это абстрактный сценарий одного законченного и полезного взаимодействия с системой, рассматриваемого с точки зрения намерений или цели пользователя. Это обобщенное описание вида или категории использования системы, которое передает действия пользователя и ответы системы[1] в упрощенной и обобщенной форме. Суть идеи состоит в том, что разрабатываемые интерфей-

[1] Наверное, эти две стороны модели еще лучше охарактеризовать как *намерения пользователя* и *функции системы*. Именно так они обозначены в методе проектирования, ориентированном на пользователя и основанном на существенных пользовательских ситуациях (Constantine и Lockwood, 1999 [30]).

сы должны соответствовать намерениям пользователей – тому, что пользователи хотят достичь. Исходные условия, связанные с технологией, – форма визуальных компонентов или даже вид устройств, применяемых для взаимодействия, – должны быть минимальны.

Для примера возьмем задачу по извлечению наличных денег из банкомата. Физическая модель может быть следующей: клиент вставляет карту; система читает информацию с магнитной полоски и запрашивает PIN-код[1]; пользователь вводит PIN-код; система подтверждает PIN-код и предлагает меню возможных операций; пользователь выбирает нужную операцию; система предлагает меню счетов; пользователь выбирает счет; система запрашивает сумму; пользователь вводит сумму и нажимает на кнопку подтверждения; система выдает наличность; пользователь берет деньги и убегает. Что может быть проще?

Магнитные полоски

Лишенная физических подробностей и технологических предпосылок, сущностная пользовательская ситуация этой задачи заключается в простом процессе: клиент идентифицирует себя, система предлагает варианты действий, пользователь выбирает, система выдает деньги, пользователь берет их. Такое описание допускает намного больше вариантов для интерфейса пользователя. Оно открывает дорогу для множества способов, с помощью которых пользователю можно предложить различные варианты действий, а он сможет из них выбрать. К числу таких способов относятся сенсорный экран и голосовая система. Ясно, что карты или PIN-коды не являются необходимыми, так как их основная задача состоит в том, чтобы просто идентифицировать пользователя. В некоторых случаях может применяться сравнение отпечатков пальцев, сканирование радужной оболочки глаза, распознавание голоса, а также устройства чтения жетонов. Сущностная модель предоставляет много возможностей, тем самым увеличивая вероятность того, что отдельные части любого решения смогут быть повторно использованы, даже если условия и предпосылки изменятся.

Эта сущностная пользовательская ситуация также прокладывает путь к более простому интерфейсу, поскольку выделяет суть дела с точки зрения пользователя, а именно получение наличных денег, что является самой распространенной операцией, выполняемой с помощью карточек. Большинство пользователей регулярно снимают одну и ту же сумму с одного и того же счета. Первый вариант действия, предлагаемый системой, может соответствовать типичному выбору данного пользователя и воспроизво-

[1] PIN (Personal Identification Number) – личный номер, код. – *Примеч. ред.*

диться из памяти: «Обычные 250 долларов с текущего счета, мистер Чатворт?». Или что-то в этом роде.

Сущностная модель отражает идеализированную цель проектирования. Хорошо разработанный пользовательский интерфейс требует ровно столько шагов или столько информации, сколько определено в сущностной пользовательской ситуации. Клиенту банка должно быть достаточно сказать: «Это я. Как обычно. Спасибо!» – и уйти. Мы определяем идеальный случай – если бы мы его не смоделировали, то не смогли бы проектировать в соответствии с ним. А без проектирования в соответствии с идеальным случаем можно не увидеть, где нас ограничивает технология или технические условия, и упустить возможность использования альтернативных подходов.

Re: редизайн

Другой областью, в которой можно применять сущностные модели, является модернизация процессов. Если модернизация процессов не является простым эвфемизмом, за которым скрывается приостановка производства или сокращение штата, то она позволяет сделать бизнес-процессы более эффективными и результативными. Для успешной модернизации некоторого процесса вы должны знать, для чего этот процесс предназначен, каким фундаментальным задачам или какой организационной цели он служит. Самым существенным вопросом в любом процессе или системе является телеологический вопрос: почему это существует? Почему это должно существовать? Для чего оно, вообще говоря, нужно?

Рассмотрим процесс обмена валюты в банке. В некоторых странах в некоторых банках бывает необходимо два или даже три раза встать в очередь, заполнить многостраничные бланки, несколько раз выполнить расчет сумм и воспользоваться услугами множества банкоматов и банковских служащих. С точки зрения клиента банка такая операция является сверхпростой: даешь деньги в одной валюте, получаешь эквивалентную сумму в другой. Банк обеспечивает правильный обменный курс, безошибочность расчета денежных сумм и извлекает небольшую выгоду из проведения обменной операции. Однако у банка нет заинтересованности в бумажных листах или занятости клерков – если целью является модернизация процесса.

Вовсе необязательно, чтобы клиент банка или банковский служащий заполнял бланки в трех экземплярах с указанием имен, адресов и с личными подписями и суммами (в цифрах и прописью) на каждом экземпляре. Курсы обмена не нужно проверять вручную, если они поступают из центральной базы данных. Дисплей, на который смотрит клиент, может служить для подтверждения суммы – подобная услуга практикуется в мага-

зинах. Распечатка о выполненной операции, выдаваемая клиенту, может пригодиться ему для отчетности. Сохранение информации об операции в архиве может использоваться для аудиторского отслеживания выполненных сделок. И в самом деле, в Европе я пользовался автоматами по обмену валюты, которые работали именно по такому, ускоренному методу.

Транс-действия

К сожалению, многие профессионалы испытывают затруднения в выявлении сути задач. Не каждый способен к абстрактному мышлению, необходимому для сущностного моделирования. Обдумывание сути требует отказа от технических шор, мешающих увидеть предметы в новом свете. Меилир Пейдж-Джонс (Meilir Page-Jones) называет это «разыменовыванием», мысленным выходом за рамки принятых понятий и представлений. Особенно это удается одаренным комедийным актерам и талантливым дизайнерам.

Вспоминаю, как однажды я ехал с Меилиром на машине вдоль реки Рейн. По дороге он переводил мне смысл надписей на немецком языке, мимо которых мы проезжали. На одном из знаков какая-то черная масса клонилась влево к волнистым линиям. Между изображениями насыпи и самой реки была показана машина, зависшая в воздухе.

Меилир сразу же перевел мне: «Остерегайтесь машин, выпрыгивающих из воды слева».

Разыменовывайтесь. Попробуйте. И старайтесь думать по существу.

Из журнала *Software Development*, том 2, № 11, ноябрь 1994 г.

23

Будущие формы

CASE умер? Многие жрецы программирования объявляли, что смерть уже наступила. Однако на каждого плакальщика всегда находится тот оптимист, который верит в воскресение из мертвых, а на каждого очернителя находится свой ликующий поборник. Ведомое духом времени, в 1994 году Австралийское компьютерное сообщество (Australian Computer Society, ACS) собрало в Мельбурне две звездных команды австралийских консультантов и специалистов-практиков. Руководили командами сторонник Грэме Симшн (Graeme Simsion) и скептик Роб Томсет (Rob Thomsett). В ходе дискуссии, названной «великим спором о CASE», обсуждались все «за» и «против» в вопросе, как обстоит дело с CASE в Австралии. Этот спор стал весьма значительным событием для такого консервативного сообщества, как ACS, – настолько, что привлек даже больше внимания, чем презентация книги бывшего премьер-министра, которая проводилась в этом же зале несколькими часами ранее. По неизвестным причинам меня втянули в этот спор в качестве члена опровергающей команды. Томсет, которого Симшн называл самым диким консультантом Австралии, привел нашу команду к победе, хотя к концу обсуждения было видно, что обе стороны находились приблизительно на одном уровне.

Действительно, слухи о грядущем погребении CASE вызывают непонимание, поскольку CASE – это (дословно) не более чем проектирование и создание программ с помощью компьютеров. Неужели компьютеры перестали оказывать в этом помощь? Или отрасль проектирования и создания программ прекратила свое существование? Будем надеяться, что не случилось ни то ни другое. CASE не исчез и не обречен, хотя и может замет-

ным образом dйclassй[1]. Следуя своему вечному стремлению приукрашивать и восхвалять то, что они когда-то вознесли до уровня святости, ученые мужи от промышленности все чаше заменяют термин CASE на «интегрированные среды разработки». Во всяком случае в последний месяц это было так.

Конечно, отчасти проблема заключается в необоснованных ожиданиях, которые подогревались рекламными кампаниями производителей программного обеспечения, а также в надеждах, питаемых любителями выдавать желаемое за действительное. Даже самый великолепный из реальных инструментов будет вызывать разочарование, поскольку почти каждый в нашем деле находится в вечном поиске меча короля Артура для программирования. Другая трудность состоит в том, что разработчики и производители инструментов CASE часто не разбирались ни в моделях и моделировании, ни в тех методах, для поддержки которых предназначались их инструменты. Более того, эти CASE-инструменты просто-напросто не были тем, что необходимо для ускорения разработки и повышения качества программ. Необходимо нечто, дающее одновременно и больше и меньше того, что дает большинство инструментов CASE. Нам следует посмотреть на визуальные среды разработки, чтобы понять, чем должны были и чем еще могут стать инструменты CASE.

Визуальные среды разработки – это одни из самых ярких и крепких нитей в клубке современных методов и продуктов. Визуальная разработка связана с большим разнообразием инструментов и подходов, позволяющих разработчикам создавать код посредством прямого манипулирования визуальными объектами в графическом пользовательском интерфейсе (ГПИ). Самым старым и известным образчиком этого жанра является Visual Basic компании Microsoft. Более поздние продукты, такие как VisualAge компании IBM и Delphi компании Borland, стали вбирать все лучшее из возможностей визуального проектирования. Хотя изначально большинство таких продуктов в основном предназначались для разработчиков приложений, парадигма визуального проектирования может в конце концов стать будущим каждого.

Программное обеспечение для визуальной разработки является всего лишь технологией, но такой технологией, которая может радикально изменить методику разработки программного обеспечения. В самом чистом виде визуальное проектирование позволяет создавать полностью рабочие системы – в значительной степени или исключительно с помощью перемещения визуальных объектов по экрану монитора. Прямое манипулирование графическими элементами является очевидным подходом к раз-

[1] Быть деквалифицированным (фр.). – *Примеч. науч. ред.*

работке графических пользовательских интерфейсов. Именно здесь и началось визуальное проектирование – с применением так называемых конструкторов графических пользовательских интерфейсов. При использовании многих ранних инструментов сложность состояла в том, что за внешней стороной приятного на вид интерфейса скрывался самый отвратительный или беспорядочный код на Basic или C++ из всех, что когда-либо придумывались. Код «за экраном» был в полной неразберихе, и вам приходилось пробираться через это месиво, чтобы заставить работать хоть что-то.

Между ГПИ[1] и гравием

Когда проектируемые системы становятся действительно сложными, разработчики стремятся строить модели на более высоких уровнях абстракции, чтобы описывать и отражать архитектуру системы, а не только ее конструкцию или поверхностные проявления в виде пользовательского интерфейса. Для этого программные единицы и взаимосвязи между ними должны быть представлены визуально. Вам нужно видеть модули, классы и объекты, а также сообщения и ссылки, которые возникают между ними. Вам нужно видеть структуру вашего кода, выраженную с помощью знакомой системы обозначений. Необходима возможность перемещения по этой структуре с помощью устройства, названного Дж. Д. Хилдебрэндом визуальным броузером (J. D. Hildebrand, 1994). Вам нужно, чтобы вы могли, проведя линию, передать информацию из одного места в другое. Вам хочется иметь возможность перенести метод из одного класса в другой с помощью перетаскивания мышью. В общем, вам нужен неизменно активный инструмент CASE с динамическими встроенными механизмами конвертации кода, которые позволяют сразу переводить в код все то, что вы делаете в виде картинок, и наоборот.

Другими словами, вы хотите иметь истинно интегрированную среду визуальной разработки – сочетание CASE-инструментов и приложений WYSIWYG[2], полученное в результате добавления в среды разработки приложений возможностей CASE и более тесной интеграции средств построения ГПИ и генерации кода внутри самих CASE-инструментов. VisualAge представляет собой направление, согласно которому моделирование осуществляется в самих конструкторах приложений. В таких средах с помощью линий можно соединять ГПИ-объекты, а также невидимые объекты, которые скрыты за экраном. В свете основных принципов CASE

---

[1] ГПИ – графический пользовательский интерфейс. – *Примеч. ред.*

[2] WYSIWYG (What You See Is What You Get) – что видишь, то и получаешь. – *Примеч. ред.*

такие инструменты, как Together C++, позволяют синхронизировать все составляющие модели реализации – код, проектные модели, схемы и диаграммы.

Вместо обособления генерации кода от построения модели и графического дизайна истинная графическая среда разработки должна поддерживать в полной синхронизации визуальные элементы системы (ГПИ), ее визуализированную архитектуру (схемы анализа и проектирования) и лежащий в основе всего этого код. Разработчику нужна возможность легко перемещаться между моделями разработки и реализации и говорить на языке либо кода, либо картинок, в зависимости от того, что соответствует текущей задаче, приоритетам или просто его настроению. Такие возможности значительно приблизят инструменты разработки к тому, как люди решают задачи. Обычно люди излагают мысли как с помощью слов, так и с помощью картинок, разрываясь между ясностью высокоуровневой абстракции и мелким гравием деталей решения (см. также главу 18).

Визуальная разработка в полной форме – это важная новая парадигма в отношениях между программистами и компьютерами. Потенциально она позволяет аналитикам и разработчикам применять одни и те же ментальные и ручные навыки для описания задачи, разработки ее решения и построения системы. Это выглядит возвратом к прошлому, когда код проектировался при кодировании, однако в этом подходе проектные решения кодируются с помощью простого проектирования. Действительное отличие, которое несет в себе новая парадигма, заключается в том, что визуальная разработка позволяет создавать и манипулировать на более высоком уровне, большими частями, с большим числом видимых связей и последствий. Это может помочь вам в буквальном смысле видеть, где вы находитесь, что вы делаете и как все это соответствует общей схеме проектируемой системы.

Двойственные процессоры

Визуальное проектирование дает возможность думать в двух режимах одновременно. Первый режим – аналитический, последовательный – сопровождается использованием языка и выражением процессов в виде кода. Второй режим – пространственно-визуальный – мы применяем, когда что-то рисуем, или строим с помощью конструктора Lego, или составляем диаграмму. Таким образом, визуальное программирование напоминает апгрейд центрального процессора вашего мозга, позволяя увеличить скорость ваших ментальных процессов и давая возможность решать более сложные задачи или быстрее строить системы.

Среды визуальной разработки также дают возможность изменять ход разработки. Люди, содействующие продвижению объектной технологии, го-

ворят, что она обеспечивает «бесшовность» процесса разработки – прежде всего потому, что для разработки и описания программного решения применяются те же самые понятия и термины, которые используются для описания задачи и для взаимодействия с пользователями и клиентами. Но это просто объекты. Однако исполнение обещания – совсем другое дело, и традиционные объектно-ориентированные инструменты разработки сами по себе не были настолько бесшовными. Сочетание визуального проектирования с методами объектно-ориентированной разработки и программирования может в конце концов появиться на практике. Вы получите возможность выполнения всей задачи с помощью прямого манипулирования объектами. Кроме того, будущее поколение инструментов может предоставить возможность изменять и расширять сами инструменты при помощи тех же методов визуального программирования, которые вы применяете для построения приложений.

Вместо того чтобы устраивать поминки по CASE, может быть, есть смысл визуализировать процесс разработки.

Из журнала *Software Development*, том 3, № 5, май 1995 г.

24

Цели программного обеспечения

Программист, проспавший большую часть последних десяти или двадцати лет, наверное, не знает о том, что такое объектно-ориентированное программирование. Остальные из нас уже сыты этими объектами по горло. Я испытываю некоторое сочувствие к этим ван Винклям[1] нашей отрасли, поскольку в 1986 году, когда я имел неосторожность вернуться в компьютерную индустрию, объектная технология уже была звездой, восходившей с головокружительной скоростью. А ведь за десять лет до этого, в пору моего последнего отсутствия в компьютерной области, объектно-ориентированное программирование было неизвестным хмурым карликом, скрытым в космосе непонятных компьютерных методов. Конечно, спустя десяток лет очень легко определять, какие тенденции проявлялись ранее.

Нельзя сказать, что объектно-ориентированный подход является особенно новым. Хотя некоторые авторы связывают провал структурных методов с возникновением объектно-ориентированных, на самом деле, структурное программирование и проектирование появилось почти одновременно с объектами, возникнув из одного первобытного болота неструктурных методов. Дейкстра (Dijkstra), Константин (Constantine), Кэй (Kaye), Даал (Dahl) и Сатерленд (Sutherland) – все эти люди работали параллельно в конце 60-х и начале 70-х годов, разрабатывая и распространяя основные понятия новых способов мышления о программах и программировании.

---

[1] Рип ван Винкль (Rip van Winkle) – герой одноименной новеллы В. Ирвинга (W. Irving, 1819), житель голландской колонии в Америке, который проспал 20 лет, выпив волшебное вино, поднесенное ему гномами. – *Примеч. науч. ред.*

К середине 80-х годов концепции объектно-ориентированных методов уже сложились, хотя они и были не всегда понятными. Конечно, не было недостатка и в тех, кто уже тогда хорошо освоил ООП и мог решить с его помощью любую задачу. Не хватало только людей, которые могли бы дать ясные объяснения. В соответствии с риторикой того времени следовало сжиться с объектами, чтобы научиться думать и проектировать с их помощью.

«Как же можно объект уподобить информационному кластеру или модулю с информационным сцеплением?», – может спросить дотошный студент.

«Послушайте, – отвечает нетерпеливый преподаватель, – я же сказал вам забыть обо всей этой старой структурной чепухе. Это не работало. И пока вы не очистите кору головного мозга от этих вещей, вы ничего не поймете». (Подразумевается, что преподаватель даже и не знает, о чем спрашивает студент, поэтому пусть студент закроет рот и слушает.)

Многие из педагогов и демагогов, которые в то время занимались ОО, утверждали, что единственный путь к «объектной эффективности» – это отказ от старых способов. Нужно забыть все, что вы знаете (или думаете, что знаете) о разработке программного обеспечения, и изучить совершенно новую парадигму. Следовало начать все сначала – с чистой грифельной доски в голове.

Если бы это было возможно! Предложения этих людей были больше похожи на религиозное обращение, чем на приобретение новых технических навыков и изучение новых понятий. Некоторые гуру объектно-ориентированной парадигмы все еще проповедуют таким образом. «Их нужно привлекать еще молодыми, пока их сознание еще не замутнено процедурами, и тогда вы сможете сделать из них истинных верующих». Приблизительно так говорят эти евангелисты.

Конечно, вплотную занявшись ОО, я быстро понял, что многие из тех ранних поборников были вынуждены занять такую евангелическую позицию. Ничего не зная о структурных методах или реляционных моделях данных, они не могли сказать что-то разумное о связи этих понятий с ОО и поэтому не могли рисковать, отвечая на серьезные вопросы своих более опытных студентов. Когда они все же упоминали принятые в то время методы разработки программного обеспечения, это всегда были нападки, напоминающие легкую игру в стрельбу по хорошо поставленным и структурированным целям.

Некоторая расплывчатость определений и полное отсутствие консенсуса относительно того, что можно считать основными понятиями в ОО, не улучшали положения бедного странника, ищущего объектно-ориентированного просветления. Однако со временем все установилось, и сегодня есть общепринятая терминология и концепции объектно-ориентированной парадигмы.

Упаковка

В ОО есть множество аспектов. Для некоторых этот подход является способом прожить, для других это образ жизни. Для того чтобы постичь ОО в полном его великолепии, следует понимать не только инкапсуляцию или сокрытие описания, но также и наследование, делегирование, универсальность, динамическое связывание и полиморфизм. (И люди еще говорят, что структурная революция навязала слишком много новых терминов!) Все эти понятия представляют собой важные аспекты данной парадигмы и ее практического применения, но реальная причина того, что в разработке программ объекты ждет большое будущее, намного проще.

В короткой статье невозможно описать все тонкости объектно-ориентированной парадигмы, но главные вопросы, связанные с человеческим фактором в программировании (peopleware), являются довольно простыми. (Написав эти строки, я уже слышу, как стучат клавиши, чтобы подготовить негодующие электронные письма, которые скоро будут переполнять мой ящик.)

Забудьте о шумихе, которую поднимают истинные верующие, говорящие вам, что объектно-ориентированные методы – это новый подход к обдумыванию задач. На самом деле это всего лишь «новый, улучшенный» завтрак из хлопьев – все дело в хорошей упаковке. Классы, которые являются необходимым компонентом объектно-ориентированного программирования, – это всего лишь улучшенные контейнеры для кода. Объектные классы имеют ряд преимуществ в сравнении с функциями и подпрограммами. Во-первых, классы – это большие и средние контейнеры. Чем больше составные блоки, тем большие системы можно построить из данного набора компонентов. Хотя объектные классы имеют большие размеры, они выглядят меньше и проще потому, что все объекты хитрым образом плотно упакованы в соответствии с каким-то одним общим понятием (например, КомпьютерныйКлиент или ЛазерныйПринтер). Благодаря этому образуется необходимая связка, которую легко узнать и перемещать. Да, вы складываете данные и процедуры в один блок, но это как раз те необходимые данные и процедуры, которые все вместе принадлежат одному блоку.

У такого блока даже есть утешительная надпись, во многом напоминающая о реальном мире клиентов и пользователей, что является вторым большим преимуществом объектов. Все остальное – это дополнительные преимущества, лежащие на дне коробки с хлопьями. Брад Кокс (Brad Cox), один из первопроходцев ОО, имел обыкновение говорить всем, кто смотрел в его сторону хотя бы с малейшим интересом, что объектно-ориентированная революция в упаковке – это абсолютно то же самое, что и

переход от дискретной электроники к микросхемам, то есть от транзисторов к чипам. Это небольшая разница, которая имеет большое значение.

Остальные детали этого подхода можно найти у Роланда Рако (Roland Racko), Меилира Пейдж-Джонса (Meilir Page-Jones) и других настоящих ОО-гуру. В своих статьях и книгах они показывают, что даже во всей своей красе ОО не обязательно должно быть трудным – даже для тех заржавелых, старых умов, которые до основания заражены структурными методами.

Субъективное программирование

Основным свойством объектно-ориентированного программирования является сама идея. Она настолько потрясающа, что в конце концов останется практически единственный способ, с помощью которого будет создаваться серьезное программное обеспечение. Поэтому если только вы не занимаетесь разработкой примитивного и глупого программного обеспечения, рано или поздно вы обязательно сориентируетесь. Хороших источников знаний и советов по объектно-ориентированным методам предостаточно.

Если в глубине души вы программист, то, следуя своим наклонностям, вы захотите написать кусок кода. Использование правильного языка может быть очень полезным. В качестве такого языка можно выбрать почти любой, но, вероятно, больше всего для этого подходит Eiffel – язык, который заслуживает большей известности и большего коммерческого успеха. Хотя он и не является самым лучшим языком из тех, что создавались для разработки программного обеспечения, этот язык является виртуальной противоположностью намного более известного языка C++. Eiffel помогает научиться думать и проектировать с помощью объектов, тогда как C++ помогает заставить себя думать, что вы изменились, даже если вы выдаете все тот же старый гуталин. В отличие от C++, Eiffel – это четкий и компактный язык. С помощью Eiffel легко программировать хорошо и трудно программировать плохо. Если бы Eiffel применяли больше производителей и на большем количестве платформ, этот язык мог бы выиграть за счет своей жизнеспособной и разнообразной среды визуальной разработки, а его инструменты поддерживали бы принятые методы и системы обозначений. Вероятно, собственный компилятор кода также был бы полезен. А еще языку не помешало бы, чтобы некоторые из его сторонников были менее жесткими. Smalltalk – больше, Java – это напиток дня, а Eiffel – это тот язык, который по-прежнему любят многие. (*Je suis ton ami*[1], Bertrand.)

[1] Я твой друг (фр.). – *Примеч. науч. ред.*

В качестве учебника по ОО для начинающих программистов я многие годы рекомендовал книгу, написанную человеком, который в конце концов сделал структурное проектирование доступным для обычных смертных. «What Every Programmer Should Know About Object-Oriented Design» (Что должен знать каждый программист об объектно-ориентированном проектировании) (Page-Jones, 1995 [55]) была не первой книгой на эту тему, но никогда не поздно писать о сложных предметах ясно и понятно. (Сейчас ее заменила другая, не менее полезная книга того же автора: «Fundamentals of Object-Oriented Design» (Основы объектно-ориентированного проектирования) (Page-Jones, 2000 [56])).

Так что сориентируйтесь и вы.

Из журнала Software Development, том 3, № 6, июнь 1995 г.

25

Шито белыми нитками

Время от времени в некоторых старых и никуда не годных фантастических фильмах можно заметить швы или застежки-молнии на том, что должно казаться кожным покровом инопланетного мутанта. Несмотря на частые возражения, в объектно-ориентированной разработке программного обеспечения швы нередко не менее заметны, чем на дрянных костюмах пучеглазых инопланетян в низкопробных научно-фантастических картинах. Если судить о процессе по конечному продукту, то обещание бесшовности объектно-ориентированного проектирования в значительной степени остается невыполненным.

Согласно современным мифам об объектной технологии, бесшовное проектирование обеспечивает создание более качественного программного обеспечения с помощью более простого процесса. Каким образом? На протяжении всего процесса разработки применяется один общий и согласованный набор компонентов – понятий предметной области. Понятия представлены согласно их трактовке реальными пользователями, обитающими в сказочной стране, называемой «реальным миром». На основе таких понятий создаются модели, с помощью которых разработчики анализируют и решают свои задачи. Таким образом, понятия воплощаются в самой структуре кода. Разработчики упрощают свою задачу благодаря мышлению в терминах объектов и непрерывному использованию одного и того же набора идей на всех этапах процесса. Так во всяком случае это преподносится.

Однако не только разработчики выиграли бы от бесшовности процесса проектирования, будь она когда-либо материализована на этой планете.

Большая часть моей работы связана с облегчением участи пользователей, ограждением их от страданий и унижений, приносимых плохим программным обеспечением с непостижимыми инопланетными интерфейсами. Пользователи вкушают плоды, которые может дать бесшовное проектирование. В свою очередь, компоненты предметной области, которые заполняют классы и объекты программного обеспечения, могут быть отражены в пользовательских интерфейсах, организованных на основе тех же понятий. Это удивительно, но продукт бесшовной разработки, ориентированный с помощью объектов из мира пользователя, является продуктом, говорящим на его языке. Такой продукт представляет собой узнаваемое продолжение мира, в котором работает пользователь. В нем объединены как раз те предметы, которые пользователь воспринимает или применяет одновременно (см. главы 24 и 28).

Как пользователи, так и разработчики предпочитают программное обеспечение, которое легко освоить и которое не требует активной технической поддержки. Однако для многих организаций оказалось трудным улучшить юзабилити своих продуктов. Пользователи и клиенты не получают удобство и простоту программного обеспечения по многим причинам. Тем не менее можно выделить и общие препятствия, стоящие на пути к успеху. В одних случаях проблемой является сама организация, в других – неэффективные методы и инструменты.

Некоторые организации становятся жертвами нехватки собственной решимости. Эти организации, так же как и их конкуренты, готовы прислушиваться к голосу потребителя или приспосабливать интерфейс под нужды пользователя, если только это не потребует дополнительных расходов или времени. Такие группы могут говорить и говорить в защиту разумных пользовательских интерфейсов или программного обеспечения, дружественного пользователю. Однако без последовательного и устойчивого внимания организации к вопросам юзабилити дело не сдвинется с места. Все чаще мне встречаются организации, в которых разработчики программного обеспечения более привержены улучшению юзабилити, чем само руководство. Такое руководство не выделяет средств на обучение. Оно не считает, что улучшение юзабилити – это часть работы проектировщиков и программистов. Оно отдает предпочтение большему количеству примочек в программе, а не удобству ее использования.

Другие проектные группы, несмотря на свое сильное стремление улучшить юзабилити программного обеспечения, могут испытывать трудности из-за применения неадекватных методов. Например, они могут уповать на тестирование с целью выявления изъянов в юзабилити, а не на хорошие методы проектирования, позволяющие избежать таких изъянов. Они готовы расходовать средства на юзабилити, но не знают, на что именно их следует потратить. В некоторых отношениях таким группам помочь

легче всего – они уже стремятся идти в этом направлении и готовы тратить на это свои ресурсы. Им нужно только показать дорогу. Как только они поймут, как применять эффективные модели, они смогут достичь чрезвычайных успехов с точки зрения юзабилити конечных продуктов.

Время инструментов

Однако даже сильного и целенаправленного стремления к улучшению юзабилити и правильному структурированию процесса бывает недостаточно. Зачастую современные средства разработки сами по себе создают препятствия на пути к повышению юзабилити программного обеспечения и бесшовному проектированию. Некоторые инструменты не только не поддерживают действительно бесшовный процесс разработки, но и затрудняют применение системного подхода к проектированию, который основан на моделировании с учетом юзабилити.

Инструменты для создания программного обеспечения продолжают совершенствоваться, особенно визуальные средства разработки. Начиная с Visual Basic и Delphi и заканчивая Visual C++ и J-Builder, визуальные инструментальные средства сделали большой шаг вперед в области разработки приложений и программного обеспечения. Это относится не только к технологии, но и к тому, как такие инструменты соответствуют методам, с помощью которых люди решают задачи. Лучшие из этих инструментов помогают разработчикам легко перемещаться между визуальным представлением программного продукта (пользовательским интерфейсом) и кодом, лежащим в его основе. Другими словами, они позволяют думать либо о видимых объектах, либо о невидимом коде – в зависимости от того, что требуется для решения текущей задачи.

Швы в инструментах начинают проявляться именно между пользовательским интерфейсом и теми моделями, которые разработчики применяют для представления объектов, взаимосвязей и задач. Обещание полной и совершенной интеграции, о которой шла речь в главе 23, еще не материализовалось. Инструменты моделирования все еще являются лишь инструментами моделирования, а инструменты проектирования – лишь инструментами проектирования. Даже если они импортируют файлы друг у друга и способны обмениваться информацией через API, швы и молнии зачастую очень заметны. Более того, инструменты не способны распознать ключевые связи и отношения. Например, пользовательские ситуации повсеместно признаны в качестве модели, имеющей важное значение для объектно-ориентированного проектирования. Они поддерживаются некоторыми инструментами моделирования, однако связь пользовательских ситуаций с пользовательским интерфейсом не распо-

знается большинством таких инструментов и пребывает исключительно в умах разработчиков.

Что же в таком случае мы хотим получить от наших инструментов, чтобы создавать высококачественные программы с помощью бесшовного проектирования? Нужно, чтобы поставщики инструментов выходили за пределы своих привычных представлений. Нужно, чтобы они поняли саму идею визуального проектирования – идею, которая так долго ждала своего часа (см. главу 18).

Ракурсы

Программную систему можно рассмотреть с различных ракурсов. Каждый ракурс или точка зрения высвечивает определенные характеристики или аспекты системы, игнорируя или затеняя другие. Модель процесса, например, известная всем блок-схема, удобным образом описывает структуру алгоритмов, но почти (или совсем) не показывает данные. Модель предметной области эффективно представляет объектные классы и взаимосвязи между ними, но является бесполезной для дизайна экранных изображений. Тот факт, что общепринятые модели, применяемые в проектировании программного обеспечения, столько внимания уделяют конкретным аспектам систем, является не изъяном, а достоинством. Каждый ракурс позволяет упростить систему для определенных целей, тем самым помогая разработчикам говорить и думать о конкретных вопросах и задачах, возникающих при создании программного обеспечения.

Пользовательский интерфейс сам по себе является одним из таких ракурсов, представляя программу так, как она будет выглядеть с точки зрения конечного пользователя. Контент-модель показывает пользовательский интерфейс с точки зрения проектировщика и моделирует его содержание в отрыве от визуального представления или каких-либо графических интерфейсов (Constantine, 1998 [29]). Карта контекстной навигации представляет собой отображение всех составных частей пользовательского интерфейса и взаимосвязи между ними (более подробно о контент-моделях и навигационных моделях рассказано в главе 44).

На самом деле даже справочные файлы и документация представляют собой альтернативный взгляд на программное обеспечение. Хотя зачастую эти ракурсы считаются у разработчиков раздражающими мелочами, они являются частью пользовательского интерфейса, поскольку служат связующим звеном между пользователями и системой. На практике юзабилити программного обеспечения зависит не только от компоновки элементов на экране и дизайна диалоговых окон, но и от качества справочных файлов и руководств.

Конечно, все модели и ракурсы данной системы очень тесно взаимосвязаны между собой – хотя бы потому, что они описывают одну и ту же систему. Визуальные объекты в пользовательском интерфейсе связаны с внутренними объектами и их методами, а абстрактные компоненты в контент-модели могут проявляться в виде всяких «штучек» в пользовательском интерфейсе. Различные контексты, в которых происходит взаимодействие с пользователем (окна, формы, диалоговые окна и т. п.), связаны переходами, осуществляемыми с помощью элементов управления в пользовательском интерфейсе. Пользовательские ситуации моделируют не только взаимодействие между пользователем и системой, но и взаимодействие между коммуникационными программными объектами.

В редких случаях, когда в наличии есть точная онлайновая и оффлайновая документация, она тоже тесно связана с другими ракурсами. Документация описывает видимые и невидимые характеристики системы с помощью той же терминологии, которая применялась в объектной модели и в дизайне пользовательского интерфейса. Документация сообщает о пользовательских ситуациях, которые можно вызвать с помощью данного программного обеспечения.

Согласно методам разработки программ на основе ракурсов, целью разработчика становится построение системы с помощью различных ракурсов, которые полностью ее описывают и конкретизируют. Назначение инструментов проектирования – поддерживать создание основы программы, обеспечивая согласование между кодом (одним ракурсом) и всеми другими ракурсами. В любой момент каждый ракурс может применяться для обзора и модификации системы. Измените код, и вместе с ним изменится интерфейс. Измените модель, и изменится код. Создадите новую контент-модель, и в интерфейсе появится пустая форма. Добавите пользовательскую ситуацию, и в файле справки появится еще один раздел. Разработчик, который испытывает затруднения в одном ракурсе, может мгновенно переключиться на другой ракурс и продолжить свою работу.

Следы

Один из выигрышей, которые дает разработка с помощью ракурсов и соответствующих инструментов, заключается в увеличении возможностей отслеживания. В разрабатываемом программном обеспечении все взаимосвязано с моделями и документацией, которые описывают систему вплоть до исходных требований к ней.

Пока разработчик устанавливает элементы форм или диалоговых окон, ему необходимо видеть пользовательские ситуации, которые будут поддерживаться в этом контексте взаимодействия. На самом деле в системе должна быть отражена связь каждого шага пользовательской ситуации с

элементом или элементами, предназначенными для ее поддержки. Если контент-модель представляет собой абстрактное описание, являющееся промежуточным звеном между моделью пользовательской ситуации и реализованным пользовательским интерфейсом, то контент-модель должна быть доступной для разработчика, пока он размечает пользовательский интерфейс с помощью инструмента визуального проектирования.

Вообще разработчику нужна возможность время от времени переключаться с одного ракурса на другой посредством тех связей, которые видны разработчику и известны системе. Разработчик должен видеть все пользовательские ситуации, поддерживаемые тем или иным контекстом взаимодействия внутри пользовательского интерфейса, причем для этого должно быть достаточно щелчка или выделения мышью. И наоборот, разработчик должен иметь возможность указать на какой-то шаг в пользовательской ситуации и после этого перейти к визуальному компоненту или компонентам, которые осуществляют этот шаг, или углубиться в объектную модель, чтобы увидеть, с помощью каких объектов он выполняется.

В ходе решения задачи в одном ракурсе разработчик может перейти к другому ракурсу, который более удобен для выработки решения. Разработчик не должен создавать код, чтобы увидеть рабочий результат, или восстанавливать структурную схему по исходным текстам программы, чтобы увидеть модели. Не должно быть необходимости экспортировать модели из одного инструмента и импортировать их в другой. Все ракурсы должны поддерживаться в совершенной синхронности с помощью интегрированной среды проектирования.

Многие аналитики начали применять пользовательские ситуации в качестве основной модели для определения требований. В соответствующей среде и в процессе разработки, пользовательские ситуации позволяют комплексно отслеживать выполнение необходимых условий. Модель пользовательских ситуаций связана с большинством други, важных ракурсов процесса проектирования. Она связана с моделью пользовательских ролей и опирается на нее. Она взаимодействует с контент-моделью и навигационной моделью, представляющими абстрактную структуру пользовательского интерфейса, и, конечно, с самим пользовательским интерфейсом. В плане терминологии она связана с моделью объектных классов или моделью данных, а также с моделью взаимодействия объектов внутри программы. В справочном руководстве пользователя она отображается в виде документов, описывающих пользовательские ситуации и то, как они разыгрываются. Она связана с тестовыми программами, предназначенными для выполнения регрессивных и приемочных испытаний, а также со сценариями для выполнения юзабилити-тестирования.

Безусловно, некоторые инструменты проектирования частично поддерживают такой ракурсный подход, однако ни один инструмент не способен применять его в полной мере. У каждого из инструментов есть свои недостатки. Главным образом они проявляются в наведении мостов между пользовательскими ситуациями и объектными моделями с одной стороны и визуальным дизайном и пользовательским интерфейсом – с другой. Это наиболее значительный шов, накладываемый при создании практичного и удобного программного обеспечения. Пора этот шов разгладить.

Из журнала *Object Magazine*, декабрь, 1997 г.

V

Совершенствование процесса

Введение

Создание программного обеспечения – это процесс. Приложения не случаются ни с того ни с сего, а программы не выскакивают готовыми из голов программистов. Это очень хорошо, так как один из главных принципов в нынешнем движении к «качеству» заключается в том, что процессы могут быть усовершенствованы. Действительно, целью так называемого движения к «абсолютному качеству» и его различных воплощений в мире программного обеспечения является не улучшение шагами или рывками, а непрерывность этого процесса. Самые эффективные организации не только изучают процессы, с помощью которых создается программное обеспечение и приложения, но и возвращают полученные данные в процесс разработки. Такой замкнутый цикл позволяет им непрерывно улучшать свои системы и процедуры.

Достижение этого несколько утопического идеала непрерывного совершенствования может потребовать усилий всей организации. Компании, которые встают на дорогу к просвещению в области разработки программ, часто начинают с тщательной оценки собственной внутренней культуры и традиций. Например, они могут приглашать консультантов для оценивания организации с помощью модели развития функциональных возможностей. Эта модель разработана Институтом проектирования программного обеспечения (Software Engineering Institute) и представляет собой сложную систему рейтингов, выражаемых числами от 1 до 5. Если все, что вам нужно, – это определить, насколько зрелой является организация, то лучше сэкономьте свои деньги. Возможно, вы находитесь на уровне 1, то есть ваши технологические процессы ненадежны и ни на что не годны. Однако это ничего не говорит о том, насколько они ненадежны и в какой степени бесполезны, поскольку никаких измерений не проводится. Почти все находятся на уровне 1, так что по этому поводу можно смело заключать пари. (Если хотите, я могу нанести вам визит или выслать счет до того, как объявят предсказанный мною результат.)

Действительной причиной, по которой следует пережить неудобства и затраты более тщательного и формального исследования, является не же-

лание узнать номер уровня, а желание выяснить более подробно, что и где идет неправильно и с чего нужно начать, чтобы привести все в порядок. Может быть, после нескольких лет работы и траты кучи денег на консультантов вы сможете достичь уровня 3. Распространяемые слухи о том, что где-то в сельских районах Вирджинии есть группа разработчиков с необычным уровнем 5, остаются неподтвержденными. Поймите меня правильно. Я не нападаю на консультантов – я один из них. Я также не подвергаю сомнению возможную выгоду от проведения тщательных оценок организаций – на самом деле моя фирма выполняет такие оценки для своих клиентов. И должен сказать, что я *не имею ничего против* разработчиков программного обеспечения, готовящихся к тому, чтобы вложить большие суммы и ресурсы в совершенствование процессов. Однако я хочу сказать, что, по моему мнению, существуют и другие способы улучшения процессов. Эти способы менее драматичны, не требуют больших затрат и не являются слишком сложными.

В этом разделе мы рассмотрим некоторые из простых идей и скромных предложений, относящихся к улучшению качества программного обеспечения посредством усовершенствования процессов разработки. Речь пойдет не о крупных социальных программах, которые могли бы оправдать астрономически высокие гонорары, выплачиваемые за модернизацию бизнес-процессов. Мы поговорим о том, что небольшие группы или даже отдельные разработчики и менеджеры могут сделать для улучшения своей работы.

26

Преимущества видимости

Помню, как я писал свою первую программу. Это было упражнение по Фортрану в рамках курса, известного под названием 6.41. В те дни в Массачусетском технологическом институте все шло под своими номерами, в том числе и моя программа. Написать ее не составило труда – она всего лишь не скомпилировалась с первого раза. Или со второго. Я был подавлен и опустошен. Когда мне все-таки удалось ее скомпилировать, я был просто шокирован тем, что она не работала. Я не мог понять, почему – мне казалось, что все написано правильно. Я показал ее своему соседу по комнате. Маршалл специализировался на математике. Я думаю, что он начал писать алгоритмы еще до появления компьютеров, на которых их можно было применять.

«Никак не могу понять, где ошибка в этой программе. Посмотри, может быть, это...» Он хватал распечатки, которые мы с любовью называли листингами 80–80 (те, кто помнят, что такое перфокарты, меня поймут).

«А это что?» – спрашивал Маршалл, тыча своим корявым бруклинским пальцем.

«Похоже, что С.»

«Здесь это не нужно.» И он был прав. Этот оператор компилировался, однако компьютер прочитывал его не так, как я предполагал. Маршалл возвращался к чему-то, связанному с градиентами и операторами Гамильтона, и оттачивал свою интерпретацию песни Марлона Брандо (Marlon Brando) из «A Streetcar Named Desire». Из всего этого я уяснил, что Маршаллу никогда нельзя показывать то, что я написал, особенно если это код. Он всегда найдет в нем что-нибудь неправильное, а мне придется на-

ходить в себе силы, чтобы отнестись к этому, как к одолжению. Кроме того, это неизменно вызывало у него очередной приступ музыкального мычания: «Стелла! Стелла!». Скоро начало занятий, а я все еще не был готов. Конечно, со временем я понял, что обсуждение проблемы с приятелем-программистом часто оказывается самым эффективным способом обнаружения какого-нибудь незаметного бага или составления какого-нибудь мудреного алгоритма. На самом деле многие из нас знают, что изложение своих идей коллеге, играющему роль «звукового экрана», или выяснение его мнения по поводу чего-то неясного может быть не только полезной и ценной процедурой для конечного продукта, но и веселым занятием, помогающим построить хорошие рабочие отношения.

Динамические дуэты

П. Дж. Плоджеру (P. J. Plauger) все же пришлось потратить силы, чтобы научить меня пользоваться преимуществами видимости. Вскоре после того как он создал компанию Whitesmith, Ltd., я стал приходить в его нью-йоркскую «штаб-квартиру» – небольшую квартиру на Манхэттене. В углу одной из комнат стоял мини-компьютер, принтер был засунут в шкаф, чтобы не было шума, а по всей комнате, которая считалась «гостиной», были расставлены несколько терминалов. И за каждым терминалом находилось по два программиста!

Конечно, кодировать за каждой клавиатурой мог только один программист, а другие заглядывали ему через плечо и давали надоедливые советы (нью-йоркцы, особенно школьники, называют это kibitzing[1]). В комнате стоял непрерывный гул. Сыпались вопросы по алгоритмам или о правильности начального значения, выдвигались предложения о том, как выйти из цикла, раздавались призывы обратить внимание на синтаксическую ошибку или на тест, выполняемый в неверном порядке, или на неправильный регистр. Время от времени два программиста менялись местами, и тот, который сидел за клавиатурой, теперь становился профессиональным «раздражителем».

Я предположил, что недостаток оборудования связан с нехваткой денежных средств, но Плоджер заверил меня, что на самом деле это был особый режим работы. Довольно неэффективный режим, да? Нет. Приняв такой подход, они стали производить законченный и протестированный код быстрее, чем когда-либо. При близком рассмотрении становится понятно, почему. Код, который выходил из этих терминалов-с-двумя-программистами, был почти на 100% свободен от багов. Программа не только содержала меньше ошибок, но и была более рациональной, краткой и эффек-

[1] От kibitz (амер. англ.) – давать непрошеные советы. – *Примеч. ред.*

тивной благодаря тому, что над ее созданием работали две светлых головы, а также благодаря ценному диалогу, происходящему между сидящими за терминалом напарниками. Такую модель работы я стал называть «динамическим дуэтом». Принцип, который здесь работает, довольно ясен: увеличение рабочей видимости приводит к увеличению качества! Два программиста, работающие в тандеме, – это не излишество, а прямой путь к большей эффективности и лучшему качеству. (Об этом знают те, кто занимается XP (extreme programming, экстремальное программирование.)

Этот же принцип можно применить к процессу обучения. Большинство людей – не все, но большинство – быстрее обучаются в небольших группах, а не самостоятельно. Это относится даже ко многим твердолобым, которые убеждены, что не могут учиться таким способом, или не любят работать в группе. В таком подходе есть множество благоприятных факторов. Обсуждение выявляет темы и идеи, которые никогда бы не пришли в голову одному человеку. Коллеги, общающиеся на равных, зачастую могут объяснить друг другу трудные понятия, если это не может сделать наставник. Новые идеи и точки зрения возникают в самом диалоге. Вероятно, самое главное – это то, что студенты в группах могут учиться друг у друга, а не просто смотреть на учителя или в учебник. Такова одна из причин, по которым на своих семинарах и практических занятиях я почти всегда создаю проектные или учебные группы.

Наверное, это правило особенно подходит для изучения языков программирования. Плоджер заметил, что при изучения языков существует оптимальное количество студентов, которые садятся за один терминал. Это не один студент. Скорее, два. Три тоже подходит, однако в одиночку студент чаще всего изучает язык значительно медленнее, чем в паре в партнером. С другой стороны, четыре и более студентов, сидящих за одним компьютером, всегда мешают друг другу – как в переносном, так и в прямом, физическом смысле. Такие большие группы часто распадаются на более мелкие или начинают применять «разделение машинного времени», что в конечном итоге приводит к снижению эффективности. Поэтому, если вы действительно хотите добавить в свой репертуар язык C++ или Smalltalk, не запирайтесь с руководствами и учебниками, а хватайте своего приятеля-программиста и вместе садитесь за клавиатуру. Знание этого принципа может быть полезным для школ с ограниченным бюджетом, а также для компаний, в которых был расформирован отдел обучения.

Виртуальная видимость

Интересно отметить, что для получения выигрыша от видимости в работе необязательно, чтобы кто-нибудь сказал хоть слово. Большинство из нас

сталкивалось со случаем, когда какая-то ошибка никак не находилась. Вы изучили тестовый прогон и листинг. Вы знаете, что проблема скрыта в одном из блоков кода, но сколько бы вы по нему не проходили, вам не удается обнаружить, где собака зарыта. Отчаявшись, вы стучите в дверь соседней комнаты и настойчиво требуете помощи. В конце концов, Шарлотта имеет степень Ph.D в Computer Science. Вы начинаете описывать ей суть проблемы. Как только вы добираетесь до цикла, что-то бросается вам в глаза. Шарлотта еще не успела ничего сказать, а вы тихо восклицаете: «О!» – и начинаете робко выходить за дверь, на ходу бормоча благодарности. «Пожалуйста, в любое время», – слышите вы в ответ.

Сам процесс объяснения или описания чего-либо кому-либо может изменить наш ход мыслей. Я не знаю, действительно ли для этого нужен кто-то еще. Возможно, достаточно только представить, что вы объясняете проблему кому-либо, хотя мне кажется, что такой способ менее эффективен.

Как и все руководства для построения лучших систем, «принцип видимости в работе» может завести слишком далеко. Одной из форм «доведения до абсурда» (или «расширения» до него?) является модель программирования, к которой прибегают некоторые крупные производители программного обеспечения. Этот подход можно назвать «подходом стаи дворняг». Просто соберите много-много программистов и поместите их в большую комнату – некую «арену для быков», заполненную столами и терминалами. Обеспечьте их всем необходимым для доступа к онлайновым базам данных и электронным рассылкам, чтобы все могли читать информацию и обмениваться мнениями. Все мы знаем, к чему приводят такие подходы.

В большинстве ситуаций преимущества видимости существенно возрастают, если к процессу подключается второй человек, однако с каждым новым человеком отдача снижается. Конечно, бывают и исключения из правил, особенно на самых первых или самых последних стадиях разработки системы. Метод мозгового штурма лучше работает при большом количестве мозгов, способных пробудить этот шторм[1]. Небольшие группы смогут создать только пылевой вихрь или могут просто сесть и посидеть спокойно. В разборе кода и анализе дизайнерских проектов также полезно участие нескольких критиков. Чем больше людей просмотрит работу, тем с большей вероятностью могут быть обнаружены ошибки и упущения. С другой стороны, в реальных задачах, связанных с поиском хорошей архитектуры программного обеспечения при наличии сложных взаимосвязанных ограничений, участие слишком большого количества

[1] Автор обыгрывает разные значения слова «storm». Brainstorming (англ.) – мозговой штурм (групповой метод решения сложных задач). Storm (англ.) – шторм, буря, штурм. – *Примеч. ред.*

умов зачастую приводит к взаимным стычкам. Вероятно, в качестве оптимального значения здесь подойдет известное правило «7 ± 2».

Видимость структуры

Принцип видимости в работе достигает зенита в некоторых специальных моделях командной или групповой работы, предназначенных для проектирования программного обеспечения. В модели под названием «Совместное проектирование приложений» (Joint Application Design, JAD) (Wood и Silver, 1989 [69]) группа конечных пользователей и разработчиков проводят анализ требований и выполняют высокоуровневое проектирование с помощью высокоструктурированного процесса обсуждения. Видимость процесса для пользователей и возможность активно вносить свой вклад приводят к созданию превосходных систем и улучшению взаимопонимания между пользователями и разработчиками.

Так называемая «структурно-открытая» команда, описанная в главе 16 (Constantine, 1989 [11]; Thomsett 1990 [62]), является другой моделью, в которой принцип видимости применяется для улучшения качества системы и, в конечном итоге, эффективности разработки. На протяжении всего рабочего цикла участники такой команды проектирования большую часть своей работы выполняют в присутствии друг друга. Марк Ретиг (Marc Rettig, 1990 [60]) сообщает, что одна успешная команда тратила половину своего рабочего дня на общие собрания. Конечно, это были не просто приятные встречи, а рабочие сессии. Их цель заключалась не в том, чтобы обсудить черновики или похвалить друг друга за успехи, а в том, чтобы заниматься работой. При «структурно-открытой» командной работе члены группы анализируют задачи, модули и даже разрабатывают детали кода. Такие рабочие сессии проводятся под руководством одного из членов команды, что позволяет сделать их более эффективными и плодотворными. Но самую главную роль здесь играет видимость, которая достигается за счет открытости процесса разработки.

Любая группа, разрабатывающая программное обеспечение, может улучшить качество своих продуктов, если повысит видимость процесса программирования. Конечно, некоторые участники будут мешать и сопротивляться этому. Мы все сталкивались с программистами, которые приходят, когда вокруг еще или уже никого нет. Они зашифровывают свои исходные файлы и закрывают собой компьютеры, чтобы кто-нибудь, случайно проходящий мимо, ненароком не увидел их программу.

Помните, Программист Скрытный – это исчезающий вид.

Из журнала *Computer Language Magazine*, том 9, № 2, 1992 г.

27

Повторение и вознаграждение

Многие вещи, начиная от сумок для покупок и заканчивая картриджами, мы используем повторно. Почему бы тогда не попробовать применять повторно код? Почему бы не использовать повторно макеты и модели вместо того, чтобы создавать их заново? Преимущества повторного использования кажутся огромными. Какой код написать дешевле, если не тот, который писать уже не нужно? Высокая степень повторного использования в сочетании с большими библиотеками компонентов позволит удвоить или даже утроить фактическую продуктивность. Все, что нужно сделать, – это полностью изменить культуру разработки программного обеспечения, а может, и особенности характера программистов.

Старые проблемы

Повторное использование вряд ли является новой идеей. Общеизвестные подпрограммы были придуманы для того, чтобы одни и те же инструкции не приходилось переписывать каждый раз, когда требовалось выполнить соответствующие вычисления. Библиотеки повторно используемых компонентов существуют почти столько же лет, сколько люди занимаются программированием. Первый урожай повторного использования был собран в математических процедурах и в управлении вводом-выводом. Ни один разработчик приложений или инструментов больше не пишет своих собственных синус-косинусных подпрограмм, разве что ради удовольствия или из-за упрямого желания сделать это.

Тогда в чем же проблема? К сожалению, большинство программистов любят программировать. Некоторые из них могут даже не есть и не мыться,

а только заниматься программированием. Большинство из них предпочитают писать код, а не рыться в документации, или искать в каталогах, или разбираться в идиотской работе какого-то тупого программиста. Разработчики программного обеспечения создают продукты, а пользователи их применяют. При прочих равных условиях программисты будут разрабатывать и строить с нуля, а не использовать что-либо повторно. Каждый из них убежден, что сможет написать более короткую и более элегантную программу, чем код его предшественника. В силу своего характера программисты имеют предубеждение против повторного использования, даже если оно может повысить их продуктивность. Как побудить их изменить свои привычки? Внезапно с балкона раздается контрапунктное пение хора: «Премии. Рыночные меры. Поощрительные схемы. Гонорары. Графики поощрений. Изменение культуры». Какая приятная какофония!

Поддержка взаймы

На самом деле не так уж и трудно понять всю выгоду. Применение повторного использования в больших масштабах начинается с библиотек повторно используемых компонентов. Существуют только две основные проблемы, связанные с такими библиотеками: помещение компонентов туда и извлечение оттуда! Для того чтобы программисты могли повторно пользоваться компонентами из библиотеки, эти компоненты должны там находиться. Откуда они там возьмутся?

Предлагались и тестировались различные модели. Одна из них заключалась в принятии добровольных вкладов от всех желающих разместить свой код в библиотеке. Некоторые организации сталкивались с трудностями в получении таких вкладов до тех пор, пока не стали предлагать оплату за них или, по крайней мере, обещать не беспокоить авторов по поводу обслуживания или модификации их кода. Такой подход является разновидностью «магазина подержанной книги». Похоже, он не требует больших затрат и позволяет создать большую библиотеку компонентов без прямого вложения денежных средств (или с вложением небольших сумм). Теоретически компоненты должны появляться как побочные продукты в обычных проектах по разработке программного обеспечения.

Такой подход работает, но библиотеки действительно напоминают знакомые нам магазины подержанной книги. Много счастливых часов я потратил, шаря по полкам и коробкам с подержанными книгами, но думаю, что это, скорее, развлечение, а не модель получения информации и повторного использования кода.

Обычно такой подход не позволяет должным образом контролировать качество программ и не предусматривает ответственности. Для привлечения добровольных вкладов программистов освобождают от ответствен-

ности. Необходимая подпрограмма может быть в библиотеке, но, скорее всего, вы ее не найдете. Если на поиск уходит больше 2 минут и 27 секунд, типичный программист решит, что в библиотеке нет того, что он ищет, и будет изобретать это заново.

Компоненты в библиотеках, построенных с помощью открытого приема материала, напоминают тысячи старых книг разных лет и ценности, нагроможденные в беспорядке. Рыться в пыльных стопках книг в поиске неожиданного сокровища может быть интересным занятием, но вряд ли вы станете этим заниматься в условиях жестких сроков. Если вам нужен хороший современный источник по постсоветской экономике, вы зайдете в солидный книжный магазин в университете, а не в подвальную лавку книготорговца, хранящую разнообразные экзотические экземпляры.

Меилир Пейдж-Джонс (Meilir Page-Jones) рассказывает об одном клиенте, у которого библиотека «подержанных» компонентов стала чересчур громоздкой. Руководство решило назначить библиотекаря, чьей обязанностью было *извлечение* компонентов *из* библиотеки и контроль за тем, чтобы в хранилище включались только самые лучшие подпрограммы. Легенда сообщает, что этого человека стали называть Конан-Библиотекарем (Conan the Librarian). Тем не менее для эффективности такого подхода должны быть серьезные силы, которые побуждали бы разработчиков вносить свои пожертвования в библиотеку.

Гонорар программиста

С другой стороны, некоторые компании не смущают авансовые расходы. Они формируют группы штатных разработчиков, чьей обязанностью становится создание компонентов для повторного использования. Эти разработчики – не обычные ворчливые кодеры, а высококвалифицированные специалисты со способностями к выявлению общего, описанию абстракций и построению пуленепробиваемого кода в конкретной предметной области. Они получают вознаграждение за создание качественных компонентов, допускающих высокую степень повторного использования. Они могут даже получать гонорар за каждый случай применения какого-либо из компонентов, созданных ими для библиотеки.

Если такие разработчики просто получают зарплату, то в их интересах может быть создание более изощренных и совершенных компонентов, чем это необходимо, поскольку одна из самых трудных частей работы – это понять, что именно нужно создать. Вначале в библиотеку попадает несколько сотен небольших и универсальных компонентов общего назначения, но по мере расширения библиотеки становится все сложнее определить, что нужно включить туда еще. Никто не захочет слишком быстро сворачивать какой-нибудь проект по разработке компонентов, потому что

потом придется либо возвращаться к творчеству, либо сидеть сложа руки с риском получить замечание.

Если снова обратиться к терминологии книжной торговли, эту модель можно представить как модель «школьные учебники». В этой модели команды специалистов стараются определить, что следует преподавать и как, и работают над книгами коллегиально. Результаты часто оказываются великолепными и отличаются яркой графикой, таблицами, упражнениями и удобными примечаниями на полях каждой третьей страницы. К учебникам прилагаются всевозможные рабочие тетради, руководства для учителей, тексты для дополнительного чтения и наглядные пособия. Обычно они неинтересны, безвкусны и даже скучны, а зачастую совершенно не отвечают реальным нуждам школ и учеников.

Авторские гонорары за повторное использование ставят другие проблемы. Например, отдача от компонентов, за которые выплачены гонорары (в виде наличных денег или кредита), обычно слишком запаздывает. Деньги уже потрачены – не только на те компоненты, которые часто используются повторно, но и на те, которые оказались бесполезными. Для эффективности процесса отбора необходим большой набор компонентов, соответствующий разнообразным требованиям рынка или среды. Это означает, что средние библиотеки будут достаточно большими. В них будет много посредственных или плохих компонентов, из которых можно отобрать лишь небольшое количество «хороших». Трудно даже понять, как проводить такой процесс отбора. Всегда ли часто используемый компонент является подходящим? Может быть, его часто применяют потому, что другого в библиотеке нет? А может быть, частота обращения к нему отражает его позицию в индексе и броузере? А может, его просто лучше разрекламировали, хотя существует меньший по размеру и более мощный вариант?

И что считать «использованием»? Каждый вызов или конкретизацию? Каждый случай наследования или ссылки? Одна программа может 130 раз обратиться к компоненту, который ни разу не применялся ни в одном продукте, а другой компонент может однократно применяться в каждой из 27 систем, разработанных в течение года. Какой компонент является более полезным? Какой продукт заслуживает большего гонорара: простой, понятный и часто используемый компонент или искусный и оригинальный класс, который помог сэкономить несколько дней работы в одном проекте?

Приобретенный вкус

Конечно, возможны и другие модели. Возьмем, например, специалиста по закупке новых книг в публичной библиотеке. Его обязанности – сле-

дить за интересами и потребностями читателей, а также тенденциями в книгоиздательстве. В соответствии с ними он обеспечивает пополнение библиотечных фондов. Подобно магазину подержанной книги, публичная библиотека собирается из компонентов (книг), которые уже изданы и не имеют повреждений. Такие компоненты включаются в библиотеку без редактирования и пересмотра.

Для нахождения модели, наиболее подходящей для библиотеки программных компонентов повторного использования, нам нужно пристальнее рассмотреть книгоиздательство и порядок приобретения работ, предназначенных для печати. В издательском деле редакторы серий и специалисты по приобретению книг играют активную роль в поиске новых имен. Они сотрудничают с авторами не только в соответствии с текущим спросом или потребностями читателей, но и для поддержания более полного «списка книг» и опережения будущих требований аудитории. Эта модель, или идея Пейдж-Джонса (Page-Jones) об «искусном покупателе», который не только приобретает работы, но и заказывает их, возможно, больше соответствует модели, необходимой для индустрии программного обеспечения.

Простое коллекционирование *разумных* компонентов не создаст библиотеку компонентов повторного использования. Нужно еще убедить разработчиков применять эти компоненты, а не изобретать их заново. Во многих компаниях для этого разработаны системы надбавок и премий, однако я не уверен в том, что они необходимы для продуктивного повторного использования компонентов. Более того, в конечном итоге такие схемы даже могут оказаться непродуктивными. Иногда бывает достаточно просто предоставить информацию о производительности. В одной организации понадобилось лишь изменить способ информирования группы о степени продуктивности. Инвестировав средства в развитие библиотеки компонентов, они были разочарованы из-за ее ограниченного использования. Они ежемесячно вывешивали лист с диаграммами, которые отражали продуктивность программистов, измеренную в строках написанного и сданного кода. Вместо этого их убедили сообщать об общем количестве строк, взятых из библиотеки компонентов и включенных в сданный код. Уровень повторного использования резко возрос. Однако является ли количество-строк-библиотечного-кода правильной меркой повторного использования? Возможно, корректное, разумное и уместное повторное использование – это намного более тонкий показатель, который сложно описать и измерить.

Даже если мы получим правильный показатель, сопоставление уровня повторного использования с материальным вознаграждением может создать проблемы. Тесная связь между простым и измеримым количеством обращений к компонентам и прямым поощрением, будь то кредит или

ежеквартальная премия, может подорвать профессиональные ценности, на которых основано эффективное повторное использование. Когда работники начинают подстраивать свои действия под поощрительные схемы, то значение базовых ценностей и качественных подходов снижается, а зависимость от хорошо налаженной структуры вознаграждений возрастает.

Вот почему в одной из компьютерных компаний, логотип которой состоит из трех заглавных букв (IBM), деятельность группы, обеспечивающей возможность повторного использования, сосредоточена на создании особого корпоративного климата и культуры. Создаются условия, способствующие повторному использованию компонентов, что является общепринятой практикой в мире программ и программирования. Наличие или отсутствие таких условий может соответственно облегчить или затруднить процесс совместного использования замыслов и реализаций. Наверное, мне повезло в том, что мои мозги промыли учителя, считавшие, что в их интересах и в интересах наших будущих работодателей не заниматься изобретением колеса. Сначала в системах обработки данных по ядерной физике, а потом в стандартных прикладных программах для бизнеса мы создали и широко применяли библиотеки компонентов повторного использования. Этот опыт показал мне, что богатая библиотека компонентов в сочетании с эффективными инструментами сама по себе является вознаграждением. Польза здесь состоит в том, что вы можете в разумный срок находить нужный вам компонент, а затем легко его применять или адаптировать под текущую задачу. Каждый раз, когда это происходит, ваша решимость обращаться к архиву усиливается. Привычка заглядывать сначала в репозитарий или библиотеку становится естественной и перестает быть связанной с получением денежного вознаграждения или квартальной премии.

Наверное, хитроумные трюки и дополнительные вознаграждения более важны для слабых библиотек и неудобных инструментов. Возможно, это сигнализирует о необходимости пересмотреть корпоративную культуру и профессиональные ценности, которые в ней поощряются или не одобряются.

Из журнала *Computer Language Magazine*, том 9, № 7, 1992 г.

28

Суперобучение

Вы собираетесь изучить Smalltalk, или стать специалистом по Java, или освоить UML, но в сутках есть только двадцать четыре часа. Вы уже опаздываете со сдачей проекта «клиент-сервер», а ваши дети хотели бы хоть иногда видеть вас до того, как закончат школу. Попробуйте суперобучение! Как утверждает реклама, вы сможете научиться объектно-ориентированному программированию за один час. Или иностранный язык – за три секунды. Суперобучение, ускоренные методы, мультисенсорное обучение. Новые методы, разработанные то ли русскими, то ли болгарами, то ли техасцами. Разве не здорово? Надеваете наушники, клюете носом и просыпаетесь, уже зная COM или API в Windows. Или проскальзываете на лекцию, недолго похрапываете и уходите, думая «объектами», словно всегда только этим и занимались. Хотите?

Сегодня это называется методами обучения во сне и подсознательного перепрограммирования бессознательного, но все начиналось с программ аудиотренинга, курсов изучения иностранных языков на кассетах. А еще раньше были долгоиграющие пластинки и даже пластинки на 78 оборотов, что возвращает меня во времена запуска Спутника.

Запуск языка

Спутник совершенно изменил мою жизнь. В 1957 году русские удивили и воодушевили весь мир, запустив первый искусственный спутник планеты. Это перевернуло представления американцев о космосе, о русских и о мозговитой молодежи, любящей науку. До запуска Спутника я был недо-

оцененным неудачником. Почти за один день я стал неудачником, оцененным по достоинству.

И я решил изучать русский язык. Курсов по этому языку не было, поэтому я заказал недавно изданный, но уже устаревший курс «живого русского языка» на трех долгоиграющих пластинках. Надпись на конверте обещала быстрое и легкое освоение этого трудного языка. Уже через несколько месяцев я знал русский алфавит и мог на ломаном русском произнести несколько предложений, однако по-настоящему язык так и не выучил. Это трудно и требует много времени. На самом деле мне понадобилось совершить три консультационных поездки в Россию, прежде чем я смог говорить на улицах Москвы и Ленинграда. Но даже тогда все думали, что я говорю по-русски с болгарским акцентом.

Конечно, изучить Smalltalk намного легче, чем русский язык. Однако ни тот ни другой язык невозможно выучить во сне. Несмотря на то что вокруг такого обучения возникла целая индустрия, не существует никакого надежного или непротиворечивого подтверждения того, что процесс обучения можно ускорить какими-либо формами музыки или ритма, или инфразвуковых импульсов. Нет подтверждений и тому, что информация, записанная на пороге слышимости на (или под?) музыку в стиле New Age, помогает повысить продажи или сбросить вес, или стать более уверенным в себе. Никакие средства подобного рода не стоят денег, исчезающих с вашей кредитной карты. Они выгодны лишь тем, кто предлагает такое «суперобучение».

Итак, как же человек изучает сложные языки или языки программирования? Как человек учится программировать или лечить людей? Как человек обучается думать в терминах процедур или объектных классов, или шаблонами блокированных коммуникаций?

Экономист Кеннет Боулдинг (Kenneth Boulding), один из создателей современной теории систем, сказал, что ноу-хау – работающее знание – это не то же самое, что знание. Знание, даже очень сложное знание, может быть получено многими способами, однако ноу-хау требует обучения на собственном опыте. Вы можете много узнать о программировании или психотерапии с помощью чтения, или наблюдения за другими людьми, или посещения ярких ознакомительных показов. Однако стать программистом или терапевтом вы сможете только в том случае, если будете писать программы или лечить людей.

Наверное, лекции – это самый неэффективный и неподходящий способ обучения. По очевидным причинам они почти бесполезны для передачи ноу-хау. Учителя и инструкторы должны отвечать за то, насколько успешно им удается передать информацию, а не за то, какой объем материала они смогли поверхностно затронуть. Когда преподаватель бубнит за-

нудным голосом об одном, потом о другом, затем о третьем и читает из списка, содержащего множество пунктов, мало кто из присутствующих почерпнет много информации.

А как насчет мультимедийных презентаций? Я был преподавателем в Институте исследования систем (Systems Research Institute) в IBM, когда приблизительно в 1970 году Джеймс Мартин (James Martin) поразил коллег и студентов тем, что использовал не один, а целых два проектора для 35-миллиметровых слайдов. С тех пор многое изменилось. Теперь уже все применяют по два проектора! Теперь, чтобы быть впереди всех, требуется намного больше. Цвет. Звук. Графика. Анимация. Игры.

Быстрое изучение

Есть ли польза от мультимедиа? Помогают ли цветные анимации узнать больше или быстрее? Мигающий свет и яркие рубашки действительно помогают. По крайней мере, аудитория дольше не засыпает, что очень важно, так как обучение во сне не работает. Визуальное и звуковое акцентирование может способствовать запоминанию. Однако эти средства работают лишь в том случае, если они тесно, неразрывно связаны с тем, что предназначено для запоминания. Я хорошо помню эффектное видеопреобразование, которое видел на лекции в прошлом году, но совершенно не помню о чем, собственно, шла речь. Хотя мне понравилось то, как машина превращалась в стодолларовую купюру.

Для облегчения процесса обучения одних только фенечек и фитюлек недостаточно. Если мультимедиа, мультисенсорная коммуникация действительно могут принести хоть какую-то пользу, то она происходит от тщательного, точного и уместного применения каждого средства. На заре структурной революции я работал с компанией, которая занималась производством мультимедийных обучающих пакетов по различным темам в области разработки программного обеспечения. Печатные, аудио- и видеоматериалы усиливали и дополняли друг друга, поскольку каждый элемент в процессе обучения был представлен именно в той форме, которая наиболее эффективно передавала соответствующие понятия или умения.

Более десяти лет я занимался тем, что обучал людей одним из самых трудных и сложных межличностных и когнитивных навыков – пониманию семей как работающих систем и оказанию помощи тем семьям, которые работали недостаточно хорошо (Constantine, 1986 [10]). Поэтому я знаю, что людям действительно можно помочь изменить их взгляды на окружающий мир и понимание этого мира. На современном жаргоне это называется «сдвигом парадигмы». Однако это происходит не за час, и даже не за день.

В чем же суть всей этой мультисенсорности и мультимодальности? Существуют различные стили обучения. Одни люди лучше обучаются через прослушивание, другие – через чтение, третьи – через просмотр. А некоторым подходит только самостоятельное изучение. Для одних людей изображение стоит тысячи слов, а другим оно заменяет только 17 или 18. Некоторым «аудиалам» графические или визуальные средства могут быть бесполезны. Одновременное воздействие на разные каналы восприятия увеличивает шансы на то, что все слова, образы и навыки останутся в памяти. Один канал восприятия может усилить обучающий эффект в другом.

Какая-то информация запоминается легче и на более длительное время. Запахи могут оставлять в памяти очень долговременные следы даже после однократного вдыхания. Пространственно-моторное запоминание, то есть память о выполненных движениях и действиях, не просто долговечна – она может способствовать вспоминанию мыслей и ощущений, связанных с движениями. Физическое обращение с предметами, манипулирование материалами, которые ясно представляют некоторые идеи, помогает закрепить понятия – при условии, что физические манипуляции непосредственно с ними связаны.

Мультимодальная коммуникация не обязательно должна вовлекать сложные мультимедийные презентации. Например, во время дискуссии, проходившей на конференции в Лондоне, на вопрос о связи между повторным использованием и объектной технологией я не раздумывая ответил с помощью удерживания двух карандашей в виде буквы «L». Таким способом я хотел показать, что эти концепции – ортогональные, то есть не зависят друг от друга. Позже, уступая под градом острых вопросов, я признал взаимосвязь этих концепций. Объектная технология может способствовать более эффективному повторному использованию. Я составил из карандашей широкую букву «V», чтобы показать, что эти концепции связаны друг с другом. Два дня спустя участники конференции все еще обсуждали эту мини-демонстрацию.

Взаимосвязь между повторным использованием и объектной технологией продемонстрировать легко. Хорошая презентация, прежде всего, должна делать простые вещи простыми в изучении. Трудные вещи всегда будут трудными. Сколько бы систем восприятия вы не применяли, вы не сможете изучить объектно-ориентированное проектирование за один час так же, как не сможете изучить за час программирование. За это время вы сможете узнать только упрощенные – некоторые сказали бы «разбавленные» – варианты основных понятий.

Из журнала *Software Development*, том 1, № 11, ноябрь 1993 г.

29

Вверх по водопаду

Может ли роботам сниться электрическая овца? Преследуют ли менеджеров программных проектов кошмары о прыжках с водопада?

В традиционном виде цикл разработки программного обеспечения проходит через ряд последовательных этапов, начиная от определения требований, анализа, проектирования и заканчивая построением и тестированием. Высокоуровневое проектирование завершается до начала детального проектирования. Анализ задач и проектирование необходимо проводить до рассмотрения вопросов кодирования. Процесс разработки постепенно переходит с верхних уровней абстрагирования к низкоуровневым деталям, от общего и абстрактного к частному и конкретному.

Конечно, на самом деле так обычно не происходит. Так называемая «водопадная» модель разработки программного обеспечения все еще подвергается жарким обсуждениям. Порой она кажется кошмарной частью нашей коллективной мифологии. Бросаясь вниз с водопада, вы теряете контроль над своим движением и направлением. Вы просто летите вниз, хотите вы того или нет. Самое большее, на что вы можете надеяться – не утонуть. Я могу взять часть вины на себя за то, что когда-то давно ввел такие чрезмерно упрощенные понятия. Однако в сравнении с неуправляемым хаосом, который царил в то время, даже такие примитивные модели последовательного цикла работы были прогрессом. С тех пор мало что изменилось.

Спешка

Разработчики программного обеспечения и другие представители индустрии программирования отличаются тем, что они всегда бегут впереди

себя. Едва увидев титульный лист спецификации, они тут же начинают придумывать код или дизайн интерфейса. Анализируя абстрактные сценарии использования, они уже представляют иконки для панели инструментов. При планировании связей между основными модулями они начинают придумывать оригинальные способы применения программного интерфейса.

Так обычно и происходит. По сути, эта зарисовка показывает, как обычные люди обычно решают задачи. В известном смысле они начинают работу с двух концов и двигаются к середине. Они мечутся между целями и средствами и перескакивают от высокоуровневой абстракции к низкоуровневым деталям и обратно.

Но разработчики не обязаны действовать таким образом. В больших и сложных проектах забегание вперед может создать реальные проблемы. В спешке вам и вашему руководителю трудно оценить длительность проекта. Слишком раннее рассмотрение деталей впоследствии может привести к необходимости их изменения, вызывая поток других переделок и исправлений. Кроме того, углубление в детали раньше времени может отвлечь внимание от основной работы.

Традиционно у разработчиков и руководителей проектов было две альтернативы, когда возникал этот импульс забежать вперед. В этом случае они могли либо поддерживать дисциплину и преодолеть этот порыв, либо поддаться ему, натворить дел, а потом опять вернуться на правильный путь. Любой путь связан с риском. Если продолжать работу над текущей задачей и не отвлекаться, то можно потерять или забыть важные и полезные идеи. Если вы всегда спешите и переключаетесь на детали, как только о них подумаете, то вы можете никогда не вернуться в основной поток. А если вернетесь, то можете не обнаружить основу вашего великолепного замысла.

Рационализированная реальность

В действительности нам нужна модель рабочего цикла, которая вбирает в себя преимущества всех способов мышления и действий. В то же время такая модель не должна позволять разработчикам становиться врагами самим себе. Сложность заключается в том, как, не нарушая хода решения текущей задачи, собрать достаточное количество информации, которая пригодится впоследствии.

Недавно мой шеф и я работали над дизайном пользовательского интерфейса к апплету для настольной системы. Мы старались действовать как дисциплинированные разработчики, применяя методичную стратегию проектирования интерфейса. Однако мы неоднократно спотыкались о

собственное стремление забежать вперед. Мы постоянно обдумывали отличные идеи, связанные с деталями интерфейса и их особенностями. Это происходило в то время, когда нам нужно было заниматься определением абстрактных сценариев, описывающих потребности пользователей.

Мы твердо решили сохранять организованность и методичность в своей работе, но хотели извлечь наибольшую пользу из того, что нам удавалось само собой. Мы стали использовать «корзины», в которые складывали все замечательные идеи, возникающие в самый неподходящий момент. Корзины, изначально предназначенные для ведения собраний, являются идеальным и простым инструментом для группового решения задач. Корзина – это место (папка, или файл, или блокнот), куда заносятся идеи, не относящиеся к текущему обсуждению, но которые все-таки нужно рассмотреть.

В конце концов, мы придумали новую концепцию цикла разработки, назвав ее моделью «задел-возврат» («feed-forward/work-back»). Это улучшенная преемница различных моделей последовательной разработки, включая не только модель «водопад», но и так называемую модель «водоворот», предусматривающую разработку по спирали. Модель «задел-возврат» – это как раз одна из тех очень маленьких идей, которые могут существенно изменить порядок работы. В принципе, эта модель может быть интегрирована практически в любую другую модель цикла разработки программного обеспечения.

Челнок

Свое название модель «задел-возврат» получила из-за тех циклов, которые она вносит в процесс разработки. Когда вы забегаете вперед, вы «забрасываете» информацию «в будущее», но саму работу не выполняете. Это этап «задел». Когда вы замечаете упущения или ошибки, совершенные ранее, вы сразу же их исправляете. Это этап «возврат». «Задел» идей на будущее и работа в обратном направлении. Вот и все. Каждый раз при забегании вперед вы делаете заметку для себя и коллег. Позже на соответствующем этапе разработки вы рассматриваете эту идею.

Можно представить, что у каждого этапа в цикле разработки программного обеспечения есть свой собственный ящик. Не имеет значения, какую именно модель цикла разработки вы применяете, сколько видов работы она предусматривает и даже то, является ли она строго последовательной или допускает возвращение к предыдущим этапам. Для каждого этапа или шага, или вида работы вы создаете хранилище идей, «корзину для входящих сообщений». В ней будут накапливаться идеи, которые подлежат рассмотрению и обработке в свое время.

Важно создать настоящий журнал или файл, который служил бы в качестве такой корзины. Если вы применяете неформальные групповые методы, то заведите в журнале отдельные листы для каждого этапа или вида работы. Если в вашем цикле разработки есть что-нибудь вроде «Подтверждения физического проекта» («Physical Design Validation»), то заведите лист с названием «Подтверждение физического проекта – Входящие сообщения». Если вы используете системы обработки документов или инструменты CASE, то создайте для каждой корзины свой файл, или папку, или другой документ.

Когда вы достигаете определенного этапа или вида работы, вы начинаете с просмотра соответствующей корзины и сортировки ее содержимого. Некоторые идеи могут оказаться неподходящими. Часть из них уже была применена или отклонена. Идеи, казавшиеся удачными, могут быть признаны неэффективными. Все остальное, признанное действенным и ярким, теперь можно применить в работе.

Иногда отклонения от основного потока разработки ведут в другом направлении. Это касается тех вопросов, которые следовало рассмотреть ранее. Ясно, что в таких случаях самый верный путь – сразу же заняться этими вопросами, чтобы процесс разработки продолжался без осложнений. Множество скрытых проблем или нерешенных вопросов не должны создавать неразбериху в системе. Если вы обнаружили, что некоторое требование является двусмысленным, то вам следует вернуться назад и задать требования более четко. Если выясняется, что некоторые вопросы системной архитектуры остались нерешенными, то вам следует решить их. Когда вы убеждаетесь, что схема файловой организации была выбрана неверно, вы заново разрабатываете файловую структуру перед тем, как продолжить работу.

Безусловно, наша цель – эффективное применение «заделов» для сокращения объема «возвратов».

Из журнала *Software Development*, том 2, № 1, январь 1994 г.

30

Своевременная поставка

Как и предупреждали представители авиакомпании «Alitalia», наш вылет из Бостона в Рим задержался. Тем не менее мы приземлились вовремя, чтобы успеть на прямой поезд до Флоренции. Там мы собирались провести несколько дней, чтобы насладиться искусством, едой и винами Тосканы. После этого нам предстояло вернуться в Рим для проведения классов по разработке программного обеспечения, ориентированного на пользователя.

Обратите внимание: я не сказал, что наш самолет приземлился «по расписанию». Он приземлился «вовремя». Кроме тонкого различия в понятиях, здесь есть и культурологический вопрос большой важности. Прибытие в 6 ч. 30 мин. на прием, назначенный на 6 ч. 30 мин., есть прибытие «по расписанию». Прибытие в то время, когда очередь у стойки бара уменьшилась и уже выносят горячую *hors d'oeuvres*[1], означает прибытие «вовремя». Вовремя подразумевает функциональную своевременность. Об итальянцах нельзя сказать, что они не способны действовать в атмосфере неотложности, например при приготовлении макарон, когда нельзя пропускать *al dente*[2]. В Италии, как и во многих других странах мира, культурологическое ощущение времени не совпадает с реальным временем. Немного раньше, немного позже – какая разница, важен результат, конечный итог. Северный сосед Италии, Швейцария, является ее полным культурологическим антиподом. В этой стране движение поездов и

---

[1] Закуска (фр.). – *Примеч. науч. ред.*

[2] Al dente (ит.) – в итальянской кухне: момент варки, когда макароны находятся в идеальной готовности. – *Примеч. науч. ред.*

жизнь людей происходит по расписанию, по которому можно сверять швейцарские часы.

Разработка программного обеспечения и прикладных программ тоже может находиться в этом культурологическом континууме. На одном конце – разработка «по расписанию» в соответствии с календарными сроками. На другом – разработка «вовремя», при которой разумных результатов важно достичь к сроку, когда они еще имеют ценность для пользователей. В некоторых случаях внешние силы задают скорость движения стрелок на часах. Некоторые приложения для федеральных учреждений и властных структур отдельных штатов включают в себя определенные компоненты, которые должны быть готовы к юридически установленным датам. Но даже в этом случае сдвоенные часы разработки «по расписанию» и разработки «вовремя» отсчитывают одно время. Новая редакция пакета по подготовке налоговых документов должна быть готова строго к 15 апреля, но на практике не ясно, следует ли начать поставку уже в начале января, или же поставка в середине февраля также отвечает потребностям пользователей. Тем не менее сегодня почти во всех случаях руководство настаивает на том, чтобы программное обеспечение поставлялось по расписанию. Крайние сроки исполнения приближаются даже быстрее, чем сроки, диктуемые жесткой конкуренцией, которая царит в мире программирования с ограниченным финансированием.

Быстро и разумно

В результате быстрое проектирование приложений стало одним из методологических безумств в нашем деле. Процесс разработки зажат в тиски абсолютных календарных планов по продвижению и сдаче проекта. Сдача в срок становится главнейшим, если не единственным критерием успеха проекта. Высокопроизводительные «SWAT[1]»-команды лучших программистов вооружаются новейшими программными инструментами, позволяющими сократить время цикла разработки продукта. Осуществление проекта в какой-то выдуманный, но жестко заданный рынком период времени становится важнее других целей. Когда подходит крайний срок, сдается все, что получилось, – вместе с ошибками, изъянами и всем остальным.

Действительно, без предельных сроков и определенных целей некоторые команды могут никогда не справиться с реализацией проекта. Они либо застынут в аналитическом параличе, либо, стремясь достичь идеального результата при непрерывно меняющейся технологии, окажутся выбро-

[1] Special Weapons and Tactics (team) – «команды быстрого реагирования». – *Примеч. ред.*

шенными на остров устаревших систем. Для некоторых людей ничто не является таким стимулом к командной работе и продуктивности, как ужас приближающихся сроков.

На одной конференции по программному обеспечению я был свидетелем крайнего проявления культуры быстрой разработки. Представьте: начальник вручает вам спецификацию нового приложения как раз в тот момент, когда вы входите в комнату, и говорит, что через 40 минут вы должны представить рабочую программу. Здесь уже не до моделей и методов. Это авральное программирование, не больше и не меньше.

Такой режим был условием на «Суперкубке по Visual C++» на Software Development'94 (с тех пор он стал обычным явлением на конференциях). Толпившаяся аудитория наблюдала за тем, как команды программистов из Microsoft и Borland с молниеносной скоростью пишут программы (программисты из Symantec на этом соревновании не появились). Ричард Хэйл Шо (Richard Hale Shaw) из журнала *PC Week* играл роль Донахью (Donahue), в то время как судьи и публика то изумлялись, то смеялись, то аплодировали. Словом, как сказал мой друг и штатный судья Дж. Д. Хилдебрэнд (J. D. Hildebrand) из журнала *Windows Tech Journal*, это было захватывающее зрелище. К тому времени я уже забыл, насколько веселым может быть занятие программированием. Конечно, мне не приходилось демонстрировать «штамповку кода» на 12-футовом экране перед толпой в 1100 человек. Хотя не так уж и много программистов сталкиваются с такими экстремальными ситуациями, как эта, все большее число разработчиков чувствуют приставленный к виску пистолет, вынуждающий их создавать программное обеспечение все быстрее и быстрее. На занятиях и во время консультаций все больше и больше программистов говорят мне о драконовских сроках и усиливающемся давлении графиков, словно вручаемых с Олимпа. Часто я слышу: «Мы хотим тестировать более тщательно, но нам сказано сдавать все как есть» или «Мы хотели бы сделать более удобный интерфейс, но босс не позволяет», или «У нас нет времени на то, чтобы сделать это как следует, – у нас едва есть время на то, чтобы сделать это вообще».

Действительно, так называемая быстрая разработка приложений возможна, но сроки, назначаемые для выполнения такой разработки, часто оказываются нереальными. Руководители и представители отдела маркетинга берут целевые даты исполнения из воздуха. Я помню, как члены одной команды сказали мне, что им назначен абсолютный срок сдачи проекта через три месяца, и поэтому они не могут тратить время на моделирование требований потребителей или разработку пользовательского интерфейса в полном соответствии с требованиями. Конечно, этот абсолютный срок был уже дважды продлен на три месяца!

Просто делайте это

Я начал заниматься программированием именно в этой атмосфере, разрабатывая стандартные деловые приложения по контрактам с фиксированными расценками. Нам не нужно было обладать ясновидением, чтобы увидеть, что большая часть нашего времени снова и снова уходила на решение одних и тех же задач. Не требовалось быть семи пядей во лбу, чтобы понять, что библиотека повторно используемых компонентов, выполняющих основные операции по обработке деловой информации, позволила бы строить системы быстрее и дешевле. Однако руководство не давало нам времени на программирование такой библиотеки. Значение имело только то время, которое могло принести деньги. У нас не было времени на построение инфраструктуры. Высшее руководство настаивало на том, чтобы мы просто писали код и «отгружали» готовое программное обеспечение.

И все же мы выкраивали время и строили эту библиотеку. Мы создавали новые компоненты во время работы над другими проектами. Конечно, иногда мы выбивались из графиков, но в то время так было почти всегда. В конце концов мы создали свою библиотеку и стали демонстрировать весьма высокий рост производительности. На самом деле вам не нужно спрашивать разрешение на то, чтобы хорошо делать свою работу. Если вы знаете, что оценка сроков исполнения проекта нереалистична, то бессмысленно «срезать углы» на анализе и проектировании. Так или иначе, сейчас или потом, но вы к ним вернетесь. Поскольку более полное понимание задачи и более тщательное проектирование в конечном итоге обычно снижает расходы, действуйте так, *как будто* у вас достаточно времени. Зачастую это может оказаться наилучшей стратегией. В долгосрочной перспективе, когда откладывание сроков сдачи неизбежно, издержки компаний-производителей и их разработчиков из-за поставки плохих продуктов будут больше, чем от просроченной поставки хорошего программного обеспечения. Как говорится, если у вас нет времени на то, чтобы сделать это хорошо, то где же вы найдете время на то, чтобы это переделать?

Поставка достойного программного обеспечения вовремя – это именно то, что в конечном итоге имеет значение для делового и профессионального успеха. Разработчикам никогда не следует принимать как данность нереалистичные сроки исполнения. Для выполнения своих истинных обязанностей перед работодателями им нужно научиться вести переговоры о сроках на основе компромиссов по объему работы и ее качеству.

В некоторых случаях, даже если начальник говорит вам гнать халтуру, надо идти вперед, делая свою работу хорошо.

Из журнала *Software Development*, том 2, № 7, июль 1994 г.

31

Под давлением

Городок Лорн (Lorne) в Австралии – это ворота на Великую океанскую дорогу (The Great Ocean Road), которая вьется вдоль одного из самых прекрасных побережий в мире, под звуки оглушающего прибоя волн, мимо крутых обрывов и белоснежных морских берегов, по которым можно идти и на протяжении многих миль не встретить ни души. Когда я жил в Австралии, Викторианское отделение Австралийского компьютерного сообщества пригласило меня в Лорн на свою ежегодную конференцию. Я должен был помочь в судействе на их первом конкурсе по разработке программного обеспечения Software Challenge. Это был 6-часовой марафон по разработке приложений, на котором с помощью новейших инструментов быстрого визуального проектирования создавалось приложение для взвешивания грузовиков, работающих на переработке отходов. Это было третье соревнование по быстрой разработке, в котором я участвовал в качестве судьи, поэтому у меня уже был какой-то опыт в этом деле (см. главу 30). Например, я научился не издавать стоны в тех случаях, когда система вызывала исключения GPF[1], и сдерживать свое волнение, когда соперники приближались к финишной ленточке, на больших оборотах завершая отладку. Кроме того, я узнал, что во время соревнований и сам смогу чему-то научиться, даже не соревнуясь.

Через час после начала соревнований я стал прохаживаться по «турнирной комнате», выделенной для программистов. По всей комнате расположились команды, состоящие из трех человек. Все они лихорадочно писа-

[1] GPF (general protection fault) – общее нарушение защиты. – *Примеч. науч. ред.*

ли код на Visual Works, PowerBuilder, SQL for Windows – все, за исключением одной группы невозмутимых молодых ребят из команды Ernst & Young. Когда я подошел к ним, я едва поверил своим глазам. Команда под руководством Крейга Брайта (Craig Bright) рисовала диаграммы! Вместо того чтобы согнуться над клавиатурами, как это сделали их соперники, они сгрудились вокруг листков бумаги. Они уточняли определение требований и усердно планировали архитектуру создаваемой системы. Заметив мой интерес, один из них дал мне копию своего боевого плана. Это был блокнот, содержавший краткое описание их обычной методологии, которая была модифицирована специально для этого соревнования. В блокноте все было расписано по пунктам, по этапам, с учетом промежуточных готовых компонентов. Это было полное, упорядоченное описание всего процесса быстрого проектирования. Даже в пылу гонки «ноздря в ноздрю», когда до сдачи результатов по более чем 200 функциональным пунктам оставалось меньше дня, эта группа демонстрировала невозмутимость под давлением и спокойно анализировала задачу и разрабатывала ее решение. Я был поражен.

Вера в процесс

Не раз я говорил студентам, что чем меньше остается времени до сдачи системы, тем важнее знать, что именно вы сдаете. Если на программирование приложения есть только четыре дня, то лучшее, что можно сделать для достижения успеха, – это потратить первые два дня на тщательное проектирование. Конечно, мне никогда не приходилось подтверждать свою веру программированием сложного приложения за один день. Легко быть праведным, когда ваша вера не подвергается испытаниям.

Безусловно, команда Ernst & Young была именно той командой, чья вера в процесс проверялась на прочность. Уже в самом начале, едва очутившись в конференц-зале, они умудрились сжечь свою совершенно новую систему на базе Pentium. Быстрый перенос программного обеспечения на лэптопы позволил продолжить работу, хотя и с меньшей мощностью аппаратного обеспечения.

Наконец, когда они перешли от журналов и блокнотов к клавиатурам и мониторам, дело опять осложнилось. На первый взгляд казалось, что соперники добились непреодолимого превосходства, тем более что некоторые из них уже демонстрировали какие-то рабочие компоненты. Однако в программировании поверхностные впечатления зачастую оказываются обманчивыми. На самом деле, к середине дня в контрольной точке, когда судьи оценивали команды программистов и проверяли их результаты по выполненным функциональным пунктам, парни из Ernst & Young превратили свое методичное спокойствие во внушительное лидерство. Тща-

тельный анализ и планирование давали свои результаты – даже в этих жестких условиях.

Тем не менее, как говорится, был еще не вечер.

По результатам начальной оценки на последнем месте оказалась команда из Xpedite Professional Services Pty. Ltd. под руководством Сью Стивенс (Sue Stevens). Пока другие команды усердно наращивали функциональность, они сосредоточились на деталях и тщательно продумывали код, направив все усилия на обработку ошибок и исключений. Однако этот метод, очевидно, не подходил для чрезвычайно плотного цикла разработки, установленного на соревновании. На полпути команда Xpedite решила поменять свою стратегию.

Когда подошло время финальной оценки, команда Ernst & Young начала объединять различные компоненты своей методично построенной системы. В спешке, когда все файлы переносились на одну машину, они нечаянно записали черновые версии поверх рабочих файлов. Мы все совершали такую ошибку, случайно перепутав направление копирования. Я и сам совершил ее всего за несколько месяцев до этого, когда переносил файлы Power Point, созданные для другой конференции. К счастью, у меня были резервные копии. К несчастью, у команды Ernst & Young их не было. Затерев большую часть своих данных, к финалу они пришли с меньшим количеством функциональных пунктов по сравнению с результатами в середине соревнований.

Польза

Тем временем команда Xpedite переключила свое внимание с деталей на человеческий фактор пользовательского интерфейса. Когда другие команды все еще строили грубую функциональную схему, они стали рассматривать реальные потребности оператора по переработке отходов и создавать более ценные компоненты. Вместо стандартных таблиц и меню они применили особые окна, разработанные с учетом нужд потребителя. В результате с последнего места, которое они занимали во время ланча, к обеду они переместились на первое.

Итак, что же можно извлечь из опыта этих доблестных команд? Более чем когда-либо очевидно, что быстрое создание прототипов не может заменить методичность в работе. По существу, именно сочетание инструментов быстрого визуального проектирования (см. главу 23) с разумным процессом анализа и разработки будет более эффективным при слишком сжатых сроках. Время, затраченное на осмысление и планирование, экономит время на разработку.

Также следует помнить о том, что именно мы разрабатываем. Мы производим не код, мы производим продукт. Большая польза и небольшой код намного ценнее, чем большой код и малая польза. Сырая функциональность менее ценна для наших клиентов, чем простые в использовании системы, которые отвечают потребностям их бизнеса.

Кроме того, важно сохранять гибкость и готовность адаптировать наши методы и процедуры к изменяющимся условиям, а также к изменяющемуся пониманию целей. Вероятно, формулы из учебников не отвечают требованиям современной быстрой разработки.

И наконец, еще об одном, хотя это и кажется очевидным. Хорошее управление версиями необходимо даже при очень коротких циклах программной разработки. Даже под непомерным давлением сроков – или, возможно, особенно под таким давлением – время, затраченное на создание резервных копий и версий, идет только на пользу. По сути, управление резервными копиями и версиями было необходимо включить в ускоренный метод разработки, применявшийся командой Ernst & Young. Именно это они и пообещали сделать к следующему соревнованию.

Из журнала *Software Development*, том 3, № 10, октябрь 1995 г.

32

Re: Архитектура

Что произошло с архитектурой программного обеспечения? В типичном приложении для малого бизнеса или в стандартном коммерческом пакете зачастую бывает трудно обнаружить присутствие хоть какой-то структуры. Архитектура – будь то внутренняя функциональная структура или структура пользовательского интерфейса – сегодня часто оказывается первой жертвой сжатых по времени разработок, коротких периодов между релизами и быстрого проектирования приложений. Иногда просто не остается времени на обдумывание всех последствий архитектурных решений. Зачастую едва хватает времени на то, чтобы просто подумать. Приехали. Системы собираются сразу, как только придумываются их новые компоненты, без особого внимания к общей структуре. Темпы настолько высоки, что разработчики не видят дальше следующей строки кода или следующего элемента.

Новая технология помогла мало. Инструменты визуального проектирования (см. главу 23) открывают новые пути к быстрому созданию сложных систем, однако скорость визуального проектирования и легкость построения рабочих приложений также способствуют недостаточному планированию. Инструменты даже могут приводить к такому стилю разработки, когда небольшие куски кода, связывающие логику ведения бизнеса с элементами интерфейса и базовой функциональностью, подвешиваются к визуальным компонентам. Все это объединяется паутиной из передаваемых сообщений и цепочек событий, в которой никто не может толком разобраться. В результате сегодня у нас есть классический «спагетти-код», в котором все связано со всем. Когда каждое изменение не-

предсказуемо передается по всей сети взаимосвязанного кода, возможности для дальнейшего развития программного обеспечения иссякают.

Второй шанс

Добавьте к быстрому созданию прототипов итеративную доработку, и последние остатки архитектуры могут утонуть в программном болоте. Это вызывает сожаление, потому что итеративное создание прототипов является мощным методом для разработки более удобного программного обеспечения за меньшее время. Создание прототипов позволяет своевременно поставлять функциональные программы или перепробовать различные подходы. Прототипы компонентов необходимы для получения отзывов пользователей, основанных на их реальных потребностях. Даже при создании пользовательского интерфейса для относительно скромных систем все предусмотреть с первого раза невозможно. Это не зависит от того, сколько времени и сил вы затратили на проектирование. Создание прототипов и итеративная доработка дают второй шанс (а также третий и четвертый) для того, чтобы довести до ума интерфейс пользователя. С каждой новой итерацией интерфейс и его внутренние компоненты становятся качественнее и функциональнее, обеспечивая все большие возможности и большую эффективность.

К сожалению, структура первого прототипа, которая удачно вписалась в общую концепцию, сильно зависит от основной архитектуры развивающейся системы. Система растет с каждой новой доработкой. Появляются новые слои компонентов, создаются новые функциональные возможности. Все это продолжается до тех пор, пока основная структура, казавшаяся такой разумной при небольшом размере системы, не начинает разваливаться под грузом доработок и уточнений.

Было бы большим соблазном просто вернуться к тем легендарным дням функциональной декомпозиции, когда дисциплинированные разработчики с помощью CASE-инструментов проектировали устойчивую и упорядоченную архитектуру всей системы. Такое проектирование велось до написания первых строк кода. Однако к тому времени, когда одна команда при помощи традиционных методов создает лишь системную архитектуру, RAD-команда[1], оснащенная инструментами быстрого визуального проектирования, уже сдает готовую рабочую систему. Поэтому методы построения системной архитектуры необходимо включить в ускоренный цикл разработки. Построение надежной структуры не должно существенно замедлять процесс разработки в целом.

---

[1] RAD (rapid application development) – быстрая разработка приложений. – *Примеч. науч. ред.*

Возможно, для этого надо заново пересмотреть место архитектуры программного обеспечения в процессе разработки. Обычно под архитектурой мы понимаем нечто, предшествующее построению. Однако так же, как можно переписать код, можно перестроить и архитектуру (в мире экстремального программирования такая процедура называется рефакторингом). Например, одному большому банку в Австралии пришлось вернуться к своим традиционным системам после того, как был прекращен один претенциозный и чрезмерно крупный проект построения новой общебанковской информационной системы. Прежняя система объединяла многие поколения клуджей[1], написанных на COBOL. Она получила заслуженную репутацию хрупкой конструкции, которая ломалась всякий раз, когда в нее вносились малейшие изменения или исправления. Руководство банка было в отчаянии, поскольку не удавалось внедрить новые финансовые услуги, без которых в те дни было невозможно выжить в динамичной банковской отрасли. Но еще не все было потеряно. Пока все сотрудники в неистовстве разрабатывали новую систему, один увлеченный программист из отдела технического обслуживания тихо перестраивал старую систему, малыми порциями приводя код в порядок и реструктурируя архитектуру процесса. Перестройка важных подсистем способствовала дальнейшему развитию проекта.

Обновление

Опыт этого банка указывает на необходимость радикальной реорганизации методов быстрого итеративного проектирования. Для того чтобы изменения, внесенные при итеративной доработке на основе прототипов, служили дольше, а само итеративное проектирование могло применяться для больших систем, нужно непрерывно возвращаться к улучшению системной архитектуры. Этот процесс можно назвать итеративной архитектурной доработкой. На каждом успешном этапе проектирования и разработки вся структура программы пересматривается для определения того, как можно улучшить интеграцию новых системных компонентов. Доработка архитектуры может привести к реорганизации структуры данных, к разделению системы на различные подсистемы или к замене неудачных алгоритмов на более эффективные. Зачастую улучшение архитектуры может потребовать редизайна и переписывания работающей части кода. Хотя такое программирование прибавляет хлопот на следующем этапе разработки, оно позволяет сделать систему более устойчивой для последующего улучшения и снижает затраты на будущих итерациях. Пересмотр архитектуры особенно подходит для изменения структуры объект-

[1] Клудж (от англ. kludge) – устройство, программа или ее часть, которые теоретически не должны работать, но почему-то работают. – *Примеч. ред.*

ных классов. Он помогает найти потенциальные или необходимые компоненты повторного использования и выявить нереализованные возможности для их применения.

Такой процесс является разновидностью концентрической разработки – рационализованной моделью быстрого создания прототипов с итеративной доработкой. Такая доработка начинается с базовых функций, обеспечивающих основной набор возможностей для пользователя. Эти функции определяются на основе отобранного подмножества пользовательских ситуаций или абстрактных сценариев, которые готовая система должна поддерживать (см. главу 22). Создание ядра законченных пользовательских ситуаций обеспечивает поддержку всех задач, а не только отдельных фрагментов функциональности. Далее в систему концентрическими слоями добавляются дополнительные возможности и украшения. С каждым новым слоем обзор архитектуры позволяет определить, какие доработки и изменения необходимо провести для того, чтобы повысить устойчивость системы к последующим улучшениям. Такие архитектурные изменения включаются в общий план работ на текущий цикл концентрической разработки.

В течение четырех лет одна австралийская компания с большим успехом применяла вариант этого подхода. Она выпускала по три релиза программной системы в год. Часть «спокойного времени», которое неизбежно появляется в периоды между выпуском релизов, выделялась на обзор архитектуры и планирование. Таким образом, архитектура обновлялась почти непрерывно.

Архитектурное планирование может существенно сэкономить время даже при очень жестких графиках работ. У саги о доблестной команде Ernst & Young, которая во время однодневного соревнования нашла время на проектирование архитектуры своей системы (см. главу 30), есть продолжение. Эта команда в полной готовности появилась на следующем соревновании Software Challenge, проходившем на ITWorld в Брисбене (Австралия). На этот раз они применяли новый инструмент (Delphi от Borland) и пересмотренный план для «безумного проектирования приложений» (так этот процесс назвали их соперники). Кроме того, они использовали управление версиями и резервное копирование, а также быстрый, но тщательный анализ и планирование архитектуры. Они выиграли.

Из журнала *Software Development*, том 4, № 1, январь 1996 г.

33

Пошаговое улучшение качества

«Тотальное управление качеством» (Total Quality Management), «Непрерывное улучшение рабочего процесса» (Continuous Process Improvement) или ISO – в большинстве современных терминов, касающихся качества процесса и продукта, акцент ставится на стремлении к качеству в масштабах всего предприятия и солидные инвестиции с долгосрочной перспективой. Продуманные схемы для оценки и повышения «зрелости процесса», такие как общеизвестная модель развития функциональных возможностей (Capability Maturity Model), разработанная Институтом разработки программного обеспечения (Software Engineering Institute), могут быть весьма полезны. Однако применение подобных моделей может потребовать много ресурсов (Humphrey, Snyder, Willis, 1991 [41]) и привести к непредусмотренным последствиям (Bollinger и McGowan, 1991 [5]). Для получения наибольшей, долговременной отдачи могут понадобиться программы для значительной реструктуризации и обеспечения всестороннего качества. Однако даже небольшие практические шаги будут способствовать быстрому и ощутимому улучшению качества программного обеспечения и эффективности самого проекта.

Нехитрые изменения в организации рабочего процесса могут разительно улучшить качество разработки программного обеспечения. Эти подходы не основаны на технологии. В них не подразумеваются ни автоматизированное проектирование и создание программ, ни объектно-ориентированные репозитарии, ни новые методы организации рабочего цикла, ни экспертные системы для метрик программного обеспечения, ни статистический контроль качества, ни любые другие технические приемы, которые могут считаться эффективными. Все эти шаги возвращают к основам, к

тому основополагающему факту, что даже в новейших технологиях работа выполняется людьми. Все эти подходы объединяет то, что в них рассматриваются способы организации работы и управления людьми и процессом. Основную часть этих подходов можно почти сразу же применить на практике без больших затрат на обучение, инструменты или агитационные плакаты.

Установка приоритетов

Первые шаги зачастую оказываются самыми важными, а первым шагом в улучшении качества является правильное определение приоритетов. Для улучшения качества продукта и процесса его создания само качество должно стать приоритетом. Если для вас и для организации в целом оно не важно, а стремление к нему не проявляется в действиях руководства, то качество не будет важным критерием и для разработчиков. Все это не означает плакаты с надписями «Качество – наша первая задача!» или напоминания работникам о необходимости стремиться к отсутствию ошибок. Здесь наиболее важно то, что вы делаете, а не то, что говорите.

- *Придайте качеству важность*

К сожалению, привычные представления и методы действий современных руководителей часто мешают им сделать качество реальным приоритетом. Одним из наибольших препятствий является чрезмерное стремление многих компаний «успеть выйти на рынок». В области новейших технологий, например в разработке программного обеспечения, видение руководства особенно заужено. Руководители не видят что-либо за пределами так называемого «рыночного окна». Если вы пропускаете это окно, все можно считать потерянным. Суть состоит в том, чтобы выйти на рынок раньше других, пусть даже с продуктом низкого качества и большим количеством ошибок. Когда забота о рыночном окне превосходит заботу о качестве, то качество начинает страдать. Это вполне очевидно. Безусловно, своевременность имеет значение, но все же это вопрос приоритетов. Когда выбор стоит между упаковыванием и продажей, по сути, бета-тестовой версии и проведением еще одного цикла тщательного тестирования и доработки, то что обычно выбирается?

Идея о рыночном окне все еще продолжает царить в головах многих разработчиков программного обеспечения, не позволяя им создавать высококачественные системы. Тем не менее история нашей отрасли изобилует призраками компаний, которые первыми вышли на рынок и на этом для них все закончилось. Известно множество новаторских продуктов, которые появлялись в недоработанном виде и не доживали до исправления.

- *Смотрите шире «рыночного окна»*

Современные руководители не присваивают качеству высокий приоритет еще и потому, что большинство компаний, особенно в Соединенных Штатах, больше беспокоятся о затратах и их снижении, чем о получении выгоды от инвестиций. Это легко понять, когда экономика переживает спад, а прибыли уменьшаются. Образование, обучение и развитие персонала – все это признается важным для качества, но заносится в графу затрат и не считается инвестированием. Посещение конференций и семинаров хотя и является важной частью поддержания конкурентоспособности компании, но все же воспринимается как накладные расходы и часто оказывается первой мишенью при снижении затрат.

Речь идет не о глупых понятиях «любезности» с сотрудниками, а о финансовой основе подходов управления и принятия решений и их влиянии на возможности улучшения качества. Если затраты от 6-месячной задержки в выпуске продукта окупятся еще через 6 месяцев, то стремление «ворваться на рынок» не снижает себестоимость.

Механизм расходования средств требует обоснования, однако анализировать нужно не только затраты, представляющие только одну часть уравнения, но и прибыль от инвестиций. Например, австралийский консультант Роб Томсет (Rob Thomsett) показал, что одинаковую отдачу можно получить как от инвестирования в CASE-технологии, так и от вложений в формирование команды, однако конечная прибыль от инвестирования в командную работу будет на порядок больше. Тем не менее CASE – это яркая технология, которую можно показать, тогда как эффективная командная работа невидима, поэтому многие компании предпочитают тратить деньги на аппаратные и программные средства, а не на человеческий фактор (peopleware).

- *Думайте о прибыли от инвестирования, а не о сдерживании расходов*

Поощрение и признание

Для придания качеству высокого приоритета людей нужно оценивать и поощрять за выполнение качественной работы. Но что следует поощрять? В области разработки программного обеспечения именно продуктивность, измеренная либо в функциональных пунктах, либо в строках кода, служит основанием для премии, или повышения по службе, или признания, или других поощрений. Либо награждаются геркулесовые, самоотверженные усилия по выполнению работы в невозможные сроки. Как это ни странно, но во многих компаниях руководителям проектов выгодно создать предпосылки, при которых под конец проекта понадобятся крайние усилия. Такая нарочитая самоотверженность имеет больше шансов получить одобрение независимо от успешности самого проек-

та. «Ну, хорошо, контракт провалился, но никто не может осудить Пита, ведь он круглосуточно работал до самого дня сдачи».

Проблема не в том, что люди не заботятся о качестве, как жалуются некоторые руководители. Исследование (1991 г.), проведенное компанией Brooks International и охватившее 11 000 человек из 6 отраслей, показало, что более 90% работников чувствовали личную ответственность за качественное выполнение работы. Однако семь человек из десяти заявили, что качество не было важным фактором в оценке их деятельности. И только 25% опрошенных утверждали, что руководство действительно оценивало повышение качества. Так что же мы поощряем? На самом деле признание и поощрения любого рода намного более редки, чем думают большинство руководителей. Почти 80% руководителей искренне утверждают, что их подчиненные получают достаточное поощрение, но только один из семи подчиненных соглашается с этим (Lickert, 1989 [48]).

Для улучшения качества необходимо воспользоваться принципом Фербера (Ferber Principle). Психиатра Эндрю Фербера (Andrew Ferber) однажды спросили, что является самым важным для начинающих терапевтов, стремящихся помочь семьям, с которыми они работают. Он ответил:

- *Когда вы видите то, что вам нравится, начинайте хлопать в ладоши как сумасшедший*

Измерение и контроль

Почти каждый слышал изречение, утверждающее, что нельзя контролировать то, что нельзя измерить (DeMarco, 1982 [32]). Часто такие слова оказываются прелюдией к активной кампании по запуску проекта, связанного с измерением параметров программного обеспечения или обеспечением статистического контроля качества. У формальных измерений есть множество преимуществ. Однако если на мгновение задуматься, то можно понять, что в жизни есть много важных вещей, которые родители, учителя, руководители контролируют, но не измеряют. Многое вообще нельзя измерить. Когда речь заходит о людях, то важным становится внимание, а не измерение. Значение имеет то, что вы контролируете. Любой хороший родитель знает: чем больше обращать внимание на капризы, тем больше будет капризов. У систем вообще и у человеческих систем в частности есть особенность, согласно которой само наблюдение изменяет предмет наблюдения. На этом основан хорошо известный эффект Хоторна (Hawthorne effect): если сделать группу объектом наблюдения и просто уделять больше внимания ее работе, это приведет к повышению производительности.

- *Уделяйте внимание*

Безусловно, объект ваших наблюдений имеет значение, потому что именно на него и будет оказываться воздействие. Если оценивать программистов по краткости написанного кода, они будут производить меньшие по размеру системы. Если критерием является дружественность системы по отношению к пользователю, то вы получаете более удобные программы (Weinberg и Schulman, 1974 [66]).

В Австралии новый руководитель группы по разработке программ технического обслуживания захотел улучшить не только эффективность своей команды, но и ее статус и значение в компании. Для этого среди прочего он стал посылать основным программистам отчеты об ошибках, которые были обнаружены и исправлены в системах после их передачи «в производство». Программист мог получить записку, в которой просто сообщалось, что в минувшие выходные в системе произошел сбой и некорректное закрытие файла вывода, однако программист Куинфорп из отдела технического обслуживания обнаружил ошибку в цикле модуля Z091, который был исправлен, перекомпилирован и протестирован за 1,6 часа.

Такая практика принесла интересные результаты. Новые системы, находящиеся в производстве, получались более надежными и быстрее проходили приемо-сдаточные испытания. Сам факт наблюдения за качеством и сообщение о результатах наблюдений может способствовать повышению качества.

- *Создавайте обратную связь*

В другой компании каждый месяц вывешивались диаграммы, отражавшие эффективность каждого программиста, которая измерялась в строках написанного и отлаженного кода. Отчеты изменили. В объем сданного кода стали включать не только код, написанный программистами, но и код всех модулей, взятых из библиотеки компонентов повторного использования. После этого уровень повторного использования существенно возрос (см. главу 27).

Обратная связь – это важный ингредиент. Когда работники получают информацию о собственной производительности и ее связи с организационными целями, качество повышается. Это основа открытой модели руководства, в которой работники получают не только отчеты о производстве и ошибках, но и финансовую информацию о затратах, статьях расходов и прибылей (Case, 1990 [6]; Finegan, 1990 [36]). Обладая такой информацией, работники проще оптимизируют распределение времени и улучшают собственные рабочие процессы. Ключом здесь является обратная связь, которая объединяет индивидуальную и командную производительность с общей финансовой картиной. Работникам известны не только выявленные ошибки и время, затраченное на программирование, но и общая стоимость работ, и получаемая в результате выгода (или убыток) с точки зре-

ния всего проекта. Многие руководители поняли, что это двусторонняя дорога. Чем больше информации доверяется работникам, тем больше работники доверяют руководству. В результате возникает непрерывный обмен идеями о возможных улучшениях. Обычно техническое руководство склонно думать об измерениях, выполняемых с точностью до третьего знака после запятой или еще точнее. Однако для проведения оценки и контроля процесса вполне достаточно качественных методов или даже приблизительных сравнений. В теории измерений, являющейся разделом статистики, приняты различные уровни измерений. Числа, которые можно перемножать и разделять, находятся на одном уровне (так называемая шкала отношений). Числа, которые можно складывать и вычитать, располагаются на другом, более низком уровне (интервальная шкала). Статистический анализ возможен, даже если результаты показывают лишь то, что одно лучше другого на некоторую неизвестную величину.

Не нужно измерять высоту с точностью до метра, чтобы найти башню на вершине горы. Требуется знать лишь то, куда ведет вас каждый шаг. Вверх или вниз. Для многих процессов эффективные стратегии улучшения могут быть основаны на приблизительном измерении, дающем представление о спаде или подъеме.

Данные и информация

Наверное, большинство руководителей заявят о ценности информации. Они думают, что их решения основаны на данных. К сожалению, те же самые руководители зачастую избегают возможности получать нужную им информацию и имеют склонность забывать о тех данных, которые у них есть.

Истинный ученый знает, что не существует такого явления, как неудачный эксперимент. Все происходящее дает информацию, которая может привести к пересмотру гипотезы или изменению технологии. Изучая семейную терапию, я усвоил, что все происходящее во время сеанса является информативным. Мы говорим нашим практикантам:

- *Запомни, это все – данные*

Настоящий руководитель знает, что все новости являются хорошими. Информация о процессе сама по себе является ценной и должна цениться. Ваше восприятие информации влияет на то, насколько она будет доступной для вас в будущем. Если поощряются только хорошие новости, правда не будет известна. Когда вы караете сотрудников, принесших плохие вести, проблема состоит не только в наказании посыльных, но и в том, что в итоге к вам будут поступать только хорошие новости. «Пло-

хие» новости, знать о которых зачастую важнее всего, доставляться не будут.

Однажды у меня был начальник, который сказал мне, что никогда не будет ругать меня за плохие вести. Особенно ему хотелось знать о трудностях, которые могут угрожать агентству, на которое мы работали. Он не только сдержал свое слово, но и оставил за мной и моими подчиненными право решать такие проблемы. Это помогало поддерживать открытость во взаимоотношениях и обеспечивало начальнику доступ к важной информации, необходимой для принятия решений.

Безусловно, для улучшения любого процесса самой важной является информация о трудностях и неудачах, хотя получения именно таких данных руководители стремятся избегать. Обнаружение ошибки в программе должно быть поводом для праздника. На самом деле все программные ошибки должны не только фиксироваться, но и изучаться.

- *Фиксируйте и изучайте ошибки*

Ведение детального протокола всех трудностей – изъянов, упущений, жалоб потребителей, изменений дизайна, ошибок в анализе, «улучшений» при бета-тестировании – является важным шагом. Другой шаг заключается в регулярном и методичном изучении всех неполадок. Это означает, что в каждом проекте необходимо предусматривать время для методичных размышлений. Если мы не изучаем свои ошибки и не учимся на них, то как мы сможем избежать их в будущем?

Для улучшения качества особенно важно никогда не путать оппозицию и критику с нелояльностью.

- *Поощряйте критику*

Зачастую именно противоположный взгляд или критическое рассмотрение дает самую ценную информацию о возможностях улучшения процесса. На самом деле качество решения задач в чрезвычайной степени зависит от наличия критики. Группы, в которых есть «свой критик» или «спорщик», а также группы, практикующие диалектические процессы столкновения идей и активной критики, работают с большей производительностью (Constantine, 1989 [11]; Priem и Price, 1991 [59]).

Конечно, мало просто знать о чем-либо неправильном и даже причины этого. Важно что-то предпринять. Программные ошибки – это не просто информация о том, что в той или иной программе что-то неверно. Они также указывают на неполадки в самом процессе создания программы. Первый вопрос: как возникли ошибки? Цель заключается не просто в их исправлении, но и в получении информации о том, как надо изменить процесс для снижения количества ошибок в будущем. В организациях, в

которых рабочие процессы непрерывно совершенствуются, каждая неудача воспринимается как возможность для собственного переобучения и улучшения рабочего процесса.

- *Исправляйте процесс, а не только программу*

Видимость рабочего процесса

В названии одной известной песни 60-х годов можно найти действенный принцип улучшения качества:

- *Пусть светит солнце*

Невидимость – враг качества. Нельзя улучшить то, что не видно. Один из самых лучших способов для обнаружения неполадок – сделать более видимым то, что делают разработчики.

Опыт показывает, что качество программного обеспечения можно значительно улучшить, если просто увеличить объем работы, выполняемой лицом к лицу (глава 26). Когда двое или более людей вместе работают над одной задачей, качество повышается. Как правило, повышение видимости рабочего процесса повышает его качество. Почему? В принципе, для того чтобы два программиста, вместе создающие программу, внесли ошибку или отступили от стандартов и правил, они должны сговориться об этом. Однако заметить ошибку или отступление от правил может и один человек. Забудьте о том, что вы слышали про «групповое мышление» или коллективную посредственность. Все эти эффекты существуют, но они зависят от определенных условий. Групповые лидеры могут легко способствовать улучшению качества решения задач и избежанию недостатков группового мышления. Для этого лидерам любых групп достаточно просто воздерживаться от выражения своего собственного мнения или откладывать его на какое-то время (Anderson и Balzer, 1991 [2]).

Модель программирования «два человека на терминал», которую я назвал «динамическим дуэтом», возникла в эпоху появления так называемого «обезличенного программирования». Обезличенное программирование основывалось на том представлении, что программисты вкладывают в программы слишком много из своего внутреннего «я». Если бы они работали в безличном стиле, выстраивали меньше защит и были более открыты для анализа, предложений и критики со стороны других, то они могли бы производить более совершенный код. Такой подход имеет ряд слабых мест, не в последнюю очередь связанных с тем, что у людей есть собственное «я». Вместо уничтожения или подавления эго современные методы управления стремятся учесть эту реальность человеческой природы и обратить ее на общую пользу.

Паролем нынешнего дня является «собственность» или «участие». Прогрессивные организации стремятся повысить значимость личного владения. Это своего рода эго-инвестиции служащих в продукты, создаваемые их усилиями. Например, открытая модель командной работы (см. главу 16) является подходом к организации проектных команд, в котором применяется решение задач на основе консенсуса. Такая модель повышает видимость рабочего процесса и увеличивает долю индивидуального участия каждого.

Видимость рабочего процесса тесно связана с идеей о разделении труда. Такое разделение присуще методу «динамический дуэт». Пока один программист сидит за клавиатурой, другой «смотрит через его плечо». Программист за клавиатурой имеет свой круг обязанностей, связанных с определением алгоритма и планированием кода. Другой программист следит за «дырами» в логике и пытается отследить ошибки и слабые места в коде.

- *Применяйте разделение труда*

Этот принцип является важнейшим компонентом метода «чистого» (clean room) программирования. С помощью этого подхода были созданы некоторые средние и крупные системы, которые практически не содержали ошибок (Cobb и Mills, 1990 [8]). В этой модели один человек или группа пишет код, стараясь «все сделать правильно». Еще кто-нибудь выполняет компиляцию и тестирование, стараясь обнаружить ошибки и погрешности. В этой модели есть и другие приемы, но даже простое разделение обязанностей само по себе может повысить качество. Знание того, что кто-то другой в команде не только просматривает код, но и занимается компиляцией и тестированием, повышает ответственность и побуждает принимать правильные решения с первого раза.

Навыки и таланты

Десятилетия исследований и практический опыт показали нам, что по продуктивности лучшие программисты зачастую на порядок отличаются от худших и в два раза от средних программистов (DeMarco и Lister, 1987 [33]).[1] Некоторые группы значительно повысили качество своей работы и продуктивность путем простого сокращения штата программистов, оставив только самых лучших. Один подход к повышению качества заключается в том, чтобы взять только самых лучших игроков, предоставить в их распоряжение все ресурсы, создать мотивацию для достижения наилуч-

[1] Т. ДеМарко и Т. Листер «Человеческий фактор: эффективные проекты и команды». – Пер. с англ. – СПб: Символ-Плюс, I кв. 2004 г.

ших результатов в работе и позволить им ее сделать. Такой подход может быть особенно привлекательным в эпоху «сокращения расходов».

- *Применяйте только самые лучшие ингредиенты*

Конечно, каждый руководитель знает, что в любой организации есть «звезды», но не каждый желает избавляться от вспомогательных игроков. На самом деле мы хотим найти способ помочь другим «стать лучше», что приводит нас к принципу перекрестного обучения. Идея заключается в том, чтобы разработчики программного обеспечения имели больше возможностей учиться друг у друга.

- *Пусть все друг друга учат*

Один из самых эффективных способов осуществления перекрестного обучения заключается в том, чтобы встроить обучение в сами проекты. Это опять возвращает нас к видимости рабочего процесса. Когда члены команды выполняют больше работы лицом к лицу, они автоматически учатся друг у друга. Кроме того, ротация обязанностей как часть процесса разработки дает возможность применять полученную информацию, помогая постепенно распространять навыки и знания среди членов группы.

Различия в природных талантах и достижимых уровнях мастерства существовали и будут существовать. Одни программисты неизменно пишут более совершенный код, другие хорошо моделируют основные абстракции. Один участник команды всегда лучше справляется с ведением обсуждений, чем другие. Однако в организации, в которой поощряется и обеспечивается перекрестное обучение и распространение умений, средний уровень способностей в любой из областей всегда повышается. Со временем люди все больше и больше начинают разбираться в специальностях друг друга. Они никогда не достигнут той точки, когда каждый сможет выполнять все обязанности с равным умением, но различия все же будут уменьшаться. Еще более важно то, что члены команды могут все успешнее заменять друг друга. Организация как целое становится менее зависимой от навыков и присутствия отдельных ее членов. Весь проект не останавливается только из-за того, что кто-то заболел или уехал в другой город.

Степени свободы

Как это ни странно, во многих организациях, искренне стремящихся улучшать качество, существуют правила и условия, которые препятствуют его улучшению. Даже такие простые вещи, как способы определения сроков исполнения и составления бюджетов, значительно влияют на качество проектов. Обычно все факторы – бюджет, распределение ресурсов,

подбор персонала, методология и сроки исполнения – уже определены на тот момент, когда проект передается разработчикам. На какой стадии можно достичь улучшения качества? Нам нужна по крайней мере одна степень свободы. Если все переменные ограничены, система становится сверхопределенной, а путей к победе у нас нет. Чем при возникновении проблем жертвуют в первую очередь? Качеством! При жестких сроках исполнения, например, устанавливаемых в соответствии с каким-то выдуманным «рыночным окном», дело зачастую доходит до таких высказываний: «У нас нет времени на то, чтобы сделать все как надо». Это указывает на одно из простейших изменений, способных улучшить качество программного обеспечения:

- *Обсуждайте сроки исполнения*

Разработчики должны непосредственно участвовать в определении сроков поставки продукта и стадий исполнения проекта. Определение сроков следует рассматривать как переговоры, в ходе которых могут быть найдены компромиссы. «Да, проект можно сдать к концу года, если вас устраивает 15 ошибок на каждую тысячу строк. Либо мы можем пообещать более низкую частоту ошибок, если вы согласны вдвое сократить количество экранов».

Заключение

Методы повышения качества программного обеспечения не всегда сложны и необязательно требуют больших затрат. Некоторые простые приемы могут привести к значительным изменениям. Прежде всего, определите свои приоритеты – присвойте качеству высокий приоритет. Не допускайте, чтобы ваш бизнес зависел от рыночных окон. Старайтесь думать о прибыли от инвестирования, а не о сдерживании затрат.

Далее, признавайте и поощряйте качество. Обращайте на него внимание. Обеспечьте обратную связь и широкий доступ к информации. Прислушивайтесь ко всему. Помните, что все новости являются хорошими, особенно плохие. Поощряйте критические оценки. Отслеживайте и изучайте ошибки и корректируйте рабочий процесс, а не только программу. Пусть светит солнце – сделайте рабочий процесс более видимым. Способствуйте перекрестному обучению – пусть каждый учит каждого. Когда вопрос качества имеет особое значение, применяйте только лучшие ингредиенты. И всегда ведите переговоры о сроках исполнения!

По материалам журнала *American Programmer*, февраль 1992 г.

VI

Юзабилити программного обеспечения

Введение

Пользовательский интерфейс – это точка соприкосновения программы и человека. Это река, которая очерчивает границу между пользователем и используемым. Разработчики программного обеспечения стоят по обе стороны этой реки. Как программисты, они, в известном смысле, находятся внутри компьютера, где видят обычную путаницу или исключительную четкость кода, лежащего в основе всего. В то же время они являются пользователями компьютерного программного обеспечения. Они могут смотреть вовнутрь со стороны и видеть не только код, но и компоновку элементов или полей, делающие их инструменты и системы более или менее удобными. Таким образом, они имеют двойную заинтересованность в пользовательском интерфейсе: как разработчики и как пользователи.

Наверное, юзабилити можно считать главной мерой качества программного обеспечения. Не имеет большого значения, снабжена ли программа эффектной графикой или быстрыми алгоритмами, или даже ее безошибочная работа, если при этом такой программой невозможно пользоваться. Если система не дает полезного результата, возможностей или сервисов, отвечающих потребностям пользователей, то может ли такая система считаться хорошей?

Когда сами компьютеры стали более доступными и большое количество людей начало непосредственно и регулярно взаимодействовать с ними, вопросы юзабилити и разработки пользовательских интерфейсов привлекли повышенное внимание в мире программного обеспечения. Сфера интересов разработчиков программного обеспечения постепенно расширилась за пределы внутренней структуры программ и данных до пользовательского интерфейса. Вначале основное внимание уделялось технологии, то есть программной стороне разработки. Постепенно разработчики программного обеспечения научились более четко продумывать детали устройств и механизмов, предназначенных для взаимодействия с пользователем, а также их компоновку внутри пользовательского интерфейса. Далее с технических деталей пользовательского интерфейса внимание переключилось на самих пользователей. Разработчиков программного обеспечения и приложений убеждали обратить внимание на реальных

пользователей, общаться с ними почаще и обстоятельнее для того, чтобы понять их предпочтения и учесть их идеи и предложения. А еще лучше – привлекать типичных пользователей к самому процессу проектирования и разработки. Программисты, которые когда-то старались держаться подальше от пользователей, за исключением тех случаев, когда они были вынуждены получать одобрение спецификаций, теперь стали встречаться с конечными пользователями на собраниях и плановых совещаниях или даже общаться с ними как с действующими членами проектной команды.

Сегодня с пользовательской точки зрения мы видим больше внимания к использованию, чем к пользователям, больший акцент на намерения пользователей, чем на их предпочтения. В таком «телеоцентрическом» или целеориентированном подходе к построению систем реальные потребности пользователя считаются более важными, чем его желания. Суть заключается в создании такого программного обеспечения, которое способно лучше поддерживать работу реальных людей. Такие программы служат полезным целям и позволяют облегчить и упростить решение задач.

Пока не ясно, сохранится ли эта тенденция и насколько далеко она продвинется. Тем не менее мы обсудим некоторые из тех важных вопросов, которые касаются пользовательских интерфейсов, пользователей и применения программного обеспечения – как мы сегодня это понимаем.

34

Согласованность и условности

Мы окружены пользовательскими интерфейсами. Возможно, этот термин получил широкое распространение благодаря компьютерному программному обеспечению, однако каждая система и каждый компонент оборудования, имеющие пользователя, по определению имеют и пользовательский интерфейс. Как показал психолог и бывший исследователь компании Apple Дональд Норман (Donald Norman, 1988 [53]), мы можем многое узнать о том, как разрабатывать и строить хорошие пользовательские интерфейсы для программного обеспечения, если посмотрим вокруг. Нужно просто подумать о том, как средства управления приборами, устройствами и инструментами делают их применение проще или сложнее.

Вспомните о том, как вы последний раз брали машину напрокат или одалживали ее у друга. Вероятно, это была другая модель или марка, отличавшаяся от модели той машины, которой вы обычно пользовались. Вы садились на место водителя, пристегивались, подстраивали зеркала и потом трогались с места. Вопрос: сколько секунд у вас ушло на то, чтобы изучить пользовательский интерфейс данной системы? Посещали ли вы для этого специальные курсы? А может быть, просматривали видеофильм о том, как пользоваться этой машиной? Или же вы смогли разобраться в этом самостоятельно без чтения руководства по эксплуатации?

Пользовательские интерфейсы большинства современных автомобилей, за небольшим досадным исключением, подчиняются Великому Закону Юзабилити (Constantine, 1991 [14]). Этот закон гласит, что пользовательский интерфейс должен давать возможность пользователю, обладающему знаниями в данной области, применять систему без дополнительного обучения и штудирования руководства по эксплуатации или других инструк-

ций вне самой системы. Другими словами, хороший пользовательский интерфейс дает пользователям, уже знающим то, что они делают, возможность приступить к работе без необходимости изучения чего-либо еще.

Ноу-хау

Конечно, вы уже умели водить машину. Возможно, вы уже были опытным автолюбителем – не обязательно профессионалом, но квалифицированным и подготовленным водителем. Для опытного водителя вождение автомобиля становится, как говорят психологи, избыточным навыком. Вы можете делать это, применяя только часть сознательного внимания. В следующий раз, когда вы будете вести машину и говорить с пассажиром, попробуйте провести простой, но показательный эксперимент. Во время разговора постарайтесь осознать факт того, что вы в этот момент ведете машину. Как это может происходить? Вождение автомобиля – очень сложная задача с точки зрения обработки информации. Это выяснили военные ученые и инженеры, когда пытались создать компьютерную программу, способную управлять автофургоном. Что касается обычного разговора, то это еще более сложная задача, чем управление машиной. Тем не менее опытные водители способны следить за нитью разговора и с помощью устойчивых подпрограмм переводить большую часть задач, связанных с движением автомобиля, в фоновый режим.

Когда вы сели за руль непривычной для вас машины, то подстроиться под новый пользовательский интерфейс вам помогли два обстоятельства. Во-первых, этот интерфейс, вероятно, очень хорошо соответствовал вашему обычному и «естественному» диалогу с машиной. Во-вторых, интерфейс, видимо, не сильно отличался от интерфейса в вашей машине. Приборная панель и элементы управления в большинстве машин выполнены в соответствии с несколькими основными соглашениями. Переключение передач находится либо на рулевой колонке, либо на полу между водителем и пассажиром; спидометр, как правило, расположен на приборной доске прямо перед глазами; руль обычно круглый, а переключатель сигнала поворота, расположенный слева на рулевой колонке (за исключением стран с левосторонним движением), поворачивается по часовой стрелке при повороте направо и против часовой стрелки при повороте налево – так же, как и сам руль управления. Этот интерфейс соответствует принятым условностям и является внутренне непротиворечивым. Время от времени вы можете столкнуться с чем-нибудь необычным, и тогда вам придется немного повозиться с тем, как, например, включать фары, но на это понадобится лишь несколько мгновений.

Многие люди удивляются, когда узнают, что лишь единичные элементы типового пользовательского интерфейса обычного автомобиля регулиру-

ются государственными или федеральными стандартами. Закон не требует даже наличия руля и не настаивает на его круглой форме. Некоторые специальные автомобили действительно строятся с другими рулевыми механизмами, рассчитанными на особые классы пользователей. Несколько лет назад одна германская автомобильная компания установила на одну из своих спортивных моделей руль овальной формы, однако водители его возненавидели. Руль управления принято делать круглым, потому что вращение круглого колеса, как показал долгий опыт, является одновременно хорошей метафорой и эффективным механизмом для задания направления поворота.

Повышение стандартов

Однако так было не всегда. В начале автомобильной эволюции было испробовано множество других механизмов поворота. В ранних моделях использовались рычаги – отчасти потому, что механическая часть тогда была проще. Но в результате естественного отбора, производимого инженерами и водителями, в конце концов победили круглые рули. Эта эволюция стала возможна именно потому, что конструкторы автомобилей не были ограничены непродуманными стандартами. Кроме того, их ничто не вынуждало отличаться ради различий – просто из-за того, что какая-то компания заявила об интеллектуальном праве собственности на общий вид круглых рулевых механизмов.

Для большинства действительно важных аспектов пользовательских интерфейсов стандарты не нужны. Более совершенные устройства и механизмы постепенно вытеснят все остальные на рынке продуктов и идей. Рулевые рычаги, как и рули управления, довольно просто преобразуют управляющее движение в изменение направления, но они имеют существенные ограничения. Если бы группа лидеров индустрии или государственных чиновников, занимающихся стандартами, из самых лучших побуждений разработала стандарт рулевого управления в тот момент, когда рулевые рычаги были более распространены, автомобили остались бы только локальным средством перемещения на небольших скоростях.

Четверть века назад К. Д. МакКензи удачно сформулировал Первый Закон Условностей (K. D. Mackenzie, 1966 [50]). Если есть больше одного способа сделать что-либо, а выбор между вариантами может быть произвольным, то выбирайте какой-нибудь один способ и *всегда* применяйте только его. Если же способы неравнозначны, то важно выбрать хороший вариант.

П. Дж. Плоджер (P. J. Plauger), принципиальный и убежденный разработчик стандартов, посвящающий их созданию значительное время, говорит о Принципе Достаточного Блага (P. J. Plauger, 1993 [58]). Стандарт

для информационного обмена или для языка программирования, или для телефонного интерфейса не обязательно должен быть идеальным или совершенным; он даже не должен быть «правильным». На практике идеальные стандарты так или иначе оказываются политически или технически невозможными. Все, что нужно, – это «достаточно хороший» стандарт. Как правило, человеческая смекалка и развивающаяся технология преодолевают большинство ограничений или недостатков.

Вопрос заключается в том, что считать «достаточно хорошим» в отношении таких широко применяемых средств, как графические пользовательские интерфейсы. Самый важный вид согласованности – это согласованность способов мышления и действий людей, когда программное обеспечение не вынуждает их выбирать те или иные методы. Кора человеческого мозга обладает поразительной гибкостью. Люди способны учиться. Они могут подстраиваться под самые сложные интерфейсы, однако у всего этого есть своя цена.

Большая часть знаний и действий людей не связана с компьютерами (как это ни печально для программистов, но это суровая реальность!). Хотя некоторые механизмы действительно «зашиты» в человеческий мозг, большая часть действий, которые люди считают интуитивными, на самом деле являются обусловленными. Сегодня психологи определяют интуицию в терминах комплексных ассоциаций и процессов обработки информации, которые настолько известны мозгу, что перестали быть полностью осознаваемыми.

Противореча разуму

Когда вы вынуждаете пользователей взаимодействовать с системой способами, которые противоречат соглашениям, запрограммированным на практике или полученным в результате эволюции, вы увеличиваете недовольство и усталость, а также создаете дополнительный и постоянный источник ошибок. Даже небольшое увеличение вероятности ошибки из-за пользовательского интерфейса может быть значительным. Представьте, к чему может привести даже малейшее увеличение частоты ошибок в процессе ввода информации в современные гигабайтные базы данных. Или представьте последствия неполадок в интерфейсах тех инструментов, с помощью которых проектируется и создается само компьютерное программное обеспечение.

К сожалению, большинство современных графических пользовательских инструментов просто-напросто не являются хорошими. Они противоречивы, чрезмерно сложны и изобилуют условностями, которые объективно и субъективно неверны. Публичное признание Microsoft в том, что Windows не представляет собой идеальную платформу для управления

домашними устройствами и другими потребительскими продуктами, – это довольно сдержанное высказывание. Проблема не в тонкостях стиля, а в принципиальных изъянах базовых механизмов.

Приведем только один пример. Возьмем полосы прокрутки, которые применяются как метафорические механизмы для управления перемещением рабочей области относительно меньшего по размерам «окна», отображающего только часть поверхности. Полосы прокрутки для экранной навигации выполняют ту же роль, что и рычаги управления для автомобильной навигации (Constantine, 1994 [25]). Они замедляют и ограничивают работу пользователя, вводят его в заблуждение, а также вызывают лишние движения, увеличение ошибок и нарушение нормального хода мыслительных процессов. Они вынуждают пользователя выполнять такие движения, которые абсолютно противоречат работе мозга. Для перемещения влево или вправо пользователю приходится сначала переместиться вниз, а для перемещения вниз или вверх нужно сначала переместиться вправо.

Для того чтобы понять, насколько проблематичным может быть простой, но «противоинтуитивный» интерфейс, проведите небольшой эксперимент. Разверните мышь на четверть оборота против часовой стрелки и попробуйте с ее помощью перемещать курсор на экране. Такая пространственная манипуляция кажется простой, но приспособиться к ней почти невозможно.

За пределами мира компьютеров существуют системы и устройства, которые предназначены для решения аналогичных задач навигации с помощью панорамирования, прокрутки и масштабирования. С точки зрения графических пользовательских интерфейсов интересно рассмотреть два из них: аппараты чтения микрофиш и видеокамеры. Среди десятков механизмов для управления экранной навигацией (в том числе и тех, которые были разработаны автором этой книги) есть несколько явно и существенно лучших, чем полосы прокрутки (Constantine, 1994 [25]). Если вам не кажется, что это важно в реальной работе, просто подумайте, сколько раз в день вы или ваши потребители прокручивают документы, диаграммы и изображения на мониторах.

Введение стандартов для графических пользовательских интерфейсов может стать плохим решением, хотя идея временами кажется хорошей. Вопрос только в том, хотите ли вы всегда попадать в заторы, управляя своим компьютером с помощью рычагов.

Из журнала *Computer Language Magazine*, том 9, № 11, ноябрь 1992 г.

35

Сложность и прогрессирующий функционизм

Я ненавижу переезжать. Я ненавижу заниматься апгрейдом программного обеспечения. Я ненавижу пересаживаться на новую машину. В принципе, переход к другой платформе или версии прост и эффективен. На практике это приводит мою и без того слегка неупорядоченную повседневную жизнь в полный и непроглядный хаос. Но больше всего я ненавижу изучать новый текстовый редактор.

Мне, как поставщику новых идей, больше всего нужны два программных продукта: графический пакет и текстовый редактор. Эти инструменты – мои постоянные попутчики. Я хочу, чтобы мне было удобно с ними работать и чтобы они хорошо совмещались друг с другом. Они определяют границы того, что я могу сделать в плане самовыражения с помощью слов и графики; они устанавливают потолок моей продуктивности. Они помогают мне одним движением доставать до дальних ящиков или же ставят подножку в простых задачах. Подобно выбору нижнего белья или дезодоранта, мои предпочтения одних инструментов другим являются глубоко личными, категоричными и иррациональными.

Изначально я мигрировал на Windows потому, что графический пакет, который был мне нужен (или мне просто хотелось им пользоваться?), работал под Windows. Миграция ассоциируется с устойчивым прогрессом, но в моем случаем это напоминало дефенестрацию[1]. Решившись на такой

[1] Дефенестрация (от лат. fenestra – «окно») – выкидывание человека из окна. – *Примеч. перев.*

шаг, я столкнулся с раздражающей необходимостью переключаться между текстовым редактором и пакетом инструментов, работающих под DOS, и графическим Windows-инструментом. Как и рецидивирующий артрит, это не смертельно, но причиняет постоянную боль. Для более гладкого взаимодействия с коллегами, многие из которых переходили на один и тот же текстовый редактор в Windows/Macintosh, я клюнул на одно из предложений провести апгрейд. Чтобы ощутить смысл термина «апгрейд», представьте себе подъем дороги, которая тянется... вертикально вверх![1]

Нельзя сказать, что этот текстовый редактор был недружелюбен к пользователю. В нем больше красивых кнопок, чем пуговиц у профессиональной портнихи, и больше файлов контекстно-зависимой справки, чем у вашего научного руководителя в институте. В нем не сложно выполнять простые повседневные операции. Наверное, с его помощью можно сделать все, что мне когда-нибудь может понадобиться. Однако можно потратить месяц на поиск всех этих удобных функций, особенно тех сокровищ, которые определяют разницу между исполнением работы в срок и не в срок.

Прогресс

По дюжине версий на счету популярных текстовых редакторов, но проще в освоении они не стали. Такой призыв на слуху, но все сводится лишь к сокращению времени, необходимого для начала работы. Едва ли не каждый может создать что-либо полезное в первые же минуты после инсталляции. Затем вы упираетесь в стену. Лучший из современных текстовых редакторов существенно сложнее для изучения, мастерского овладения, нежели его предшественники на CP/M и Apple. Забытые жемчужины прошлого, такие как Electric Pencil и Spellbinder, с высот своих восхищают нас, пользователей пусть более мощных, но и чересчур громоздких программ.

Отчасти причина кроется в том, что первые текстовые процессоры, в основном благодаря добавленным функциям, выросли в текстовые редакторы, которые, в свою очередь, превратились в «текстовые издательства» с еще большими возможностями. Далее системы высокого класса, такие как WordPerfect и Word for Windows, стали почти неотличимы от полноценных настольных издательских программ с точки зрения их возможностей, хотя средства и методы у них довольно разные. Эти системы битком набиты фенечками и фитюльками, и нужно приложить немало усилий, чтобы заставить их работать в нужное время и в нужном месте.

---

[1] Автор обыгрывает слово «upgrade» (англ.); grade – здесь: уклон, подъем; up – вверх. – *Примеч. ред.*

Текстовые редакторы и растущий легион других важнейших инструментов стали жертвами прогрессирующего функционизма, серьезной болезни пользовательских интерфейсов, которое поражает программный продукт в его зародыше и, если ее не остановить, калечит пользователя. Невылеченный, прогрессирующий функционизм может привести к вспышке агорафобии у пользователей перед многочисленными открытыми диалоговыми окнами и даже к мучительному страху перед незнакомыми меню. Иной раз очевидным тому симптомом является смутное тревожное предчувствие, что за нежданным каскадом разворачивающихся меню скрывается замечательная комбинация клавиш, которая соответствует Ctrl-Alt-F5 на вашей прошлой системе.

Успешность продаж

Функции продаются. Обозреватели программного обеспечения уделяют им внимание и выделяют их в аккуратных сравнительных таблицах с помощью различных галочек, пунктирных линий и кружочков. Поставщики программного обеспечения соперничают за большее количество пунктов в списках функций, приводимых в рекламных буклетах. Потребители учатся одним взглядом отличать «полнофункциональный» персональный информационный менеджер от подобного пакета с ограниченной функциональностью. Большинство покупателей никогда не воспользуются всеми пунктами и операциями, однако им приятно просто сознавать, что такие функции есть на всякий маловероятный и непредвиденный случай. Чем больше, тем лучше, верно?

Прогрессирующий функционизм – это тяжелое хроническое заболевание. Синдром проявляется не в количестве функций, а в истории их появления, реализации в программном обеспечении и представлении пользователю. Прогрессирующий функционизм возникает в результате медленного разрастания функциональных возможностей и выражается в неоднородности пользовательских интерфейсов. Такие интерфейсы перегружены индивидуальными особенностями и специальными функциями, которые, как бородавки и карбункулы, могут появляться в самых необычных местах.

Прогрессирующий функционизм истощает систему, поскольку при добавлении нового элемента для него нужно найти разумное место, которого может и не быть. Если бы в ранних версиях текстовых редакторов не было непечатаемых комментариев, заранее предусмотренных в интерфейсе, то функции для их создания и редактирования были бы просто куда-нибудь приклеены. Примером может служить функциональная клавиша для импортирования/экспортирования текстового файла DOS, применение которой маловероятно, но вполне реально.

Прогрессирующий функционизм часто приводит к разбросу взаимосвязанных операций или пунктов выбора по разным частям пользовательского интерфейса. После четвертой или пятой редакции внесение десятков «добавлений» и «исправлений» приводит к тому, что пользовательский интерфейс покрывается слоем небольших довесков, разнообразных переключателей и приложений к меню. Со временем форма интерфейса все больше отражает внутренние программные ограничения. В ней проявляются трудности, с которыми сталкивались программисты при поиске места для размещения функций. Бывалые пользователи, чьи пальцы загрубели и искривились за многие годы применения этих функций, настолько привыкли к ним, что редко задумываются перед нажатием Ctrl-F5-C-C. Таким образом, они поощряют недобросовестных поставщиков программного обеспечения к еще большему интерфейсному варварству, поскольку новые функции не должны мешать применению старых.

На самом деле пользователь хочет иметь (или ему нужен?) простой интерфейс для управления этими сложными системами. К сожалению, современное программное обеспечение очень часто оказывается простым только на поверхности. Вы убеждаетесь в его запутанности при попытке выполнить с его помощью какую-то реальную работу. Естественно, к этому моменту вы уже находитесь за пределами магазина по продаже программного обеспечения и связаны соглашением, упакованным в целлофановую оболочку.

Сталкиваясь со сложностью, человек применяет фрагментирование. Простые или взаимосвязанные компоненты объединяются в фрагменты, которые можно мысленно воспринимать как целые единицы. Хороший интерфейс делает то же самое, сокращая до минимума общее количество идей и методов, которые пользователь должен усвоить. Для этого, во-первых, необходимо тщательное продумывание, а во-вторых, регулярная переработка во избежание беспорядка, вызываемого прогрессирующим функционизмом.

Например, с позиции человека, применяющего текстовый редактор, прямые линии, являющиеся линиями, есть линии. Пользователь хочет нарисовать линию и поместить ее в какое-то место. Независимо от того, проходит ли она под исправленным текстом, или отделяет основной текст от сносок, или формирует элементы таблицы, или обрамляет боковой комментарий, или представляет собой линейку размером 3 пункта в красивом заголовке для письма, – это всего лишь линия. Она выглядит как линия при выводе на печать. Мы называем ее линией, когда говорим коллеге: «Убери эту линию снизу». Однако в большинстве текстовых редакторов каждый из этих вариантов рассматривается как отдельное явление.

В моем старом текстовом редакторе было два основных способа рисования линий, или линеек, как говорят наборщики. Такое разделение в интерфейсе было обусловлено не внешними соображениями, исходящими от пользователя, а внутренними особенностями программы. В старой, более примитивной функции для рисования линий применялись текстовые символы. В последующих, более универсальных продуктах использовалась графика. Как это ни странно, старый способ был полу-WYSIWYG – вы рисовали с помощью клавиш управления курсором и могли сразу видеть результат. Новый способ требовал введения значений для типа линии, позиции и толщины. И даже после этого вы не могли увидеть результат без перехода в режим предварительного просмотра перед печатью, что было в традициях старых текстовых ограничений DOS.

Но представьте, какой текстовый редактор у меня теперь! В нем есть пять совершенно разных способов создания линий. В руководстве все они обозначены по-разному, а доступ к ним осуществляется с помощью разных меню и кнопок. Некоторые линии можно убрать, выделив их и нажав на кнопку удаления, а некоторые – нельзя. Очевидно, что это еще один случай прогрессирующего функционизма. В итоге появляется больше сложностей, чем нужно, а внешне все выглядит более сложным, чем есть на самом деле.

Дарвиновский дизайн

Безусловно, я твердо верю в эволюцию и в гибкость разработки программного обеспечения. Оно постоянно перерабатывается в соответствии с углубляющимся пониманием того, что мы хотим создавать с помощью инструментов и как их можно сделать наиболее полезными для нас. С другой стороны, эволюционный процесс в области разработки программного обеспечения может обернуться мешаниной из частей и компонентов, которые могут работать, но создавать при этом трудности для пользователя. После двух или трех основных релизов пользовательский интерфейс необходимо целиком перестраивать, чтобы для доступа к функциям требовалось меньшее количество элементов управления.

Пока же мне приходится продираться через заросли прогрессирующего функционизма, чтобы разобраться, как в моем чудесном (?) текстовом редакторе для Windows провести от края до края простую линейку размером 3 пункта. Я был уверен, что она там есть, но понадобилось некоторое время, чтобы обнаружить ее среди всех этих прыщей на пользовательском интерфейсе.

Из журнала *Computer Language Magazine*, том 9, № 10, октябрь 1992 г.

36

Назад к истокам

Что же все-таки хотят пользователи? И как можно об этом узнать? Разработчикам программного обеспечения рекомендуется производить такие системы, которые хотят получить их клиенты и покупатели, – системы, более «ориентированные на пользователя». В любой отрасли компании стремятся быть более конкурентоспособными, прислушиваясь к «голосу покупателя» и руководствуясь его мнением. Уже недостаточно производить программное обеспечение с нужными функциями. Чтобы быть легким в изучении и применении, программное обеспечение также должно иметь хороший пользовательский интерфейс. Но как узнать, что потребители хотят видеть в пользовательском интерфейсе? Многие компании обращаются к маркетинговым исследованиям, телефонным и письменным опросам пользователей или потенциальных пользователей, спрашивая их о том, что они хотят получить.

Иногда оказывается, что пользователи и сами не знают этого. Зачастую их желания могут совсем не совпадать с их потребностями. Один из крупных производителей программного обеспечения для ведения бухгалтерского учета собрал от своих покупателей более 15 000 просьб и предложений в промежутке между двумя основными релизами программы. Многие из этих предложений были явно нелепыми, а более тщательное исследование показало, что и часть других предложенных нововведений было бы ошибочно включать в программу.

Желание добра

В народных сказках крестьяне, которым даруют исполнение трех желаний, почти всегда навлекают бедствия на свою деревню. Если вы прямо

спрашиваете пользователей об их желаниях, обычно они просят о дополнительных возможностях. Если вы исполните эти пожелания как покорный джин, то вызовете очередную эпидемию прогрессирующего функционизма (глава 35). Еще хуже то, что опрашиваемые пользователи могут не сообразить, о чем именно нужно просить, но, будучи польщенными таким вниманием и охваченные чувством ответственности, они что-нибудь придумают. После этого возникают затруднения.

Для потребителя пользовательский интерфейс *и есть* система. Для выяснения того, что требуется или что является правильным или неправильным в данной системе, нужно вернуться к истокам. Если не спрашивать, то можно ничего не узнать. Разработчики, которые полагаются только на свой опыт и здравый смысл или доверяют спонтанным откликам и жалобам клиентов, ставят себя в невыгодное положение с точки зрения конкурентоспособности.

Опросы пользователей – наглядный инструмент, однако большинство пользователей не хотят тратить время на заполнение вопросников, а отвечающие зачастую делают это невнимательно и не учитывают деталей. Телефонные опросы или личные встречи могут дать немного больше информации, чем письменные отклики пользователей. Тем не менее во всех опросах есть трудности с вспоминанием. При заполнении вопросника меня сбили с толку 3D-функции в новом графическом пакете, но сейчас я понял, как ими пользоваться, и даже не могу вспомнить, что именно меня смутило поначалу. Важная для разработчиков информация, а именно место, где я нажал не на ту кнопку или получил не тот результат, уже потеряна.

Для большинства разработчиков сайты бета-тестирования являются основным источником обратной связи. Таким способом можно получить ценную информацию, особенно если компания устанавливает тесные рабочие отношения с рядом хороших сайтов. Однако с точки зрения разработки и совершенствования интерфейсов в бета-тестировании есть серьезные ограничения.

В частности, если продукт довольно «хрупок» или сыроват, покупатель может столкнуться буквально с сотнями сбоев или небольших трудностей за один день обычного использования. Попытки отметить все неполадки по мере их появления настолько нарушают ход работы, что все, за исключением самых заинтересованных и озабоченных бета-тестеров, фиксируют только часть возникших осложнений. Обеспечение пользователей диктофонами увеличивает долю выявляемых ошибок, но это также мешает обычному порядку работы.

Для преодоления этих ограничений в больших компаниях по производству программного обеспечения стало модным создавать лаборатории или

исследовательские центры по анализу юзабилити. В этих отделах применяется аудио- и видеоаппаратура. Как правило, в них устанавливается зеркало для наблюдения за использованием системы. Такие центры обычно оснащаются разнообразными компьютерами и рабочими станциями.

Главным недостатком здесь является то, что в лаборатории люди ведут себя не так, как у себя дома, не говоря уже о затратах на создание и функционирование такого отдела. Если вы хотите узнать, как люди работают с тем или иным программным обеспечением, такие исследования нужно проводить *в контексте*. Для этого можно применить какой-нибудь вариант контекстного опроса, например метод, впервые предложенный Карен Холтцблатт (Karen Holtzblatt) и ее коллегами из Digital Equipment Corporation (Holtzblatt и Beyer, 1993 [40]).

Суть контекстного подхода заключается в исследовании действий пользователей при выполнении обычной работы в обычных условиях. Это напоминает исследования на местах, которыми занимаются антропологи и этнографы. За людьми наблюдают во время их работы, а в беседах выясняются детали ее выполнения. Почти не вмешиваясь в рабочий процесс, опытный интервьюер может получить довольно подробную информацию о том, как в действительности применяется продукт.

Конечно, для наблюдения за тем, как некто пользуется некой системой, такая система должна быть. На ранних стадиях разработки часто применяют прототипы, а на более поздних этапах – альфа- и бета-версии. Иногда хороший исследователь может получить полезную информацию всего лишь с помощью бумажных прототипов – простых статических рисунков с предлагаемыми макетами экрана.

Идеи для новых инструментов и компонентов можно получить, наблюдая за применением уже созданных инструментов. Даже небольшие изменения, привносимые пользователями, могут существенно упростить рабочий процесс. В распоряжение пользователей можно предоставить программное обеспечение, производимое вашими конкурентами, чтобы узнать его узкие места. Или же можно изучать действия людей при выполнении работы без программных инструментов. Суть не в автоматизации ручного труда, а в выяснении того, как и где программное обеспечение может быть полезным.

Посещение офисов

Центральная идея контекстных методов – выйти из *своего* офиса и взаимодействовать с пользователями в *их* офисах. Это не только помогает получить более точные данные для проектных решений, но и сокращает расходы. Конечно, создание отдела юзабилити-исследований стоимостью в

полмиллиона долларов может вызвать шум на бирже. Это сигнал рынку – ей-богу, наша компания действительно стремится производить хорошие пользовательские интерфейсы и применять принципы разработки, ориентированные на пользователя. Однако даже простой блокнот, диктофон и поездки на автомобиле могут привести к появлению хорошего программного обеспечения.

С другой стороны, большинству людей не по душе, когда кто-нибудь стоит рядом и наблюдает за их работой. Кроме того, сам процесс наблюдения оказывает влияние на их действия. Когда разработчик интерфейсов или юзабилити-исследователь садится рядом с пользователем и начинает делать записи, то возникают взаимоотношения, изменяющие образ действий пользователя. Разработчик бессознательно дает пользователю подсказки: резкий вдох, быстрые пометки в блокноте, тихое «ах» или «хм-м», откидывание на спинку стула или наклон вперед, или ерзание на стуле. Мириадами путей пользователь получает тонкие сообщения о своих действиях или о том, что он должен предпринять вместо своих невероятно глупых метаний. Возникает большой соблазн «помочь», и типичные разработчики любят вмешиваться и подсказывать, когда клиент не находит оптимального способа применения «их» продукта. Неопытные и неквалифицированные исследователи часто могут вмешаться еще более явным образом: «Ну, давайте я покажу вам более простой способ», «Ах, ну, щелкните здесь мышью», «Нет, совсем не так».

Это работает и по-другому. Пользователь знает, что вы находитесь рядом и наблюдаете. Он знает, что вы лучше разбираетесь в системе, поэтому он рассчитывает на вашу помощь и либо напрямую задает вопросы, либо оглядывается на вас.

В общем и целом, наверное, лучше не находиться рядом. Означает ли это, что нужно возвращаться в офис и строить ту самую юзабилити-лабораторию с зеркалом? Не обязательно.

Простая технология может быть удивительно эффективной. Закрепите видеокамеру за спиной пользователя и сфокусируйте ее на клавиатуру и экран. Рядом с экраном расположите переносное зеркало таким образом, чтобы в камеру было видно лицо пользователя. Вот и все.

В типичном исследовании ведется видеозапись того, как пользователь применяет какой-либо компонент программного обеспечения. Сначала запись просматривают исследователи, чтобы изучить действия пользователя. Возможность видеть его лицо дает исследователям дополнительную информацию о происходящем. Когда видно, что пользователь удивлен, раздражен, озадачен или нетерпелив, это многое говорит о деталях интерфейса.

Затем исследователь просматривает отрывки из этой записи вместе с пользователем. Это необходимо для прояснения намерений или реакций пользователя, его мыслей и действий при работе с тестируемым программным обеспечением. Именно здесь зеркало оказывается полезным. Когда при просмотре люди видят самих себя, особенно выражение своего лица, они часто могут с поразительной точностью вспомнить свой «внутренний диалог» и собственные ощущения, которые были во время записи. Если постараться получить эту информацию и понять ее, то такие данные могут показать, что именно *нужно* вашим пользователям.

В краткосрочном периоде стратегия, когда вы даете покупателям то, что они хотят, или то, о чем они заявили в маркетинговом опросе, может быть успешной. Но в долгосрочном плане, вероятно, более выгодно дать покупателям то, что им действительно нужно. Особенно если упаковка подтвердит, что это как раз то, что покупатель хотел, или то, что, по его мнению, ему нужно. В конце концов, хитрость на благо пользователя не является злом!

Из журнала *Computer Language Magazine*, том 9, № 12, декабрь 1992 г.

37

Цветной язык

Один современный мастер Дзен спрашивает: «Каков цвет хлопка одной ладонью?» Здесь есть о чем подумать, тем более что сейчас мы будем говорить о роли цвета в пользовательском интерфейсе.

Цвет стал важным аспектом графических пользовательских интерфейсов, по крайней мере с точки зрения продаж программного обеспечения. Многие годы я отказывался от приобретения цветного монитора, поскольку моя работа в основном была связана с обработкой текстов. Древняя Hercules-совместимая видеокарта давала мне все необходимое, и за этим плоским монохромным монитором янтарного цвета я мог часами работать без головных болей и усталости глаз. Зачем же тогда переходить на блеклые краски, меньшую разрешающую способность и мерцающее изображение? У меня был цветной монитор – огромный индюк CGA, который шел в придачу к моему первому лэптопу. Но обычно он служил в качестве стола. Я не видел мир программного обеспечения во всем цвете до тех пор, пока не пересел на мою теперешнюю рабочую лошадку. В этой системе есть монитор с прогрессивной разверткой: 72 Гц, 256 цветов, 1024×768 пикселов. Мои глаза открылись.

До того времени я считал цвет только хитрой уловкой при продаже видеоигр и программ для поддержки принятия решений, которые зачастую мало чем отличались друг от друга. Хотя я никогда не сходил с ума от палитры рабочего стола в Windows («Смотри, Мод, какие яркие цвета!»), я подверг переоценке роль цвета в коммуникации. Цвет может быть не просто украшением. Он может стать важной частью взаимодействия

между пользователем и программным обеспечением. Суть, естественно, только в том, когда и как его использовать.

В цветах есть скрытый смысл, в основном вносимый культурой. Однако некоторые особенности восприятия цвета связаны с психологией человека. Дети, даже самые маленькие, проявляют больше интереса к ярким цветам и броским контрастам. Красный, оранжевый и желтый цвета, особенно в черно-белом контексте, привлекают наше внимание. Но люди не всегда реагируют на цвета так, как об этом принято думать. В привычном представлении синий и зеленый цвета являются холодными и успокаивают человека, однако исследования это опровергают. Некоторые изыскания указывают на то, что ярко-розовый цвет может оказывать наибольший успокаивающий эффект в психологическом и физиологическом отношении, хотя большинство людей не любят такой цвет в окружающей обстановке.

В некоторых корпоративных стандартах для ГПИ (графических интерфейсов пользователя) установлено, что потенциально опасные или необратимые шаги следует помечать красным цветом. К сожалению, действия типичных пользователей не согласуются с этой идеей. На самом деле выделение красным цветом увеличивает вероятность случайного выбора данного элемента или пиктограммы. На многих людей красный элемент в меню действует наподобие красной тряпки для быка; заметив его, они просто не могут не щелкнуть по нему мышью.

Цветовые сигналы

Цвет можно применять как дополнительное измерение в процессе коммуникации, способствующее интерпретации сложной информации. Мои дочери обучались в экспериментальной средней школе, в которой применялся новаторский подход к развитию навыков чтения. Если бы английский язык имел строгий фонетический строй и единообразные правила орфографии, то необходимость в системах проверки правописания была бы небольшой, а споры на тему «обучение фонетическим методом или простая зубрежка» вряд ли возникали бы. В английском языке один и тот же звук может быть записан десятками способов. По существу, основные отклонения от общих правил в орфографии и грамматике часто относятся к той базовой лексике, которая ведет начало от протоиндоевропейского языка. Это еще более усложняет задачу юного проточитателя.

В системе Гатагно (Gatagno) «слова в цвете» цвет применялся как дополнительный ключ к «декодированию» звуков в слове. Все формы звука «ау»[1]

[1] [ei]. – *Примеч. науч. ред.*

(«eigh», «ei», «ai», «ey», «a» и т. д.) печатались одним цветом и служили своего рода визуальными тренажерами, на которых дети могли учиться.

Эта замечательная схема кажется эффективной, однако может также привести к крайней глупости при применении цвета в образовательных целях. Один специалист по объектно-ориентированным методам проповедует мультисенсорные образовательные методики, комплексно воздействующие на мозг с помощью звука, цвета и действия. Хотя представленные им видеоматериалы и проецировались на изумительный полноцветный дисплей с высоким разрешением, это была всего лишь лавина обычного скучнейшего монохромного текста. Интерес для правого полушария (или для того восприимчивого семилетнего ребенка, который сидит в каждом системном аналитике) представляли только черно-белые графические изображения, демонстрируемые в нижнем правом углу каждого слайда. Увы, на каждом слайде была одна и та же картинка, которая ничего не иллюстрировала и не разъясняла.

Дети и другие человекообразные существа воспринимают понятия более легко и более точно, если процесс подкрепляется информационными средствами, вовлекающими различные каналы восприятия, особенно если абстрактные понятия представлены яркими и ощутимыми предметами, которые можно подержать в руках. Дети легче изучают алфавит, если они могут взять в руки цветные буквы и поиграть с ними. Хорошие учителя годами применяют такие методы. Успех так называемых CRC-карт в объектно-ориентированном проектировании (Wirfs-Brock и др., 1990 [68]) отчасти объясняется тем, что эти маленькие (3 на 5 или 4 на 6) заменители объектных классов являются конкретными предметами, с которыми разработчик может поиграть при обдумывании различных архитектур и сценариев.

С другой стороны, наш серьезный методист использовал цветные игровые элементы для привлечения аудитории к изучению его любимой методологии. К сожалению, примененные цвета (приглушенные, пастельные тона почти одинаковой насыщенности) мало различались и не были информативными. Педагогические достоинства цветов не были замечены, поскольку карточки, обозначавшие тесно связанные понятия, имели различные, а не схожие или близкие цвета. В то же время карточки, представлявшие совершенно разные понятия, имели светлые сине-зеленые и зелено-синие оттенки, которые были настолько схожими, что при недостаточном освещении в аудитории большинство людей их не различало. Для многих из присутствующих карточки сливались в сплошную серовато-синюю массу. Приблизительно один мужчина из 12 и одна женщина из 20 не различают цвета.

Это приводит нас к важному правилу цвета в разработке пользовательских интерфейсов: никогда не полагайтесь только на цвет для обозначения важного различия между визуальными элементами. Например, если нужно отличать статическую связь классов от динамической связи экземпляров, то недостаточно отобразить одну связь красным цветом, а другую – синим. Линии должны также разниться по толщине или по стилю (сплошная, пунктирная линия и т. д.).

Если цвет применяется как существенная часть пользовательского интерфейса, то скромный разработчик должен сохранить для конечного пользователя возможность выбора или, по крайней мере, последнего слова. В Windows вы можете менять цвет почти каждого элемента, но я никогда не мог сообразить, как изменить цвет текстового курсора в приложениях для Windows. Маленький, худосочный курсор, присутствующий в большинстве текстовых приложений, почти не виден, если он того же цвета, что и текст. В конце концов, я сделал цвет текста темно-синим. Текст стал читаться на белом фоне почти так же хорошо, как и черный, но это намного облегчило поиск курсора.

Цветовые решения

По мере того как цветные дисплеи высокого разрешения начинают преобладать не только в офисах, но и в лекционных и конференц-залах, графика приобретает новое значение. При этом корпоративные спонсоры конференций ошибочно пытаются применять стандартную форму принципа «посмотри и ощути» на всех конференциях, распространяя «шаблоны» для таких презентационных пакетов, как Persuasion и PowerPoint.

Отчасти сложность состоит в том, что шаблоны разрабатывают либо программисты, которые не имеют никакого понятия об эстетике и мало разбираются (или совсем не разбираются) в вопросах передачи информации, либо графические дизайнеры, у которых есть чувство эстетики, но тоже нет глубокого понимания передачи информации. Первые производят все тот же пестрый мусор, который можно увидеть в палитре настраиваемых рабочих столов, а последние создают красивые шаблоны с сочными фоновыми цветами, которые отличают продукты высшего аудиовизуального класса. Типичная комбинация цветов представлена от глубокого пурпурного до ярко-синего, с голубыми заголовками и ярко-зелеными маркерами. Прекрасный вид и незабываемые ощущения! Единственная неувязка в том, что половина аудитории не может прочитать половину слайдов. На одной из недавних конференций почти каждый участник старше 40 лет признался мне, что не мог читать с этих великолепных проекционных дисплеев и вместо этого полагался на распечатки, которые, разумеется, были черно-белыми.

Ваши сотрудники, озабоченные хорошим «видом» и хорошим «ощущением», находятся на неверном пути. Восприятие формы зависит от контрастности. Как правило, чем больше контраст, тем легче читать. Уже давно известно, что при нормальном освещении текст удобнее всего читать, если он напечатан черным шрифтом на белом фоне. Светлое на темном (и другие сочетания цветов) читать значительно труднее.

Это поднимает другой вопрос: шрифты. По шрифтам пользовательского интерфейса иногда можно определить возраст людей, занимавшихся его разработкой. Слишком часто в программном обеспечении применяются или устанавливаются по умолчанию шрифты, которые я называю «до 40», – небольшой размер символов и маленький междустрочный интервал. Эти шрифты могут читать только те, кому меньше 40! Проблема усугубляется дисплеями высокого разрешения, поскольку системные шрифты обычно являются растровыми. Не раз я был свидетелем того, как проваливалась демонстрация замечательного программного обеспечения, так как собравшиеся в демонстрационной кабине не могли прочитать текст меню и диаграмм; они только вежливо кивали головами, как будто понимали, что происходит.

Укрепляющаяся гегемония «презентационных пакетов» порождает и другие трудности. Такие пакеты создают, как я это называю, «шестизарядные слайды» – ну, вы знаете: маркер, маркер, маркер, маркер, маркер, маркер. Наглядные пособия должны быть именно наглядными пособиями. Они должны способствовать пониманию и запоминанию, визуально иллюстрировать, расширять или подкреплять комментарии докладчика. Шестизарядные визуальные материалы являются очень слабым и наименее эффективным методом применения потенциально мощного инструмента. От использования красивых цветов мало толку. Возможности цвета и графики необходимо применять для достижения конечной цели передачи информации, а не просто так, волей-неволей следуя дизайнерскому принципу сэра Эдмонда Хилари (Edmond Hillary) – раз уж они есть.

Подобно Сискелу (Siskel) и Эберту (Ebert), я убежден в том, что некоторые компоненты лучше всего воспринимаются в своей оригинальной, нецветной форме. Иногда цветной язык лучше всего переводить в простой черно-белый. А что касается цвета хлопка одной ладонью, я думаю, он может быть цвета окна.

Из журнала *Software Development*, том 1, № 1, январь 1993 г.

38

Совершенствующиеся середнячки

Лыжные трассы бывают трех видов – зеленые, синие и черные – поскольку лыжники тоже бывают трех видов: начинающие, середнячки и эксперты. Я середнячок, был им многие годы и, наверное, буду середнячком всю свою жизнь. Таких как я инструкторы снисходительно называют классическим «совершенствующимся середнячком», подразумевая под этим, что я – человек среднего возраста, который продолжает улучшать свое мастерство, хотя и не достигает больших и быстрых успехов. Я счастлив. Я люблю кататься на лыжах и даже сумел выжить на некоторых ужасных (или прославленных) черных трассах с двумя ромбами, пропуская повороты на более мягкие и спокойные спуски. Но все же, главным образом, я стараюсь держаться поближе к дружелюбным синим квадратам и успокаивающим зеленым кругам.

Катание на лыжах помогает узнать многое о взаимоотношениях между пользователями и системами и о том, как разработчики программного обеспечения могут улучшить эти взаимоотношения. Это было бы жестоким испытанием – учиться катанию на лыжах на каком-нибудь из тех крутых бугристых склонов, которые предназначены для опытных лыжников. Некоторые новички, в основном подростки до двенадцати лет, после легких склонов норовят сразу же отправиться на трассы для могула[1], но я всегда подозревал, что они просто-напросто биомутанты. Для начального обучения большинству из нас подойдут широкие и легкие склоны, обозначенные этими милыми зелеными кружочками.

[1] Могул – разновидность фристайла, езда по крутым горкам. – *Примеч. ред.*

С другой стороны, вы сможете научиться только элементарным приемам, если весь день будете проводить на простых склонах. Рано или поздно, вам придется перейти к более сложным синим и черным (иногда черно-синим) трассам.

Опытные лыжники, эксперты, одолевающие слалом и перелетающие через бугры, внушили нам, что это единственный способ катания на лыжах. Но совершенствующиеся середнячки тоже могут наслаждаться необыкновенными минутами, спускаясь с Небесной долины (Heavenly Valley). Внизу вы видите жемчужину – сияющее бирюзовое озеро Тахо (Lake Tahoe), к востоку – пустыню и коричнево-желтое море, простирающееся до горизонта. Морозный утренний воздух щиплет лицо, а под лыжами скрипит свежевыпавший снег.

Трехфазный дизайн

Хорошо, но какое отношение это имеет к разработке программного обеспечения? Мечтая о катании на лыжах у Тахо, я хотел рассказать о трехфазной модели пользовательского интерфейса (Constantine, 1994 [13], 1994 [24]). Согласно трехфазной модели, у пользователей системы есть различные потребности на различных стадиях своего развития, а значит, пользовательский интерфейс должен разрабатываться с учетом этих изменяющихся потребностей. Для этого в программном обеспечении необходимо предусмотреть разные виды интерфейса для пользователей с разными уровнями владения продуктом. Каждый из этих вариантов должен иметь свои характерные черты и отдельные технические задачи. Как и сеть трасс на хорошем лыжном курорте, эти компоненты интерфейса должны быть не разделены, а тесно связаны между собой.

В трехфазной модели предусмотрены три интерфейса: интерфейс приобретения, переходный интерфейс и интерфейс производства. Интерфейс приобретения – это система, которую начинающий пользователь видит при первой встрече с продуктом. Хороший интерфейс приобретения позволяет начинающему пользователю сразу приступить к работе. Интерфейс производства дает возможность опытному, полностью подготовленному пользователю с большой эффективностью решать сложные задачи. Между этими двумя интерфейсами находится переходный интерфейс, предназначенный для совершенствующихся середнячков. Они уже прошли стадию медлительного и неуверенного новичка, но еще не стали и, наверное, никогда не станут опытными пользователями, способными при помощи «горячих клавиш» легко и стремительно лавировать между многими приложениями, которые каскадом расположены на экране. Трассы на лыжных курортах для начинающих, совершенствующихся и опытных лыжников различаются, но соединяются между собой. Точно

так же в хорошей системе есть трехуровневый интерфейс, позволяющий легко переходить от одного уровня к другому.

Обделенное большинство

Я думаю, что сегодня одной из самых серьезных проблем в разработке пользовательских интерфейсов является преобладающее внимание к гладким склонам программного обеспечения. Потребности опытных пользователей неохотно обеспечиваются с помощью довольно грубых и уродливых «продвинутых» сервисов или кое-как составленного набора горячих клавиш и неудобного средства составления макросов. А «совершенствующиеся середнячки», которые могут быть самой многочисленной и самой важной категорией пользователей, почти всегда игнорируются.

Если система полезна и хорошо разработана, то пользователи перестанут быть в ней новичками. Лишь некоторые из них со временем станут экспертами или очень опытными пользователями, а большинство, вероятно, будет проводить свои дни в качестве совершенствующихся середнячков. Их потребности отличаются от запросов экспертов или новичков; им нужен пользовательский интерфейс, позволяющий постоянно и последовательно увеличивать свои знания о данном программном продукте и улучшать навыки его применения. Такой интерфейс не должен наказывать за незнание. Он не должен неожиданно швырять их на склоны диалогов с двойным черным ромбом, которые понравятся только программисту, пишущему на C++.

В лучшем случае коммерческие программные продукты и инструменты разработки программного обеспечения имеют зеленую трассу и черную – и почти никаких трасс между ними. Переходный интерфейс, помогающий бывшему новичку повышать свою эффективность и приобретать новые навыки, должен разрабатываться как особый набор функций, в котором учитываются знания начинающего и знания, необходимые эксперту. Настраиваемые панели кнопок и инструментов хороши, но недостаточны, поскольку они либо навязывают пользователю стандартный набор (всегда слишком сложный или слишком ограниченный), либо вынуждают его самому определять назначение функций.

Весь интерфейс мог бы быть модальным, предлагая режим для начинающих пользователей и ряд промежуточных режимов со все более разнообразным и расширяющимся набором функций. Функции, предназначенные для опытных пользователей, вероятно, не следует предлагать новичкам, разве что с помощью четко обозначенных «подъемников» или «трасс». Специальный флажок в настройках говорит системе, каким уровнем пользователь владеет или какой уровень хочет применять. При запуске интерфейс может загружаться в одном из стандартных режимов.

Как это ни странно, многие бесплатные компьютерные игры очень легко настраиваются в соответствии с уровнем игрока, а дорогие рабочие пакеты и инструменты разработки программного обеспечения поставляются в одной, прокрустовой конфигурации.

Карты

Обычно горнолыжные курорты предоставляют и другие полезные возможности, которые должны входить в каждый программный интерфейс. Карты лыжных трасс показывают, каким образом можно добраться из одного места в другое, и помогают лыжнику найти или обойти определенные трассы. В программных системах необходимо предусмотреть свои «карты трасс», служащие визуальными руководствами по расположению тех или иных функций в лабиринте кнопок, диалоговых окон, а также выпадающих и всплывающих меню. Онлайновые справки – по крайней мере, справочные системы популярных графических пользовательских интерфейсов – имеют ограниченную ценность. Перемещаться по такой онлайновой «помощи» так же сложно и неудобно, как и по самим меню и диалоговым окнам, и, что еще хуже, справка и интерфейс организованы по-разному.

На лыжных склонах и на картах трассы четко размечены. В программном обеспечении заголовки меню часто дают неполное представление о том, что за ними скрывается. В свою очередь, в системе «помощи», отделенной от функций интерфейса, которые она предназначена описывать, сплошь и рядом применяется другая терминология. Если я пытаюсь выяснить, как убрать сноску в моем текстовом редакторе, эта чертова справка должна просто показать мне, как это сделать. Для этого справочная система должна последовательно открыть нужные меню и указать на пункт, который следует использовать.

Разные инструменты и методы и даже разные правила и принципы можно применять в переходном интерфейсе, но не в интерфейсе приобретения и интерфейсе производства. Юзабилити-тестирование (глава 36), в котором исследуется взаимодействие пользователей с системой, может помочь точно настроить интерфейс приобретения. Однако, скорее всего, такое тестирование не будет полезным для переходного интерфейса и интерфейса производства. Почему? Потому что к тому времени, как начинающий пользователь превратится в середнячка или эксперта, в видеокамере уже закончится пленка или у психолога рассеется внимание.

Благодаря хорошему проектированию и всестороннему юзабилити-тестированию в некоторых ГПИ удалось создать разумный интерфейс приобретения. Например, интерфейс в Apple Macintosh был разработан так, что-

бы с момента открытия упаковки до начала выполнения полезной работы проходило не более 20 минут.

Проблема состоит в том, что по мере прогресса пользователя типичный интерфейс остается неизменным. Начинающие пользователи системы нуждаются в программных учебных роликах. Ролики помогли многим детям научиться ездить на велосипеде, но они не предназначаются для постоянного использования. По существу, такие ролики не помогают детям научиться ездить на велосипеде. Они учат ездить на велосипеде с присоединенными опорными роликами! Для формирования навыков равновесия и движения на велосипеде опорные ролики нужно снять.

Итак, когда вы неохотно готовитесь к разработке и тестированию пользовательского интерфейса для вашего следующего продукта, вы обнаруживаете, что одного интерфейса недостаточно. Нужна зеленая, синяя и черная версии, да еще с картой трасс и указателями. А может быть, даже необходимы отсоединяемые опорные ролики. Хорошо, эта аналогия небезупречна.

Ладно, поеду кататься на лыжах!

Из журнала *Software Development*, том 1, № 2, февраль 1993 г.

39

Пригодны ли вы

2.70! Это число может вам ни о чем не говорить, но курс машиностроения, который обозначен этими цифрами в Массачусетском технологическом институте, довольно знаменит. Студенты получают наборы разнообразных деталей и затем соревнуются в проектировании и построении лучшего робота, собирающего шарики для пинг-понга, или компьютеризированного передвижного моста, или другого устройства, выдуманного преподавателями. Профессор Вуди Флауэрз (Woody Flowers), научный сотрудник Государственной службы радиовещания (Public Broadcasting System, PBS) и основатель курса 2.70, хочет помочь будущим конструкторам и инженерам создавать более практичные продукты, которые действительно работают для человека. Его университетские курсы способствовали организации соревнований между студентами высшей школы. Даже учеников средней школы он вооружает видеокамерами для того, чтобы вокруг себя они находили и документировали недружелюбные для пользователя или просто непригодные конструкции и схемы.

Если бестолковое проектирование так легко распознать, то почему на миллионах индикаторов видеомагнитофонов или микроволновых печей по всему миру загораются цифры 12:00 или 88:88? Почему программное обеспечение, основанное на ГПИ, не ускоряет нашу работу? Несмотря на созданные «отделы изучения человеческих факторов» и «лаборатории юзабилити-тестирования», ведущие производители продолжают выпускать глупое программное обеспечение с серьезными изъянами в юзабилити, которые даже двенадцатилетний ребенок может заметить с другого конца комнаты.

Представьте: мой компьютер сообщает, что 8 283 файла занимают почти 550 Мбайт дискового пространства (о, конечно, это были старые добрые времена... еще до появления Win95 и 98). Для удержания лидерства в области консультирования необходимо разбираться в этих цифровых джунглях и находить в них нужный материал. Размер файла и дата его создания зачастую помогают быстрее найти нужную версию или вариант, поэтому нам нужно видеть эти данные, когда мы собираемся открыть файл из приложения. При открытии или сохранении файлов, когда мы видим их и думаем о них, нам удобнее всего заниматься уборкой в доме – перемещением файлов или папок, переименованием или удалением. Каждый знает об этом, но только небольшая часть продуктов предоставляет такую возможность. Нам необходимо расширение/дополнение, способное стандартизировать и расширить набор диалогов «Открыть», «Сохранить», «Сохранить как» в Windows. Каждое приложение должно отображать информацию о статусе файлов и их описание непосредственно в диалоговых окнах, а также поддерживать в них выполнение рутинных операций с файлами.

Это хорошая идея. В некоторых больших продуктах эта возможность предусмотрена. В одном из таких чудесных пакетов вы можете нажать Ctrl-O, и в верхней левой части диалогового окна открытия файла (там же, где и всегда) появляется знакомое комбинированное окошко для выбора имени файла. Внизу располагается прокручиваемый список файлов, в котором, как всегда, видно слишком мало файлов и совсем не видно их статуса. Выделите одно из имен файлов, и в сером окне внизу справа (!) появится дата, время создания и размер выделенного файла. За раз можно увидеть описание только одного файла. Нужно постоянно переключать внимание между верхней левой и нижней правой областями экрана. Два разных и далеко отстоящих друг от друга визуальных элемента приходится мысленно совмещать; описания файлов нельзя сравнивать без запоминания; вместо быстрого и легкого просматривания списка нужно выполнять неудобную, последовательную, механическую операцию. Такая неразумная конструкция нарушает основные принципы дизайна интерфейсов и не позволяет пользователю выполнять свою работу.

Речь не идет о чем-то сверхсложном; эти изъяны очевидны даже для неподготовленного подростка. Все ведущие компании-разработчики занимаются подготовкой специалистов по пользовательским интерфейсам и создают отделы по проведению юзабилити-тестирований. Из разговоров с сотрудниками этих компаний я знаю, что они думают о юзабилити и, по-видимому, имеют представление о принципах разработки пользовательских интерфейсов. Однако на пути от намерений к готовому программному обеспечению что-то происходит. Я изучаю причины, по которым хорошие компании поставляют никчемное программное обеспечение. Детальное исследование еще проводится, но кое-что стало ясно и на данном этапе.

Должностные обязанности

Для получения удобного программного обеспечения разработка пользовательского интерфейса должна стать чьей-то обязанностью. Без ответственности и подотчетности хорошие пользовательские интерфейсы не появятся. Об этом я говорил в течение многих лет. Теперь я начинаю думать, что забота о юзабилити – это обязанность *каждого*. В команде каждый разработчик должен обращать серьезное внимание на юзабилити конечного продукта, начиная с первого мозгового штурма и до появления коробки с готовым продуктом.

Одним из путей к поддержанию такой сосредоточенности является проведение систематических юзабилити-инспекций (Constantine, 1994 [23]). Они напоминают традиционный анализ программ и проектных решений, но особое внимание они уделяют пользовательскому интерфейсу и юзабилити – для выявления изъянов в этой области. Такие инспекции заставляют разработчиков думать о пользователях и юзабилити программного обеспечения. Одной-единственной инспекции, проводимой непосредственно перед принятием окончательной версии продукта, недостаточно. Разработчики и специалисты по построению интерфейсов должны внимательно проверять модели рабочих процессов, первые бумажные прототипы, начальные варианты дизайна, рабочие прототипы, а также альфа- и бета-версии программного продукта. Как показал Якоб Нильсен, юзабилити улучшается с каждой инспекцией (Jacob Nielsen, 1993 [52]). С каждой новой инспекцией разработчики узнают немного больше о хорошем дизайне пользовательского интерфейса и ошибках, которых следует избегать.

Слишком мало, слишком поздно

Разработчики программного обеспечения часто отбрасывают полезную информацию о юзабилити продуктов, поскольку получают ее слишком поздно. Ведущие производители инструментов разработки и другого программного обеспечения, упакованного в целлофан, любят демонстрировать свою приверженность юзабилити. Они демонстрируют сияющие блеском новые лаборатории тестирования, в которых при помощи сложной видеоаппаратуры и мощных компьютеров можно наблюдать и оценивать действия репрезентативных конечных пользователей при использовании программного обеспечения. Эмпирическое юзабилити-тестирование более эффективно, чем юзабилити-инспектирование. Тем не менее у лабораторных тестирований есть серьезные недостатки. Во-первых, юзабилити-тестирование проводится слишком поздно. Для получения реалистичной оценки взаимодействия между конечным пользователем и программным обеспечением требуется рабочая система – обычно это бета-версия. Но к

этому времени все ошибки в пользовательском интерфейсе уже совершены. Обнаружить все изъяны будет сложно, почти невозможно. Основная структура и функции программного обеспечения закреплены в коде, поэтому обычно уже слишком поздно предпринимать что-либо, кроме доработки и улучшения поверхностных деталей пользовательского интерфейса. В результате пользовательский интерфейс, который лишь слегка отполирован, остается неудобным. Главные изъяны зачастую скрываются в архитектуре программы и пользовательского интерфейса – в согласованности различных функций и в базовой модели, на основе которой создано программное обеспечение.

Зачастую юзабилити-тестирование не обнаруживает даже фундаментальные ошибки в архитектуре и не выявляет необходимость в коренной перестройке взаимодействия между программным обеспечением и пользователем. Оно помогает найти лишь небольшие изъяны в данном общем подходе. Подобно тому как с помощью тестов не найти путь к безошибочному коду, с их помощью не найти путь к пользовательским интерфейсам, свободным от изъянов.

Даже экспертные оценки, которые обычно дешевле и более эффективны, чем юзабилити-тестирование, часто проводятся слишком поздно, чтобы быть полезными. В одном приложении тщательное инспектирование пользовательского интерфейса выявило немногим более сотни ошибок, которые были отсортированы по степени влияния на юзабилити продукта. Клиент зафиксировал несколько поверхностных изъянов из нижней трети списка и только одну из них посчитал действительно серьезной. Все остальные ошибки были расценены как слишком сложные для исправления, поскольку они были связаны с базовой архитектурой программы.

Поверхностные функции

Разработка программного обеспечения сводится к созданию различных функций. На рынке выигрывает тот, кто предлагает наибольшее количество функций или самые необычные возможности. Однако программное обеспечение с притягательной 3D-графикой, перегруженное функциями, может оставаться несовершенным с точки зрения пользовательского интерфейса. Юзабилити программного обеспечения касается общей организации пользовательского интерфейса. При анализе юзабилити не рассматриваются функции программы и их взаимодействие, призванное облегчить работу пользователя. А значит, каждый участник команды должен обращать внимание на архитектуру и детали интерфейсов в течение всего процесса разработки.

Возможно, производители программного обеспечения уделяют столько внимания композиции и внешнему виду не потому, что это так важно для юзабилити, а потому, что кроме этого они ничего не умеют отлаживать.

Из журнала *Software Development*, том 2, № 4, апрель 1994 г.

40

Редактирование интерфейсов

Моя компания только что закончила редактировать пользовательский интерфейс. Официально контракт предусматривал проведение «экспертной оценки юзабилити», но на самом деле мы занимались редактированием. Хороший пользовательский интерфейс начинается с хорошей архитектуры, но для обеспечения удобства и практичности программного обеспечения необходимо выявить и устранить изъяны в пользовательском интерфейсе. Часто для исправления ошибок приходится не один раз проводить критический анализ, пересмотр и доработку. В этом состоит процесс редактирования, а навык редактирования является одним из важнейших навыков для разработчиков программного обеспечения и прикладных программ. Редактирование программы, книги или пользовательского интерфейса во многом очень похожи.

Я всегда завидовал писателям, чьи пальцы с легкостью парят над клавиатурой, создавая первые черновики, которые можно считать окончательным вариантом. Я пишу медленно, иногда мучительно, зачастую нерегулярно, а первые плоды моего труда редко бывают съедобными. К счастью, я довольно хороший редактор. За многие годы я научился «отключаться» от своих сочинений, чтобы спустя время окинуть их свежим придирчивым взглядом и найти места, требующие переделки. Статья может пройти через четыре или пять раундов безжалостной редактуры, прежде чем я смогу с удовлетворением сказать, что теперь она готова.

На самом деле многое из того, что мне известно как писателю, я узнал в процессе многолетней работы в качестве редактора книг, журналов и брошюр. Если заниматься этим долго, то редактирование входит в плоть и

кровь. Редакторский склад ума становится способом восприятия мира. Некоторые привычки редактирования настолько во мне укоренились, что стали почти автоматическими и бессознательными. Я могу просматривать книгу в магазине и заметить опечатку прямо при перелистывании страниц, как будто слова с ошибками напечатаны жирным шрифтом и прыгают на страницах, чтобы обратить на себя внимание.

Разговор о меле

Такая способность иногда приводит к неловким ситуациям, особенно если не умеешь себя сдерживать. Однажды я ждал, когда освободится столик в одном из тех небольших стильных французских кафе, в которых меню написано мелом на доске. Без колебаний и раздумий я подошел к доске и по привычке исправил ошибку в слове. Мои дочери были в шоке.

Некоторые люди от рождения пишут грамотно. Но не я. Программист-писатель П. Дж. Плоджер (P. J. Plauger) однажды заметил, что он органически не способен писать слова неправильно. Он сказал это перед тем, как предложить мне приглашать кого-либо для вычитки моих рукописей. Вместо того чтобы последовать его совету, я сам научился писать правильно.

В этом мне помогли два обстоятельства. Начало положила многолетняя работа в качестве редактора журнала, пытающегося спасти малопонятные, неэстетичные и громоздкие сочинения ученых и исследователей. Завершили дело программы проверки орфографии. В конечном итоге после многолетнего применения текстового редактора со встроенной проверкой орфографии я стал писать правильно. Программы проверки орфографии – это замечательные учителя. На самом деле они не проверяют текст и не исправляют ошибки, а просто сообщают вам о том, что они не распознали какое-то слово. Первые программы проверки орфографии даже не предлагали варианты для исправления, поэтому вам приходилось обращаться к словарю для того, чтобы убедиться в своей правоте либо исправить ошибку. В любом случае вы сразу получали обратную связь и активно занимались самообучением. Поэтому теперь мне трудно не заметить ошибку в правописании.

То же самое относится и к редактированию пользовательских интерфейсов. Когда вы начнете думать в терминах юзабилити, вам будет трудно не замечать изъяны юзабилити, где бы они не скрывались. Вы обращаете внимание на то, что указатели в больнице могут вводить посетителей в заблуждение или сбивать с пути. Вы видите, что назначение кнопок на дистанционном пульте видеомагнитофона трудно запомнить, а объяснения вашего соседа по поводу направления движения неоднозначны.

Конечно, находить недостатки в других системах нетрудно – намного труднее увидеть изъяны в своей собственной работе. Например, экспертная оценка юзабилити, упомянутая мной ранее, была втиснута в довольно плотное рабочее расписание. Наш отчет перечислял более сотни изъянов юзабилити в проекте пользовательского интерфейса, выполненном нашим клиентом. Отчет прошел тщательную вычитку, прежде чем мы передали его для печати и переплета в один из круглосуточных копировальных центров. Утром в день проведения консультации, когда вместе с нашим клиентом, его помощниками и его начальником мы собрались вокруг стола, я открыл отчет и в тот же миг заметил вопиющую опечатку на самой первой странице. Всегда ли так происходит?

Редакторы и писатели знают об этом эффекте и пользуются им. Если вы хотите вычитать свой собственный текст, то уберите его на пару дней в ящик письменного стола перед тем, как перечитать. Если вы хотите выявить изъяны юзабилити в своем пользовательском интерфейсе, то отложите схемы, наброски и снимки экрана в сторону на день или два, и займитесь чем-нибудь другим. Когда вы обратитесь к ним вновь, вам будет легче посмотреть на вашу работу свежим взглядом и увидеть в ней возможные недочеты.

Как стать чемпионом

Редакторы журналов часто подходят к своей работе с позиции заведомо несведущего читателя. Пит Бикфорд (Pete Bickford), который некогда носил титул чемпиона по пользовательским интерфейсам (User Interface Champion) в Apple, так же оценивает пользовательские интерфейсы. Он говорит, что его работа подразумевает вхождение в роль самого глупого пользователя на планете. Это такой настрой ума, при котором вы отключаете все, что вам известно о компьютерах, или приложениях, или программировании. Если вы сможете посмотреть в верхний правый угол окна Windows-приложения и увидеть кнопку для увеличения, то у вас есть перспектива стать более эффективным редактором интерфейсов.

Однако нарочитой невосприимчивости и намеренного незнания недостаточно. Редактирование предполагает не только критическое отношение, но и творчество. Вы не можете просто вычеркивать, вам нужно еще и вписывать. То, что вы пишете, определяется вашей способностью суждения, основанного на представлении о хорошей форме. Так же как вы не можете быть редактором газеты, не зная основ журналистики и правил грамматики, вы не можете редактировать пользовательские интерфейсы, если не понимаете принципы юзабилити. Вы должны распознавать чрезмерную сложность или неудобный процесс. Далее, вы должны знать, как упростить запутанную структуру или сгладить неровности операций.

Вам даже нужно научиться видеть то, чего нет. Примером может служить отсутствующая обратная связь для пользователя или функция, которая не видна в нужном месте и в нужное время.

Защита от ошибок

При редактировании пользовательских интерфейсов полезно занимать негативную позицию типичного нью-йоркского театрального критика или книжного обозревателя, который обращает внимание на слабые стороны и ошибки, на проблемы и изъяны. Это может быть самым простым занятием для программистов и инженеров, которые по природе своей способны к критике. Поставьте двух из них перед белой доской и вы получите три твердых мнения с последующим обменом артиллерийскими снарядами-словами. Они могут быстро замечать недостатки в проектах своих коллег или ограничения в предложенных решениях. С другой стороны, их типичная реакция на прямую или косвенную критику – это объяснение.

Поэтому главное правило для редактирования интерфейсов – никогда ничего не защищайте и не объясняйте. Почти всегда найдется объяснение, почему вы разместили эту кнопку именно там или почему изображение пиктограммы обновляется, когда щелкают мышью по какому-нибудь элементу. Но главным с точки зрения юзабилити являются не внутренние причины, а внешние последствия. Для сохранения способности к творческой критике нужно удерживать себя от поиска оправданий и объяснений для собственного дизайна и программного решения.

Редактирование может многому научить с точки зрения писательского ремесла. Точно так же, чем больше вы будете практиковать критическое редактирование пользовательских интерфейсов, тем больше вы узнаете о том, что есть хорошая архитектура пользовательского интерфейса. Если вы заметите, что ваша рука тянется к синему карандашу, когда кто-то просит вас взглянуть на снимок экрана, возьмите карандаш. Возможно, это начало вашей карьеры редактора пользовательских интерфейсов.

Из журнала *Software Development*, том 3, № 11, ноябрь 1995 г.

41

Сервис

Это можно назвать вниманием к клиенту или «голосом покупателя», но речь идет о сервисе. Я получил возможность узнать об этом больше, когда впервые открывал банковский счет в Австралии. Я послушно заполнил все бланки, предъявил все документы и внес первый вклад. Буду ли я пользоваться чеками? Да, конечно. Нужна ли мне карточка-ключ? Какая карточка? Карточка для EFTPOS (читается «эфтпос»). Слово EFTPOS в Австралии является повседневным. «Эта очередь по EFTPOS?» – такой вопрос можно услышать в супермаркете. Или покупатель может сказать: «Я хотел бы оплатить этот заказ по EFTPOS». Работники финансовых служб знают, что EFTPOS означает Electronic Funds Transfer, Point Of Sale (терминал системы электронных платежей). В Австралии покупка продуктов, оплата счетов, получение наличных – все это осуществляется при помощи EFTPOS.

Конечно, я попросил, чтобы банк выдал мне эту карту, тем более что по правилам я мог получить ее только лично. Я получил карту, и она неделю пролежала у меня в кармане, пока я не пошел снять немного денег.

Банкомат не работает

На экране первого банкомата, к которому я подошел, красовалась надпись «Банкомат не работает», поэтому мне пришлось подойти к другому. Это было древнее на вид создание с однострочным плазменным экраном, предлагающим вставить карту. Я поискал щель для карты. На поверхности банкомата не было никаких символов, помогающих правильно ее вставлять. Мне пришлось сделать три попытки, прежде чем расположен-

ные внутри валики смогли втянуть карту в эту машину. По требованию я ввел свой PIN-код, запросил наличные деньги, ввел тип счета, набрал сумму и подтвердил введенную информацию, нажав клавишу OK. Ничего не произошло. После долгой паузы появилось сообщение о том, что для данного счета эта операция не активизирована. Мне предлагали обратиться к представителю моего банка.

Я не понимал, что означало это сообщение. Может быть, я ввел неверный PIN-код или совершил какую-то другую ошибку. За мной уже выстраивалась очередь, но я попытался еще раз. И снова неудачно. То же самое произошло и на другой машине. В конце концов я зашел в один из филиалов моего банка, где у справочной стойки коротким «Да?» меня приветствовала сотрудница сервисной службы.

Я объяснил, что у меня не работала EFTPOS-карта. «Хорошо, что вы делали?» – спросила она таким тоном, как будто со мной говорили в кабинете директора школы. Я ответил, что попытался получить наличные деньги. «Нет, что вы делали?» – спросила она опять с сильным ударением на последнем слове. И вот, призвав на помощь все свои способности, я начал пересказывать ей в точности все то, что я делал: выбрал клавишу «Снять деньги» из зеленых клавиш, потом выбрал «Основной чековый счет» из синих клавиш... «А, ну вот». Что вот? «Ну, вот видите. У вас же нет чекового счета». Я стал объяснять, что уже открыл чековый счет, в подтверждение доставая чековую книжку из кармана пальто. «Нет, это не чеки. Мы не открываем чековые счета частным лицам. У вас сберегательный счет. Если бы вы правильно нажимали на клавиши, у вас не было бы этой проблемы». Я был готов потребовать ее начальника, когда увидел надпись на ее карточке – как раз она и была этим начальником. Я что-то пробурчал, повернулся и ушел.

Очевидно, что в этом случае проявилось плохое отношение или даже недостаточная квалификация представительницы банка, но ориентация на клиента или ее отсутствие может также зависеть и от программного обеспечения. Системы, которые мы разрабатываем и поставляем своим клиентам, могут сильно влиять на качество тех услуг, которые оказываются с их помощью. Если бы в моем случае программное обеспечение было хорошим, то мне не пришлось бы идти в банк, чтобы прослушать лекцию от Хельги Ужасной. Программе, которая применялась в банкоматах, было известно, что у меня нет чекового счета. Она знала, что у меня был один и только один так называемый «сберегательный» счет, и могла сообщить об этом.

Небольшие детали создают или разрушают сервис. Как и многим консультантам, мне удобно заказывать программное обеспечение и оборудование по телефону. Однажды, перечитывая последний пункт сделанного

заказа, я все еще вертел в руках мою кредитную карту. И тут я внезапно понял, что вместо банковского счета компании я сообщил оператору данные моей личной кредитной карты. Я попросил ее изменить номер карты. Последовала длинная пауза и вздох. «Мне придется заново оформить этот заказ», – сказала она. «Разве вы не можете просто вернуться назад к этому полю и изменить его?» – «Нет, это поле уже заблокировано после выхода из того окна, поэтому все придется начать сначала». Клик, клик. «Назовите, пожалуйста, свою фамилию еще раз». Я заявил, что уже называл свое имя. «Но оно в том заказе, который нужно удалить».

Как видно, безмозглые программисты, которые создавали эту программу, даже никогда не задумывались о возможном применении этой системы. Они не предусмотрели способ вернуться назад или отменить часть совершенной операции. В результате – лишняя суета для продавца и по крайней мере один раздраженный покупатель, подумывающий о размещении следующего заказа в другой компании.

Все может быть и по-другому. Хорошее программное обеспечение помогает людям оказывать более качественные услуги. Расскажу о своем опыте общения с представителями винного клуба «Cellar Door Direct» в Южной Австралии. Через несколько месяцев после получения рекламного объявления я решил позвонить им и заказать немного вина. Все, что у меня было, – это листок с моими каракулями. К сожалению, код того вина, которое было мне нужно, не отображался на экране у телефонного оператора. Я слышал, как она «листала» файлы с данными, пока не нашла то, что нужно. «Это Jarrondale'92 Shiraz, правильно?» Она назвала цену, которая показалась мне слишком большой, – я напомнил, что это вино было выставлено на распродажу. «А какая цена указана у вас?» – спросила она. Я назвал ей верную, на мой взгляд, сумму. «Я больше не вижу здесь этой цены, но давайте я сделаю исправление». Клик, клик. И через пару дней у моей двери стоит посылка. Сочетание гибкого программного обеспечения с гибким персоналом позволило укрепить отношения с покупателем.

Служба доставки

Приведу еще пример. В Сиднее впервые создали замечательную службу под названием «Cuisine Courier». Вы выбираете блюда из меню местных ресторанов, и за какие-то $2 заказ доставляют прямо к вашей двери. Позвонив в эту службу, вы просто сообщаете им свой телефон, а система предоставляет остальные данные. «Мистер Константин? Вы живете по тому же адресу? Хорошо, по какому меню вы хотели бы сделать заказ?» Вы называете краткий код, и оператор подтверждает его, сообщая полное название ресторана, – и так для каждого блюда. «Заказ будет доставлен через 45 минут». Через 45 минут в дверь стучат, и можно начинать пиршество.

Как им это удается? Просто их система хорошо разработана! Как только ваш заказ подтвержден, система отправляет факс с компьютера в нужный ресторан. К тому времени когда курьер прибывает в ресторан, заказ уже готов и к нему приложена инструкция по доставке. И нет никакого беспокойства, как говорят австралийцы, потому что все работают с одними и теми же данными.

Как разработчики программного обеспечения мы должны помнить, что такие небольшие детали, так же как и общая архитектура системы, имеют значение в оказании услуг конечному потребителю. От наших решений зависит, будет ли «голос клиента» недовольным или благодарным.

Из журнала *Software Development*, том 4, № 3, март 1996 г.

VII

Удобные объекты

Введение

Когда в 1986 году я вернулся в мир компьютерного программного обеспечения, тема объектов горячо обсуждалась, хотя многие разработчики и их начальники смотрели на объекты с подозрением. Они казались очередным модным веянием, чья 15-минутная слава вскоре потускнеет. Однако мне объекты виделись не такими простыми. Я полагал, что объекты представляют собой еще один краеугольный камень, заложенный в основу разработки программного обеспечения (см. главу 24). При поддержке Джорджа Шасела (George Schussel) из Digital Consulting, Inc. я организовал и провел Симпозиум по объектно-ориентированным системам (Object-Oriented Systems Symposium) – переезжающий с места на место форум, который в течение нескольких успешных лет служил трибуной для таких лидеров и мыслителей, как Гради Буч (Grady Booch), Дэйв Томас (Dave Thomas) и Ребекка Уирфс-Брок (Rebecca Wirfs-Brock).

Скептицизм и осторожность все еще сохраняются и всегда будут сохраняться, но революция почти завершилась. Объектно-ориентированный метод стал преобладающим подходом в проектировании и разработке программного обеспечения. Однако каждой революции присуща своя ортодоксальность, и объектно-ориентированный подход не является исключением. Революционные защитники новой парадигмы стали рьяно ругать все, что имело хоть какое-то отношение к отвергнутой и дискредитированной парадигме процедурного программирования и пользовательским ситуациям. Хотя сейчас разработка пользовательских ситуаций применяется повсеместно, трудно поверить, что когда-то их считали ересью (Lockwood и Constantine, 1993 [49]). Однако многие из старых и новых приверженцев объектной парадигмы считали пользовательские ситуации контрреволюционными. Тем не менее как революция, так и контрреволюция рано или поздно уступают прагматике, и большинство объектно-ориентированных методов, в том числе сильно расхваливаемый Unified Process (Jacobson и др., 2000 [43]; Kruchten, 2000 [46]), в конце концов стали включать в себя определенные формы моделирования пользовательских ситуаций.

Эти две центральные темы в современной практике разработки программного обеспечения – пользовательские ситуации и объекты – тесно связаны с темой, о которой шла речь в предыдущей части книги – юзабилити. Как объекты, так и пользовательские ситуации применяются – иногда весьма элегантно, а иногда неумело – в разработке пользовательских интерфейсов. И то и другое в свое время провозглашалось решением проблемы юзабилити. И то и другое находило неправильное понимание и применение почти так же часто, как и помогало достичь успеха. В этой части книги представлена серия тесно взаимосвязанных статей, которые были написаны для журнала *Object Magazine*. В них исследуется сложное переплетение программных объектов и пользовательских ситуаций с одной стороны и проектирование пользовательских интерфейсов и юзабилити программного обеспечения – с другой. Читатель, у которого это описание вызовет интерес, сможет найти более подробное изложение представленных идей в книге «Software for Use» (Constantine и Lockwood, 1999 [30]).

42

Объекты, которые раздражают

Графические пользовательские интерфейсы не имеют ничего общего с юзабилити – они связаны с графикой. Какой смысл в фантастическом ГПИ, если вы не применяете его для рисования красивых картинок? А с учетом того, что качество изображения на экране монитора растет, почему бы не анимировать ваши замечательные картинки? Целью является повышение продаж, а мишенями – обозреватели и покупатели программного обеспечения, которые смотрят (не очень внимательно), а потом пишут восторженные рецензии или покупают лицензию на использование системы. Потребители, которые больше заботятся о том, чтобы действительно выполнять работу, – это совсем другая группа людей, поэтому вы больше слышите о пользовательских интерфейсах, чем о юзабилити; больше говорится о философии проектирования, ориентированного на пользователя, чем о реальной поддержке процесса работы; больше рассуждений о необходимости прислушиваться к голосу покупателя, чем о внимании к проблемам пользователей.

Возьмем объектную технологию. Если когда-то всему надлежало быть «структурированным», чтобы заслуживать внимания, то теперь все обязано быть «объектно-ориентированным». Если что-то не «объектно-ориентировано», оно не считается современным. Если программное обеспечение не построено из «объектов» и «классов», то в нем не может быть ничего хорошего. В примитивном представлении объекты считаются естественными и интуитивными. Более того, они политически корректны, потому как могут использоваться повторно!

Сегодня пользовательские интерфейсы стали объектно-ориентированными. По крайней мере, так написано на коробке.

На первый взгляд

Представьте: вы подумываете об отдыхе, зная, что объекты надежно спрятаны в соответствующих библиотеках классов с огромными возможностями повторного использования – глубоко внутри кодового ядра устойчивого программного обеспечения. И как раз в ту минуту, когда вы подумали, что получили полное представление о полиморфизме, универсальности и перманентных объектах, а объектная технология обрела смысл, объекты неожиданно начинают появляться на экране вашего монитора. Если объекты полезны внутри программного обеспечения, то они должны быть полезны и вне его. Если объекты полезны для разработчиков, то они должны быть полезны и для пользователей. Во всяком случае так утверждается в рекламе.

На что же похожа эта объектная революция, когда она переходит из языка программирования в пользовательский интерфейс? Представьте такую картину. На экране показана модель карманной записной книжки-ежедневника. Вы можете щелкать мышью по вкладкам для перехода к любому разделу. Вы можете щелкнуть мышью в углу «страницы» и «перелистнуть» ее. Ну надо же, как удобно! Конечно, большую часть времени пол-экрана не задействовано, а для чтения надписей на вкладках нужно поворачивать голову – сначала по часовой стрелке, потом против, поскольку при перелистывании страниц текст переворачивается с одной стороны на другую. Ах, но зато это выглядит таким знакомым, а ведь каждый знает, что знакомые метафоры облегчают применение программного обеспечения, не так ли? Особенно не следует придавать значение тому, что на экране super-VGA помещается меньше информации, чем на старом DOS-дисплее с 80 колонками, хотя информация отображается более мелким шрифтом. Зато все в цвете. Это очень приятно, если только вы не являетесь одним из двенадцати мужчин, которые не различают цвета. Этот «персональный информационный менеджер», должно быть, очень полезная штука, раз в нем так много замечательных картинок. Но постойте, это не просто картинки, это объекты! Их можно перемещать, с ними можно работать, с ними можно взаимодействовать, и это делает вашу жизнь такой приятной.

В другом персональном информационном менеджере на экране показана картинка вращающегося файла предметного указателя (наподобие Rolodex™). Щелкаете по крышке, она открывается, и вы видите небольшие карточки, на которых написаны имена, адреса и телефонные номера. Щелкните и потяните за «ручки», расположенные по обе стороны, и карточки начнут переворачиваться.

Некоторые называют это объектно-ориентированной технологией, но из уст пользователей, на практике применяющих все это, можно услышать

менее вежливые названия и описания. Мы вовсе не против лишних движений мышью, которые повышают риск вызвать сбой из-за многократных нажатий. Мы вовсе не против того, чтобы эти восхитительные картинки отвлекали нас от работы. В конце концов, ведь это объектно-ориентированный пользовательский интерфейс!

Что же такое объектно-ориентированный интерфейс? Пиктограммы, панели кнопок и выпадающие меню – ваш персональный вариант ГПИ? Или же это методы «указать-выбрать» (point-and-click) и «перетаскивание» (drag-and-drop) , являющиеся догмами непосредственного манипулирования? (Кстати, если приводить точные аббревиатуры, то мы сейчас говорим об OOI GUI WIMP.) Может быть, это любой пользовательский интерфейс, созданный с помощью языка объектно-ориентированного программирования, или же пользовательский интерфейс любой системы, разработанной с помощью объектно-ориентированных методов, независимо от языка? Возможно, это означает, что интерфейс усыпан картинками «реальных» объектов. Или то, что пользователи могут взаимодействовать с системой, передавая объектам сообщения («Документ, распечатай себя»).

В действительности «объектно-ориентированный интерфейс» означает всего лишь то, что объектно-ориентированная революция завершилась. Новая парадигма в мышлении получила такое распространение, что даже начальники отделов маркетинга и копирайтеры открыли для себя объекты. Объекты продаются!

Конечно, мы все знаем, что объекты лучше своих предшественников. Они лучше потому, что объекты интуитивны и картинки интуитивны, а картинки – это объекты. Таковы рассуждения. Но даже поверхностный взгляд на непомеченные пиктограммы в каком-нибудь новом популярном пакете электронной почты или текстовом редакторе, или презентационной программе выявляет, насколько интуитивными являются эти интерфейсы. Сколько из этих пиктограмм можно правильно понять с первого взгляда? Сколько из них можно легко запомнить? Сколько пиктограмм можно запомнить меньше чем за год ежедневного использования? Что бы ни писали на коробке, в которой эта программа продавалась, если на понимание пиктограммы уходит больше одной секунды, она не интуитивна! Если вы забываете, на какую кнопку нажимать для добавления столбцов, она не интуитивна. Если вы постоянно открываете не то меню для создания заголовка страницы, оно не интуитивно.

Играет ли это какую-нибудь роль? Какое значение имеет форма пиктограммы или детали копирования с помощью «выбрать-перетащить» (click-and-drag)? Большое. Небольшие ошибки и небольшое раздражение постепенно накапливаются, замедляя работу, а не ускоряя ее. Работа – это то, что некоторые из нас пытаются выполнить, часами просиживая за

рабочим столом. Программное обеспечение, которое противоречит человеческим способам мышления и решения задач, неизбежно приводит к росту количества ошибок. Если программа заставляет пользователя выполнять лишние шаги, или думать дважды, или переосмысливать что-либо, то независимо от того, насколько он свыкнется с программой, количество его ошибок будет больше неизбежного минимума. На самом деле эти дополнительные ошибки вовсе не являются человеческими. Они вызваны ошибками в интерфейсе, в программном обеспечении.

Для потребителя пользовательский интерфейс и *есть* система. Это то, что конечные пользователи видят и с чем взаимодействуют при выполнении своей работы. Пользователи не видят, насколько удачным было применение множественного наследования, или насколько верно построен какой-то метод, или насколько тщательным было повторное использование классов из библиотек. Они не видят большую часть из того, над чем разработчики программного обеспечения так сильно пыхтят. Они видят пользовательский интерфейс. Точка. Пользовательский интерфейс – это не только то, что можно увидеть и ощутить. Это также и то, как система работает, как она проявляет себя. Для пользователей самым важным в интерфейсе любой части программного обеспечения является возможность системы помогать в их работе и способы этого добиться.

Позволяют ли красивые пиктограммы, хорошая анимация, диалоги в картинках и говорящие кнопки сделать программное обеспечение удобнее для реальной работы? Работа – это определенное поведение. Оно слагается из действий, шагов и процессов, взаимосвязанных с другими процессами. Работа состоит не из объектов, а из операций. С точки зрения выполнения работы более важны не объекты, а цели: какая работа выполняется, как и почему. Если разработчики будут это учитывать, они смогут создавать более совершенные системы – такие системы, которые дадут пользователям возможность лучше выполнять свою работу.

Физическое заблуждение

Еще на заре компьютерной эры, когда проектирование программного обеспечения было связано лишь с обработкой данных, аналитики и программисты поняли горькую истину: простая автоматизация старых ручных методов приводит к созданию плохих систем, являющихся всего лишь неудобными эмуляциями, а не полезными нововведениями. Сегодня у нас есть «объектно-ориентированные интерфейсы» – и что произошло со старыми продуктами Rolodex™? Они переместились с рабочего стола на так называемый «рабочий стол»! Создатели этих бездумных копий ручных систем не учли того факта, что изначально продукты Rolodex™ сами по себе были технологическим прорывом. Они позволили отказать-

ся от неудобной технологии индексных карточек, которые, в свою очередь, применялись вместо менее удобных конторских книг. Rolodex™, или DayTimer™, или DayRunner™, которые кажутся весьма полезными, когда вы держите их в руках, становятся просто неудобными и бестолковыми, когда они моделируются на экране.

Все дело в том, что объектная технология испытывает затруднения из-за наивной мифологии. Айвар Джекобсон (Ivar Jacobson), гуру в объектной технологии, отмечает, что наивные объектные модели приводят к программному обеспечению, которое ненадежно в условиях изменяющихся требований и целей. Наивные объектные модели основаны на примитивном поиске программной среды для объектов из «реального мира». Далее поведение объектов отображается в объектных классах. Практика описания поведения с помощью кнопок и пиктограмм, представляющих объекты из реального мира, является не менее наивной и ведет к созданию ненадежных интерфейсов.

Устойчивая архитектура программного обеспечения включает в себя множество компонентов (будь то объектные классы или функциональные модули), которые возникли в ответ на потребности разработчиков. Эти важные компоненты скорее являются частью решения, чем частью задачи. Они должны быть созданы, а не просто найдены среди предметов, разбросанных по комнате во время какой-то глупой «объектной игры». Для разработки хорошей внутренней архитектуры – или хорошей архитектуры интерфейса – необходимо, чтобы программист был думающим индивидуумом, способным решать задачи, а не забавляющимся художником, изображающим «репрезентативную реальность». Вместо наивного закрепления каких-то действий за «дублерами» физических объектов хороший разработчик распределяет свойства и назначение компонентов на основе здравых принципов разработки программного обеспечения. В соответствии с ними связанные функции взаимосвязаны, а несвязанные объекты – разделены.

Каждый, кто разбирается в объектной технологии, знает, что программные объекты не являются объектами реального мира и ведут себя иначе. Мы можем вполне оправданно ожидать, что объект **Пациент** в больничной информационной системе будет знать свой диагноз и состояние своей страховки, полученной от **Страхователя**, но мы не можем рассчитывать на то, что в ответ на «постукивание по колену» у этого объекта будет дергаться какая-нибудь программная «нога». Объекты не являются «реальными» несмотря на то, что об этом радостно говорят учителя объектного простодушия. Объекты представляют собой абстракции, существующие в умах аналитиков, разработчиков и программистов. Они относятся к «естественному» миру не больше, чем любые другие абстракции, которые люди придумывают для решения задач рабочего дня.

Между объектной технологией и хорошим пользовательским интерфейсом существует реальная связь, но она тонкая и отчасти косвенная. При хорошей вспомогательной инкапсуляции и сокрытии информации объекты позволяют производить более рациональное и эффективное разбиение задачи на подзадачи. Качественная объектно-ориентированная архитектура программного обеспечения помогает четко отделить реализацию пользовательского интерфейса от реализации базовой функциональности приложения и в то же время сохранить их взаимосвязь. Качественная объектно-ориентированная разработка программного обеспечения подразумевает разделение объектных классов, представляющих и обеспечивающих разные аспекты задачи (Jacobson и др., 1992 [44]). Интерфейсные объекты, которые инкапсулируют интерактивное поведение с характеристиками и элементами внешних интерфейсов, могут быть отделены от других объектов – от объектов предметной области, которые инкапсулируют данные и поведение, связанные с понятиями и структурами предметной области, а также от управляющих объектов, которые осуществляют координирование и взаимодействие множества объектов.

В той мере, в какой объекты являются технологией для более эффективного и широкого повторного использования, объектно-ориентированные методы могут способствовать разработке более совершенных пользовательских интерфейсов, поскольку непротиворечивое повторное использование помогает построить непротиворечивые пользовательские интерфейсы. Непротиворечивые пользовательские интерфейсы более удобны в применении, поскольку в них меньше информации, которую пользователям нужно изучать и запоминать для эффективной работы. Непротиворечивость позволяет пользователям повторно применять приемы и процедуры, которые были изучены с какой-то целью в других частях интерфейса или в других системах. Например, установка атрибутов выделенного текста с помощью панели инструментов не должна существенно отличаться от установки атрибутов по умолчанию для всего документа. Как бы согласованность ни была желанна, ее трудно достичь, когда каждый элемент или блок пользовательского интерфейса задумывается и строится по отдельности. Непротиворечивые пользовательские интерфейсы существенно легче создавать с помощью стандартизованных библиотек классов, а также проектных хранилищ объектных классов повторного использования.

Мимолетная «мудрость» нынешнего дня, относящаяся к взаимодействию человека и компьютера, гласит, что хорошие пользовательские интерфейсы состоят из обозримой коллекции знакомых объектов. Однако хорошая организация пользовательского интерфейса, способствующая выполнению реальной работы, намного важнее свободной навигации. Пользовательский интерфейс должен отражать внутреннюю и понятий-

ную организацию работы, а не физические объекты ее текущего воплощения. Это означает необходимость учитывать намерения пользователей и обдумывать способы помочь им в их работе. Это означает необходимость оставлять простые вещи простыми. Сохранение номера телефона коллеги не должно включать в себя вращение экранного телефонного диска. Исправление ошибки не должно вызывать пламя на экране, имитирующее сожжение корзины с мусором.

Хорошие объектные интерфейсы – это всего лишь хорошие объектные интерфейсы. Это интерфейсы, которые поддерживают выполнение работы и предоставляют людям возможность принимать решения. Они берут на себя то, с чем компьютеры могут справиться лучше, а людям позволяют выполнять то, к чему люди более способны. Разработчики программного обеспечения могут заниматься более полезным делом, чем просто копировать несложные элементы современных никчемных программ. Печатающее устройство – это не просто механический карандаш. Личный информационный менеджер, сделанный на совесть, – это не просто карманная записная книжка, отображенная на экране монитора.

Что касается разработчиков программ и пользовательских интерфейсов, то вопрос заключается в том, будут ли они применять объектно-ориентированное программное обеспечение для создания объектных интерфейсов или же объекты будут только вызывать у них раздражение.

По материалам журнала *Object Magazine*, июль 1993 г.

43

Глубокое понимание

На что похож объект? Кто его применяет и как? Какую пользу он приносит?

В известном смысле пользователи и обращенные к ним пользовательские интерфейсы есть *raison d'être*[1] объектной технологии. Именно для решения задач, связанных с графическим представлением информации и взаимодействием пользователя с устройствами, позднее названными электронно-лучевыми дисплеями, Айвен Сазерленд (Ivan Sutherland) первым сформулировал многие базовые понятия объектно-ориентированного программирования. Действительно, мой коллега-преподаватель однажды заметил, что почти все, что есть в современной объектной технологии, можно найти в диссертации по Sketchpad, представленной Сазерлендом в 1963 году (Sutherland, 1963 [61]).

С тех первобытных времен список книг, посвященных объектно-ориентированным методам, вырос до огромных размеров, однако едва ли мы сказали что-то новое о пользователях и пользовательских интерфейсах. Мы хвалимся нашим чудесным бесшовным проектированием и привносим в него термины из прикладных задач, однако наши методы почти не способствуют пониманию пользователей, их языка, а также обучению систем говорить на этом языке.

---

[1] Разумное основание, смысл (фр.) – *Примеч. перев.*

Где пользователь?

Будучи преподавателем в Сиднейском технологическом университете, я проанализировал некоторые из самых известных и успешных книг по объектно-ориентированному анализу и проектированию. Это была высококлассная выборка. Сначала я проверил каждую книгу по этой теме в моем офисе, потом прошел по коридору к профессору Брайану Хендерсон-Селлерзу (Brian Henderson-Sellers) и пробежался по его книжным полкам. Конечно, я не удивился, что у него много таких же книг, как у меня, но все же нашел 15 других, недавно вышедших книг по объектно-ориентированной разработке, включая почти все из самых известных и цитируемых. В целом, в книгах было около 6 000 страниц. И лишь 161 страница относилась к обсуждению аспектов, связанных с пользователями, пользовательскими требованиями, юзабилити или пользовательскими интерфейсами. Почти три четверти из этих страниц принадлежали трем книгам. Больше половины книг содержали три или меньше страниц, посвященных пользователям и применению продуктов. Треть книг не содержали никаких подходящих упоминаний в предметном указателе. В одной книге нашлась целая страница, посвященная полезности программного обеспечения, но она не попала в указатель. Наверное, сейчас ситуация меняется, что подтверждается появлением нескольких книг, в том числе: «Designing Object-Oriented User Interfaces» (Разработка объектно-ориентированных пользовательских интерфейсов) (Collins, 1995 [9]) и «Object Modeling and User Interface Design» (Объектное моделирование и разработка пользовательских интерфейсов) (van Harmelen, 2000 [63]). Согласитесь, выход этих книг нисколько не преждевременен, и ни одна страница этих книг не является лишней.

Для многих разработчиков, применяющих объекты, все еще не ясно, какую лепту объектная технология может внести в пользовательские интерфейсы. В конце концов, объектная технология – это технология «под капотом». Это технология реализации, а не технология взаимодействия. Эволюция объектно-ориентированных методов, как и остальных методов разработки программного обеспечения, происходила в направлении усовершенствования механизмов, находящихся «под капотом». Мало кто обращал внимание на такие мелочи, как удобное расположение замка капота.

Мы не должны чересчур винить себя за эти упущения. В них повторяется история многих новых технологий, которые при появлении зачастую ориентировались на задачи реализации и функционирования, и только потом в них проявлялось более пристальное внимание к внешним деталям. Так было и в первых проектах по выпуску автомобилей, в которых разработка надежных двигателей и решение проблем трансмиссий оправ-

данно имели больший приоритет, чем создание комфортных кресел и удобных средств управления. Конечно, этот подход изменился. Возможно, и в объектной технологии пришло время подумать о комфорте и удобстве пользователей.

Большинству пользователей не интересно, что находится «под капотом» тех систем, которые они применяют. Дон Норман (Don Norman), эксперт по юзабилити и защитник удобства применения технологий с точки зрения пользователя, говорит просто: для потребителя пользовательский интерфейс и есть система (Norman, 1988 [53]). Но бывают исключения. При определенных обстоятельствах технология, спрятанная «под капотом», становится важной для пользователя. Она становится важной, если легковой автомобиль не может разогнаться и безопасно обойти грузовик. Или когда компилятор не может выдавать качественный код. Технология приобретает значение, если двигатель нужно ремонтировать или регулировать через каждые 5 000 км. Она важна, если Windows необходимо перезагружать через каждые несколько часов из-за утечки памяти в пакете офисных презентаций.

Пользовательские интерфейсы связаны как с внешним видом, так и с поведением программного обеспечения. Если какой-то инструмент визуальной объектно-ориентированной разработки позволяет достигнуть довольно большой производительности, то технология, находящаяся под капотом, начинает буквально высовываться из-под приборной доски. Долгие ответы на запросы и инертная прокрутка данных на экране могут стать таким же элементом пользовательского интерфейса, как и пиктограммы на панели инструментов.

По мере пополнения списка литературы и расширения дискуссий об объектной технологии многие мимоходом высказывались или писали об объектно-ориентированных пользовательских интерфейсах. И лишь немногие брались за более сложную задачу – рассказать о том, что это за интерфейсы и чем они полезны.

От ГПИ к ООПИ

Что же такое ООПИ[1]? Еще одна аббревиатура с двойным «О» или что-то еще? Существует ли настоящий объектно-ориентированный пользовательский интерфейс? Захочет ли его применять уважающий себя человек? Если это всего лишь ГПИ с возможностью непосредственного мани-

[1] ГПИ – графический пользовательский интерфейс, ООПИ – объектно-ориентированный пользовательский интерфейс.

пулирования визуальными объектами, то добавление ОО почти ничего не меняет для ПИ.

Одна из постоянных трудностей в мире ОО становится особенно неприятной, когда речь заходит о пользовательском интерфейсе. Эта трудность – различие (или чаще его отсутствие) между объектами в программном обеспечении или программных моделях и объектами в банальном смысле, то есть объектами внешнего физического мира, который так часто с высокомерием называют «реальным миром». Многие новички в объектной технологии попали в ловушку, думая, что для успеха в работе с объектами достаточно найти объекты в реальном мире, настрочить немного кода для классов, к которым принадлежат объекты, и программное обеспечение уже готово для поставки. Что может быть более бесшовным? Как убедительно показали Айвар Джекобсон (Ivar Jacobson) и другие методисты, в качественных программных системах многие из наиболее важных объектов не соответствуют напрямую объектам из реального мира или предметной области.

Банальное и техническое значения слова «объект» все еще объединяют или путают. Согласно повседневной терминологии, все пользовательские интерфейсы содержат в себе объекты, которыми пользователи могут манипулировать. Тот факт, что некоторые объекты представляют собой монохромные символы, управляемые с помощью команд с клавиатуры, не делает их менее достойными именоваться «объектами».

В своей книге на эту тему Дэйв Коллинс (Dave Collins, 1995 [9]) предложил «определение» ООПИ, основанное на трех характеристиках: (1) пользователи воспринимают объекты и воздействуют на них; (2) пользователи классифицируют объекты исходя из поведения объектов; (3) все интерфейсные объекты объединены в одну общую согласованную концептуальную модель. Звучит неплохо, но первый пункт относится к пользователям, а не к пользовательским интерфейсам. Можно возразить, что восприятие объектов присуще человеческому мозгу, если речь идет об обычных объектах, либо является феноменом, существующим только для разработчиков, если речь идет о программных объектах. Второй пункт, касающийся классификации, также относится к пользователям, а не к интерфейсам. Это правило почти наверняка не подходит для всех людей (или даже для кого-то одного), если принимать во внимание все возможные обстоятельства. Люди классифицируют обычные объекты по многим свойствам и показателям, которые не относятся к поведению. Например, вы можете сказать, что все политики действуют почти одинаково, однако ваш друг при этом похож на Билла Клинтона. Что касается третьего предполагаемого свойства ООПИ, то согласованность модели является характеристикой любых хорошо организованных пользовательских ин-

терфейсов: хорошие интерфейсы внутренне непротиворечивы и воспринимаются как одно целое.

Согласно другому взгляду, ООПИ – это интерфейсы, которые обнаруживают логику и структуру объектно-ориентированной парадигмы, представляя пользователям объекты и иерархии классов, методы и сообщения как среду взаимодействия с системой. У такой позиции есть некоторые основания, но также есть и определенные ловушки. Если вести тщательное и наивное рассуждение от противного, то получается, что пользователи будут взаимодействовать с объектно-ориентированным интерфейсом посредством перемещения сообщений от одного объекта к другому, однако вряд ли такое взаимодействие можно считать естественным или очень практичным.

Ни один из пользователей, чей здравый смысл не повредился за годы болтовни с объектно-ориентированными компиляторами, не скажет, что его взаимодействие с графическим пользовательским интерфейсом заключается в «отправке сообщений объектам». Пользователи не отправляют сообщения объектам. Они что-то делают с ними или с их помощью. Даже так называемое «прямое манипулирование» является своего рода неправильным направлением, сбивающим с прямого пути. Пользователи ничем не манипулируют напрямую, когда перемещают мышь по столу для перемещения курсора или изменения картинки. Наша способность научиться таким действиям и воспринимать их как разумные является еще большим даром человеческого мозга, чем способность программировать компьютер для участия в этой пантомиме курсоров и пикселов.

Поверхностные элементы

Есть ли вообще смысл выставлять механизмы объектной технологии в интерфейсе, предназначенном для конечных пользователей? Как правило, если логика и структура программы становятся видны на экране, что-то является неправильным. Это признак конструкции вида «изнутри наружу», в которой интерфейс становится тонкой и пятнистой кожей, покрывающей бугорчатые внутренности. Наоборот, конструкция вида «снаружи внутрь» означает то, что внутренние компоненты и их внешние проявления отражают потребности пользователя, а не структуру программы.

Так называемые «объекты-фабрики» («factory objects»), помещенные в интерфейс, можно привести в качестве примера полезного исключения. Визуальные компоненты, которые при щелчке мышью или при перемещении порождают новые экземпляры класса, являются интересной и недостаточно используемой формой объектно-ориентированной технологии пользовательских интерфейсов. «Блок» «листов для факса» можно при-

менять для открытия чистой формы, готовой для заполнения и последующей передачи. Не каждому такая «документо-ориентированная» схема покажется самой удачной, поэтому следует подумать и о других подходящих вариантах.

Коллинс противопоставляет ООПИ другим, не объектно-ориентированным ПИ, но тот соломенный заборчик, который он возводит, на самом деле является древним интерфейсом командной строки. В интерфейсах командной строки глагол (команда) ставится перед дополнением (параметром). Иногда утверждается, что грамматика типа «дополнение-глагол» в ООПИ является более естественной и более гибкой – выбираете картинку и щелкаете по инструменту изменения цвета или перетаскиваете документ к принтеру.

Если вы хотите получить удобный пользовательский интерфейс, то навязывание этого синтаксиса пользователю во имя чистоты объектно-ориентированной технологии – не очень подходящая идея. Плотник сначала берет гвоздь, а потом подносит к нему молоток – не наоборот. В задачах, решаемых в реальном мире, объекты как дополнения и объекты как операторы, как правило, не являются взаимозаменяемыми. Можно ударять молотком по гвоздю, но проводить обратную операцию бессмысленно.

Если одной из сильных сторон объектно-ориентированной технологии назвать сохранение соответствия с языком предметной области, то это относится к конструкции ООПИ. На самом деле хороший дизайн ПИ всегда согласуется не только с языком пользователей и предметной областью, но также с задачами из этой предметной области, которые интересуют пользователей. Такое соответствие достигается только тогда, когда у нас есть время для осмысления намерений пользователя и его действий.

Чтобы спуститься с абстрактных высот, рассмотрим одну частную, но показательную задачу: разработка взаимодействия со степлером в системе управления документами. Так же как чистые листы бумаги или документы можно скрепить вместе и получить единое целое, так и степлер в ПИ позволяет собирать совокупности документов в группы, которыми пользователь может манипулировать как единым целым.

В реальном мире работник офиса может поднести стопку документов к степлеру и скрепить их. Это соответствует «объектно-ориентированной» идиоме drag-and-drop: щелкаем мышью по пиктограмме стопки документов, перетаскиваем ее к пиктограмме степлера и отпускаем. Обратите внимание, что операция почти идентична выделению объекта нажатием кнопки мыши, затем перемещению указателя к степлеру и выполнению еще одного нажатия кнопки мыши, но только с одной существенной разницей. Для многих пользователей операция drag-and-drop оказывается одной из наиболее сложных для освоения. Они предпочитают последова-

тельно щелкать мышью по двум объектам. Для них это проще, чуть быстрее и гораздо надежнее. Однако другие пользователи, зараженные интерфейсами командной строки или развращенные многолетним выполнением реальной работы вместо использования компьютеров, могут предпочесть сначала выбрать инструмент для скрепления и потом щелкнуть мышью по стопке документов, которые нужно скрепить.

Любое утверждение о том, что способ drag-and-drop – самый естественный и правильный, еще более ослабляется следующим аргументом. При разумной реализации стопка документов не должна оставаться в зубах объекта «степлер», так же как распечатанный документ не должен оставаться над пиктограммой принтера. Даже пуристы MacMetaphor признают возможность компьютера выполнять какую-то часть этой манипуляции. Они охотно согласятся на то, чтобы перетаскиваемые объекты чудом перескакивали на свои начальные позиции.

В реальном мире люди не путают метафоры объектов и функций. Они могут поднести бумагу к степлеру или степлер к бумаге и применить его, ничего не перепутав. Хороший пользовательский интерфейс дает такую же гибкость или даже увеличивает ее. Не создавая путаницы, такой интерфейс допускает все основные варианты взаимодействия: (1) перетаскиваем стопку бумаги к степлеру; (2) щелкаем мышью по стопке бумаги, потом щелкаем по степлеру; (3) щелкаем мышью по степлеру, потом щелкаем по стопке бумаги; (4) перетаскиваем степлер к стопке бумаги. Хорошая реализация интерфейса предоставляет пользователю все эти возможности. Степлер должен выглядеть солидно, всем своим видом показывая, что к этой пиктограмме можно перетаскивать другие. А при щелчке мышью пиктограмма степлера должна изменяться так же, как и любая кнопка. При щелчке мышью она должна «утопать». При приближении перетаскиваемой стопки бумаги ее цвет должен изменяться, чтобы показать готовность степлера. Если же вы пытаетесь перетащить к нему принтер, степлер должен ответить отказом. Соответствующая анимация, изображающая работу степлера (со звуком или без), должна сообщать пользователю о завершении операции. Когда вы берете степлер как инструмент, ничего при этом не выделив, указатель превращается в пиктограмму ручного степлера.

Является ли такой подход к ПИ объектно-ориентированным? Наверное, объектные пуристы ответят «нет», тогда как объектные евангелисты, вероятно, будут поддерживать этот подход – как все другое, что, по их мнению, является замечательным и будет работать. Конечно, можно реализовать это на каком-нибудь объектно-ориентированном языке или воспользоваться ретро-методами и собрать интерфейс из процедур. Для пользователей важнее всего соответствие интерфейса той работе, которую они выполняют.

Выбор технологии реализации может представлять интерес для пользователей, когда речь идет о добавлении новых возможностей или изменениях в соответствии с новыми требованиями. В конечном итоге, надежная технология, позволяющая быстро и качественно интегрировать новые возможности, приносит больше пользы, чем технология, которая не допускает изменений. Если под объектно-ориентированным пользовательским интерфейсом понимать интерфейс, построенный из взаимодействующих программных объектов, то в этом может быть большой выигрыш. Конечно, для создания такого интерфейса мы должны хорошо его понимать и действовать согласно нашему репертуару, конструируя гибкие, повторно используемые компоненты.

По материалам журнала *Object Magazine*, сентябрь 1996 г.

44

Абстрактные объекты

В фантастическом фильме «Темная звезда», являющемся классикой андеграунда, смелый лейтенант Дулитл пытается прочитать лекцию по феноменологии умной бомбе, готовой разнести себя вместе с кораблем. «Вселенная – это абстракция, – поспешно объясняет он, – и все, что мы когда-нибудь узнаем о ней, является абстрактными построениями нашего собственного сознания, пропущенными через наши ограниченные и ненадежные чувства». Отсюда вывод: не делайте потенциально фатальных выводов на основе возможно неверных построений.

Переходя к конкретным случаям, можно сказать, что объектная технология построена на основе абстракций. Классы абстрагируют свойства от множества экземпляров, а абстрактные классы действуют как понятийные ячейки для организации конструктивных идей. Даже экземпляры программных объектов являются тонкими абстракциями, не более чем призраками в машине, в лучшем случае дублерами, заменяющими отсутствующих актеров или существ из внешнего мира.

Временами, когда мы учим эту темную процедурную деревенщину, которая приходит в классы для избавления от своих структурных ограничений, мы начинаем с того, что обращаем внимание на реальные экземпляры стульев, или работников, или видеокассет, находящихся поблизости. Рано или поздно для ориентирования в объектном мире студенты начинают думать в терминах программных объектов и строить устойчивые абстракции. Это не новая задача с точки зрения обучения людей. До появления объектов существовали абстрактные типы данных. В те времена, когда модули в машинах были простыми подпрограммами, разложение

цельных задач на абстрактные функции лежало в основе хорошего программного проектирования. По правде говоря, вся разработка программного обеспечения основана на абстракциях и абстрактных моделях. Абстракция дает нам возможность думать о целом, объяснять противоречивое и непредставимое и исследовать внутреннее содержание будущих программ, не создавая их. С позиций разработки, абстрактное мышление является сложным и тонким методом, для освоения которого детям требуются годы. Некоторым же взрослым никогда не удается освободиться от устойчивого буквализма, но было бы неприлично называть их имена.

Абстракция на бумаге

Что касается остальной части нашей дисциплины, юзабилити-инжиниринг и проектирование пользовательских интерфейсов являются специальностями, которые можно считать отстающими составляющими разработки. Или, как сегодня можно сказать, «абстрактно востребованными». Разработчики механизмов взаимодействия и проектировщики пользовательских интерфейсов записывают свои мысли на бумаге или в компьютере. Их изображения почти всегда очень похожи на то, что они разрабатывают. Они делают эскизы или тщательные рисунки, макеты или прототипы. Иногда рисунки бывают грубыми, одноцветными и неэстетичными, но все же они выглядят как экраны, формы и диалоговые окна, напичканные меню и панелями инструментов, списками и переключателями. Истинные юзабилити-специалисты и графические дизайнеры могут даже рисовать рожицы на своих штриховых рисунках или человечков на пояснительных заметках с карандашными расчетами. Они составляют детальные и точные сценарии и увешивают свои офисы диаграммами, которые могли бы привести в восхищение работников студии Диснея.

Может показаться, что в этих фосфоресцирующих изображениях, мелькающих на стеклянном экране, есть что-то точное, пусть обманчивое. Даже дисциплинированные разработчики, которые структурируют код с помощью цветных схем или определяют алгоритм поиска с помощью математических моделей, внезапно начинают думать буквально, когда дело доходит до разметки пользовательских интерфейсов.

Появление превосходных и мощных инструментов визуального проектирования, таких как Delphi компании Borland или Visual Age компании IBM, по-видимому, способствовало еще большему распространению примитивного метода проектирования и даже возвысило его до уровня стандартной практики. При помощи современных инструментов люди проектируют пользовательские интерфейсы, не проектируя, а конструируя их. Они соединяют объекты вместе, манипулируя реальными окнами списков, сетками данных и другими компонентами из растущего разнообразия

приспособлений современного ГПИ. Конечно, ярые сторонники точности и объектно-ориентированной чистоты тут же заметят, что некоторые мнимые объектно-ориентированные среды визуального проектирования лучше было бы назвать средами, основанными на методе экземпляров, а не объектно-ориентированными. Однако как самые лучшие, так и самые худшие инструменты находятся почти на одном уровне, если вести речь о конкретных элементах управления. Внутри этих инструментов предметы означают самих себя. Форма, являющаяся формой, является формой.

Быстрая зарисовка на бумаге со множеством сырых пиктограмм и косых линеек прокрутки является в той степени абстрактной, в какой большинство разработчиков пользовательских интерфейсов может ее понять. Когда пользователи жалуются на нечитаемость или указывают на недостаток мастерства дизайнеров, разработчики быстро перестают делать рисунки для программного обеспечения. Ведь быстрее и проще накидать в форму каких-нибудь реальных «штучек» с помощью Visual Objects, или Visual Basic, или Visual Age, или Visual Goober. Всего за несколько минут можно создать работоспособный интерфейс. Он выглядит как первоклассный интерфейс, потому что он и есть первоклассный. И кроме того, как и многие первоклассные компоненты реального программного обеспечения, он с помощью клавиатуры переместился из замысла прямо на экран, минуя этап тщательного обдумывания.

Ответственность за такое плачевное состояние дел лежит не только на этих инструментах и их создателях, но и на профессионалах, которые их применяют. Объектно-ориентированные методологии и инструменты не предлагают эффективных моделей для представления абстракций пользовательских интерфейсов.

Для представления архитектуры пользовательских интерфейсов в абстрактной форме нужны три составляющих. Во-первых, необходим способ представления различных контекстов, в которых пользователи будут взаимодействовать с системой, причем не имеет значения, какой будет их конечная реализация: окно или экран, форма или панель с закладками и диалоговыми элементами. Во-вторых, нужны способы для представления сущностей, содержащихся в различных контекстах взаимодействия без необходимости принимать решение от том, как будет выбираться элемент: при помощи командной кнопки, переключателя или флажка. И наконец, нужен способ для отображения взаимодействия всех этих контекстов. Необходим способ определять, где находится начало и как можно переходить от одного контекста к другому. Эта довольно широкая территория охватывается двумя простыми концептуальными инструментами: контент-моделью и картой контекстной навигации.

Концепция контекста

Работа контекстна. Работая на складе или в офисе, в гараже или на компьютере, люди выполняют свои задачи в четко определенных контекстах с заранее известными наборами инструментов и материалов. Для выполнения какой-то конкретной задачи люди идут туда, куда нужно, с тем инструментом, который требуется. Желая нарисовать картину, вы становитесь перед мольбертом с палитрой и кистью в руках. Для приготовления поздним вечером итальянского блюда вы идете на кухню, берете несколько томатов, базилик и чеснок, кастрюли для макарон и для соуса, ложку и терку для pecorino[1]. Решив отремонтировать сломавшийся стул, вы идете в нужное место и берете нужные материалы и инструменты. Даже если место остается одним и тем же, контекст меняется в соответствии с требованиями текущей задачи. Помощник зубного врача раскладывает на подставке разные наборы инструментов для обработки канала зуба или для выполнения обычной чистки.

Как можно представить такие разнообразные контексты применения, не вдаваясь в конкретные детали? Как можно без схем и макетов экранных изображений показать, что следует связать, а что следует разделить? На помощь приходит интерфейсная контент-модель. Люди придумывали различные варианты этого метода, но решающее влияние оказала техника фиксации требований с помощью клеящихся листков, прикрепляемых к листу большого перекидного блокнота (Holtzblatt и Beyer, 1998 [40]).

В контент-модели лист бумаги представляет один контекст взаимодействия, в котором выполняется одна значимая для пользователя задача. Все, что пользователю нужно получить от системы для выполнения своей задачи, представляется с помощью наклеиваемых листков. Для активных инструментов и элементов управления берут «горячие» цвета, например розовый, желтый или оранжевый. Для представления данных, информации или другого материала, с которым производятся манипуляции, берут «холодные» цвета, например зеленый или синий. На каждом листке описывают его назначение и функцию с точки зрения пользователей, но не конкретную «штучку», с помощью которой эта функция может быть в итоге реализована. Например, на лист, представляющий контекст взаимодействия «подготовка слайдовой графики», мы можем наклеить листок инструмента под названием «изменитель цвета» и листок материала под названием «графический фрагмент». Такие листки можно легко переместить с одного листа на другой или выбросить в корзину. Контент-модель становится гибкой средой для игры с содержимым и структурой пользовательского интерфейса. Она не требует рисовать картинки или

[1] Острый овечий сыр (ит.). – *Примеч. науч. ред.*

выбирать конкретные «штучки» для реализации. Внимание в большей степени сосредоточено на задаче, пользователях и на их потребностях для выполнения своей работы, а не на деталях проектирования реального пользовательского интерфейса. Даже тот факт, что контент-модели внешне не похожи на макеты экранных изображений, помогает как пользователям, так и разработчикам думать в объектно-ориентированных терминах.

Жалкий прототип

Некоторые разработчики называют такие листы с наклеенными цветными карточками «неточными прототипами», но думать о них как о прототипах неверно. Они не являются плохими прототипами или неточными интерфейсами. Это хорошие абстрактные модели пользовательских интерфейсов. Они помогают искать различные способы разбиения задач или групповых элементов или экспериментирования с различными архитектурами пользовательского интерфейса.

Для завершения картины, отображающей архитектуру пользовательского интерфейса, нам также нужен общий вид, показывающий все контексты взаимодействия и способы перехода пользователя от одного контекста к другому. Другими словами, нужна модель, напоминающая диаграмму состояний или схему взаимодействия между объектами. Карта контекстной навигации является моделью, представляющей каждый контекст взаимодействия в виде именованного символа, а с помощью соединительных стрелок на ней показаны возможные переходы между этими контекстами. Карта навигации – это еще один уровень абстракции, который извлечен из контент-модели. Он не отражает содержимое контекстов взаимодействия, скрывая инструменты и материалы, которые в них содержатся.

Для чего же нам нужна вся эта абстракция? Так же как объектно-ориентированные программисты превращают формы в классы и аннотации в код, так и мы хотим видеть, как контексты взаимодействия преобразуются в реальные компоненты: экраны с линейками прокрутки и инструментами редактирования, текстовые редакторы с меню и панели инструментов со множеством «интуитивных пиктограмм». Тогда зачем проходить через все эти промежуточные этапы абстрактных моделей и символических переходов, когда можно просто перетащить нужную «штучку» в форму, проектируемую в Delphi?

Преимуществом применения абстрактных моделей является их простота по сравнению с реальными предметами и процессами. Они облегчают понимание целого и помогают увидеть лес за деревьями. Опуская сложные или ненужные детали, они позволяют разработчикам сосредоточиться на главном, на сути реальной задачи.

Проект пользовательского интерфейса зачастую прост, но хороший пользовательский интерфейс может быть дьявольски сложен. Он может включать в себя сложные взаимодействия между десятками экранов с десятками диалоговых окон и бессчетным количеством деталей. Все это должно быть связано между собой, работать согласованно и иметь смысл для пользователя. Интерфейс должен быть простым для неопытного пользователя и полнофункциональным для эксперта. Он должен работать таким способом, каким пользователи привыкли думать.

Так же как и в фотографии, если вы взяли крупный план, трудно увидеть панораму ландшафта. Абстрактные модели позволяют отложить на время некоторые из множества решений и деталей, пока мы занимаемся планированием общей картины. При этом внимание сосредоточено на главном – работе и пользователях, которые стараются выполнить эту работу.

Кроме того, абстрактные модели способствуют поиску новых решений. Оставляя открытыми больше возможностей, они приглашают нас к творческому заполнению пустых мест. Если мы размещаем в окне две линейки прокрутки, мы сразу же оставляем себе только один способ перемещения по рисунку. Если вместо этого мы даем розовому листку название «навигатор рисунка», то мы начинаем рассматривать другие возможности – например, окно навигации, обеспечивающее вид «с высоты птичьего полета», или режим панорамирования и масштабирования, как в видеокамере. Когда наступает подходящее время, мы можем рассмотреть разные варианты и заполнить пустые места. Чем дольше мы удерживаемся от простых и стандартных решений, тем с большей вероятностью мы найдем удачный ход.

Как любил отмечать французский пуантилист Жорж Сера (Georges Seurat), именно пустой холст, а не законченный портрет наводит на изобретение нового. Наверное, пришло время для того, чтобы объектно-ориентированная технология применила силу абстракции к проблеме взаимодействия.

По материалу из журнала *Object Magazine*, декабрь 1996 г.

45

Новая среда

Конечные пользователи, сидящие перед 19-дюймовыми мониторами, могут даже и не знать о том, является приобретенное программное обеспечение объектно-ориентированным или нет. Можно даже утверждать, что если пользователи не прочитают об этом на коробке, в которой поставлялась программа, или не откопают что-нибудь в сопроводительной документации, то особенности технологической реализации вряд ли их заинтересуют. При условии, что система работает с приемлемой эффективностью и поддерживает большую часть пользовательских задач, можно говорить о соответствии системы основным потребностям пользователей. Если система относительно проста в изучении и пользователи могут легко запомнить, как в ней работать, значит, их хорошо обслуживают.

С другой стороны, если объектная технология способна обеспечивать рационализацию и упрощение процесса разработки и проектирования, если она действительно позволяет создавать системы, более полно отвечающие потребностям наших пользователей, тогда программное обеспечение, которое является просто «хорошим», – это не совсем то, что наши покупатели по праву ожидают получить. Объектно-ориентированное программное обеспечение должно быть узнаваемым для пользователей, но не по знакомой неуклюжести его интерфейсных классов или по упоминанию Smalltalk или C++ на коробке, а по более широким возможностям, предоставляемым пользователям, и более эффективному и совершенному способу их представления. При разработке пользовательского интерфейса нет поводов не применять новые методы повышения эффективности и расширенные возможности доработки и повторного использования компонентов, которые дает объектная технология.

В таком случае, как именно следует применять наши отточенные умения и мощные инструменты? Какую форму могут иметь новые, удобные интерфейсы? Как приступить к изменению способов взаимодействия между пользователями и нашими объектно-ориентированными системами?

Непротиворечивая противоречивость

Как нам говорят, непротиворечивость является критерием хорошего пользовательского интерфейса (см. главу 34). Пользователи предпочитают интерфейсы, которые выглядят и работают подобно уже известным им интерфейсам. Такие интерфейсы для них более комфортны. Каждое предполагаемое новшество, каждое новое устройство или средство наталкивается на высокий барьер в виде некоторого естественного сопротивления. Даже более совершенные интерфейсы могут категорически отвергаться пользователями, которые не хотят или не могут научиться новым способам взаимодействия со своими компьютерами. Пользователи Apple Macintosh хорошо известны тем, что всегда настаивают на непротиворечивости и отвергают программное обеспечение с «немакинтошевским» поведением или функциями. Но многие ли сейчас помнят, что среди первых тестеров ОС Macintosh нашлись такие, которые посчитали ее крайне неудобной, ставя в вину отсутствие стандартных команд и какого-либо из принятых приглашений вида «C:>».

У новаторов есть целая масса хороших концепций, на основе которых они могут разрабатывать пользовательские интерфейсы – одновременно и оригинальные, и вполне знакомые. Многие методы создания внутренней структуры объектно-ориентированных программ также находят применение при разработке пользовательского интерфейса. Обобщение и конкретизация, компоновка и перекомпоновка, расширение и перегрузка – все эти методы могут помочь в разработке новых эффективных интерфейсных средств и способов взаимодействия с ними. Зачастую самые удачные новшества в пользовательских интерфейсах – это не впечатляющие рывки в будущее, а вариации на заданную тему, когда знакомые элементы комбинируются в новых сочетаниях. Художник-новатор Билл Бакстон (Bill Buxton) назвал это «радикальной эволюцией».

Наихудшей формой противоречивости и непоследовательности является неестественное поведение знакомого объекта. Если на дверную ручку, которую хочется повернуть, на самом деле нужно нажимать, такая ручка собьет с толку любого, кто будет открывать дверь, пытаясь высвободить щеколду. Некий элемент ГПИ, который выглядит как выпадающий список, но при щелчке мышью вызывает диалоговое окно, будет запутывать пользователей и вызывать у них раздражение. Видя перед собой табло, которое кажется пассивным отображением данных, обычный пользова-

тель может никогда не догадаться, что по нему, словно по программной иконке, нужно дважды щелкнуть для обновления данных.

Инновации и непоследовательность могут проявляться в различных аспектах пользовательского интерфейса: во внешней форме объектов, в их поведении или в идиомах, посредством которых мы взаимодействуем с ними. Алан Купер (Alan Cooper), который прямо критикует многие современные методы разработки пользовательских интерфейсов, ввел термин *идиомы взаимодействия*, обозначающий идиоматические жесты и последовательности, с помощью которых пользователи взаимодействуют с графическими пользовательскими интерфейсами. Из общепринятых идиом для общения с программным обеспечением на основе ГПИ можно назвать следующие: один щелчок мышью для выделения или активизации, двойной щелчок для открытия или запуска, обведение для выделения группы объектов, перетаскивание (drag-and-drop) для перемещения объектов.

Самые удачные инновации привносят новые идиомы представления, поведения или взаимодействия, не требуя дополнительного обучения для их эффективного использования. При рассмотрении «интуитивного» интерфейса пользователи могут легко догадаться о назначении и способах применения его элементов, даже если прежде их никогда не видели. Правильность догадок пользователей зависит от многих факторов. Важна узнаваемость и ясность, а также выразительность интерфейса – его способность при помощи слов, фигур, символов и позиций донести до пользователя информацию разработчика о применении и поведении системы.

«Разведываемые» интерфейсы позволяют пользователям исследовать возможности системы и применять ее средства, не вызывая своими действиями серьезных последствий. Устойчивые внутренние объекты, разработанные для обеспечения обратимости процессов, могут способствовать улучшению юзабилити, поскольку возможность отменять любые действия или «откатываться назад» является первым помощником в исследованиях. Простая навигация по многоуровневым меню или диалоговым окнам является еще одним известным способом поддержки исследований.

В основе объектно-ориентированной парадигмы лежит взаимодействие посредством сообщений. Хорошее взаимодействие с пользователями также является ключом к созданию новых интерфейсов, способных обучить пользователей работе с ними. Психолог и гуру в юзабилити Дональд Норман (Donald Norman) ввел два понятия, которые могут помочь разработчикам придумывать новые компоненты и идиомы – новые, но понятные и удобные. *Ограничители* – это элементы пользовательского интерфейса, которые сообщают об ограничениях той или иной операции. Они помогают пользователям избежать бесполезных или бессмысленных действий.

Например, диалоговые окна и рабочие зоны имеют границы. Модальные диалоговые окна приостанавливают взаимодействие до тех пор, пока пользователь не выполнит какую-нибудь операцию в данном диалоге. Бегунки двигаются только в одном измерении. При попытке перетащить элемент в окно, которое не может его принять, курсор изменяет свою форму на кружок с диагональной линией, который принято использовать для сообщения «дороги нет». Судя по этим примерам, разумное применение ограничителей позволяет мягко управлять действиями пользователей для успешного выполнения работы.

Подсказки представляют другую сторону обучающих интерфейсов. Подсказки – это элементы пользовательских интерфейсов, которые стимулируют, поощряют и подкрепляют определенные формы взаимодействия. Прямоугольный элемент, тень от которого подсказывает, что он «выступает» над поверхностью, побуждает пользователя щелкнуть по нему мышью, то есть воспользоваться им как кнопкой. Если в результате этого действия элемент «утопает», данная подсказка подтверждается. Треугольники или стрелки, указывающие вверх, подразумевают, что компонент можно передвинуть вверх или увеличить его размер, если воспользоваться этим элементом управления. Соответственно, направление стрелок вниз подразумевает обратное действие.

Смешивание сред

Подобно тому как обычные внутренние возможности можно комбинировать посредством множественного наследования и смешанного программирования, так и визуальные компоненты и идиомы взаимодействия можно комбинировать для создания новых возможностей, которые сразу распознаются и применяются пользователями без дополнительной помощи или инструкций. Например, работу во многих приложениях, задействующих базы данных и перманентные объекты, можно упростить с помощью специальных таблиц. В этих таблицах можно непосредственно редактировать данные, при этом ячейки данных служат как для отображения текущих значений, так и для ввода новых. Если ячейки для редактирования похожи на поля для ввода, то пользователю становится понятным их потенциальное поведение. Новые пользователи, вероятнее всего, смогут правильно понять назначение ячеек и попытаются отредактировать такие поля в процессе изучения интерфейса.

В некоторых системах предпринят еще один шаг в комбинировании функций внутри таких сеток отображения – в отдельные ячейки некоторых колонок встраивается список для выбора. Если возможен только определенный набор значений (например, в поле, предназначенном для кода отдела компании), то прямой выбор из списка возможных значений

поможет уменьшить количество нажатий на клавиши и сократить количество ошибок. Если ячейки, работающие как выпадающие списки, внешне ничем не отличаются от других ячеек, то при переходе к ним поведение программы может показаться странным и неожиданным для пользователя. Он может просто не осознать наличие тех или иных возможностей. Если же такие ячейки будут похожи на выпадающие списки или комбинированные окна, их назначение будет ясно, а применение – очевидно. Небольшой прямоугольник в виде кнопки с правой стороны каждой такой ячейки – с изображением направленного вниз треугольника и с «подсказкой нажатия» – приглашает пользователя щелкнуть мышью по кнопке, чтобы увидеть выпадающий список.

В другом простом обобщении и рекомбинации применяются окошки счетчика. Это типичные элементы ГПИ, которые позволяют пользователю «прокручивать» числовые значения вверх или вниз, щелкая мышью по небольшим треугольникам, направленным вверх или вниз. Обычно эти треугольники расположены справа от поля отображения и редактирования. Для переустановки времени при помощи только таких элементов потребуется два окна счетчика – одно для часов, а другое для минут. В результате время показывают отдельные, неудобные и непонятные, визуальные элементы. В альтернативном варианте время отображается в обычном формате ЧЧ:ММ внутри обычного текстового окна, к которому с двух сторон добавлены окошки счетчика. Пользователи сразу понимают, что окошко счетчика, расположенное слева от текстового окна, регулирует значения часов, хотя такое расположение является нестандартным внутри нестандартного компонента.

Перегрузка без перегрузки

Перегрузка операторов, которая так дорога программистам и компьютерным лингвистам, при переносе на пользовательский интерфейс может быть либо полезной, либо вредной. Некоторые перегрузки могут быть вполне разумными и нравиться пользователям, другие будут отвлекать их. Если название, расположенное вверху отображаемой колонки, имеет вид кнопки управления, то пользователь может подумать, что по ней можно щелкать мышью. В популярных программных продуктах такое действие интерпретируется как запрос на сортировку данных по этой колонке. Налицо перегрузка элемента, который служит одновременно пассивным заголовком и активной командной кнопкой. В некоторых контекстах, таких как Windows Explorer, перегрузка применяется еще шире, а кнопка с заголовком превращается в переключатель, с помощью которого при повторном щелчке мышью порядок сортировки можно изменить на обратный. Такая перегрузка может быть еще более удобной,

если ее «увеличить», обеспечив активную обратную связь. Однако здесь следует проявлять осторожность, чтобы избежать выдачи пользователю противоречивых сообщений. Элементы управления сортировкой в колонках не должны выглядеть как выпадающий список или окошко счетчика, поскольку в этом случае они дают неверную «подсказку». Небольшой треугольник в заголовке однозначно сообщит пользователю как о направлении текущей сортировки, так и о колонке, по которой сортируются данные. Уже после беглого изучения этого механизма начинающий пользователь не будет испытывать трудностей в его понимании и применении.

Приведем еще один пример эффективной перегрузки. В интерфейсах многих приложений есть визуальные компоненты, к которым можно перетаскивать данные для запуска процесса их обработки. Известными примерами таких «зон сброса» являются пиктограммы принтеров и корзин для мусора, расположенные на экране. В некоторых случаях весьма разумно иметь как зону, в которую можно перетаскивать документы, так и специальную управляющую кнопку для применения операции к уже выбранным данным. Одни пользователи предпочитают идиому drag-and-drop, другим этот механизм кажется утомительным и трудным, поэтому они предпочитают выбирать данные, а потом щелкать мышью по элементу управления. Иногда в приложениях можно увидеть условность, согласно которой «зоне втаскивания» придается соответствующая «подсказка» в виде углубления или «колодца». В соответствии с принятой практикой «подсказка нажатия» передается при помощи создания выступающей, выпуклой внешней формы. Объединенный элемент управления состоит из выступа в виде кнопки, помещенного в центр неглубокого «колодца». Он говорит пользователю: «Или тащи сюда, или нажимай – как хочешь».

Неправильная «подсказка» может приводить к ошибкам, путанице или отказу пользователя от применения какого-либо элемента. Внизу большинства стандартных окон приложений находится строка состояния, которая отображает различные настройки в углублениях, похожих на неглубокие «колодцы». В некоторых программах эти «колодцы» могут работать как активные элементы управления, хотя при отсутствии соответствующих «подсказок» пользователю не ясно, какие именно «колодцы» обладают таким свойством. Если один щелчок мышью никак не влияет на настройку, то двойной щелчок в некоторых случаях может переключить элемент управления из одного состояния в другое или вызвать дополнительное диалоговое окно.

В объектной технологии существуют разумные правила и паттерны эффективной конкретизации, а также возможность деления объектных классов на подклассы. Точно так же существуют разумные способы для расширения компонентов и идиом интерфейсов, которые позволяют соз-

давать новые, эффективные возможности. Повторное использование внутренних программных компонентов и интерфейсных классов обычно способствует непротиворечивости, однако знакомая внешняя форма должна сопровождаться знакомым поведением – если требуется, чтобы результат не сбивал с толку пользователей и не досаждал им. Скорее всего, последовательные усовершенствования, вносящие небольшие изменения в существующие компоненты и механизмы, принесут больше пользы, чем радикальные отступления от стандартов и условностей. С помощью непротиворечивых расширений, обобщений и комбинирования можно создавать новые, оригинальные компоненты. Посредством разумного применения «подсказок», ограничителей и перегрузки эти компоненты можно сделать более понятными и удобными для пользователя.

По материалу из журнала *Object Magazine*, март 1997 г.

46

Полезные ситуации

Сегодня чуть ли не каждый системный аналитик или разработчик программного обеспечения превращается в сценариста, словно в каждом из них спрятан талант голливудского кинодраматурга. Как только скучающие теоретики придумывают очередную статью, а консультанты, применяющие устаревающие методы, выпускают очередную антологию, так сразу же появляются варианты проектов, основанных на сценариях. Каждая из главных объектно-ориентированных методологий, вплоть до самых современных и самых унифицированных, включает в себя сценарии, или пользовательские ситуации, или какую-нибудь другую последовательную модель задач с удачным или неудачным названием. О пользовательских ситуациях пишут статьи, выпускают книги, сочиняют заметки, проводят занятия и дискуссии, поэтому сегодня большинство профессионалов в компьютерной отрасли говорят о них спокойно и осмысленно, не запинаясь и не останавливаясь в недоумении по поводу того, действительно ли термин взят из английского языка.

Ряд ведущих специалистов-практиков также пришли к заключению, что пользовательские ситуации полезны при разработке пользовательских интерфейсов (см. главу 22). Если пользовательские ситуации можно применять для управления разработкой объектного взаимодействия и разбиения методов по классам, значит, их можно применять для проектирования взаимодействия между человеком и компьютером и для распределения элементов интерфейса между интерфейсными контекстами. Логика может показаться неубедительной, однако эта идея обладает привлекательностью, являясь своего рода технологическим вариантом волшебства, основанного на внушении. В конце концов, оба термина имеют общий ко-

рень и могут даже рифмоваться. «Идя в рейсы по пользовательскому интерфейсу, не забывайте пользовательские кейсы»[1] – это вполне может быть девизом часа.

Скучная история

Конечно, проектировщики взаимодействий и другие специалисты по пользовательским интерфейсам уже много лет применяют разные формы сценариев, включая раскадровки – визуальные эквиваленты киносценариев. И вот к чему это привело. Офисные программные пакеты теперь все больше и больше напоминают широкоэкранные голливудские киноэпопеи с распределением ролей между смешными закадычными друзьями, вызывающими раздражение. (Честно говоря, я каждый день пускал бы дорожный каток по анимированной скрепке.)

Айвар Джекобсон (Ivar Jacobson) объясняет, что сценарии и пользовательские ситуации – это не одно и то же, несмотря на все заявления «ну и что, мы тоже их применяем» от людей, всегда говорящих «мы тоже», и вопреки всем творческим переопределениям, сделанным батальонами практиков. И то и другое – это модели задач, в которых применяются описания последовательности событий, однако пользовательская ситуация является абстракцией, отдельным случаем (видом) использования. В сценарии документируется конкретный пример взаимодействия. В нем представлено буквальное, если не литературное описание (Constantine и Lockwood, 2000). Чтобы написать сценарий взаимодействия пользователя и пользовательского интерфейса, нужно представлять себе и пользователя, и интерфейс. Кроме того, необходима возможность ссылаться в описании на сам интерфейс и его компоненты. Это значит, что сценарии не могут помочь в разработке пользовательских интерфейсов, поскольку пользовательский интерфейс является одним из персонажей этой истории. Перед созданием сценария вам обязательно нужно придумать хотя бы частичную схему пользовательского интерфейса. Сценарии не бесполезны – они помогают понять задачи и доработать формы взаимодействия с уже созданным пользовательским интерфейсом. Однако обычно сценарии слишком конкретны, чтобы помочь в разработке структуры и содержимого совершенно нового пользовательского интерфейса. (Constantine и Lockwood, 1999 [30]).

Пользовательские ситуации стали «денежной единицей» в объектно-ориентированной технологии, потому что они полезны. Подтвердив свою ценность в разработке технических условий и создании объектно-ориентированного программного обеспечения, пользовательские ситуации вполне

[1] Going places with use cases for user interfaces. – *Примеч. науч. ред.*

могут быть полезны и в проектировании пользовательских интерфейсов. Пользовательские ситуации крепко связаны с внешней стороной системы, относясь к ней как к черному ящику. В то же время сценарии, подобно коротким рассказам, могут быть написаны почти с любой точки зрения. И в самом деле, некоторые программисты склонны рассматривать свою систему как центр вселенной и писать сценарии с внутренней перспективы.

Пользовательские ситуации не только дают внешнее представление, но и отступают от буквального языка сценариев, обращаясь к более абстрактному языку переменных и классов. Например, в сценарии может быть написано: «Программист Паула щелкает по пиктограмме на рабочем столе, чтобы запустить мастер технической поддержки. Ей предлагаются два варианта соединения: сетевое или модемное. Она выбирает модемное. Затем предлагается ввести номер пользователя – она набирает 4477-610. Далее, в предложенном списке проблем она щелкает по пункту «Проблемы с печатью» и т. д». В пользовательской ситуации, наоборот, дается более общее описание: «Пользователь щелкает по пиктограмме запуска программы технической поддержки, производит соединение с системой, вводит номер пользователя, выбирает пункт из предложенного списка проблем». Таким образом, мы делаем шаг назад, который приближает нас к пониманию реальной задачи, стоящей перед пользователем.

К сожалению, большинство пользовательских ситуаций все же содержат много скрытых, невыраженных допущений, связанных с пользовательским интерфейсом системы. В данном примере система технической поддержки обозначена пиктограммой, а у пользователя есть свой номер. Этот номер необходимо ввести для получения доступа к системе, после чего нужный раздел выбирается из списка. Такие допущения могут показаться не слишком существенными, но мы знаем по опыту, что именно невысказанные, а значит, и не подвергнутые сомнению допущения могут в конечном итоге нанести наибольший вред. Они привязывают нас к конкретным дизайнерским решениям, принятым без подробного описания задачи и без учета альтернативных вариантов.

Условный, конкретный вариант пользовательских ситуаций представляет собой блюдо, состоящее из неявных решений в сочетании с описанием текущей задачи. Зачастую эти ингредиенты трудно различить или отделить друг от друга. Действительно ли применение пользовательского номера является частью проблемы, или это одно из возможных решений неуточненной задачи по идентификации пользователя? Для создания хорошего пользовательского интерфейса нужна более абстрактная и очищенная модель, напоминающая пользовательские ситуации, но не засоренная допущениями относительно дизайна интерфейса, который еще не разработан.

Хорошие цели

По этим причинам Люси Локвуд (Lucy Lockwood) и я разработали метод сущностных пользовательских ситуаций как упрощенную, обобщенную и более абстрактную модель в сравнении с ее конкретными аналогами (см. главу 22). Абстракция (как об этом говорилось в главе 44) способствует новаторскому мышлению и созданию более надежного дизайна. Целью моделирования сущностных пользовательских ситуаций является описание пользовательских задач как таковых – без каких-либо технологических допущений и ограничений. Сущностная пользовательская ситуация представляет собой абстрактную сущность целей пользователя. Она выражается в терминах намерений, а не в терминах действий пользователя. Например, пользователи системы технической поддержки намереваются получить помощь – сообщить, кто они, и объяснить, с какой проблемой или проблемами они столкнулись.

Сущностные пользовательские ситуации не только отделяют пользовательскую проблему от дизайнерских решений, но и различают пользовательские намерения и системные ответы, обеспечивающие реализацию этих намерений. Вместо одного свободного описания взаимодействия – того описательного подхода, который применяется в сценариях и большинстве пользовательских ситуаций, – сущностные пользовательские ситуации принимают форму структурного описания, в котором диалог разбивается на две колонки: то, что пользователь хочет сделать, и то, что система должна представить в качестве ответа. Названия этих колонок: *модель пользовательских намерений* и *модель системных ответов*. Такая модель имитирует формат, предложенный Ребеккой Уирфс-Брок (Rebecca Wirfs-Brock) для конкретных пользовательских ситуаций. В задаче, связанной с разработкой системы технической поддержки, сущностную пользовательскую ситуацию можно назвать получение технической помощи. Она может принять следующую форму:

```
получение технической помощи

  Намерение пользователя       Ответ системы
  запросить помощь             опознать
  идентифицировать себя        предложить помощь
  конкретизировать проблему    дать ответ
```

В сущностном варианте пользовательской ситуации опускаются посторонние детали взаимодействия и не затрагивается визуальная или физи-

ческая форма интерфейса. Высокая степень абстракции и сосредоточенность на цели позволяют разработчику рассмотреть разные пути достижения одних и тех же результатов. Например, приведенная сущностная пользовательская ситуация может описывать как применение системы сетевой технической поддержки, так и программу обработки голоса при телефонном общении. Так как сущностные пользовательские ситуации обычно короче и проще конкретных, они часто помогают создать более компактные системы с более простыми и удобными интерфейсами. Поскольку пользовательские ситуации являются целевыми моделями задач, они помогают сосредоточиться на целях и реальных потребностях пользователей.

Закономерно, что сущностные пользовательские ситуации для данного приложения почти всегда взаимосвязаны между собой. Конечно, можно описать каждую сущностную пользовательскую ситуацию как законченный, самодостаточный и независимый документ, но тогда в сумме мы получим кучу лишних документов. Мы можем сэкономить бумагу, если организуем и упростим совокупность сущностных пользовательских ситуаций, выделив общие элементы и создав взаимные ссылки между ситуациями.

Подобно применению наследования для определения классов и подклассов внутри объектно-ориентированных программ, мы можем создать квалификационную иерархию сущностных пользовательских ситуаций. В этой иерархии абстрактные ситуации содержат типичные или совместные части описаний, а подситуации содержат конкретные детали. Точно так же с помощью объединения простых ситуаций мы можем построить комплексные пользовательские ситуации. Одним из самых мощных средств для организации и упрощения сущностных пользовательских ситуаций является новаторский метод, представленный Джекобсоном (Jacobson и др., 1992 [44]). Части описания, которые являются необязательными или особыми вариантами, можно выделить и определить как расширения. С помощью каждого расширения можно расширить любое количество других пользовательских ситуаций. Например, в системе технической поддержки пользователь, следуя пользовательской ситуации, может запрашивать помощь у оператора. Эта пользовательская ситуация, запрос помощи оператора, является расширением других пользовательских ситуаций, поскольку она представляет собой дополнительный или особый случай взаимодействия, основанный на альтернативной или второстепенной цели пользователя.

Основное назначение моделирования сущностных пользовательских ситуаций – вести разработчика к простой и надежной форме дизайна пользовательских интерфейсов. Однако при разработке продукта сущностные пользовательские ситуации могут служить и для других целей. Модели сущностных пользовательских ситуаций являются идеальной средой для

утверждения и уточнения системных требований. Короткие и понятные описания сущностных пользовательских ситуаций в сочетании с картой определения всех взаимосвязей между описаниями дают чрезвычайно компактную и легко интерпретируемую картину всех внешних функций, обеспечиваемых системой.

Так как описание сущностной пользовательской ситуации представляет собой диалог, записанный в виде двух колонок, пользователи и покупатели могут легко понять их без дополнительных объяснений или какой-либо специальной подготовки. Сущностные пользовательские ситуации становятся простой и естественной средой для взаимодействия между разработчиками и пользователями, позволяя совместно определить масштабы и требования к создаваемой системе. Далее, в процессе программирования системы, из сущностных пользовательских ситуаций можно извлечь внешние тестовые ситуации, так же как и приемосдаточные тестовые ситуации для приемных испытаний.

Рука помощи

Хотя документация и справочные системы зачастую бывают упрощенными, они являются важными компонентами для обеспечения юзабилити систем – настолько важными, что самая большая глава в книге «Software for Use» (Constantine и Lockwood, 1999 [30]) целиком посвящена разработке справочных систем. Хорошая документация и правильно организованная, отзывчивая справочная система могут облегчить применение сложной системы; и наоборот, неадекватная, плохо организованная справочная система способна полностью испортить систему, которая во всех других отношениях разработана грамотно.

Организация документации и онлайновой помощи на основе сущностных пользовательских ситуаций – это новый и эффективный способ улучшения юзабилити программного обеспечения. Сущностные пользовательские ситуации представляют собой все возможные потребности пользователя и цели, которых можно достичь при помощи данной программы. Каждая сущностная пользовательская ситуация, написанная должным образом, является отдельной задачей, которая закончена и четко определена с точки зрения внешнего пользователя. Эта задача описана на языке пользователей с учетом данной предметной области. Вот почему сущностные пользовательские ситуации идеально подходят для организации справочной системы, ориентированной на задачи («how to...?»). При этом каждая ситуация имеет свой раздел и содержит набор инструкций для активизации данной пользовательской ситуации. Даже взаимосвязи внутри карты пользовательских ситуаций – расширение, конкретизация

и объединение – могут быть уместно и естественно отображены в документации и справочной системе в виде связей и перекрестных ссылок.

Таким образом, сущностные пользовательские ситуации позволяют последовательно отслеживать выполнение требований – не только в период от начального анализа до пользовательского интерфейса, внутреннего проектирования и тестирования, но также и в ходе документирования и создания онлайновой помощи. Внимание к сущности (в виде сущностных пользовательских ситуаций) может помочь нам, разработчикам и дизайнерам, дать пользователям именно то, что им нужно: компактные системы с простыми пользовательскими интерфейсами, которые снабжены хорошей документацией и просты в применении.

По материалам журнала *Object Magazine*, июнь 1997 г.

47

Эффективные объекты

Проектирование основано на измерении. Мост должен быть достаточно длинным, чтобы соединить берега, и достаточно крепким, чтобы выдержать заданную статическую нагрузку. Он должен обладать структурной стабильностью, чтобы выдерживать предполагаемую максимальную скорость ветра и сохранять устойчивость при землетрясениях определенной силы. Измерение не менее важно и в проектировании программного обеспечения, однако метрика программного обеспечения до сих пор часто воспринимается как углубленный и эзотерический предмет, в основном представляющий интерес для теоретиков, которые стремятся получить исследовательские гранты, или для подрядчиков военных ведомств, которые стараются попасть в правительственные квоты.

Метрика программного обеспечения связана с преобразованием чего-либо в числа, поэтому она может быть получена на основе всего, что можно подсчитать, оценить или измерить. Самыми известными являются измерения размера и сложности программы. В диапазоне методов измерения самым простым и непосредственным является старый способ подсчета длины кода, который со временем расширился до меры KLOC, тысяч строк кода. На другом конце этого диапазона находятся методы оценки функциональных пунктов, а также пункты свойств и другие замысловатые техники этого рода. Более сложные методы оценки имеют свои достоинства и своих сторонников, но когда вся риторика произнесена и все исследования сделаны, становится ясно, что для многих целей даже простой подсчет классов и методов может быть не менее полезным, чем самая сложная и теоретически обоснованная измерительная схема.

По многим причинам руководители проектов по разработке программного обеспечения в первую очередь применяют метрику размера и сложности. То, что измеряется, является очевидным и может быть легко интерпретировано. Измерения размера и сложности позволяют отслеживать продвижение разработки и количественно оценивать продуктивность разработчиков. В сочетании с хорошей базой статистических данных такие измерения дают возможность намного точнее и надежнее оценить время разработки и связанные с ней расходы, чем это можно сделать при помощи более распространенных субъективных подходов, которые зачастую сводятся к выдумыванию *ex nihilo*[1] и последующему умножению на какой-нибудь взятый с потолка коэффициент.

Профпригодность

Для разработчиков количественные измерения качества проектирования потенциально являются более важными, чем простые измерения количества кода. Проектная метрика основана на исчисляемых и измеряемых аспектах проекта, которые описывают важные стороны законченного продукта – все, что интересует разработчиков с точки зрения предполагаемого метода реализации, работы и обслуживания данного программного обеспечения. Например, компонентное сцепление и межкомпонентное связывание, два известных измеримых параметра проектирования, описывают, насколько легко можно будет модернизировать или расширить программу с помощью частичной декомпозиции на взаимосвязанные элементы. Термины «сцепление» и «связывание», впервые появившиеся в раннюю эпоху структурного проектирования (Yourdon и Constantine, 1979 [70]), со временем перешли в объектно-ориентированное проектирование в виде мер сцепления классов и межклассового связывания. Формы этих мер были широко исследованы – как классическая, так и объектно-ориентированная (Henderson-Sellers, Constantine и Graham, 1996 [39]).

В ответе на основной вопрос «лучше ли этот проект, чем тот?» измерение проектных параметров дает ряд ощутимых преимуществ для разработчиков и дизайнеров. Эти измерения позволяют сравнивать разные подходы и определять наилучшее решение, не прибегая к постановлениям руководства или бросанию летающих тарелок с 30 шагов. Уже на ранних этапах разработки они позволяют узнать, идет ли она по правильному пути и на каком повороте могут возникнуть серьезные трудности. В процессе проектирования или последовательной доработки они позволяют нам оценить, изменяется ли проект в лучшую сторону или просто изменяется. Правильные измерения, выполненные в правильный момент, могут

[1] Из ничего (лат.). – *Примеч. науч. ред.*

сказать нам о том, насколько совершенным является наш проект в некотором абсолютном смысле. Является ли эта версия достаточно близкой к оптимальной или к оптимуму еще нужно проделать долгий путь? Принесут ли какую-нибудь существенную пользу дополнительные испытания или доработки?

В разработке программного обеспечения измерения играют и стратегическую роль – с точки зрения бизнеса. Цифры обладают таким влиянием, которое может отсутствовать в простых словах – особенно сейчас, когда ценность слов была уменьшена многолетними безосновательными утверждениями и пустыми обещаниями. В условиях жесткой конкуренции, сокращения расходов и отсева разработчики программного обеспечения и прикладных программ все чаще вынуждены оправдывать свое существование, причем с приведением весомых аргументов. Измерения позволяют сравнить производительность с индустриальными стандартами и наилучшими достижениями. Измерения позволяют документировать постепенные улучшения производительности и качества работы. Измерения позволяют продемонстрировать конкурентное преимущество внутреннего знания или внешней независимости. Руководителям бизнеса мало что говорит более красноречиво, чем цифры. Легко заявлять о простоте использования и повышенном удобстве программного обеспечения, однако количественные сравнения могут сделать эти утверждения более убедительными. Сокращение избыточности данных на 37% и повышение производительности транзакций на 28% может быть наиболее убедительным аргументом в поддержку нового продукта.

Юзабилити – один из самых главных критериев качества программного обеспечения, однако метрика юзабилити программ была открыта совсем недавно и пока мало исследована. Большинство измерений юзабилити проводятся уже после разработки и оценивают простоту и эффективность применения готовых программных систем. Долгое время юзабилити-проектирование сводилось к проведению лабораторных и полевых тестирований, для которых требуется наличие рабочих систем или хороших имитаций. Хотя юзабилити-тестирование, проводимое в лаборатории или на рабочем месте, может дать ценную информацию для разработчиков, эта информация зачастую поступает слишком поздно и за слишком высокую цену. В современном мире ускоренной разработки и ограниченных сроков разработчикам нужны более быстрые и простые способы оценки качества создаваемых ими интерфейсов – пока они еще на бумаге, и их можно быстро и легко изменить.

Игра в числа

Пользовательские ситуации (см. главы 22 и 46), которые служат для многих целей в объектно-ориентированном проектировании, могут также применяться как основа для проведения эффективных и практических измерений параметров пользовательских интерфейсов. Три таких параметра – сущностная эффективность, целостность задач и согласованность задач – в течение нескольких лет разрабатывались как часть метрического комплекса, предназначенного помочь разработчикам в оценке и улучшении проектов. Эти параметры были созданы на надежных и последовательных принципах юзабилити программного обеспечения: простота, эффективное функционирование, видимость возможностей и разумная организация.

Пользовательские ситуации определяют, какую внешнюю функциональность должна обеспечивать система. Сущностная пользовательская ситуация воплощает каркас, идеализированную сущность некоторой четко определенной задачи, стоящей перед пользователем (см. главу 46). Таким образом, описательная часть сущностной пользовательской ситуации представляет взаимодействие между пользователем и системой в самой простой, абстрактной и обобщенной форме, свободной от ссылок на конкретные компоненты пользовательского интерфейса или технологию его разработки. Она описывает идеал с точки зрения минимального взаимодействия, необходимого для выполнения данной задачи. Конечно, в действительном пользовательском интерфейсе для решения некоторой задачи может потребоваться большее количество отдельных шагов, чем при идеальном взаимодействии. Таким образом, описание сущностной пользовательской ситуации представляет собой ориентир, с помощью которого можно проверять качество реального проекта.

На практике в качестве меры можно выбрать количество шагов-действий пользователя. Можно просто сравнивать количество шагов, за которое пользователь «проходит» пользовательскую ситуацию с помощью данного проекта пользовательского интерфейса, и количество шагов в сущностной пользовательской ситуации. Перевод соотношения этих чисел в проценты дает показатель сущностной эффективности данного проекта. Чем выше сущностная эффективность, тем проще пользовательский интерфейс и тем больше пользователь может рассчитывать на быстрое и эффективное решение задачи с его помощью. Этот параметр дает нам в некотором смысле абсолютную меру качества. Например, если наш первый проект дает 98% сущностной эффективности, это означает, что дальнейшая доработка интерфейса не приведет к его значительному улучшению.

Пользовательские интерфейсы должны упорядочивать различные визуальные объекты на экранах и в диалоговых окнах. В хороших интерфей-

сах организация визуальных объектов соответствует задачам, выполняемым пользователями. Как говорит специалист по пользовательским интерфейсам Алан Купер (Alan Cooper), экраны, окна и диалоговые элементы подобны разным комнатам (Cooper, 1995 [31]). Большинство людей предпочитает выполнить всю задачу в одной комнате, не переходя в другие в поисках необходимых инструментов и материалов. Они не делят работу на части, выполняемые в разных комнатах. Каждый раз, когда программное обеспечение вынуждает пользователей менять контекст взаимодействия, переключать экраны или переходить к другому диалоговому окну, ход их мыслей и работа прерываются. С каждым таким прерыванием пользователи не только работают все медленнее, но и делают больше ошибок. Хороший пользовательский интерфейс позволяет «пройти» пользовательскую ситуацию с меньшим количеством изменений контекста взаимодействия – с точки зрения общей сложности данной пользовательской ситуации. Поскольку в таких интерфейсах необходимые данные и функции объединены, эти интерфейсы не только проще применять, но и (в первую очередь) легче изучить.

Простая мера организационной эффективности может быть основана на количестве переключений, которые пользователь должен выполнять при обычном «прохождении» пользовательской ситуации. Показатель целостности задач отражает процент необходимых (обязательных) шагов, которые можно выполнить без переключения контекстов. Высокое значение показателя целостности задач означает, что все данные и функции, необходимые для пользовательской ситуации, собраны в одном месте, то есть в одном контексте взаимодействия. Низкое значение этого параметра свидетельствует о том, что пользователям придется часто и неоправданно переключаться между окнами или экранами, тем самым снижая общую эффективность и увеличивая количество ошибок. В данном параметре учитывается то, что контекстные изменения более допустимы для необязательных или особых подзадач, а также то, что продолжительные или сложные пользовательские ситуации можно распределить по нескольким экранам или диалоговым окнам.

Как показатель сущностной эффективности, так и показатель целостности задач относятся только к одной пользовательской ситуации. В то же время очевидно, что их можно применять и как параметры, сообщающие о средневзвешенных значениях совокупности пользовательских ситуаций. Наоборот, параметр согласованности задач является непосредственной мерой соответствия между целым набором пользовательских ситуаций и общей организацией законченного пользовательского интерфейса. С точки зрения эффективности использования хорошей архитектурой пользовательского интерфейса является та, которая позволяет более просто «проходить» обычные пользовательские ситуации. Редкие или

особые пользовательские ситуации допустимо делать более сложными, при условии, однако, что это не оказывает значительного влияния на общую эффективность применения системы. Другими словами, если отсортировать все пользовательские ситуации в порядке ожидаемой частоты их применения, то полученный список будет похож на список ситуаций, отсортированных по количеству шагов. Таким образом, согласованность (корреляция) между ожидаемой частотой применения и количеством шагов является еще одной мерой качества пользовательского интерфейса с точки зрения эффективности его организации. Этот параметр принимает наивысшие значения, когда наиболее часто выполняемые задачи имеют наименьшую продолжительность, а задачи с наибольшей продолжительностью выполняются наиболее редко. Юзабилити-параметр согласованности задач выражается в процентном соотношении ранжированной частоты и ранжированной длины.

Что важно

Одной из привлекательных сторон математики является то, что все, кто правильно извлекает квадратный корень из 9, получают один и тот же результат. К сожалению, не все измерения программного обеспечения происходят таким же образом. Разные люди, выполняющие функциональную оценку, могут получить совершенно разные результаты. Для получения повторяющихся результатов необходимо тщательно разработать правила подсчета, а аналитики должны быть хорошо подготовлены и, желательно, сертифицированы.

Правила подсчета также необходимы для оценки таких параметров юзабилити, как сущностная эффективность, целостность задач и согласованность задач, но оценивать их относительно несложно. Для получения одинаковых результатов необходимо ответить на следующие вопросы: что можно считать за один шаг пользователя и что является ненужным изменением контекста. Эти организационные вопросы сравнительно просты – подробные ответы займут лишь несколько страниц.

Вычисления, основанные на пользовательских ситуациях, помогают разработчику найти изъяны в пользовательском интерфейсе и варианты улучшения его проекта. Это не пассивные или малопонятные цифры, а пример «инструктивных показателей», то есть измерений, которые ведут к построению более совершенных проектов. Выполнение таких измерений позволяет увидеть неэффективные пользовательские ситуации, ненужные изменения контекстов и неоптимальные части проекта – в сравнении с теми, которые уже более чем «достаточно хороши».

Три описанных показателя оценивают только отдельные аспекты качества пользовательского интерфейса – главным образом те, которые связаны

с эффективностью применения. Для проведения более глубокой оценки эти показатели необходимо объединить с другими параметрами и образовать большой метрический комплект. Хотя измерения не являются ответом на все вопросы разработки, «численная» разработка пользовательского интерфейса, основанная на хорошо продуманных измерениях параметров, позволяет на порядки поднять уровень юзабилити объектно-ориентированного программного обеспечения.

По материалам журнала *Object Magazine*, сентябрь 1997 г.

48

Связанные объекты

Что делает тот или иной предмет легким для понимания? Что делает тот или иной предмет простым в использовании? Что превращает совокупность объектов – не отдельных, а представленных в определенном контексте – в набор рабочих инструментов? Возьмем объекты, взаимосвязанные внутри программного обеспечения, или визуальные объекты, отображаемые в графическом пользовательском интерфейсе. Что делает их понятными? Что делает их удобными?

Эти главные вопросы знакомы всякому, кто когда-нибудь пробовал готовить на чужой кухне. Чтобы разобраться, где что находится, нужно несколько минут, и все равно придется спрашивать, где находится нож для чистки овощей или дуршлаг. Тем не менее постепенно, по мере того как вы обнаруживаете разные наборы предметов, вы начинаете замечать определенную логику в их размещении, и вскоре все становится на свои места.

Если только хозяин кухни не является человеком неорганизованным или если вы сами не привели эту кухню в полный беспорядок, можно предположить, что в одном из ящиков найдутся ножи. В банке, стоящей в буфете, может храниться набор деревянных ложек. Вероятно, кастрюли и сковородки составлены в шкафу, а их крышки находятся на специальной подставке. Или же кастрюли вместе с крышками стоят в кладовке.

Для приготовления пищи важно наличие посуды и нужных приспособлений, однако размещение кухонной утвари не менее важно для ощущений при готовке – будет ли это радость или полное разочарование.

Представьте, что все предметы на кухне переставлены так, что мерные емкости оказываются вместе с блюдцами, ножи для чистки овощей – со

спичками, а крышки от кастрюль лежат на тарелках. Ставим микроволновку в кладовку, а холодильник – в комнату. В раковину составляем консервы. Тарелки убираем в холодильник, а все ножи, ложки и вилки раскладываем на шкафах. И тут вы понимаете, что в этой, некогда хорошо обставленной кухне, теперь почти невозможно готовить.

Некоторые из современных пользовательских интерфейсов имеют приблизительно такую же степень хаоса. Форматирование абзаца и строки располагается в меню Формат, форматирование колонтитулов – в меню Вид, а форматирование страницы – в меню Файл. Человеческий мозг обладает такой способностью к приспособлению, что со временем может изучить почти любую схему расположения, какой бы хаотичной она ни была. Однако разумные схемы делают процесс изучения быстрее, а приготовление пищи – и работу с текстом – намного проще.

Конечно, кухни, как и пользовательские интерфейсы программного обеспечения, можно организовать несколькими способами и при этом сохранить в них разумный порядок. Большинство схем размещения утвари в кухне совмещают в себе больше одного вида логики. Некоторые предметы распределяются по категориям, другие – по способу применения, а третьи – по иным соображениям и ограничениям, таким как наличие ребенка. Тем не менее, хотя личные предпочтения могут отличаться большим разнообразием, одни способы разделения по категориям могут быть более разумны, чем другие. Некоторые формы организации могут быть концептуально красивы, но совершенно не подходить для практического применения. Для большинства людей было бы трудно готовить на кухне, организованной только по принципу формы – квадратные предметы в выдвижных ящиках, плоские и круглые в нижних шкафах, круглые и глубокие в верхних шкафах и т. д., – несмотря на простоту и очевидность такого порядка.

С другой стороны, некоторые виды категоризации (например, по материалам, из которых изготовлены предметы) являются более разумными и практичными, чем может показаться на первый взгляд. Собирать предметы из стекла, или серебра, или фарфора в одном месте разумно и удобно – отчасти потому, что такие наборы зачастую объединяют предметы не только по схожести материала, но и по схожести функционального назначения. Другой эффективный способ группирования – собирать предметы, которые обычно применяются вместе. Например, в одном шкафу можно хранить инструменты для выпечки: миксерные чашки, миксеры, измерительные стаканы и т. п. Также могут быть и промежуточные варианты: вилки обычно хранятся рядом с другими приборами, но каждый столовый набор, состоящий из ножа, вилки и ложки, обычно не хранится в виде отдельных комплектов, разве что в кухнях на самолетах.

Таким образом, качество организации кухни или компоновки пользовательского интерфейса зависит от того, как объекты связаны друг с другом и с выполняемой задачей. Целостность задач (см. главу 47) является мерой качества интерфейсов. Она оценивает степень, с какой данный дизайн интерфейса объединяет в одном пространстве элементы, которые применяются вместе в обычной задаче – в одной пользовательской ситуации. В этой главе речь пойдет о визуальном сцеплении. Это мера, с помощью которой можно оценить качество организации интерфейса в показателях объединения связанных между собой объектов и разделения несвязанных.

Крепим вместе

В объектно-ориентированной технологии давно признается важность хорошей организации программного обеспечения. Нас учат, что классы следует хорошо подбирать и определять. Они должны взаимодействовать между собой оптимальными и рациональными способами. Крепкая объектная архитектура сможет сохранить свою устойчивость под неизбежным и безжалостным натиском изменяющихся условий, добавлений и многократных исправлений. Она принесет пользу не только самим разработчикам, но и множеству других профессионалов, которые будут заниматься расшифровкой и пересмотром этой архитектуры.

Повседневный язык, который мы применяем для обсуждения этой темы – будь то кухни или программное обеспечение, – обнаруживает в своих этимологических корнях основы хорошей организации. Например, о хорошем дизайне, или об убедительном доказательстве, или об удобном интерфейсе мы говорим как о вещах связных и ясных. Связанные группы включают в себя предметы, скрепленные вместе. Если они крепятся друг к другу, то их легче понять, «схватить» – либо умом, либо руками.

Разумная объектно-ориентированная архитектура начинается с разумных объектных классов, построенных на устойчивой концептуальной основе, – с таких классов, которые объединяют в себе те методы и атрибуты, которые составляют связанное целое. Этот принцип связанности воплощается в одном из основных критериев качества программного обеспечения – сцеплении. Показатель сцепления известен давно. Впервые он был представлен как одно из ключевых понятий в структурном проектировании (Yourdon и Constantine, 1979 [70]), а затем стал одним из основных элементов современного проектирования программного обеспечения. Этот критерий применялся в различных видах, в том числе и в хорошо известном и широко обсуждаемом методе LCOM (Lack of Cohesion of Methods, метрика отсутствия сцепления в методах) (Chidamber и Kemerer, 1994 [7]; Henderson-Sellers, Constantine и Graham, 1996 [39]). Позднее

критерий сцепления был доработан и расширен для того, чтобы лучше оценивать качество объектно-ориентированных проектов.

Что же означает сцепление и почему оно так важно для ОО-проектирования? Сцепление – это мера смысловой и понятийной взаимосвязанности между частями какого-либо программного целого. Она основана на том принципе, что для более легкого понимания общей картины взаимосвязанные предметы необходимо размещать в одном месте, а менее зависимые предметы – отделять друг от друга. Таким образом, части соединяются в блоки, которые легче воспринять и понять в виде гештальтов – целых и интегрированных единиц. Столовые приборы находятся в одном шкафу, тарелки – в другом, и все это не смешивается с чайными чашками. Сцепленные объектные классы и системы, построенные на их основе, более просты для понимания как при создании программного обеспечения, так и в процессе применения, повторного использования или модификации.

Подобное назначение и у сцепления объектов, которые представлены в пользовательских интерфейсах и отвечают на действия пользователей. Люди быстрее поймут пользовательский интерфейс с хорошей организацией, в котором связанные между собой предметы собраны вместе, а несвязанные – разделены. Пользователь легче воспримет интерфейс и сделает меньше ошибок при взаимодействии с ним.

Сам по себе критерий сцепления – по крайней мере в традиционном определении, принятом в проектировании программного обеспечения, – недостаточен для оценки организации визуальных объектов в пользовательском интерфейсе. Сцепление и удобопонятность нужно измерять исходя из расположения объектов в пользовательском интерфейсе – их визуальной организации. Визуальное сцепление является именно таким критерием – критерием, позволяющим оценить качество организации визуальных объектов в пользовательских интерфейсах.

Согласующиеся идеи

Визуальное сцепление – это мера соответствия между визуальной организацией объектов, существующих в пользовательском интерфейсе, и концептуальной организацией идей, представленных этими объектами. Более согласованный и удобопонятный пользовательский интерфейс возникает в результате того, что предметы, воспринимаемые взаимосвязанными, располагаются вместе, а предметы, которые воспринимаются независимо, располагаются раздельно.

Для практического измерения визуального сцепления следует знать две вещи. Во-первых, в какие смысловые группы можно собрать понятия, представленные с помощью визуальных объектов пользовательского ин-

терфейса. Во-вторых, какие визуальные группы возможны в самом интерфейсе.

К счастью, современные методы разработки графических пользовательских интерфейсов позволяют довольно легко определить визуальные группы. Разработчики пользовательских интерфейсов помещают линии и полосы между разделами диалоговых окон. Они рисуют границы вокруг кнопок выбора вариантов или предусматривают рамку вокруг панели инструментов. Они могут изменить цвет фона или создать эффект «выступания» или «вдавливания». В конце концов, они могут просто оставить больше пустого пространства между двумя группами командных кнопок. В любом случае различные визуальные группы обычно можно легко увидеть при первом взгляде на пользовательский интерфейс.

Другое дело смысловые группы, которые невидимы и неощутимы. К счастью, здесь может помочь хорошая модель предметной области. Предметные классы, их методы и атрибуты определяют один из аспектов смысловой организации, связанной с данным приложением. С помощью простого упорядочивания основных понятий, воплощенных в приложении, мы можем определить группы взаимосвязанных понятий. В основном (но не всецело) такие группы базируются на модели предметных классов.

После этого легко вычислить значение параметра визуального сцепления. Необходимо только рассмотреть каждую пару визуальных компонентов в каждой визуальной группе и подсчитать количество пар, в которых оба визуальных компонента связаны с одной и той же смысловой группой. Визуальное сцепление – это просто количество пар, в которых оба объекта связаны с одной смысловой группой. Обычно это количество выражается в процентах от всего количества возможных пар. Вычисление можно продолжить рекурсивно, чтобы учесть группы групп и оценить визуальное сцепление пользовательского интерфейса в желаемом масштабе (Constantine, 1996 [27]).

Сцепленное приложение

Теоретически критерий визуального сцепления может показаться превосходным, но вряд ли можно сказать, что на практике он является самоочевидным. К счастью, тема визуального сцепления была хорошо изучена. В одном из исследований (Constantine, 1996 [27]) были старательно сконструированы три различные версии квазистандартного диалогового окна печати. Все три версии имели реалистичный (с виду) дизайн и содержали одинаковое количество визуальных компонентов и визуальных групп. Все они были скомпонованы в соответствии с одинаковыми эстетическими принципами. Различалась только степень визуального сцепления. Один из проектов, так называемая «структурированная» версия,

был высокоорганизованным, а его показатель визуального сцепления составлял 62%. Другой проект, похожий на обычное диалоговое окно печати Windows, имел промежуточный показатель визуального сцепления, равный 42%. Наибольшую сложность представлял внешне правдоподобный, но концептуально хаотичный вариант дизайна. Показатель визуального сцепления этой «беспорядочной» версии составлял 29% несмотря на ее разумный внешний вид.

Для оценки проектов привлекались профессиональные разработчики и дизайнеры интерфейсов из Соединенных Штатов и Австралии. Каждый вариант дизайна рассматривался по различным сценариям задач. Критериями служили легкость понимания, изучения и применения. Если не говорить об одном или двух ляпах, которые случаются в любых реальных проектах, результаты во многом соответствовали ожидаемым. Варианты дизайна с большим значением показателя визуального сцепления были признаны более простыми в понимании, изучении и практическом применении. Такими они и были на самом деле.

Дело всего лишь в том, что хорошая визуальная организация имеет значение.

По материалам журнала *Object Magazine*, март 1997 г.

VIII

Это превосходное новое программное обеспечение

Введение

Вы не можете не общаться. Иногда это называется Первым законом человеческого взаимодействия (Watzlawick и др., 1967 [65]). Всякое действие или бездействие является формой общения, неким утверждением, а всякое общение, в известном смысле и в некоторых сферах, является политическим. Всякое общение, будь то пустая болтовня или многозначительное молчание, оказывает свое воздействие. С языком и общением связаны вопросы синтаксиса, семантики и прагматики. Именно прагматика человеческого общения делает его неизбежно политическим.

Программное обеспечение является формой общения. Как говорили Эд Йордон (Ed Yourdon) и я еще в бурные 70-е годы, хорошее программное обеспечение пишется для того, чтобы его прочитали. Это сообщение от одного программиста другому – или самому себе в будущем. Это также сообщение от программиста пользователю, которое пропущено через медленное и в какой-то мере неподходящее устройство-интерпретатор, известное под названием «пользовательский интерфейс». Такое сообщение говорит пользователю о том, как разработчик видит и воспринимает окружающий мир. Система и ее пользовательский интерфейс являются средой, посредством которой программист рассказывает конечному пользователю о своей ментальной модели. Достигнет ли сообщение своего адресата и в какой форме оно будет передано, зависит от многих обстоятельств. Важнейшими среди них являются намерения и способности разработчика.

Таким образом, программное обеспечение в качестве некого сообщения является политическим утверждением и играет политическую роль. Как программа изображает людей, что она разрешает и что запрещает, на что обращает внимание и от чего отвлекает – все это является политическими сообщениями. Сегодня из-за своей вездесущности программное обеспечение вполне может стать одной из самых влиятельных политических сил. Например, видеоигры, в которых восхваляется насилие и кровопролитие, нельзя считать нейтральной технологией. Это яркие сообщения о состоянии мира. Независимо от того, нравятся они вам или вы их ненавидите, создаете вы их или же хотите запретить, видеоигры, в которых

мужчины являются героями, а женщины – жертвами, играют свою роль в политике полов.

Разработчикам программного обеспечения платят за создание более совершенных систем, а не за создание более совершенных миров, хотя могут быть и другие мнения. Наука и технология кажутся спокойным прибежищем для профессионалов, которые просто хотят делать свою работу или получать удовольствие от игр с технологиями. Но мы не только профессионалы. Мы – люди, живущие на одной небольшой планете. Здесь нет балконов для равнодушной публики, только одна огромная сцена – мир.

Для чего же все-таки создаются программы? Кому они служат? Я постараюсь не читать вам проповедей, но все же есть вещи, о которых нам следует думать при планировании и программировании нашего превосходного нового программного обеспечения. Что говорит продукт публике, своим пользователям? Что он сообщает тем, кто вступает с ним в диалог? Расширяет ли он возможности людей или ограничивает их? Облагораживает ли их или унижает их достоинство? Просвещает или сбивает с толку? Возбуждает ли он интерес или отражает равнодушие? Имеет ли продукт какое-то значение?

Так много вопросов. Ваши ответы определят вашу конечную оценку.

49

Высокомерное программирование

Если бы Данте жил в цифровой век, он обязательно предусмотрел бы еще один круг ада для преступников, пишущих подлый код. Я не говорю о создателях вирусов и взломщиках систем – для них ад может быть слишком хорошим местом. Нет, я имею в виду тех, кто создает коммерческие программы с таким эгоистичным поведением, что они игнорируют операционную систему, другие приложения, а зачастую и самих пользователей и покупателей. Приложения с подобным поведением пишутся так, словно это единственное резидентное программное обеспечение на автономной системе. Такие приложения работают с неким чувством начальственной осведомленности и беспечного безразличия к другим, будь то люди или программы.

Такой стиль я стал воспринимать как высокомерное программирование. Высокомерное программирование – это разновидность практики программирования. Она совершенно отлична от многих разновидностей глупой или плохой работы, с которой мы так часто встречаемся. Высокомерное программирование – это эгоцентричное программирование, программирование с завышенным чувством собственной значимости.

Я думаю, что есть по крайней мере три подвида высокомерного программирования: эгоцентрическое, ленивое и претенциозное. Классической разновидностью является эгоцентрическое программирование. Для этой формы высокомерного программирования характерна некая отстраненность от мира, которая, вероятно, отражает социальную замкнутость таких разработчиков. Программисты могут настолько углубиться в задачу, настолько укрыться в своем уголке программной вселенной, что способны полностью забыть о существовании других программ и программистов.

Ленивая форма высокомерного программирования отражает стремление избежать тяжелой работы. Конечно, проще быть высокомерным и ленивым, чем заниматься сложными проблемами совместимости кода с сетями, или анализировать ошибки при вводе, или уважительно относиться к пользователю. Легче считать, что ваша программа работает одна на отдельной машине.

И наконец, есть высокомерие претенциозности. Это чрезвычайно сложный код, написанный исходя из того, что авторы все знают лучше, чем пользователи. Такие программы отбирают у пользователей контролирующие функции и лишают их возможности выбора. На самом деле высокомерно сложный код написать труднее, чем простой эгоцентрический, и еще сложнее, чем высокомерно ленивый код, но результат получается тот же самый. Во всех трех разновидностях высокомерного программирования продуктами являются приложения, которые говорят слишком громко, распространяют себя по всему диску, занимают все экранное пространство и отнимают все ресурсы операционной системы.

Невоспитанные драйверы

Возьмем одного из лидеров производства принтеров. Устройства, выпускаемые этой компанией, являются одними из самых надежных. То же самое можно сказать о встроенном программном обеспечении. Увы, их драйверы совсем другой породы. Эти драйверы поставляются с предупреждением о том, что их нельзя устанавливать в Windows таким же образом, как любые другие принтеры – словно они хотят сказать: «Мы особенные». Да уж, конечно, особенные! Один из релизов программы инсталляции, разработанной этой компанией, заменил менеджер печати Windows на свою версию, даже не сказав при этом хотя бы формального «с вашего позволения».

Более поздние релизы все так же продолжали настаивать на своей собственной процедуре установки, которая в некоторых случаях даже не носила стандартного имени Setup или Install. После установки программа показывает автопортрет принтера размером с четверть экрана при каждом выполнении задания на печать, хотя вам уже известно, что именно происходит, или вы просто не хотите знать об этом – например, когда другой компьютер из одноранговой сети инициирует печать на вашем общем принтере. Эта цветистая бесполезная картинка огромных размеров мешает ходу ваших мыслей и заслоняет экран до тех пор, пока вы не убираете ее принудительно. После чего вы пытаетесь разобраться, где вы были до этого. Ну, так вот, о чем же я говорил до того, как меня так грубо прервали? А, ну да, о высокомерном программировании.

Окно сообщения подобного рода вызывает раздражение, как будто вся система предназначена для драйвера принтера. Такой драйвер ведет себя так, словно это единственная интересная программа в системе, как будто вы больше всего хотите видеть именно эту громадную пиктограмму, напоминающую вам о том, какой принтер вы купили. Собаку можно воспитать, друга можно попросить говорить потише, но это программное обеспечение совершенно невозможно научить хорошим манерам. С ним нельзя справиться, даже если принтер стоит на столе, и вы можете видеть мигающий индикатор и слышать шум лазеров.

Если принтер был подключен через распределитель и в нем заканчивалась бумага, драйвер выдавал сообщение «принтер не отвечает», которое нельзя было убрать. Программисты никогда не думали, что вам может понадобиться что-то еще, кроме их принтера – единственного центра печатной вселенной. Этот принтерный драйвер был настолько эгоистичным, что крутился в жестком цикле, ожидая ответа принтера, в то время как Windows работала со скоростью черепахи.

Такая программа говорит: «Мой принтер, мой драйвер, мои проблемы с отсутствием бумаги настолько важны и безотлагательны, что я вынуждена блокировать систему до преодоления кризиса». Если не удавалось определить статус принтера, этот кусочек эгоистичного программного обеспечения так и сидел в памяти. Либо все застывало окончательно, либо вы добирались до менеджера печати, либо поддавались порыву и отдавали системе салют с помощью трех клавиш[1]. И так происходило всякий раз, когда заканчивалась бумага – всякий раз!

Задом наперед

Принтерные драйверы – не единственные преступники. Из-за огромного количества гигабайт, которые занимали программы и данные на каждом лэптопе в моем офисе, было решено перейти на использование портативных ленточных накопителей большой емкости. Устройства работали превосходно, хотя и немного шумели, но вот драйверы были спрограммированы высокомерно. Резервное копирование начиналось с запроса накопителя по параллельному порту SCSI-адаптера, однако программа драйвера подразумевала, что накопитель уже подсоединен, включен, готов к работе и снабжен картриджем. Если это было не так, программное обеспечение начинало глючить, нарушая работу всей операционной системы.

Или возьмем одну из версий программы, предназначенной для работы со внешним многофункциональным факсом по параллельному порту. Если

[1] Ctrl-Alt-Del на PC-совместимых компьютерах. – *Примеч. ред.*

вы запускали Windows до присоединения и включения факса, драйверы начинали жаловаться и выдавать сообщения об ошибке. Если вы хотели переключить печать на принтер, то перед этим вам нужно было сначала удалить драйвер факса, а затем выйти из приложений и перезагрузить систему, иначе драйвер начинал забрасывать вас бесконечными сообщениями об ошибке.

Среди моих любимых кандидатов на попадание в тот особый круг ада являются компании, которые слишком скупы, чтобы вкладывать деньги в испытанную инсталляционную технологию. Такие компании применяют установочные программы собственной разработки. В результате частенько встречаются системы, способные презрительно поправить файлы операционной системы, не спрашивая на то разрешения и не создавая резервных копий. Продукт одной творческой лаборатории, производящей звуковые карты, не только требовал установки Windows-драйверов под MS-DOS, но и занимал столько памяти, что для его запуска приходилось убирать из CONFIG.SYS и AUTOEXEC.BAT все остальное.

Высокомерное программирование можно обнаружить не только в драйверах для периферийных устройств. Вспомните, как несколько лет назад вполне оправданно поднялся шум, когда стало известно, что один крупный провайдер онлайновой информационной службы автоматически «обновлял» ваше программное обеспечение, записывая что-то на ваш диск при подключении к системе. (Конечно, современные броузеры, антивирусные программы и многие другие приложения запросто делают это через Интернет, внезапно превращая неизменный рабочий стол в незнакомую территорию или мистическим образом снижая производительность работы без вашего разрешения.) Кроме того, операционная система провайдера считывала данные с вашего диска и сообщала их вам, когда вы подписывались на предоставляемые услуги. Здесь важно не то, что была возможность отменить эту опцию, а то, что это было высокомерие, расчетливое высокомерие: запрограммировать систему так, чтобы эта опция была активна изначально.

Наверное, самым ярким примером высокомерного программирования является активная электронная почта, которая запускается при открытии почтового сообщения. Такая эгоистичная программа не только запускает себя, не дожидаясь завершения инсталляции, но и широко открывает двери для новых способов распространения вирусов, червей и троянских коней. (Конечно, это было дальновидным, но никем не замеченным предупреждением, которое предсказало появление таких вирусов, как Melissa и Love Bug.)

Более смиренный код зачастую более прост и менее амбициозен. Его конфигурацию можно изменять, и пользователи сохраняют над ним конт-

роль. Он сообщает пользователям необходимую информацию, но не становится при этом навязчивым. Он любезно предлагает не делать то, что уже сделано. Такой код работает тихо, надежно и сотрудничает с другим кодом.

Ах, прекрасная Беатриче, будь ты здесь, ты научила бы нас истинному смирению.

Из журнала *Software Development*, том 4, № 2, февраль 1996 г.

50

Интерфейсы разнообразные

Разнообразие не только политически корректно. Оно полезно. Оно полезно в командах, способствуя творческому сотрудничеству, а также на рынке, где слово «разнообразие» стало еще одним заклинанием в лексиконе рекламных агентов и представителей. Теперь PC-продукты еще и P.C. продукты, а разнообразие проникло и в пользовательский интерфейс.

На одной конференции по CHI (Computer Human Interaction, взаимодействие между компьютером и человеком), проходившей в Бостоне, один из участников ратовал за подгонку дизайна пользовательского интерфейса под разнообразные этнические группы и сообщества пользователей. Каким образом? Разные пользовательские интерфейсы для разных «демографических групп». Это утверждение было проиллюстрировано на примере обыкновенного диалогового окна для настройки атрибутов текста: шрифта, размера, стиля и т. п. Унылый и утилитарный дизайн – не оптимальный, но и нельзя сказать, что ужасно плохой. Затем были показаны разнообразные варианты.

Круговое мышление

После превосходного короткого рассказа о так называемом «европейском» дизайне (может быть, шрифт в нем назывался Eurostile?) речь зашла о дизайне для женщин: круглое диалоговое окно с маленькими круглыми кнопками и текстом, расположенным по дуге. Многие женщины говорили, что это выглядело как коробочка для контрацептивных таблеток. Или как пудреница. Как точно!

Неважно, действительно ли криволинейный рисунок женщинам нравится больше, чем мужчинам, как утверждал этот дизайнер. Диалоговое окно для настройки стиля текста – это инструмент. Суть здесь заключается в том, что круглое диалоговое окно просто-напросто намного неудобнее для любого человека – будь то женщина или мужчина. Текст, который расположен по дуге или наклонен, значительно труднее читать, чем обычный горизонтальный текст, а круговая прокрутка просто нелепа.

По признанию самого докладчика, этот дизайн подвергался критике со времени первой публикации в ACM годом раньше. Отвечая на критику, докладчик показал слайд с традиционным диалоговым окном, которое предположительно разработали дизайнеры женского пола. Но через пару мгновений, как будто в подтверждение общей состоятельности концепции культурной настройки, он с гордостью показал диалоговое окно, разработанное для афро-американцев. Кнопки и блоки этого окна покачивались среди ярких разноцветных фигур. В качестве обоснования идеи был показан следующий слайд, на котором изображались аналогичные образцы африканского искусства. Здесь дело не в эстетической привлекательности африканских народных искусств. Даже не говоря о пользовательских предпочтениях и юзабилити-тестировании, можно отметить, что текстовые окна и кнопки в том дизайне были почти не видны на ярком фоне.

Это напоминает мне действия компании, которая несколько лет назад выпустила линейку механизированных инструментов для женщин. Компания посчитала, что мастеровитые женщины предпочтут покупать легкие инструменты с маленькими ручками. Предполагалось, что «дизайнерские цвета» инструментов отнюдь не помешают. К сожалению, эти инструменты были менее мощными и не слишком прочными. В итоге любая женщина, которая действительно хотела сделать что-нибудь сама, выбирала только настоящие инструменты: черные, некрасивые и полезные.

Хужеры[1] и африканцы

Такое подстраивание под разнообразных пользователей, проиллюстрированное на примере разноцветных диалоговых окон и красивых механических инструментов, нельзя считать ни уважительным, ни полезным. Вместо того чтобы учитывать и оценивать истинные различия, такой подход лишь закрепляет глупые и ошибочные стереотипы. Реальное разнообразие свалено в надуманные и бессмысленные категории, создавая то, что писатель Курт Воннегут назвал словом «гранфаллон» – ошибочным группированием.

[1] Уроженцы штата Индиана. – *Примеч. науч. ред.*

В конце концов, кто же этот архетипический европеец, который предпочитает выровненные по левой стороне заголовки и художественность оформления? Итальянец или, быть может, немец? Какую эстетику и художественные предпочтения разделяют жители Сицилии и Стокгольма? Как говорит реклама газеты *The European*: «The European – без нее вы не европеец». «Европейская культура», противопоставленная провинциальному французскому, тосканскому или датскому среднему классу, существует только в умах директоров рекламных агенств и руководителей корпораций.

Мы создаем гранфаллоны из многих составляющих, но особенно из Африки – целого континента с десятками очень разных народов. В средствах массовой информации Африку часто воспринимают как одну страну, а африканцев – как один народ. Такое всеобъемлющее определение стирает все интересные и важные различия. В одном рекламном объявлении, опубликованном в недавнем воскресном приложении, даже говорилось о круизе «по таким экзотическим портам, как Рио-де-Жанейро, Сингапур и Африка». Это правда. Наверное, в порту Африка находится громадная гавань!

Подстройку под разные вкусы можно легко выполнить без создания пользовательских стереотипов и без ущерба для юзабилити. Например, в соревновании по разработке проектов интерфейса для компьютерной голосовой почты команда из Клариса (Claris) продемонстрировала главное диалоговое окно, в котором пользователь мог выбирать «лицевые панели» разной формы и внешнего вида. При этом основная функциональность и простота применения оставались неизменными. В другом углу помещения для соревнований – и на другом конце диапазона отношения к пользователям – находилась команда одного из крупнейших производителей программного обеспечения. Она представляла незамысловатый, общий для всех пользователей интерфейс на основе окон, пиктограмм, меню и указателей. Ого, хм.

Хороший дизайн учитывает реальные потребности и особенности пользователей. Возьмем обыкновенное велосипедное седло. Современные туристические велосипеды имеют седла, которые изредка признаются комфортабельными для всех людей с нормальной нервной системой. Однако они больше подходят для мужского строения тела, чем для женского. Группа женщин, инженеров и любительниц велосипедной езды, предложили совершенно новый тип седла, которое соответствует особенностям женского телосложения. Оно черное, непритязательное, но значительно более удобное – во всяком случае, так мне говорили.

Самое важное в разработке пользовательских интерфейсов – предоставить пользователю возможность легко подстраивать интерфейс под свои

вкусы и привычки. Такие изменения должны способствовать, а не мешать выполнению задач с помощью данного программного обеспечения.

Эстетические опасения

Эстетика является важным элементом пользовательского интерфейса, но она не должна сказываться на юзабилити. Я могу изменять содержимое и расположение панели инструментов, но мне не нужно, чтобы у сообщений были кружевные края. На мой взгляд, в простой практичности есть своя красота. Перечница или солонка, защелкивающиеся штекеры. Инструменты, которые хорошо работают, которые точно соответствуют своему назначению и хорошо лежат в руке пользователя, прекрасны сами по себе. Нам не нужно договариваться обо всех деталях, поскольку эстетика программного обеспечения должна быть под контролем самого пользователя.

В течение дня я много времени провожу перед монитором. Для меня «обои» на рабочем столе становятся виртуальным окном, как любимое кухонное окно над раковиной, через которое можно смотреть в сад, пока моешь посуду. Иногда на моем рабочем столе открывается панорама каскадов и потоков Города Водопадов (Waterfall City) в изображении художника Джеймса Герни (James Gurney). А иногда на нем можно увидеть грубые, но красивые пролеты мостов Рона Валотски (Ron Walotsky) и виадуки, пересекающие воображаемый город. Иногда золотые краски чужого солнца вырисовывают прекрасные воздушные арки Aristoi Джима Бернса (Jim Burns). Можно ли назвать этот интерфейс женским? Или типично мужским? Американским? Европейским? Интеллектуальным? Чувственным? Кто знает. Об этом не говорится в учебниках по пользовательским интерфейсам. Хотя стили этих художников, мастеров современного фантастического реализма, совершенно отличаются друг от друга, их объединяет одно эстетическое звено: я. Именно я собрал эти изображения вместе. Именно я могу смотреть на эти виды, размышляя над следующей главой.

Если вы хотите продать свой интерфейс большему количеству людей, не создавайте стереотипов и не ставьте свои догматические представления о дизайне между пользователем и программным обеспечением. Дайте пользователям возможность самим настраивать то, что имеет для них значение, будь то внешний вид программного обеспечения или его работа. А затем просто отойдите в сторону.

Из журнала *Software Development*, том 2, № 8, август 1994 г.

51

Мастеры

Согласно одной из аксиом научной фантастики, непосвященным любая новая технология кажется магией. Даже для тех, кто в курсе дела и знает, что все это лишь сложение с плавающей точкой и условная логика, компьютеры в какой-то степени представляются черной магией. Некоторые разработчики программного обеспечения даже усиливают ощущение волшебства, добавляя в свои продукты всевозможные мастеры (wizards), агенты и элементы активного интеллекта.

Подобно специалисту по приготовлению коктейлей, программный мастер задает наводящие вопросы, а затем делает выводы. Мастер в презентационном пакете может спросить о виде презентации, основной и второстепенных темах, о составе аудитории и о ваших целях. Далее мастер выбирает цветовую схему, шаблон слайдов и затем показывает вам заглавный слайд. Может быть, это хороший способ, а может быть, не очень, но, по крайней мере, он дает вам возможность не думать. И если вы не против визуальных материалов, которые выглядят как у всех, тогда вам и не нужно иметь хороший вкус или знать что-либо о композиции и дизайне.

Просто игра

Стивен Уейс (Steven Weiss), программный методист и волшебник, однажды сказал мне: «Волшебник – это актер, который играет роль волшебника». Тогда мастер – это, должно быть, тупая программа, которая играет роль разумного актера. Хотя мы все знаем, что настоящей магии не бывает, по каким-то причинам мы готовы верить, что искусственный интеллект реален – когда видим его подобие на экране.

Даже эксперты могут доверять экранным псевдо-экспертным системам в программном обеспечении, имитирующем интеллект. Однажды я оценивал один CASE-инструмент, который включал в себя комплексную экспертную систему для автоматического преобразования аналитической модели (схемы потоков данных) в модель проектируемой системы (схему модульной структуры). В то время шло ее бета-тестирование и расширенное внутреннее применение в реальных проектах. Ни один пользователь так и не заметил, что независимо от изменения условий и критериев система всегда выдавала один и тот же вариант дизайна – выполнялась обыкновенная механическая трансформация входных параметров. Это вовсе не интеллектуальный дизайн и даже не особенно хороший. Экспертные возможности этой «интеллектуальной» программы никогда не подвергались сомнению. Рекомендации системы принимались без анализа и критической оценки.

Но мастеры хотя бы ждут просьбы показать фокусы или же стоят в стороне. Активные агенты – другая история. Эти резидентные программы следующего поколения сидят и наблюдают за тем, что происходит. Время от времени они предлагают помощь, или совет, или выдают какое-нибудь сообщение. В одном превосходном новом ГПИ каждый агент представлен карикатурной фигурой. Старуха с седыми волосами, стянутыми в пучок, представляет библиотекаршу. Строгий мужчина изображает менеджера ресурсов, а странный тип с опущенной на глаза шляпой – агента поиска. Когда эти агенты хотят что-нибудь сказать, выражение их лиц меняется. Они могут слегка хмуриться, чтобы указать на возможную ошибку, или поднимают палец и показывают, что у них есть какое-то предложение или сообщение. Если не говорить о сексизме и культурных предрассудках, можно ли считать эту идею хорошей?

Некоторые производители программного обеспечения думают, что электронная почта должна быть активной и по своей воле выполнять что-либо на машине получателя – например, запускать какое-нибудь приложение или удалять себя, если она долго лежит непрочитанной. Мне это больше всего напоминает вирус. Не знаю, как остальной корпоративный мир, но я хочу, чтобы моя электронная почта ничего не делала на моей машине, а только сидела бы тихо, пока я не решу, что с ней делать.

Люди очеловечивают все, что видят вокруг себя – от машин до кошек. Многие пользователи компьютеров дают своим машинам личные имена. Очень многие, в том числе и немало разработчиков программного обеспечения, приписывают личностные качества программному и аппаратному обеспечению. Для некоторых программистов это может быть своеобразной манерой говорить о технике, но для многих людей это действительно анимизм – форма магического мышления, в которой неживым и неразумным вещам приписывается способность действовать и думать. Имен-

но так мы и поступаем, когда что-то не понимаем или не можем с чем-нибудь разобраться. Мы все этим грешим.

Мой первый микрокомпьютер получил имя Большой Чародей. Даже с ограниченной памятью и слабоватой графикой он мог выполнять некоторые довольно впечатляющие действия, связанные не только с проектированием. После целого дня работы в его памяти то тут, то там появлялись провалы. Я прекрасно понимал, что под термической нагрузкой часть микросхем работала на пределе, однако это действительно *выглядело* так, словно компьютер становился капризным и раздраженным, будто уставший ребенок.

Если бы не моя скептическая натура и не знания, полученные в Массачусетском технологическом институте, я, наверное, поверил бы, что Чародей испытывает некую неприязнь к моей подруге, которая иногда набирала на нем какие-нибудь бумаги. У меня он работал идеально, но стоило ей сесть за него вместо меня и начать набирать, как через пять минут Чародей начинал выбрасывать на экран какой-то мусор и глючить, зациклившись на какой-нибудь подпрограмме. Когда я снова садился за него, все опять работало хорошо. Я знаю, что этому можно найти какое-то рациональное объяснение (хотя мы так его и не нашли), но было трудно не поверить в то, что моя подруга и Чародей просто не ужились.

Тупые терминалы

Кончено, компьютеры не думают. Даже самый лучший, так называемый машинный интеллект представляет собой не что иное, как человеческий интеллект, скрытый за ширмой пикселов и мегабайтов. Ни один из современных «умных» механизмов не имеет показатель искусственного IQ выше нуля. Вся эта «интеллектуальность», если ее можно так условно назвать, является вымышленной. На недавней конференции один юзабилити-каббалист, рассказывавший об «обольстительных интерфейсах», сказал, что при помощи интеллектуальных пользовательских интерфейсов можно даже «имитировать реальную интимную близость». «Это вовсе не трудно, – уверял он. – Ну, вы знаете, точно так же, как вы делаете это с вашей женой». С его женой, может быть.

Возможно, мы уже готовы признать реальность такой механической «интимности» фабричного производства. Машина теперь «персонализирует» письма, поздравительные открытки и рекламные сообщения. Многие из нас получали такие фотокопии рекламных посланий с кое-как прикрепленной запиской: «Ларри, я подумала, что это тебя заинтересует. С.». Каждый из нас знает кого-нибудь, чье имя начинается с «С», поэтому предполагается, что мы подумаем, будто это личная рекомендация от друга или знакомого. На самом деле это очередная массовая рассылка.

«Мастерство» Windows – это форма отвлечения внимания. Отвлечение внимания – это один из фундаментальных рабочих принципов магии, ее реальная, вернее, обманная сущность. Если фокусник показывает на что-либо, или машет чем-нибудь, или о чем-то говорит, он не хочет, чтобы вы заметили что-то другое. Одно из первых правил разгадывания фокуса – никогда не смотреть туда, куда вам предлагают смотреть. Обращая лица пользователей к панелям инструментов и создавая диалоговые окна, имитирующие интимность, мы отвлекаем пользователей от вопросов, касающихся предназначения программного обеспечения. Мы предлагаем им посредственность, называя ее магией. Мы предлагаем иллюзию разумного программного обеспечения, соблазняя пользователей возможностью переложить задачу мышления на машину. Естественно, машина не думает. В итоге получается, что не думает никто.

Возможно, в программном обеспечении должны быть предупреждения наподобие тех, которые наклеиваются на продукты питания и другие товары. «Данное программное обеспечение содержит подпрограммы, которые прикидываются умными». Или «Применение мастеров, агентов и других форм искусственного интеллекта не заменяет применение собственной головы». Одно скромное, но показательное исследование выявило: чем больше пользователям говорят о внутренней логике экспертной системы и технологии ее проектирования, тем менее вероятно, что они будут слепо доверять этой системе. Так что надежда еще есть.

Из журнала *Software Development*, том 2, № 9, сентябрь 1994 г.

52

Образы будущего

Какого цвета ваш персональный компьютер? Модный ли у вас лэптоп? Моя дорожная лошадка темно-серая, а офисная машина обычного серо-бежевого цвета, который в магазинах офисной мебели иногда называется «цветом оконной замазки». Но не все так плохо. Новая линейка мультимедийных компьютеров со встроенным радио- и ТВ-тюнером, а также видеомагнитофоном – настоящие «информационные устройства» – появятся в различных цветах, модных и ярких. Будут ли они гармонировать с другими нашими домашними приборами, которые в основном окрашены в цвет слоновой кости или черного анодированного алюминия? Не спрашивайте.

Когда-то давно логарифмические линейки выпускались в кожаном футляре, который можно было пристегнуть к ремню, но никто их на ремне не носил. Это было тогда, а что теперь? Теперь переносные компьютеры и просто компьютеры – это модные аксессуары. По словам одного из докладчиков на конференции, посвященной интерфейсам человек-компьютер, именно об этом сейчас фантазируют светлые умы, которые дали миру компьютер, чтобы остальные могли им пользоваться.

Мягкое обольщение

Все старое теперь становится новым. Помните те слащавые «дружественные» интерфейсы, которые приветствовали вас по имени и желали вам удачного дня, даже если вы ошибались и вводили неправильную дату? Некоторые гуру в ГПИ, которые тогда посчитали это нормальным, сейчас идут еще дальше и предлагают «обольстительные интерфейсы», интер-

фейсы для массовых рынков, которые представляют компьютер как друга, как приятеля, как партнера. Но хотите ли вы, чтобы компьютер был вашим приятелем? Или же вы предпочитаете, чтобы он просто помогал вам выполнить нужную работу?

Эти фантастические интерфейсы не зависят от какого-нибудь эфемерного искусственного интеллекта. Или, по крайней мере, они не зависят от того истинного искусственного интеллекта, который хотят получить ученые-компьютерщики. Обольстительные интерфейсы – это маркетинговая уловка, стремящаяся вызвать неодолимый интерес к какому-либо программному обеспечению. Дружба и привлекательность имитируются с помощью подстройки интерфейса под личность покупателя. Один дизайнер имел смелость сказать, что такое программирование является простым, поскольку есть только четыре типа личности, а все необходимые знания о человеке можно получить, задав около двадцати вопросов. Последний раз, когда я интересовался этой темой, нашлось около двадцати *теорий* личности, и большинство из них было сложнее, чем эта.

Судя по исследованиям, проводимым крупными производителями программного обеспечения, интерес может представлять не только примитивная псевдопсихология, но и изучение способов сделать программное обеспечение действительно полезным. Нам говорят, что люди хотят получить эту милую чепуху. Что им больше нравится, когда с ними правильно говорят, чем возможность нормально работать. Что они предпочтут передать все управление машине. Но действительно ли вы хотите, чтобы компьютер вас обольстил, особенно тот компьютер, который вас обманывает?

Есть еще более дикие идеи относительно интерфейсов: например, «накидка знаний» – сочетание театральной накидки и гибкого (в прямом смысле этого слова) компьютера. Просто берите с собой эту накидку, предварительно загрузив в ее память необходимую информацию: систему проектирования программного обеспечения при помощи объектно-ориентированной технологии – для офиса, новости и остроты – для вечеринки, базу данных с откликами покупателей – для редкого похода в универсам. Накидка будет шептать вам на ухо все необходимое для того, чтобы быть экспертом в любой области, в какой пожелаете. Она даже сможет напомнить имя знакомого, стоящего перед вами, – по всей видимости, идентифицируя личность с помощью встроенной оптической и звуковой аппаратуры или, возможно, с помощью некоего нейронного интерфейса, который пока еще не придумали (это всего лишь детали; не следует беспокоить мечтателей такими мелочами). Благодаря «накидке знаний» мы сможем перейти от компьютеров, имитирующих интеллект или интимность, к компьютерам, которые помогают людям это имитировать. Это будет превосходный новый мир, в котором очарование и легкость общения будут доступны тем, кто сможет себе позволить наибольший объем

памяти «накидки» и самое свежее программное обеспечение. Но действительно ли вы хотите быть среди людей, чье понимание основано на магнитно-дисковых схемах, прикрытых вельветовой материей?

Наручная архитектура

Но погодите, есть еще кое-что, как говорят в ночном телемагазине. В современной киберпанковой моде уже появились булавки для галстука и ушные кольца, сделанные из лишних микросхем и цифровых прибамбасов, а завтра, как нам говорят, мы будем носить вещи еще круче. У нас будут целые компьютеры, помещенные в модные браслеты (наверное, это случится тогда, когда «накидка знаний» уже станет повседневной одеждой). А волшебники интерфейсного проектирования даже предлагают, чтобы в наручных компьютерах применялись инфракрасные или коротковолновые передатчики. Они могли бы автоматически связываться, будучи в непосредственной близости друг от друга, – естественно, с применением соответствующего протокола и верификации цифровой подписи, которые предназначены для управления соединениями и установки связи с нужным абонентом. (Полагаю, миниатюрные версии таких устройств будут называться «запонками».) Когда друзья будут встречаться и пожимать друг другу руки, их компьютеры будут автоматически обмениваться сообщениями, что придаст новый смысл выражению «протокол установления связи» (handshaking protocol)[1]. Какой прогресс! Технология служит человеку, устраняя необходимость таких неприятных человеческих действий, как обычный разговор или еще более ужасная интимность тет-а-тет. Не проще ли обмениваться файлами, чем говорить друг с другом?

Взгляд в будущее расширяется. С развитием технологий станет возможно не только соединение близко расположенных устройств, но и широковещание. Вам не придется выслушивать какого-нибудь нахала, который пытается каждому из присутствующих на вечернике рассказать глупую историю о том, как он однажды застрял в лифте с Джоном Скали. Этот отчет просто скачается на ваш наручный процессор, когда вы войдете в комнату. Потом ваше умное программное обеспечение, запрограммированное выявлять всякую чепуху даже на расстоянии, быстро удалит ее. Не нужно будет выслушивать и даже перетаскивать этот мусор в корзину.

Это наручное чудо может связываться со стационарным компьютером при въезде в гостиницу, загружая ваш профиль в комнату. Температура и освещение будут настроены по вашему вкусу, а на экране размером со стену появится необходимый хранитель (screen-saver). Или же этот экран будет демонстрировать интересующие вас заметки из *Wall Street Journal*,

[1] Handshaking protocol (англ.); дословно – протокол рукопожатия. – *Примеч. ред.*

или *Software Development*, или alt.wierd – по вашему вкусу. Действительно, одной из самых распространенных фантазий о грядущем микрокомпьютерном тысячелетии является программное обеспечение, предназначенное для поиска и отбора информации из огромного цифрового потока, смывающего информационные границы.

У меня уже есть доступ к подобной системе. Мой партнер просматривает публикации в различных изданиях – от *Journal* до *DBMS* – и неизменно сообщает мне о том, что может меня заинтересовать, просветить или быть для меня полезным. То же самое я делаю и для него, просматривая другие издания. Попытка переложить такую работу на компьютерную программу ставит два почти непреодолимых вопроса. Большая часть заметок под заголовками, которые меня заинтересовывают, не стоят прочтения, а многие из самых ценных публикаций оказываются в новых рубриках, которые я никогда не могу определить заранее. Сможет ли компьютер понять вас так хорошо, чтобы справиться с этой функцией? Сможете ли вы доверять компьютеру, если будете знать, что его «понимание» притворно?

Возврат управления

«Накидки знания» или компьютерные браслеты, или текстовые процессоры, способные переписывать вашу прозу, или программное обеспечение, умеющее определять, кто вы есть, – наверное, все это еще так далеко в будущем, что пока мы можем чувствовать себя спокойно. Мы знаем, что, подобно предсказанию останова машины Тюринга, некоторые задачи не поддаются решению ни в теории, ни на практике – даже если их можно легко описать. Если крупные производители программного обеспечения не могут создать презентационный пакет без утечек памяти или операционную систему, которая не обрекает себя на забвение в виде пустого экрана, то нужно ли нам беспокоиться о том, смогут ли они создать интеллектуальное программное обеспечение или запрограммированную интимность?

К сожалению, дизайнеры ГПИ, подобно дизайнерам модной одежды, зачастую больше заботятся о внешнем виде, чем о возможностях. Гибкие интерфейсы подгоняются под пользователей с помощью настройки несущественных деталей. Вы можете выбрать почти любую цветовую схему, но не надейтесь, что программа будет подстраиваться под ваши реальные нужды и различные стили работы.

Как напоминает нам юзабилити-гуру Бен Шнейдерман (Ben Shneiderman), люди хотят управлять, но также хотят ощущать достижение результата. Им нужна возможность контролировать важные процессы. В конце дня они хотят думать не о работе программного обеспечения, а о том, чего *они* добились. Нам же, разработчикам, нужно определить для

себя, чего мы хотим достичь. Хотим ли мы создавать модное программное обеспечение или же полезное? Хотим ли мы, чтобы пользовательские интерфейсы предоставляли пользователю возможность контроля, или же они просто должны хорошо имитировать выполнение работы?

Из журнала *Software Development*, том 2, № 10, октябрь 1994 г.

IX

Култура и качество

Введение

Организационная культура настолько глубоко проникла в нашу деятельность, что кажется разумным завершить эту серию заметок исследованием культурного ландшафта разработки программного обеспечения – не только организационную культуру отдельных групп и компаний, но и некоторые более общие культурологические вопросы в нашей индустрии и профессии в целом. В разделе III были описаны основные культурологические разновидности рабочих организаций. Этот заключительный раздел возвращает нас к теме культуры. В нем рассматривается связь культуры с эффективностью работы всей организации и ее частей, а также с качеством конечных продуктов.

Слово «cultured»[1] может звучать как одобрение, если речь идет о человеке, или как признак подозрительного происхождения, если речь идет о жемчуге. Культура организации или индустрии может работать как на пользу, так и во вред. Чем характеризуется успешная организационная культура? Как можно изменить рабочую культуру, чтобы повысить шансы на успех? Культура разработки, в которой ценится и поощряется качество работы, будет способствовать достижению успеха. Индустрия или профессия, построенная на таких принципах, как ответственность и непрерывность обучения, повышает ценность и потенциал ее представителей.

Однако при общем рассмотрении культуры и организации можно легко потерять из виду личность и ее место в *peopleware*. Поэтому в завершение поговорим о той ключевой роли, которую должны играть люди как в передаче, так и в преобразовании культуры. Культура не является неизменной абстракцией. Она может меняться, в том числе и в результате чьей-то личной инициативы. Вы тоже важная часть культурного ландшафта разработки программного обеспечения. Вы созданы этой культурой и формируете ее. В той или иной степени вы можете влиять на изменение культуры, развивая профессию и улучшая состояние дел в сфере разработки программного обеспечения.

Думайте нестандартно.

---

[1] Cultured (англ.) здесь: а) культурный (человек); б) cultured pearls – искусственный жемчуг. – *Примеч. ред.*

53

Культурное изменение

Когда вы находитесь в Сиднее, поступайте так, как поступают жители этого города. Много лет назад, когда американские кафе с золотистыми сводами, входящие в сеть быстрого питания, впервые появились на Пит Стрит, в них предлагали гамбургеры австралийского типа, с маринованной свеклой сверху. Не так давно американские автопроизводители были обеспокоены тем, что японские покупатели приобретали мало американских машин, тогда как модели, произведенные в Японии, хорошо продавались в Америке. Конечно, машины, привезенные из Японии в Штаты, имели руль с левой стороны, чтобы соответствовать дорожной культуре, согласно которой автомобили ездят по правой стороне дороги. А Японии, небольшой островной стране с левосторонним движением, Детройт пытался продавать большие американские машины с неправильно расположенным рулем управления!

Этот простой урок, заключающийся в том, что для успешного импорта необходимо учитывать местную культуру, относится также к импорту новых технологий, инструментов и методов в организации по разработке программного обеспечения. Будь то программные инструменты, или концептуальные инструменты, или инструменты управления, или инструменты маркетинга – если они не адаптированы к местной культуре, применять их на практике будет довольно трудно.

Снижение риска

Лица, принимающие деловые решения, всегда ищут их обоснование, если предлагают что-то новое. Девиз «Либо ведите, либо следуйте, либо

отойдите в сторону» долгое время был популярен среди руководителей компаний, однако мало кто действительно хочет «вести». Компания, которая ведет, может легко пойти по неверному пути. Или попросту исчезнуть. Руководители хотят знать, кто еще применяет тот или иной продукт или процесс и как он помогает пользователям. Они хотят слышать о компаниях, достигших успеха, желательно в той же сфере деятельности, а может быть, даже в соседнем квартале.

Перед введением объектной технологии, или архитектуры веб-серверного приложения, или среды визуального проектирования руководители разработки зачастую ищут эталонную модель – установленный план действий, уменьшающий риск, или хотя бы общее описание, которое может вселить хоть какую-то уверенность. Но было бы ошибкой искать какой-нибудь стандартный метод для переноса технологии. Нет двух полностью одинаковых корпоративных культур. Компании, работающие в одной отрасли, на одном рынке, могут иметь очень разные рабочие культуры. То, что хорошо подходит для одной компании, может быть совершенно неприменимым для компании по соседству.

Попытки применения новых методов и технологий разработки могут оказаться неуспешными по многим практическим причинам. Вы можете выбрать не тот язык программирования, столкнуться с трудностями при получении технической поддержки или выбрать поставщика, который находится на грани банкротства. Но даже несмотря на правильные решения, правильные инструменты и правильные методы, вы все равно можете ничего не достичь, если не будете учитывать культурологические реалии, в которых работает ваша группа, разрабатывающая программное обеспечение.

Культура компании складывается из многих составляющих. Культура компании – это планировка рабочих комнат и принцип, по которому распределяются комнаты с окнами. Это традиции, по которым бумаги либо аккуратно разложены на рабочих столах, либо свалены в кучу в коридоре. Это дни, когда все носят обычную одежду, и дни, когда можно принести пиво и пиццу и надеть яркий галстук. Это ежемесячные полуобязательные коктейли с канапе. Однако стратегия по введению и распространению новых подходов в группах, разрабатывающих программное обеспечение, должна придавать особое значение определенным вопросам культуры.

Меня, нас или их

Откуда организация знает то, что она знает? Как она контролирует то, что делает? Как она решает, что нужно предпринять? В каждом случае нас интересует главный центр – основное хранилище знаний, методов

контроля и принятия решений. Что это – конкретный человек, неформальная группа или формальный институт?

Первый важный вопрос: где находятся профессиональные ноу-хау организации? Хранятся главным образом в головах сотрудников? Являются частью коллективного фольклора – «здесь принято делать так»? Или же они формально закреплены в методах и процедурах? Другими словами, где «живет» практический опыт и кто им «владеет»? Сколько ноу-хау можно найти в корпоративных руководствах и инструкциях? Куда нужно пойти, чтобы узнать о ходе разработки программного обеспечения? Полагается ли ваша компания всегда и во всем на знания и умения ее блестящих и опытных сотрудников? Что можно узнать, околачиваясь в столовой или слушая разговоры в коридорах? Определяется ли «наилучший способ» на основе личных предпочтений? Или же «хорошее программирование» основано на неписаных законах о том, что работает и что не работает? Как происходит накопление знаний с каждым новым проектом – индивидуально, или распространяется по всей группе, или же эти знания добавляются в архивы и формальные инструкции, принятые в организации?

Подобные вопросы мы можем задать о контроле разработки и, в частности, о том, как обеспечивается качество продуктов. Полагаетесь ли вы на чье-либо личное мнение и на то, что у вас самые лучшие работники? Существует ли в группе неформальная взаимовыручка? Заглядывают ли сотрудники друг другу через плечо, чтобы проверить или сравнить программный код? Или же в организации есть принятые стандарты, а соответствие продуктов этим критериям регулярно проверяется?

То же самое относится и к решениям о том, что нужно делать и как. Если группа обычно работает на основе консенсуса, то указ руководства компании о применении группового инструмента G вряд ли пойдет на пользу. Группа, привыкшая к тому, что ей прямо говорят о следующем проекте и технологии для его реализации, может неверно воспринять постановку на голосование вопроса о быстром создании прототипов.

От альфы до омеги

Если новые технологии и методы применяются неверно, то их внедрение будет затруднено, а шансы на успех – малы. Например, один из методов перехода к широкому применению объектно-ориентированного проектирования может заключаться в привлечении большой группы консультантов. Они придут со своим рациональным и унифицированным «Объектным методом для разработки гигантских приложений» (Object Method for Engineering Giant Applications, OMEGA) в комплекте с семью томами технологических руководств и предложат несколько ступеней обучения с

применением интегрированного CASE-инструмента. Такая участь может ожидать группы, в которых традиционно применяется дисциплинированное проектирование, а полки заставлены замусоленными руководствами по стандартам. Однако группа свободно мыслящих программистов, работающих по своим собственным стандартам, будет увиливать от таких занятий, чтобы покодировать. Они не станут читать инструкции и не будут обращать внимание на всю эту «корпоративную чепуху». Их методы, так же как и их умения и способности, сугубо индивидуальны. Для того чтобы до них «достучаться», вам не нужно загонять их в большой класс на просмотр учебного видеофильма или отдавать в руки бездумного инструктора. Они привыкли на свой страх и риск определять, что стоит изучать, и самостоятельно доставать необходимые материалы, поэтому каждого из них нужно убеждать в отдельности, а затем обеспечивать их локальными ресурсами для индивидуального практического обучения.

То, что вы пытаетесь внедрить, должно соответствовать культуре. Возьмем инструменты моделирования. Инструменты, построенные на основе конкретных методов и подразумевающие стандартные способы их использования, будут более полно и эффективно применяться там, где уже сложилась четкая практика работы. Гибкие инструменты для создания диаграмм или пакеты произвольной конфигурации больше подходят для групп, которые опираются на индивидуальную работу или неформальную групповую культуру.

Одни и те же инструменты разные группы могут применять по-разному. Групповое обеспечение (groupware) может подойти для организации, в которой принято тесно сотрудничать при проектировании и принятии решений. Однако группа разработчиков-одиночек, которые любят работать обособленно, вряд ли заинтересуется Lotus Notes. В компании кодирующих ковбоев сложное групповое обеспечение станет не более чем причудливым средством для дружеской переписки.

Конечно, в реальных организациях культура обычно определяется соотношением индивидуальной работы, неформальной групповой культуры и официально установленных правил. В хороших стратегиях по переносу технологий учитывается специфическое сочетание компонентов, характерное для конкретной организации.

Кто-нибудь будет маринованную свеклу?

Из журнала *Software Development*, том 2, № 12, декабрь 1994 г.

54

Агенты изменения

Одна рыба, сделав правильное движение в нужный момент, может изменить курс всего косяка. В группе, разрабатывающей программное обеспечение, успешность введения нового инструмента или улучшенного метода управления версиями часто зависит от одного-двух ключевых игроков, которые действуют в качестве «агентов изменения». Эффективные агенты изменения представлены в разных лицах. Некоторые из них занимаются прямыми продажами. Они ловят вас в коридоре и устраивают демонстрацию преимуществ Java или убеждают применять библиотеку элементов ГПИ. Каждому менеджеру, которого им удается поймать, они рассказывают о достоинствах «чистого программирования». Другие агенты могут вызвать изменения, просто выполняя что-либо намного лучше, чем люди вокруг них. Например, они могут показать способ разрешения возникшей дилеммы или просто продемонстрировать, что иногда программу все же можно сдать в срок.

Однако метод уговоров, применяемый для продаж, в некоторых организациях может привести к провалу, а скромная эффективность даже самого блестящего разработчика может остаться совершенно незамеченной среди творческого хаоса, который процветает в некоторых группах. Во многих областях, где технология сталкивается с людьми, тактика и стиль должны быть согласованы с организационной культурой. Для осуществления изменений в разных организациях могут потребоваться разные типы агентов или стилей агентурной работы.

Важно не путать агентов, которые действительно вызывают изменения или побуждают к этому других, с активистами или сторонниками, кото-

рые могут проводить громкие, заметные, но ни к чему не приводящие кампании. Активисты, призывающие к техническим изменениям, склонны имитировать стандартную тактику заурядных политиков. Они предпринимают заметные и видимые шаги: рассылают уйму писем, распространяют анкеты, берутся за выпуск бюллетеней или организацию тематических групп. Такая тактика привлекает внимание и может улучшить осведомленность в чем-либо, но зачастую она имеет малое отношение к изменениям в организации, – например, действительно ли организация поменяет свой курс и будет стремиться к повышению уровня повторного использования кода. Умение подбирать людей, которые могут продвигать технологии или приспосабливать к ним свой стиль, не менее важно, чем выбранный язык программирования или поддерживаемые платформы.

Одобрение руководства

В компании, чья организационная культура построена наподобие египетской пирамиды, поддержка руководства, возможно, является главным ключом к успешному проведению изменений. Попытки внедрения новых методов, предпринимаемые без поддержки начальства с верхних уровней пирамиды, обречены на провал или на преодоление множества организационных барьеров.

Тем не менее простого мандата, полученного от «верхов», недостаточно. Поддержка руководства означает личное и организационное одобрение, активное продвижение и практическую помощь в виде необходимых и достаточных ресурсов. Руководитель-организатор может не только одобрить и профинансировать какое-то действие, но и придать ему официальный вес. Само по себе это не заставит разработчиков изучать и применять более эффективные методы визуальной разработки, но без придания изменениям официального веса вряд ли их можно провести. По крайней мере, в компаниях, основанных на иерархии власти, высокопоставленные покровители являются ключевыми фигурами в самых успешных технологических переходах. Агенты изменения могут вызывать и поддерживать такие переходы, но именно содействие руководства приводит к их реализации.

Способные коллеги

Не во всех компаниях структура напоминает Transamerica Building[1]. В современных сплюснутых иерархиях, в которых есть много открытых, неформальных групп с высоким уровнем сотрудничества, корпоративные

[1] Небоскреб в Сан-Франциско. – *Примеч. науч. ред.*

мандаты уже не имеют такого значения. Разработчики, привыкшие принимать совместные решения в сплоченных проектных командах, хотят инициировать, планировать и вводить улучшения самостоятельно. В таких условиях самыми эффективными агентами изменения зачастую являются наиболее уважаемые сотрудники, которые могут выступать в роли неформального внутреннего консультанта для людей из своей или другой проектной команды. Они обладают влиянием благодаря своему опыту и способностям, а не вследствие положения или власти. Они действуют скорее как помощники при обсуждениях или катализаторы стратегического изменения курса, а не как официальные организаторы или лидеры.

Энергичные выдумщики

Руководители-выдумщики играют очень заметную и важную роль во многих компаниях, присутствующих в отрасли программного обеспечения, однако не все группы поддаются харизматической привлекательности таких руководителей. Тем не менее в некоторых компаниях основная руководящая сила – это общее видение, которое выстраивает людей нужным образом и задает общее направление. Члены таких команд, обладающие необходимыми знаниями для превращения видения в реальность, способны работать слаженно и с ровной эффективностью.

В тех организациях, где соответствие общему видению является важным фактором в повседневной работе, самые эффективные агенты изменения чаще всего являются выдумщиками, миссионерами, способными увлечь компанию программистов своим представлением о более совершенных методах или о программном обеспечении, не содержащем ошибок. Они выступают в роли технических евангелистов, которые вызывают у людей личное одобрение новой миссии или нового образа организации. Одобрив или согласившись с новым видением, разработчики сами осуществляют переход – в той же самой тихой и эффективной манере, в какой они обычно создают программное обеспечение.

Лидеры моды

Если говорить об индустрии программного обеспечения, то громкая техническая шумиха – это, пожалуй, более типичное явление, чем монастырская команда разработчиков, смиренно кодирующих в своих кельях. В свободных группах, состоящих из индивидуалистов-новаторов, техническая харизма нескольких центральных разработчиков может быть важным фактором во внедрении новых методов и их адаптации. В каждой компании есть несколько технических суперзвезд, которые всегда стараются быть первыми в применении любых новых технологий. Эффективность этих лидеров технической моды в роли агентов изменения

главным образом зависит от типа организации, в которой они работают. В царстве творческого хаоса неформальное лидерство, основанное на признании лидером самого лучшего и блестящего среди равных, зачастую более эффективно, чем административное руководство или самая лучшая обучающая программа. Если вы окружены упрямыми и независимыми кодирующими ковбоями и все же хотите вводить новые методы и технологии проектирования, то посмотрите, кто устанавливает техническую моду. На кого все равняются в выборе инструментов и технологий? Эти люди являются естественными лидерами, которых вам следует привлечь в качестве агентов изменения.

Секретные агенты

Как и в мире политики и интриг, не все эффективные агенты работают в открытую и на видимом фронте. Тайные агенты могут незаметно продвигать свои предложения или подрывать существующий порочный режим. Агенты технического изменения, которым не хватает одобрения сверху или общей поддержки, все же могут проводить изменения, но для этого им приходится действовать скрытно или хитро. Для разработчиков-партизан, действующих в корпоративных условиях, это является одним из тех рискованных шагов, которые при успехе все окупают. Вам нужно действительно верить в этот новый язык, или объектную технологию, или архитектуру с разделением событий, или что-то другое, что вы предлагаете. Иначе, если ваш технический революционный заговор раскроют слишком рано или если вы и предлагаемая вами технология не смогут принести обещанных результатов, вы можете оказаться в ряду навязчивых консультантов.

Из журнала *Software Development*, том 3, № 1, январь 1995 г.

55

Встроено самое лучшее

Утренняя сцена: электронные радиочасы будят в 6.30 утра, и вы тащите себя вниз по кратчайшей траектории до кофеварки. Вы сливаете гущу со дна в кружку, а потом минуту нагреваете это в микроволновке. При этом вы стараетесь найти вечно теряющийся пульт вашего телевизора. Пока вы пьете свой кофе, вам удается пару минут посмотреть основные новости CNN, после чего вы направляетесь к машине, чтобы испытать на себе трудности обычной утренней поездки на работу. К тому времени как вы добираетесь до офиса и щелкаете по кнопке включения своего настольного компьютера, вы, вероятно, уже воспользовались десятком или более компьютеров, в которые загружены миллионы строк кода.

Непризнанные герои, разрабатывающие программное обеспечение, не пишут код ни для мэйнфреймов, ни для PC, ни для рабочих станций. Программное обеспечение, созданное ими, нельзя найти в архивах Staples, CompUSA или Dick Smith. И все же вы пользуетесь их кодом каждый день.

Вездесущие чипы

Конечно, я говорю о повсеместных вычислительных устройствах – о процессорах и программах, скрытых внутри наших радиочасов, в микроволновых печах, в телефонах, в магнитофонах. Они внимательно прислушиваются к нашим капризам и желаниям, выражаемым с помощью кнопок, и переводят их в инфракрасные сигналы управления тем или иным устройством. Если ваш автомобиль представляет собой одну из последних моделей, то только в нем можно обнаружить целую дюжину программных процессоров, не говоря уже о мобильном телефоне, которым вы пользова-

лись по пути на работу. Вы нажимаете на кнопку «Разговор» и соединяетесь через длинную цепь скрытых компьютеров, начиная с вышки сотовой связи и заканчивая цифровым телефонным коммутатором, стоящим в офисе вашего клиента. Это мир встроенных системных приложений, в котором компьютеры скрываются под разными масками и, похоже, делают все что угодно, кроме вычислений. В сравнении со встроенными приложениями остальной мир разработки программного обеспечения иногда кажется «низшей лигой».

Компьютеры, на которых работают встроенные приложения, могут выглядеть маленькими и незначительными, будучи скрытыми внутри CD-плейеров или детских игрушек. Другое дело программы. Цветной лазерный копир может содержать миллионы строк на C, а в лабораторном осциллографе тысячи классов, написанных на Smalltalk, ждут своей инициализации. Большинство этих приложений предъявляют высокие требования к точности, безошибочности, повторяемости и надежности. Ошибки могут проявляться самым очевидным образом. Они могут сбивать с толку и даже быть опасными, например в программном обеспечении рентгенологического аппарата, промышленного робота или автомобиля. Добавьте к этому необходимость работы в реальном масштабе времени, что характерно для многих приложений, и планка поднимается еще выше.

Несмотря на эти трудности или даже благодаря им огромное количество встроенных программ имеет впечатляющее качество. Встроенная система, созданная одной компанией для высококлассного офисного продукта, содержала более полутора миллионов строк кода. Во всем этом коде было обнаружено только четыре ошибки. Да, да, именно так, за два с лишним года существования этого продукта было выявлено только четыре ошибки. Почему и как им это удалось?

Стоимость апгрейда

На этот вопрос ответить довольно легко. Например, одна-единственная ошибка во встроенном обеспечении одного из принтеров Hewlett-Packard может помешать этой компании достичь новых успехов. Если достаточно серьезная ошибка обнаруживается после выпуска продукта, то расходы на апгрейд ПЗУ будут больше, чем прибыль, полученная за весь период существования данной серии. Это серьезный стимул не допускать ошибок с самого начала. Так они и делают. Ни одна большая программа не бывает идеальной, однако некоторые программисты встроенных систем создают огромные программы с качеством, настолько близким к безупречному, насколько это вообще возможно. Дальнейшее снижение количества ошибок просто-напросто потребовало бы слишком больших затрат.

Может быть, в этом отчасти и состоит проблема компаний вроде Microsoft и IBM: нет достаточного стимула для того, чтобы делать все правильно, особенно когда покупатели согласны мириться с нелепым программным обеспечением, содержащим ошибки. Поставщики программного обеспечения всегда могут незаметно внести исправления в выпускаемые продукты, или просто выпустить исправленную версию 6.0a по цене CD-диска, или же выложить ее в Интернете, чтобы каждый мог самостоятельно ее скачать.

Как же незаметным героям-программистам удается достичь таких результатов? Понятно, что они не всегда работают так успешно. Отрасль встроенного программного обеспечения изобилует ужасными историями о существующем коде со множественными слоями недокументированных заплат и исправлений. Модемы разрывают связь при передаче данных, в которых содержится последовательность, применявшаяся для отладки, но случайно оставленная в окончательной версии. Существуют копиры, которые внезапно теряют нить своей работы, и поэтому им приходится проводить временную «лоботомию» с помощью быстрого выдергивания шнура питания. Даже в военной ракетной радиоэлектронике случалось так, что ракеты «отключались» в середине полета, а плохо запрограммированные навигационные часы содержали накопленные ошибки. Нечего и говорить о пользовательских интерфейсах встроенных систем. Некоторые из них являются самыми отвратительными и неудобными на планете. В сравнении с тем хламом, который является нормой для персональных компьютеров, рабочих станций и универсальных машин, встроенное программное обеспечение выглядит твердым как кремень.

Безошибочность части таких программ достигается с помощью насильного внедрения личной дисциплины, которая усмиряет даже самых воинственных маньяков, помешанных на стандартах. Некоторые компании, особенно в сфере телекоммуникаций и программ военного назначения, эффективно применяют «чистые» методы программирования. Согласно этим методам, сборка кода проводится очень тщательно, а программистам запрещено возвращаться к написанному коду. Другие компании достигают таких чудес, собирая системы из огромного количества компонентов повторного использования. Некоторые из случаев наиболее сложного и строгого применения объектно-ориентированной технологии относились к созданию встроенных приложений. В основном они написаны на C++, но Smalltalk также проявил себя с лучшей стороны.

Рутина

Сложные задачи, возникающие при создании встроенных приложений реального времени, привлекают множество лучших программистов, чьи

личные и профессиональные интересы сочетаются с необычной культурой программирования. Культурный контекст встроенных приложений объединяет внимание к мелким деталям и необходимость строгой дисциплины. Программирование встроенных систем может быть утомительной работой, заставляющей программиста возиться с таким «добром», как загрузка регистров, ожидание сдвига в уровне сигнала или чередование и маскирование битов. В то же время код должен быть почти идеальным, помещаться в очень маленькой памяти и работать достаточно быстро, чтобы не вызывать паузы при выводе на экран или отклонения ракет от заданного курса.

Программисты, создающие такой код, склонны работать тщательно и скрупулезно. В первую очередь они стремятся сократить количество ошибок. С точки зрения качества программного обеспечения такие программисты имеют низкий «изначальный уровень» программных багов. Конечно, дешевле всего найти и исправить те ошибки, которым вы не позволяете проникнуть в код. Программисты встроенных систем широко применяют тактику выявления изъянов непосредственно во время рабочего цикла, вылавливая ошибки как можно раньше – пока их легко найти и просто исправить. Для сокращения количества ошибок проводятся тщательные многократные проверки расчетов и кода. Такой метод требует намного меньше затрат и является более эффективным, чем тестирование и отладка на завершающей стадии проекта.

При разработке «невстроенных» программных систем популярный метод устранения ошибок основан на бета-тестировании и отладке, проходящих на 400 000 сайтов. С точки зрения общих расходов, ложащихся на плечи покупателей, трудно представить себе более неэффективную и затратную схему. С другой стороны, для производителей программного обеспечения это выгодный подход при условии, что покупатели спешат платить за привилегию выполнять работу производителей, а у самих производителей нет стимула выпускать качественные продукты.

Отчасти успех программирования встроенных систем связан с культурой и контекстом, а также, как я убедился, с применяемым подходом и профессиональной подготовкой. Многие годы я веду мастер-классы по высокопроизводительной командной работе и обучаю методам проектирования с учетом юзабилити на проводимой два раза в год Конференции по встроенным системам (Embedded Systems Conference). Когда я высказывался по поводу того, насколько быстро команды программистов встроенных систем находили действительно хорошие решения учебных задач, почти всегда кто-нибудь объяснял это тем, что большинство из них – инженеры.

Для инженерного склада ума характерны прагматизм, склонность к решению задач и профессиональная точность, что определенно способствует хорошему качеству программирования. Мы бы все выиграли, действуя как инженеры. Мир программирования встроенных систем установил высокую планку качества, продемонстрировав, что создавать надежные и безошибочные программы действительно возможно. Будем надеяться, что остальной мир сможет ответить на этот вызов.

Из журнала *Software Development*, том 3, № 7, июль 1995 г.

56

Заметки из итальянского ресторана

Культурным сердцем Тосканы является город великого искусства и великолепной еды. Для американцев это Флоренция, для итальянцев – Firenze. Колонны часто встречаются в архитектуре Firenze. При написании этой колонки[1] я черпал вдохновение в одном ресторане. На мой взгляд, он может служить образцом самых лучших методов работы, на котором мы все можем учиться.

Ресторан Alle Murate – наверное, самый лучший ресторан Тосканы. Говоря это, я отдаю себе отчет в том, какую роль играет преувеличение в статьях о ресторанах. В один особенно незабываемый вечер, который я провел в этом заведении, я понял, что там ничего нельзя улучшить. Никакой приправы не требовалось добавлять, ни одна щепотка соли не была лишней. Даже гарнир не стоило сдвигать ни на волосок – настолько изысканной была сервировка.

Такое со мной случается редко. Подобно многим из тех, кто занимается программным обеспечением, я не могу смотреть на экран и не заметить при этом ляпы программистов или компоненты, которые следовало сделать иначе. Будучи странствующим консультантом, я также провел очень много вечеров в ресторанах, где мои собственные инстинкты шефповара всегда дают о себе знать. Когда я нахожусь в каком-нибудь ресторане, я всегда думаю о том, как бы я сам все устроил, будь он моим. Но только не в этом *ristorante*[2]. Он был совершенным и идеальным. Его нель-

---

[1] Слово column (англ.) может переводиться и как газетная или журнальная колонка, и как архитектурная колонна. – *Примеч. перев.*

[2] Ресторан (ит.). – *Примеч. науч. ред.*

зя было улучшить, не превратив во что-нибудь другое, чем он не должен быть и для чего не предназначался.

Три фактора, делающие это место совершенным, являются уроками для разработчиков программного обеспечения. Во-первых, безупречное внимание к деталям. Во-вторых, почти бесшовная командная работа. В-третьих, желание посетителя – превыше всего.

До последней детали

В программном обеспечении, где сложность и тонкость взаимодействия заявляют о себе в полный голос, внимание к деталям встречается редко. При разработке бесчисленного количества продуктов невнимательное отношение к мелким деталям приводило к провалу проектов, которые в остальных отношениях были неплохими. Например, важное диалоговое окно не сочетается с остальным интерфейсом, или неудобство регулирования параметров цвета затрудняет выбор нужного оттенка, или каждая часть документа может изменяться только пользователем, сохранившим ее. Иногда даже дизайн коммерческих продуктов кажется бессистемным. Такие промахи не являются результатом ошибочного выбора языка программирования или неверной методологии. Это результат недостатка внимания. Таких ошибок можно избежать только с помощью скрупулезного, почти маниакального внимания к деталям.

В ресторане Alle Murate внимание к деталям достигает той точки, когда оно становится едва заметным. Каждое блюдо – миниатюрное произведение искусства. Вилка, оставленная на тарелке, заменяется почти мгновенно. Каждый напиток, каждое вино имеет свой специальный бокал. Нет ни суматохи, ни вычурности – лишь внимание к каждой детали.

Работающие команды

Я часто пишу о командах и всегда с удовольствием наблюдаю за хорошей командной работой на практике – будь то игра команды на концерте или слаженность персонала в универмаге. Как раз в ресторанах я чаще замечаю отсутствие командных действий. Общепринятые порядки – например, за каждым столиком или секцией закреплен свой официант, который обслуживает только свою территорию, – даже могут быть несопоставимы с коллективными усилиями. В сфере разработки программного обеспечения территориальные претензии могут затруднять командную игру. Если какой-то сотрудник является единственным настоящим экспертом по системным архитектурам, а какая-то подсистема становится эксклюзивной и неприкосновенной собственностью одного программиста, то совместная командная работа становится проблематичной.

В том ресторане все было по-другому. Небольшое помещение, где каждый, похоже, отвечает за все. Умберто Монтана (Umberto Montana) стоит у руля, но весь персонал действует так, словно ресторан принадлежит всем им.

Как и в самых лучших технических командах, в этом ресторане есть специалисты, но специализация не мешает им сотрудничать или быть гибкими. Сониа, например, виртуозно справляется с винами, искусно вынимая пробки и с поэтичной точностью переливая вино из бутылки в графин. Крепкие тосканские красные вина должны «подышать», чтобы ими можно было насладиться в полной мере, поэтому она начинает с того, что, покрутив бутылку, ловким движением отправляет на дно бокала рубиновую каплю. Потом она добавляет еще и снова крутит бутылку, удерживая вино почти на миллиметр от края. Но если Сониа занята – например, объясняет нам тонкости рыбного блюда, – подключается кто-то другой и исполняет ее обязанности за другим столом.

Некоторые из величайших итальянских художников, в том числе и Микеланджело, были талантливыми организаторами командной работы других художников и подмастерьев. Наверное, Умберто наследовал такой талант. Он – образец «расхаживающего управляющего». Он ходит по помещению, беседует с посетителями, спокойно наблюдает за процессом и умело помогает там, где это нужно. Но в целом создается впечатление, что весь персонал следит сразу за всем и каждый обращает внимание друг друга на необходимость наполнить пустой бокал или просто наполняет его.

Такой передвижной контроль не только поддерживает внимание к деталям, но и производит впечатление расчетливого коллективного управления – внимательного, но не навязчивого. Наверное, такой метод можно применить и в разработке программного обеспечения. Нужно уделять больше внимания непрерывному совместному наблюдению за текущей работой, а не зависеть от случайного и редкого сквозного структурного контроля.

Даже посетитель становится частью этой команды. Заказ напоминает обсуждение технических требований. Какое вино вы желаете? Может быть, после этого блюда на первое вам понравится рулет из мяса ягненка *il secondo*[1]? Умберто может даже на месте придумать блюдо, которое подойдет под ваш вкус и будет сочетаться с выбранным вами вином. Он отходит к окошку небольшой кухни и говорит с шеф-поваром. Нет, она не может сделать это сегодня, но как насчет поджаренной на огне утки? Он возвращается к столу с новым предложением.

[1] На второе (ит.). – *Примеч. науч. ред.*

Когда нужно

Наверное, утонченное очарование, которое делает этот ресторан идеальным, объясняется тем, каким образом его посетителям подается не то, что они хотят, а то, что им нужно. В эпоху, когда корабль программного обеспечения, перегруженный функциями, следует по курсу списка пожеланий пользователей с 900 пунктами, нам следует с большим пониманием отличать одно от другого.

По-настоящему хорошая итальянская еда требует к себе хорошего вина – или даже более чем хорошего. Вино, которое прекрасно подходит для макарон или для блюда из рыбы, совершенно потеряет свой вкус с уткой, начиненной *spinacci*[1] и апельсиновой коркой. Для *Dolci*, десертного блюда, подойдет бокал Malvasia spumante или, возможно, Brachetto. Не все могут посчитать это необходимым для получения полного удовольствия от еды, однако отдельные посетители закажут к обеду больше одного сорта вина.

Поэтому, если только вы не против, по прибытии вам сразу же наливают бокал приятного бодрящего белого вина, что помогает вам поднять вечернее настроение и служит аккомпанементом к предстоящим *antipasti*[2]. Когда доходит очередь до *dolci*, появляется другое отличное вино.

В программном обеспечении хорошо разработанный пользовательский интерфейс не показывает пользователям все, что они когда-либо могут пожелать. Он просто показывает все то, что необходимо для текущей задачи. Хороший интерфейс является тонким компромиссом между желаниями пользователей и тем, что им действительно необходимо для работы. Именно в этом состоит разница между современными методами проектирования, ориентированными на использование (Constantine и Lockwood, 1999 [30]), и устаревающими методами типа «пользователь в центре внимания».

Меню в этом *ristorante* – сама простота. Вы выбираете из скромного списка главных блюд либо заказываете один из комплексов с фиксированной ценой и полностью полагаетесь на опыт и вкус Умберто и его персонала, которые создают для вас сюрпризы и радость.

В данном случае доверие вполне уместно. Умберто с тонкой изощренностью может отвлечь внимание посетителя от вина, которое может ему не понравиться, или от сочетаний блюд, которые могут раздражать полость рта. Как превосходный консультант, он сразу же понимает, с кем имеет дело. Когда мы просим его подсказать какое-нибудь вино, Умберто

---

[1] Шпинат (ит.). – *Примеч. науч. ред.*

[2] Закуски (ит.). – *Примеч. науч. ред.*

вспоминает, что в прошлый визит нам понравились крепкие красные вина, и предлагает чудесное «сочное» вино, полученное с небольшой фермы на севере.

Это совершенство, конечно, не для всех. Однажды рядом с нами сидела американская пара из тех, какие обычно живут в богатом районе любого большого города. Хорошая пара: она решила быть не в настроении, а он решил командовать. Он отказался от помощи Умберто в выборе вина из тысяч бутылок, стоящих в погребе. Вместо этого мужчина грубо попросил карту вин. Они отказались от помощи в выборе блюд из меню. По выражению их лиц во время еды и перед уходом можно было понять, что они достигли своих целей покомандовать и остаться не в настроении.

Умберто говорит, что встречает подобные пары приблизительно раз в неделю, но ни одна из них не появляется во второй раз. Что касается нас, то мы не можем дождаться, когда в следующий раз окажемся в Firenze.

Из журнала *Software Development*, том 3, № 8, август 1995 г.

57

Наставничество

Наставничество. В постмодернистском языковом смешении, где каждому существительному грозит превращение в глагол, а каждому слову, обозначающему действие, – номинализация, глагол «наставлять» означает довольно интересную деятельность. Наставник – это, конечно, учитель. Но это не просто учитель, а еще и руководитель, мастер – отчасти консультант, отчасти тренер, отчасти коллега. В некоторых областях разработки программного обеспечения тренерская помощь стала почти столь же популярной, как и наставничество. «Тренинг» – даже более подходящее слово, которое произносить намного легче. Попробуйте прислушаться к разговору консультантов на какой-либо конференции. Скорее всего, вы услышите, что они больше не занимаются консультированием. Вместо этого они обеспечивают «наставнический и тренировочный процесс». Вот это да!

По словам известного эксперта Эда Йордона (Ed Yourdon), наставничество стало признаком культуры компании Microsoft и краеугольным камнем ее подхода к улучшению программного обеспечения. В соответствии с этим подходом за новыми работниками закрепляется персональный наставник, который проверяет каждую строчку их кода. А «наставляемый» читает каждую строку кода, написанного наставником. Это, как нам говорят, является живым подтверждением стремления к качеству и совершенствованию рабочего процесса на большом ранчо в Редмонде.

Нелли знала

Не желая ни славить, ни хоронить Цезаря, я воздержусь от общераспространенного бичевания Microsoft и вместо этого обращу больше внима-

ния на то, чем является и чем не является наставничество. Во-первых, оно не ново. Несмотря на неологическое одеяние, представляющее наставничество как часть общего движения к совершенствованию качества и улучшению рабочего процесса, это старая идея. Наставничество – это модель «мастер-ученик» без узаконенного рабского служения. Это «переодетая» версия метода «сядь рядом с Нелли», который ранее применялся в разработке программного обеспечения. Эд Йордон и я даже писали об этой простой методике в первом издании «Structured Design» (Структурное проектирование) (Yourdon и Constantine, 1974 [70]). Новому программисту говорят: «Сядь рядом с Нелли. Она знает, как программировать». Как мы объяснили впоследствии, это симптом проблемы. Так как никто точно не знает, как Нелли удается хорошо программировать, и даже она сама не может толком это объяснить, то самый лучший способ поучиться у нее – это сесть рядом и смотреть, как она пишет код.

Ученичество – это модель обучения, которая более подходит для овладения ремеслом программирования, чем для поддержания дисциплины при разработке программ. Дисциплина требует наличия людей, которые умеют делать это; людей, которые знают, что делают те, которые умеют это делать; и людей, которые знают, как можно научить других тому, как делать то, что делают те, которые умеют делать это хорошо. Наставление – это то, к чему вы прибегаете, когда либо не понимаете свои действия, либо не можете выразить их в форме, в которой этому можно легко и надежно научить.

Естественно, что при создании программ исполнители, мыслители и учителя – не всегда одни и те же люди. Некоторые из наших лучших разработчиков и даже кое-кто из наших великих гуру имеют только смутные или неверные представления о том, что нужно для того, чтобы делать то, что делают они. С другой стороны, прекрасные учителя могут быть посредственными программистами. Обучение и программирование – это совершенно разные умения.

К этому времени Нелли уже стала прабабушкой, а все попытки программировать, которые она совершает, заключаются в том, чтобы записать на видеомагнитофон все четыре выпуска последних новостей на PBS. Тем не менее в компьютерной отрасли есть еще очень много программистов и разработчиков, которые хорошо делают свою работу, но даже под угрозой увольнения не могут объяснить, как. И чем ближе вы подходите к линии фронта, переднему краю методов, тем больше в этом убеждаетесь. На одной конференции в Сиднее (Австралия), посвященной объектно-ориентированному проектированию, наставничество упоминалось на каждом углу. На презентациях и в комментариях из зала о нем говорили как о золотом пути к мастерскому владению объектной технологией. Наверное, не случайно о наставничестве больше всего говорят тогда, когда речь захо-

дит о новейших технологиях, а надежная опора почти не видна. Теория, методики и инструменты больше всего отстают от задач и существующей практики в тех областях, где внедряются передовые технологии. Там, где могут быть драконы, нужны хорошие наставники.

У меня были самые лучшие наставники. Удивительно, что нашлись умные тренеры, которые оказались достаточно смелыми и готовыми к тому, чтобы взять на обучение такого беспокойного молодого человека. Результаты тех лет все еще полезны. Я до сих пор стараюсь найти новые повороты в методиках, о которых я впервые узнал от Джека Креминса (Jack Cremeans), Кена Макензи (Ken Mackenzie), Дэйва Джаспера (Dave Jasper) и Бада Вайтофа (Bud Vitoff).

Жесткий допуск

Однако в той степени, в какой наставничество вообще работает, – а работает оно не всегда, – оно не может непосредственно привести к улучшению качества. Наставничество лишь помогает распространить стиль и практические навыки наставника. В качестве организационной стратегии оно может быть эффективным способом поддержания корпоративных традиций, при условии, что – хорошо это или плохо – люди думают и программируют в соответствии с принятой культурой.

Это соответствие само по себе может стать первым небольшим шагом к повышению качества процесса проектирования. Согласованность – бабушка качества. Аксиома совершенствования рабочего процесса такова: перед началом улучшения процесса необходимо организовать его контроль – сделать его воспроизводимым. Пока процесс разработки программного обеспечения остается хаотичным, усилия – нерегулярными, а вы не знаете, будет ли продукт сдан в срок и в каком он будет виде, никакого систематичного совершенствования и последовательного улучшения не добиться. Надежность конечного продукта остается, главным образом, в руках Судьбы и легионов бета- и гамма-тестеров. Согласно Модели развития функциональных возможностей (Capability Maturity Model), представленной Институтом разработки программного обеспечения (Software Engineering Institute), первая ступень на лестнице качества, которая ведет к великим высотам 5-го уровня (другими словами, к работоспособному программному обеспечению), подразумевает уменьшение различий, то есть получение непротиворечивых результатов в ходе предсказуемых процессов. Конечно, это означает, что блестящие и необъяснимые успехи, равно как и неожиданные провалы, не будут распространяться. Таким образом, разработка будет вестись согласно «золотой середины» – в более стабильной, но менее эффектной среде.

Если наставничество соответствует организационной культуре, то инвестировать в признанную и систематичную программу наставничества более полезно, чем доверяться природным инстинктам неподготовленных тренеров. В принципе, каждому нужен наставник, и каждый, в свою очередь, должен попробовать себя в этой роли. Процесс наставления не должен прерываться просто потому, что вы сдали свой годовой отчет без сучка, без задоринки. Наставники могут помочь вам подняться на следующую ступень – так же как и на первую. Наставничество может служить частью эффективной стратегии распространения технологии, когда внедряются новые методы и языки. В программе по повышению качества работы тренировка может дополнять как анализ кода, так и сквозной проход дизайна.

Наставничество особенно популярно в консалтинговых фирмах, которые навязывают его в качестве эффективного средства для осуществления технологических переходов. Вместо консультирования они будут работать вместе с вами на протяжении всего проекта и мягко помогать вам достичь их уровня мастерства. По сравнению с интенсивными предварительными курсами для персонала или нерегулярными консультациями по мере возникновения проблем модель наставничества имеет очевидные преимущества, не последним из которых является более стабильный заработок консультанта.

Итак, слово «наставничество» является одним из ключевых – по крайней мере, сегодня. Однако язык продолжает преображаться прямо на глазах. Язык должен двигаться дальше и изменяться, иначе мы быстро к нему привыкнем или, что еще хуже, будем ограничены в своих действиях. Умные консультанты просто-напросто умеют называть все по-другому. Деньги на консультации и обучение можно выжать из бюджетов, но еще не все охотники за наживой открыли для себя тренировку и наставничество. Когда только это произойдет, появятся денежные ограничения на контракты по организации тренировок, а наставничество «окаменеет» на 13 страницах контрактных обязательств, компенсаций и договоренностей о неразглашении.

Возможно, так и было задумано!

Из журнала *Software Development*, том 3, № 3, март 1995 г.

58

На обучение

Темой обсуждения было ближайшее будущее, передовой край в разработке программного обеспечения и приложений. Вопрос из зала – словно мучительное стенание из прошлого: «А что вы посоветуете аналитику и программисту, который на языке COBOL создает текстовые приложения, обращающиеся к базам данных сетевых мэйнфреймов?» После целых семи или восьми миллисекунд глубокого раздумья я наклонился к микрофону и с заговорщической интонацией и в то же время командным голосом произнес: «Переобучайтесь».

Неудовлетворенный моим ответом, человек подошел ко мне в перерыве. Что же ему делать? Его компания не поддерживает дальнейшее образование, расходы на обучение в бюджете не предусмотрены, книги и курсы слишком дороги для его скромного заработка, а COBOL – это все, что он знает.

Отраслевые стилисты уже не один год объявляют о том, что мэйнфреймы ушли в небытие, однако в тот момент, когда следователь уже собирается подписывать свидетельство о смерти, эти чертовы машины опять поднимают голову. Не оплакивайте старый добрый COBOL. Наверное, на планете наберется больше строк на COBOL, чем на всех других языках программирования, вместе взятых. (Этот досадный факт подтвердился, когда возникла «проблема 2000 года», напомнившая о старом пенсионере COBOL.) Как и все живые языки, COBOL продолжает развиваться. Теперь он стал объектно-ориентированным и когда-нибудь может получить настоящую поддержку.

Как и рок-н-ролл, COBOL будет жить вечно. Конечно, «вечность» – это относительное понятие в отрасли, где языки пятого поколения следуют за языками четвертого поколения уже через несколько лет после языков третьего поколения. Тем не менее языки программирования подчиняются Четвертому закону Константина: ни один язык программирования, на котором пишут значительное количество разработчиков, не может исчезнуть полностью. Изучать программирование я начал с Фортрана – одного из самых первых «высокоуровневых» языков. Фортран до сих пор применяется, до сих пор поддерживается своими партизанами, до сих пор жизнеспособен. Еще более удивительно то, что язык RPG, уже седой к моменту моей первой встречи с ним в начале 60-х, в середине 90-х все еще оставался в ходу.

Мой твердый совет этому несчастному программисту был обусловлен не пессимизмом или оптимизмом в отношении близкой кончины больших ЭВМ и «больших» языков, а реализмом относительно происходящего в этой профессии. Вопрос не в том, сможет ли этот кодирующий консерватор сохранить свою работу, а в том, что он будет делать в мире возникающих и исчезающих возможностей. В 80-х годах мой шурин вовремя оставил COBOL и мэйнфреймы, но это было тогда, а этот программист все еще стоял и спрашивал меня, как быть дальше.

Пластичность

И я ответил ему. На манер Дастина Хоффмана в сцене из фильма «Выпускник« («The Graduate») я прошептал: «Объекты». Если вы не занимаетесь объектами, тогда займитесь ими. Иначе вы окажетесь среди тех, кто смотрит на проходящий через пески поток, который оставляет после себя высыхающую лужу возможностей. Я также говорил ему о визуальном проектировании. Я даже предложил изучить что-нибудь из программирования встроенных систем – я могу создавать трудности.

Не думаю, что он действительно услышал мои слова. Он не желал тратить деньги на свое будущее, и он совсем не хотел учиться новому. Он надеялся услышать от меня что-то обнадеживающее – может быть, какое-нибудь тайное заклинание, которое поможет остановить прилив новых тенденций, который уже подбирался к его ногам, или получить от меня какую-нибудь шлюпку, чтобы можно было немного продержаться на плаву.

Трудно сказать, что делать. Ветры меняют свое направление, а прибой может унести вас в море. Вы можете заняться Java, изучить ActiveX или освоить межплатформенную библиотеку ГПИ, и все же вы будете смыты волной, когда производитель вашего любимого инструмента «отдаст концы», или продукт перестанет поддерживаться из-за поглощения компании, или Билл и Компания опять передумают и решат все сделать по-другому.

Нет никаких гарантий. Тем не менее некоторые суда более пригодны для плавания, чем другие, и вы можете разумно управлять движением своей карьеры. Как мне кажется, здесь есть два главных условия. Во-первых, нужно построить как можно более крепкий корабль, а во-вторых – никогда не убирать рук со штурвала.

Постоянное совершенствование

Книги, курсы, конференции и новое программное обеспечение – это именно то, что позволяет профессионалам легко плавать по бурным водам. Если образовательный бюджет компании исчерпан в результате сокращения расходов, изменения рабочего процесса или из-за недостатка финансирования, то вам, наверное, следует восполнить этот бюджет из собственного кармана. Это инвестирование в себя и в свое будущее. Вы не можете лишать себя такой возможности. Это уменьшает риск остаться не у дел в случае, если коса сокращений заденет и вас. В любом случае это увеличивает ваши шансы на открытом рынке труда или может помочь организовать свое собственное предприятие dot.com.

Проблема в том, что многие организации и профессионалы думают, что образование – это то, что бывает один раз, когда мы еще молоды, а обучение – это форма дополнительного тренинга, который потом можно проходить от случая к случаю. В действительности современные общие требования организационного обучения и непрерывного совершенствования рабочего процесса относятся не только к людям, работающим в организациях, но и к самим организациям. Настойчивые профессионалы всегда чему-то учатся, непрерывно повышают свою квалификацию и улучшают свое профессиональное мастерство.

Поэтому я сказал тому «инквизитору», что да, это хорошо, когда компания оплачивает подписку на журналы и покупает билеты на две конференции в год, но, в конце концов, кому это больше нужно? Вместо того чтобы хныкать о том, что профессиональный корабль уходит без него, я предложил ему потратить деньги на обучение искусству мореплавания, пока у него еще есть хоть какие-то деньги.

Самое интересное в обучении заключается в том, что чем больше вы учитесь, тем легче это становится. Учеба становится привычкой. Если этим заниматься достаточно долго, то можно даже научиться тому, как учиться. Свой первый «второй язык» после родного изучать обычно намного труднее, чем все последующие. То же самое касается и языков программирования. Почему некоторые *не* хотят изучать новый язык? Ведь вскоре становится ясно, что все языки являются вариациями всего на несколько тем. Вместо того чтобы учиться «программировать на языке Z», можно научиться программировать.

Это приводит нас к другой стороне более надежного будущего. Стоит изучать то, что является наиболее общим.

В теории

В Бостоне существует огромное количество колледжей и университетов, в которых предлагаются разнообразные подходы к образованию. Когда я пошел в колледж, в инженерной области все равнялись на Массачусетский технологический институт и Северо-восточный университет – две школы, которые так же отличаются друг от друга, как Smalltalk и COBOL. В районе Бэк-Бэй[1] располагался Северо-восточный университет, в котором упор делался на практику и на приобретение рабочего опыта в процессе совместного обучения. На кембриджской стороне реки[2] находился Тех, где правила наука, а акцент ставился на теорию, а не на практику – на фундаментальные принципы инжиниринга. Ходили слухи, что инженеры, прошедшие подготовку в Северо-восточном, могли эффективно работать уже в день своего выпуска, а выпускники Теха могли пять лет отрабатывать свое обучение. С другой стороны, еще через пять или десять лет бывшие студенты Северо-восточного сталкивались с устареванием своих знаний, а выпускники Теха по-прежнему были на высоте.

Теория не является бедной служанкой практики – она превосходит практику. Теория позволяет заглянуть в будущее. Когда Эйнштейн разработал формулу $E = mc^2$, ядерных реакторов еще не было. Существование планеты Плутон было предсказано еще до того, как ее смогли увидеть; теоретически она должна была существовать, судя по параметрам орбиты Нептуна. В менее космическом масштабе: теория модульной сложности, на которой основано структурное проектирование, предсказала экспериментальные результаты, появившиеся спустя несколько лет. Кроме того, эта теория предвосхитила ключевые понятия объектно-ориентированного программирования, хотя при возникновении понятий связывания и сцепления методология ООП еще не существовала даже как идея.

Надежность любой работы является иллюзорной, но самым лучшим страховым полисом является крепкий фундамент того, что вы делаете. Слово *фундамент* означает «основы». Общие принципы являются более стабильными, чем те виды конкретной деятельности, которые на них основаны. Подобно гибкому коду, допускающему многократное использование, базовые принципы остаются действующими в изменяющихся контекстах. Если ваши ценности основаны на знании какого-то конкретного

---

[1] Бэк-Бэй (Back Bay) – район г. Бостон. – *Примеч. ред.*

[2] Кембридж (Cambridge) – пригород Бостона на реке Чарльз (Charles River). – *Примеч. ред.*

языка, или платформы, или среды программирования, то ваш карьерный фундамент стоит на песке, а волны технологии, несомненно, вынесут его из-под ваших ног.

Какие бы будущие волны технологии не влияли на разработку прикладных программ и программного обеспечения, вы можете наверняка знать только одно. Если вы будете просто продолжать делать то, что делаете сейчас, вам, вероятно, не придется делать это слишком долго.

Из журнала *Software Development*, том 3, № 4, апрель 1995 г.

59

Одаренные программисты

Это был долгий год. Месяц за месяцем вы, как и другие программисты из вашей команды, тяжело работали, выполняя обязательства по проекту. Вы начинаете думать о премии в конце года. Это сезон подарков – зимнее солнцестояние, Ханука, Рождество, Кванза и обыкновенный Новый год – праздники идут один за другим. Что вы подарите вашим высокопроизводительным программистам, чтобы они поняли, как вы их цените? И вообще – как вы оцениваете, награждаете или стимулируете разработчиков приложений и программного обеспечения? По существу, именно в этом заключался вопрос, поднятый аудиторией на одном из лекционных туров, который я совершал по Австралии вместе с такими индустриальными светилами, как Роб Томсет и Эд Йордон. Это заставило всех нас задуматься о более творческих и эффективных методах поощрения разработчиков.

Это вечная проблема. Когда речь идет о стимулах, многие из нас склонны думать упрощенно. Несколько лет назад я работал с русскими и украинскими менеджерами из одной недавно приватизированной фирмы, которая раньше принадлежала советскому государству. В какой-то момент я понял, что вместе с ними оказался вовлечен в одну неувядаемую управленческую игру под названием «Почему бы вам не... – Да, но...». Я предлагал способы более эффективной работы, а они отвечали: «Да, но у нас нет надежных источников снабжения», «Да, но наши работники не готовы взять инициативу в свои руки», «Да, но мы не можем уволить работников и не можем изменить их зарплату. Мы не можем стимулировать сотрудников». Сначала эти блестящие и «мотивированные» менеджеры советской закалки не находили мотивацию для своих работников, но ког-

да я предложил «мозговой штурм», они придумали десятки способов и средств мотивации, не прибегая к сокращению штатов или денежным поощрениям. Естественно, мы также способны найти эти возможности.

Игрушки для технарей

Многие из нас, чокнутых, любят технологические игрушки. Тест на родство со славным гранфаллоном довольно прост. Загораются ли ваши глаза при мысли о новом замечательном устройстве в ГПИ, или о 3D-видеокарте к 21-дюймовому плоскоэкранному монитору, или о гигагерцовом лэптопе с диском в 20 Гбайт? Вы один из тех людей, которые не могут дождаться выхода бета-версии нового продукта? Вы раздираете целлофановую обертку для того, чтобы узнать, решены ли в релизе 2.0 проблемы релиза 1.1?

Одним из вознаграждений за поставку программного обеспечения вовремя может быть первоочередное предоставление стенда для проведения бета-тестирования. Первые копии нового CASE-инструмента или недавно приобретенного пакета для проектирования можно вручать тем членам команды, которые работали лучше всех в прошедшем квартале. Команды, справляющиеся с обязательствами, могут первыми вытягивать фишки при очередном апгрейде оборудования.

Конечно, такая тактика работает не всегда. Для некоторых из нас установка бета-версии программного обеспечения – это нечто среднее между плаванием в заплесневелой овсяной каше и заменой спутниковой антенны на крыше во время грозы. За последние несколько лет я сам умудрился принять участие лишь в одном бета-тестировании. Возможно, в качестве более подходящего варианта можно предложить меню, представляющее программные или аппаратные «игрушки». Лучшие работники выбирают в первую очередь. Еще одним вариантом может быть право первого голоса в выборе инструментов, языков и библиотек, которые следует приобрести в следующий раз.

Говоря об игрушках – доводилось ли вам наблюдать помешательство, которое возникает, когда на конференции разработчиков какой-нибудь докладчик выходит с крутыми подарками? Никогда не преуменьшайте значение фирменных футболок. Целый ряд рекламных диковинок – командные пиджаки, специальные галстуки, кружки или коврики для мыши, выпущенные ограниченным тиражом, – можно применять для награждения успешных команд и программистов, чтобы отметить их особым образом. Лучшая команда может получить право разработать и выпустить за счет компании собственные знаки отличия.

Рабочий отпуск

Я должен похвалить Роба Томсета за один из самых творческих способов поощрения лучших и блестящих программистов, позволяющий получить неожиданную обратную отдачу. Он предлагает вознаграждать успешность временем. Работники, которые создают качественное программное обеспечение и представляют его в срок, получают время на участие в любых интересующих их проектах. Какой программист откажется от шанса получить несколько месяцев на изучение нового языка, или экспериментирование с методами сжатия изображений, или разработку нового текстонезависимого метода поиска? Проект может быть каким угодно, и за участие в нем программист получает зарплату! Что действительно вызывает интерес у большинства из нас, технарей, – так это возможность изучать новое, испытывать его, играться с новыми инструментами и методами.

Томсет говорит, что, судя по его опыту, большинство программистов все же хотели бы иметь больше времени, чем денег. Для компании отдача от финансируемых исследований такого рода – это не только счастливый разработчик с новыми навыками и идеями, но и новые программы или какая-нибудь новая ценная технология. Возможно, такой подход к поощрению больше подходит для целых команд. Если проектная команда демонстрирует способность превышать установленные достижения, наградите ее, позволив ее членам участвовать в любом исследовательском проекте или программной разработке на свой выбор.

Если для ваших программистов время действительно значит больше, чем деньги, вы можете предложить им в качестве награды дополнительные выходные. В более масштабной и долгосрочной перспективе интересна традиция, принятая у австралийцев и известная под названием «отпуск за долгую службу». Если вы проработали в какой-то организации или компании десять лет, вы можете получить длительный отпуск (обычно 8–12 недель) с полной оплатой. В отрасли, где лояльность не распространена, а текучесть является серьезной проблемой, довольно разумно применять стимулы, позволяющие удержать хороших людей, которых вам было так трудно найти.

Волнующий тренинг

Быть в курсе событий, происходящих в области разработки программного обеспечения, довольно трудно. С другой стороны, это очень забавно – работать в такой быстро меняющейся технической сфере деятельности. Она никогда не бывает скучной. Посещение дополнительного практического семинара или билет на участие в очередной Конференции по про-

граммированию встроенных систем (Embedded Systems Programming) или по разработке программного обеспечения может быть эффективным способом признания усердной работы. Книги и журнальные подписки – еще один недорогой метод поощрения.

Если группа научилась работать вместе, зачем ее разбивать? Наградой за высокопроизводительную командную работу может стать право продолжить совместную деятельность в следующем проекте. Другими словами, возможность подбирать коллег становится эффективным стимулом для повышения производительности. Аналогично свободный выбор следующего проекта также может быть наградой за хорошую работу в предыдущем.

А как насчет улучшения условий труда, ремонта офиса или перестановок, чтобы людям было легче работать в уединении или проводить встречи? Возможности ограничены только вашей фантазией и вашим желанием отступить от условностей традиционного мышления руководителей. Например, в одной компании мотивацией сотрудников служит доступ к внутренней информации. В ней применяется открытый управленческий подход, который позволяет каждому быть в курсе финансовых и производственных новостей, поэтому любой сотрудник может увидеть собственный вклад в работу компании.

Конечно, при разработке схем вознаграждений и поощрений нельзя не учитывать особенности людей. Один человек будет рад получить самую красивую кофейную кружку с золотой надписью, а другой будет обижен таким подарком, потому что вместо кружки этот измотанный аналитик ожидал получить дополнительный выходной, ведь он и так уже чуть ли не живет в офисе.

Вы можете даже спросить у своих «ударников труда», что они хотят получить. Таким образом вы можете получить совершенно новую идею!

Из журнала *Software Development*, том 3, № 12, декабрь 1995 г.

60

Иконы отрасли

Конференц-центр – одно из интереснейших мест в Амстердаме – представлял собой перестроенную церковь: скамьи заменили на стулья, как в театре, а все помещение со сводчатым потолком превратилось в аудиторию, которую заполнили самым современным аудиовизуальным оборудованием. Выступая с докладом на ежегодном собрании европейской группы пользователей CASE-инструментов, я не мог удержаться от того, чтобы не отметить внушающую благоговение обстановку. Заметив слева украшенную резьбой кафедру, которая возвышалась над сценой, я задумался, каково это – обращаться к группе с такой величественной высоты. Возникшая мысль вызвала желание пошутить, и я сказал аудитории, что на самом деле я обращался к руководству за разрешением прочитать свой доклад с той кафедры, но безрезультатно.

Все упали со смеху, когда я сказал, что получил отказ, – только Джеймсу Мартину[1] разрешили говорить оттуда.

Наша отрасль – это мир высоких технологий и твердолобых деловых людей, принимающих важнейшие корпоративные решения. Среди нас есть инженеры, ученые, аналитики, программисты. Мы тщательно и разумно анализируем продукты и процессы, а затем на основе их достоинств и недостатков делаем свой выбор. Однако под внешним налетом четкого здравомыслия и объективных данных лежит другой мир, в котором господствует культ личностей. Все дело в именах – в гуру, их последователях и со-

[1] Джеймс Мартин (James Martin) считается отцом CASE-технологии. – *Примеч. науч. ред.*

ревнующихся лагерях, которые выступают под знаменами истинных верующих. Ура! Ура! Объекты выигрывают у функций 4:0 – подробности в следующем выпуске новостей.

Имена и числа

У нас столько известных имен – от Коуда (Coad) и Йордона, Кодда (Codd) и Дэйта (Date) до Комафорда (Comaford), Крингли (Cringely) и Куртиса (Curtis); от Майерса (Myers) до Мейера (Meyer), Варда/Мелора (Ward/Mellor), Вассермана (Wasserman) и Вайнберга (Weinberg). Все – начиная от новичков и восходящих звезд и заканчивая пантеоном, теми богами на вершине, чьи имена признаются каждым, кто пробыл в этой отрасли больше недели, – понимают, что это бизнес гуру и личностей настолько же, насколько бизнес технологий и чипов. Есть «глобальные» гуру и есть те, чья репутация связана с какими-то конкретными областями. Алан Грайвер (Y. Alan Griver) может легко заставить все сообщество Visual FoxPro слушать себя с раскрытым ртом, но среди мастеров по кодированию встроенных систем его речь вызовет только вопросы «Кто? Что-что?».

Большие звезды приобретают лояльных последователей и создают продукты, которые называют их именами. Уже есть Booch-граммы и Chen-нотации. При создании объектных моделей одни аналитики следовали Коуду/Йордону, а другие – Шлеру/Мелору (Shlaer/Mellor). В схемах потоков данных применялись либо кружки Йордона/ДеМарко (DeMarko), либо прямоугольники со скругленными углами, введенные Гейном/Сарсоном (Gane/Sarson). Было время, когда бизнес-аналитики могли из-за этого подраться. Может быть, вы все еще верите в системное проектирование по Джексону (Jackson). А может быть, вы все еще нормализуете свои определения до формы Бэкуса-Наура (Backus-Naur).

У нас есть множество звезд, а может быть, даже наберется с десяток суперзвезд, но в созвездии компьютерных светил Джеймс Мартин – самая яркая величина. По слухам, он зарабатывает $25 000 в день. Что же нужно для того, чтобы достигнуть таких высот?

По мнению австралийского журналиста и обозревателя Грема Филипсона (Graeme Philipson), секрет заключается в особом сочетании стиля и содержания. Просто хорошее знание своего предмета и способность сказать нечто ценное не даст вам продвинуться дальше чтения лекций. Яркость и талант сами по себе работают лучше, по крайней мере хоть какое-то время, однако рано или поздно пустого болтуна раскусывают, и ему приходится либо кричать громче, либо прикусить язык.

Наличие характерной «изюминки» является необходимой частью игры. У Мартина есть смокинг, свое мультимедийное шоу и его безупречная,

учтивая английская манера речи. У Пита Коуда есть гавайские рубашки и пластиковые гудки. Среди любимчиков Филипсона есть эксперт и аналитик Джеф Тэш (Jeff Tash), которого он описывает как «резкого и самоуверенного оратора, чей стиль – кричать на свою аудиторию». Как видите, есть немало способов произвести впечатление. Я сам известен своей ковбойской шляпой, которую я ношу на конференциях в качестве напоминания о своих тирадах по поводу кодирующих ковбоев (см. часть II).

Не работая

По общему признанию, работа гуру программирования довольно приятна. На одной из бостонских Конференций по программным методам (Software Methods Conference) Меилир Пейдж-Джонс (Meilir Page-Jones), который сам считается гуру, закончил презентацию музыкальной пародией на хит Dire Straits «Money for Nothing». Вариант Меилира – это жалобная песнь работающего программиста, который изо дня в день лепит программы, а затем ему еще приходится высиживать на передвижных представлениях, устраиваемых «подиумными» демонстраторами. «Так не получится: деньги просто так и бесплатные путешествия».

На самом деле звезды в сфере программного обеспечения имеют двойственную природу. Немногие люди остаются «на сцене» так же долго, как Мартин и Йордон. Отчасти секрет выживания состоит в том, чтобы, как напоминает нам Меилир, не принимать все это всерьез. Полезно даже быть немного несведущим. Я понятия не имел о том, что сам могу быть гуру, до своей поездки в Бразилию с моей младшей дочерью в 1988 году. Наш хозяин, профессор компьютерных наук в одном из крупных университетов Рио, спросил Хивер, каково быть дочерью знаменитого отца. Она не поняла, о чем он говорил. Так же как и я.

Я вспомнил об этом, когда коллега по Сиднейскому технологическому университету представил меня как «икону отрасли». Я рассмеялся. У меня очень живое воображение, и я мгновенно представил панель инструментов, заполненную кнопками – одна из них содержала иконку[1] с изображением моего лица. Нажимаете кнопку с Йордоном и получаете предупреждение об офшорных программистах или ошибках 2000 г. Нажимаете кнопку с Гради, Айваром или Джимом – и на экране все унифицируется. Моя иконка может запускать хранитель экрана с кодирующими ковбоями, которые скачут галопом по монитору.

---

[1] Слово icon (англ.) может переводиться и как икона, и как иконка (пиктограмма). – *Примеч. перев.*

Если вам хочется видеть свое изображение на панели инструментов, то ваша дорога к звездам открыта. Просто следуйте за лидерами. Понаблюдайте за поступью вашего любимого гуру и придумайте свой вариант.

Некоторые шаги являются обязательными. Неважно, сколько систем вы построили, или сколько статей написали, или сколько провели семинаров. Никто не будет всерьез воспринимать вас как гуру до тех пор, пока вы не выпустите книгу или не станете регулярно печататься в какой-нибудь известной отраслевой газетенке. Желательно, чтобы эти заметки сопровождались вашей фотографией. Ваше имя на обложке книги должно идти первым среди соавторов. Все помнят, что метод OMT был разработан Рамбо (Rumbaugh), и каждый считает пользовательские ситуации неотъемлемой частью метода Джекобсона, но кто сейчас вспомнит имена других авторов, участвовавших в создании авторитетных трудов этих гуру?

Дары

Самый устойчивый и почитаемый гуру имеет нечто большее, чем стиль, – нечто, чему, вероятно, невозможно научить или научиться. Это харизма. Часто и неверно употребляемое слово «харизма» буквально означает дар богов. Этот дар наделяет некоторых людей способностью побуждать других идти на риск – создавать новое программное обеспечение, или стремиться к 5-му уровню в SEI, или заниматься неиспытанным языком программирования.

По-настоящему харизматические личности – те страстные истинные верующие, которые побуждают приверженцев следовать за собой в горы, в тюрьмы или в места похуже, – наверное, люди другого рода. Лен Оукс (Len Oakes), психолог из Мельбурна, изучающий психологию харизмы, утверждает, что, по сути, настоящие харизматичные гуру очень похожи друг на друга. Основополагающие идеи могут быть совершенно различными, но личности обладают сходным, чрезвычайно твердым центром. Кореш (Koresh), Жюре (Jouret), Раджниш (Rajneesh), Хаббард (Hubbard) – у харизматических лидеров подобного типа кроме всего прочего есть один общий изъян: непроницаемый эгоистичный взгляд на реальность, который может вызывать у других ощущение абсолютной, окончательной, непоколебимой достоверности. В мире, где преобладает двойственность и неоднозначность, достоверность может служить действенным эликсиром. Но, увы, достоверность такого рода основана на выдуманной реальности, которая не связана с реальным миром. Она поддерживается за счет непрерывных усилий по отгораживанию от новых идей, новых сведений и новых аргументов, которые могут поставить под сомнение абсолютную истинность личных представлений.

Под это описание могут подойти некоторые из ярых защитников тех или иных инструментов или методов разработки, однако у большинства истинных харизматических лидеров мало шансов в такой быстро меняющейся и динамичной индустрии, как наша. Чаще случается так, что истинные верующие остаются на улице и проповедуют в пустой аудитории. Хотя, с другой стороны, может быть, быстрые изменения как раз и делают нас столь восприимчивыми ко всякому спасителю, продающему серебряные пули, при помощи которых можно мгновенно создавать приложения без единой ошибки. Подходите, ребята. К каждому волшебному флакону полагается бесплатное программное обеспечение!

И в каждом змеиный жир?

Из журнала *Software Development*, том 3, № 2, февраль 1995 г.

61

Импресарио

Наш бизнес довольно большой. Даже тем, кто работает в маленьких компаниях, может показаться, что один человек вряд ли способен что-то изменить, а тем более существенно повлиять на ход событий. Я регулярно получаю сообщения от людей, которые жалуются, что не могут оказывать влияние, а их тихий голос – лишь глас вопиющего в пустыне программирования. Они не питают большой надежды на то, что в практике проектирования программного обеспечения что-то улучшится. Некоторые из них являются программистами и разработчиками программного обеспечения, желающими действительно что-то изменить, а не просто создавать больше кода. Другие – руководители, которые благоговеют перед своим техническим персоналом и чувствуют, что вносят меньший вклад по сравнению с работающим у них гением, способным редактировать программу на C в 15 разных окнах одновременно.

Влияние и воздействие может принимать разные формы. Целая отрасль или профессия может измениться благодаря действиям или вкладу одного человека. Эдсгер Дейкстра (Edsger Dijkstra) применил теоретическую работу Бома (Bohm) и Джакопини (Jacopini) к стилю программирования. Своим историческим письмом к редактору под названием «GO TO Statement Considered Harmful» (Оператор GO TO считаю вредным) (Dijkstra, 1968 [34]) он вызвал спор, приведший к революции в практике программирования. Молодой Билл Гейтс совершил несколько правильных шагов в создании системного программного обеспечения для зарождающейся микрокомпьютерной индустрии, и наш бизнес радикально изменился. Докторская диссертация Алана Кэя (Alan Kaye) стала основой революционного языка, который помог превратить объекты в новую программную

парадигму. В сущности, в основе современного компьютерного программирования лежит совсем немного базовых идей, которые были выдвинуты сравнительно небольшой группой новаторов.

Однако даже самые известные новаторы и создатели, самые заметные инициаторы и зачинатели, самые активные авторы и консультанты не всегда оказывают настоящее влияние. Иногда другие, менее известные и менее почитаемые личности могут оставлять глубокий след. Возьмем, например, импресарио.

Влияние на ход событий

Английский – жадный язык, страстно заимствующий слова почти отовсюду. Итальянское слово *impresario* означает «антрепренер». В свою очередь, оно заимствовано из французского языка, где это слово означает человека, который организует и координирует работу предприятия. Однако в английском языке слово «импресарио» часто ассоциируется с надзором за цирковыми артистами или обозначает хозяина театра. Импресарио руководит не отдельной группой, а туром. Он не исполнитель, а непосредственный начальник, не инженер, а директор лаборатории. Тем не менее значение и важность импресарио очень часто недооцениваются.

Технические специалисты, посещающие Австралию, зачастую поражаются уровню дисциплинированности и искушенности многих групп, разрабатывающих программные приложения. Это не значит, что каждый имеет первый ранг, а все программное обеспечение, как говорят австралийцы, «попадает в цель». Это означает лишь то, что эффективное применение инструментов и системных методов здесь широко распространено.

Все лучшие методы работы уходят корнями в 70-е годы, когда ведущие разработчики программных методов стали посещать Австралию. Поскольку эта страна так мала, а область информационных технологий – еще меньше, то на удивление много австралийских профессионалов-программистов прошли подготовку непосредственно у этих первопроходцев, которые были создателями и основными защитниками современных методов и подходов. Многие из тех студентов, которые рано испытали на себе влияние Константина, ДеМарко (DeMarco), Вайнберга (Weinberg) или Йордона (Yourdon), заняли руководящие позиции или стали независимыми консультантами и профессиональными лидерами, в свою очередь оказывая влияние на современную практику разработки программного обеспечения и приложений.

Формирующую роль в этом процессе культурного изменения сыграли один человек и крошечная организация, проводившая курсы по повышению квалификации. Оба персонажа – человек и организация – мало кому известны за пределами Австралии. Компания называлась DP Education

Proprietary Limited, а Деннис Дэви (Dennis Davie) был веселым импресарио, который первым завлек Эда Йордона и Ларри Константина на континент, чтобы научить австралийцев своим новым методам. В то время когда мало кто в Соединенных Штатах даже слышал о структурных методах, а рукопись книги «Structured Design» (Структурное проектирование), известной как «Orange book», еще только предполагалось отдать в печать, австралийцы уже получали этот материал из первых рук. С Дэви было очень приятно сотрудничать, и за эти годы он сумел привлечь к работе в Австралии значительную группу квалифицированных преподавателей и новаторски мыслящих людей. Оставаясь человеком небогатым и малоизвестным, Дэви тихо и счастливо ушел на пенсию и перебрался на Золотое побережье Австралии. Иногда он проводил одно-два занятия в своей компании, которую возглавил его сын. Знал ли он об этом или нет, но Дэви, который умер в начале 1998 года, изменил жизнь многих людей.

Структурирование

За примерами других влиятельных импресарио далеко ходить не надо. Эд Йордон, возможно, больше известен как автор методов, в названии которых фигурирует его имя, или как плодовитый писатель и убедительный демонстратор. Однако самый значительный вклад в данную область он внес в качестве импресарио. Вряд ли будет преувеличением сказать, что на основе структурных методов он создал индустрию. Технологии автоматизированного проектирования и создания программ имеют одни и те же корни.

Важная часть его одаренности – способность привлекать талантливых людей и ценить талант в человеке. Он не только сумел правильно распознать зарождение новых направлений, но и собрать вокруг себя самых лучших и самых ярких на то время мыслителей и преподавателей. В своем апогее компания Yourdon Inc., возглавляемая Эдом и его женой Тони Нэш (Toni Nash), была очень привлекательным местом. Почти каждый, у кого было что сказать и было желание сказать это, хотел там работать. Удивительно, как много современных самых влиятельных лидеров отрасли так или иначе начинали у Эда Йордона.

Рытье каналов

Импресарио интересны не только с исторической точки зрения; их роль сегодня так же важна, как и всегда. Например, Джордж Шассел (George Schussel) из Digital Consulting выражает себя именно в этой роли. Помимо всего прочего, он способствовал тому, что сопротивляющийся Ларри Константин вернулся на передовую линию в области разработки программного обеспечения. Шассел и еще один импресарио Рик Фридман (Rick Friedman) сыграли важную роль в привлечении внимания широкой

аудитории к объектной технологии. Одни из самых удачных конференций в данной области были организованы Коэн Тинджли Викорен (Ko-Ann Tingley Vikoren), *impresaria*, которая раньше занималась организацией Software Development Conferences и пригласила меня на самую первую конференцию в 1988 году.

Кстати говоря, в стране, которая дала нам это слово, есть свой импресарио информационных технологий. Джованни Модика (Giovanni Modica) не только великолепный повар итальянской кухни. Он всеми силами способствует тому, чтобы Италия стала мировым лидером в разработке информационных систем. Средством здесь должен стать *перенос технологий* (technology transfer), что отражено и в названии его компании. Приглашая самых лучших, самых ярких людей со всего мира, он надеется сделать будущее итальянских методов разработки более светлым. И похоже, он своего доьется. Благодаря своему упорству и искренности он сумел собрать группу специалистов мирового класса, которые теперь регулярно останавливаются в Милане и Риме, чтобы передать итальянским разработчикам новейшие эффективные методы и инструменты.

Ворота

Десятилетия исследований показали, что успех технической организации может в значительной степени зависеть от того, как выполняются некоторые ключевые роли. Руководители, которые признают талант, не тушуясь перед ним, и способны побудить самого лучшего стать еще лучше, служат катализаторами технического прогресса. Они – импресарио, которые могут стимулировать организацию к постоянному расширению своих горизонтов. Сами они могут не заниматься созданием кода или разработкой системной архитектуры, но то, как они толкают, тянут, поддерживают и противостоят, является важнейшим ингредиентом, от которого зависит либо раскрытие, либо сворачивание возможностей предприятия.

Не менее важны люди, которые выступают в роли привратников, или каналов для передачи технической информации. Привратники – это те, кто ездят на конференции, читают технические публикации, покупают книги, обращают внимание коллег на новые идеи. Зачастую привратники – это импресарио в меньшем масштабе. Они организуют семинары в неформальной обстановке и приводят на них сторонних специалистов и консультантов. Они не столько источники, сколько каналы, и без них не было бы взаимодействия.

Возможно, это вы.

Из журнала *Software Development*, том 3, № 9, сентябрь 1995 г.

Приложение

Зарегистрированный Peopleware

(Последняя заметка из серии «Peopleware» появилась в мартовском выпуске журнала *Software Development* за 1995 год. Когда колонку внезапно закрыли, у меня уже был черновик апрельской заметки, которую я предполагал написать в более шутливом стиле в соответствии с первым днем этого месяца. Мне долго казалось несправедливым, что те читатели, которые на протяжении значительного времени оставались верны этим заметкам, были лишены возможности услышать лебединую песню этой серии. Поэтому предлагаю вашему вниманию ту самую «потерянную заметку».)

Peopleware™. Хотите верьте, хотите нет, но это торговая марка. В подтверждение этого в моем почтовом ящике появилось письмо как раз в то время, когда я заканчивал подготовку первого издания этой книги (Constantine, 1995 [26]). В письме выражалось беспокойство по поводу того, что это слово может ввести людей в заблуждение. Будущие покупатели книги могут подумать, что я написал историю неизвестной компании, которая и является владельцем этого названия.

Я не особенно волновался, как, впрочем, и юристы из издательства Prentice Hall. Для удержания торговой марки нельзя допускать ее перехода во всеобщее достояние вследствие широкого, бессистемного и неконтролируемого применения. Такая участь постигла многие старые торговые марки, такие как эскалатор и керосин. Для того чтобы иметь право на обладание торговой маркой, вы должны быть первым, кто использовал ее в торговле, и вы должны тщательно охранять ее. Например, если я напишу в своей заметке, что я пил Coke®, но не напишу это слово с заглавной буквы или не поставлю после него магический значок «зарегистрирован-

ной» марки, то милые люди из Coca Cola™, которые зорко за всем следят, направят мне вежливое, но настойчивое письмо с напоминанием о том, что это зарегистрированная торговая марка и что ее следует всегда писать с заглавной буквы, и желательно, чтобы после нее стоял знакомый значок с буквой R в кружочке. О, конечно, эти приятные люди не будут судиться с каждым простым журналистом, который забывает правила игры. Они приберегут свои большие правовые ружья для тех, кто попытается без разрешения продавать хранители экранов с надписью Coca Cola™.

Как сказано в оригинальном введении к этой книге, слово «Peopleware», без украшающих знаков или сносок, возникло много лет назад. Оно широко применялось, например как заголовок довольно известной книги команды Тома и Тима (DeMarco и Lister, 1987 [33]).[1] Все это ни разу не приводило к каким-либо судебным разбирательствам. Конечно, я никогда бы не посмел отнестись с презрением к полубогам судебного процесса и не рискнул бы навлечь на себя гнев владельцев торговых марок, будь они большими или маленькими, известными или неизвестными. Поэтому, во избежание каких-либо недоразумений во время чтения этого текста, помните, что все упомянутые здесь торговые марки являются собственностью их владельцев. Во всяком случае так юридические нормы обязывают нас указать на обложке книги или внизу веб-страницы.

Имейте в виду, что я не являюсь радикальным сторонником свободного использования интеллектуальной собственности. Я не ссорюсь с торговой маркой Lexmark™ или с зарегистрированной чепухой, распространяемой в виде таких выдуманных монет информационного века, как Agilent™ или Inspiron™. Но как создатель новых слов я сам начинаю испытывать возмущение, когда компании делают торговой маркой Обычные Слова™, захватывая в своих узких меркантильных Интересах® наше общее языковое наследие, плоды многих веков письменной и устной эволюции.

Действительно ли практика изъятия обычных слов из общего пользования является Разумным Выбором℠ для гонки, в которой язык имеет такое большое значение? Может быть. Но это похоже на Парадокс®. С одной стороны, нужно защищать коммерцию и капитал, который торговцы и производители вкладывают в свои Добрые Имена®. С другой стороны, нельзя подавлять Реальное Общение™. Я, например, не знаю, что такое Отличное Решение℠, но я хорошо знаю, что мы не должны упускать из виду различие между *Компьютерным Языком*™ и обычным языком, между необходимостью идентификации торговой марки и потребностями простого общения.

---

[1] Т. ДеМарко и Т. Листер «Человеческий фактор: эффективные проекты и команды». – Пер. с англ. – СПб: Символ-Плюс, I кв. 2004 г.

При отсутствии большей разумности в этом вопросе продолжение регистрации повседневных слов и фраз может превратить в кошмар повседневную жизнь. Представьте себе такую сцену. Однажды вы поднимаетесь с постели, ваша голова функционирует только наполовину после вчерашних поздних посиделок на работе. Вы выпиваете немного кофе, чтобы разбудить остальную часть своего мозга™. Ваш партнер спрашивает вас: «Куда вы хотите пойти сегодня™?».

«Я бы хотел вернуться в постель, – ворчите вы, – но у меня есть работа в Офисе™. Обычная рутинная работа. Люди, места, заботы™.»

Вы делаете сумасшедший Бросок® к Автомагистрали®, чтобы успеть до того, как там появятся пробки. В офисе вам звонит по телефону потенциальный клиент, занимающийся интернет-торговлей. Он ищет Внутренние Пути® улучшения Юзабилити Посредством Проектирования™.

«Как и другие Крутые Организации™, – объясняет клиент, – мы заинтересованы в Ускорении своего бизнеса™. Нам нужны Реальные РешенияSM.»

Вы отвечаете: «Вам понравится то, как мы работаем™. Мы ориентируемся на сотрудничество со своими клиентами. У нас великолепная команда дизайнеров и разработчиков. Вместе мы сможем добиться удивительных результатов™.»

«Хорошо, что вы понимаете, что стандартный Подход® не для нас, – отвечает клиент. – Мы Все® обдумали. Мы хотим Выйти За Пределы® простой электронной торговли и создать Сообщество Покупателей™. Поэтому мы хотим, чтобы вы как консультанты были Смелыми® в своих идеях. Думайте Нестандартом™.»

«-тно, – бормочите вы. – Нестандартно».

После этого вы говорите более громко: «Давайте устроим Сетевую Конференцию® и обсудим эти вопросы. Я буду вашим Консультантом™, и мне будет помогать мой Дизайн-компаньон™. А Хейди®, которая работает у нас Офисным Помощником™, будет Организатором® этого собрания. Естественно, вы можете быть уверены, что все мы понимаем важность сохранения конфиденциальности и поэтому будем осторожны при Взаимодействии™.»

Вы хватаете веб-камеру и лэптоп с сетевой картой и направляетесь в свободный конференц-зал для того, чтобы начать монтировать Разъемы и Гнезда™. После того как все оказываются в онлайне, а программное обеспечение Работает Согласованно®, вы начинаете вкратце описывать некоторые новые идеи для перестройки их сайта. Вы спрашиваете: «Вы так же представляете себе Сеть™?»

Вы предлагаете другие идеи. И вдруг Все Заработало™. «Заметьте разницу™. Я думаю, это будет Лучшим Решением™. Нет ничего нового под Солнцем®, но, хотя это и не выглядит как-то особенно, Разница Есть™. Разница состоит в понимании℠. При таком варианте посетитель сайта получает Полный Контроль® и может быстро разобраться с Соединениями®. Так мир обменивается идеями℠.»

«Пусть будет по-вашему™. Мне это кажется Плохой Идеей®, – говорит один скептик из вашей команды. – Как можно узнать, что это будет работать?»

«Мы просто чувствуем, что это правильно™. В сущности, это быстрая и простая техника drag-and-drop. Возьмите ЭТО, Переместите ЭТО, Используйте ЭТО™. Мы можем работать с высокоуровневыми концепциями, но нужно Спустить информацию с небес на землю®. Цель – создать Просто Мощное Программное Обеспечение™.»

«Конечно, нужна технология, Построенная для нашего Быстрого Времени™ и способная функционировать Со скоростью информации™. Будет разумно сделать апгрейд программного обеспечения вашего сервера – это надежное инвестирование в ваш компьютер™. Не нужно ничего фантастического: только Настраиваемое Программное Обеспечение®, чтобы вы могли Настроить его и Забыть®...»

Однако вскоре становится очевидно, что только консультация на месте сможет прояснить, что в действительности нужно клиенту. К сожалению, собрание нужно провести немедленно, а офис клиента находится на Золотом Побережье Австралии – 14 часов полета из международного аэропорта Лос-Анжелеса.

«Без проблем,– говорите вы. – Мы будем там™. У меня уже есть Виза®. Мне только нужно будет сделать некоторые Путевые Заметки™. И не беспокойтесь, никто из нашей команды никогда не опаздывает на самолет. Некоторые люди просто умеют летать℠.»

Вы начинаете обсуждать условия контракта. Наконец, клиенты соглашаются. «Перед взлетом не забудьте захватить с собой дополнительные контракты,– говорит руководитель проекта, – по всем услугам, которые вы предлагаете®. Договорились?»

«ОК! – отвечаете вы. – Партнеры®!»

И так далее. Кто-то где-то подает сейчас заявку на регистрацию вашей любимой фразы, вашего любимого цвета или, быть может, даже вашего имени. Можете на это рассчитывать.

Гарантировано. Точка®.

Библиография

[1] Ancona, D. G. and D. F. Caldwell. 1992. «Bridging the Boundary: External Activity and Performance in Organizational Teams», *Administrative Science Quarterly* 37(4): 634–65.

[2] Anderson, L. E. and W. K. Balzer. 1991. «Effects of Timing of Leaders' Opinions on Problem-Solving Groups», *Group & Organizational Studies* 16(1): 86–101.

[3] Belbin, R. M. 1981. *Management Teams: Why They Succeed or Fail*. London: Heinemann.

[4] Beyer, H., and K. Holtzblatt. 1997. *Contextual Design: Defining Customer-Centered Systems.* NY: Morgan Kaufman.

[5] Bollinger, T. B. and C. McGowan. 1991. «A Critical Look at Software Capability Evaluations», *IEE Software* 8(4): 25–41.

[6] Case, J. 1990. The Open-Book Managers», *Inc.* 12(9): 104–13.

[7] Chidamber, S., and C. Kemerer. 1994. «A Metrics Suite for Object-Oriented Design», *IEEE Trans. Software Eng.* 20 (6): 476–493.

[8] Cobb, R. H. and H. D. Mills. 1990. «Engineering Software Under Statistical Quality Control», *IEEE Software* 7(6): 44–54.

[9] Collins, D. 1995. *Designing Object-Oriented User Interfaces.* Redwood City, CA: Benjamin/Cummings.

[10] Constantine, L. L. 1986. *Family Paradigms*. New York: Guilford Press.

[11] Constantine, L. L. 1989. «Teamwork Paradigms and the Structured Open Team», *Proceedings: Embedded Systems Conference*. San Francisco: Miller Freeman.

[12] Constantine, L. L. 1990. «Organization Paradigms and the Management of Change», *Proceedings: Software Development '90*. San Francisco: Miller Freeman.

[13] Constantine, L. L. 1991a. «Building Structured Open Teams to Work», *Software Development '91 Proceedings*. San Francisco: Miller Freeman.

[14] Constantine, L. L. 1991b. «Toward Usable Interfaces: Bringing Users and User Perspectives into Design», *American Programmer* 4(2): 6–14.

[15] Constantine, L. L. 1991c. «Fitting Intervention to Organization Paradigm», *Organization Development Journal* 9(2): 41–50.

[16] Constantine, L. L. 1992a. «Managing for Quality User Interfaces», *Software Management 1992 Proceedings*. San Francisco: Miller Freeman.

[17] Constantine, L. L. 1992b. «Getting the User Interface Right: Basic Principles», *Software Development 1992 Proceedings*. San Francisco: Miller Freeman.

[18] Constantine, L. L. 1992c. «Quality by Increments: Small Steps with Big Payoffs», *American Programmer* 5 (2).

[19] Constantine, L. L. 1993a. «Objects in Your Face», *Object Magazine*, July.

[20] Constantine, L. L. 1993b. «User Interface Design for Embedded Systems», *Embedded Systems Programming*, August.

[21] Constantine, L. L. 1993c. «Work Organization Paradigms for Project Management and Organizations», *Communications of the ACM* 36(10). October.

[22] Constantine, L. L. 1994a. «More than Just a Pretty Face: Designing for Usability», *Software Development 1994 Proceedings*. San Francisco: Miller Freeman.

[23] Constantine, L. L. 1994b. «Collaborative Usability Inspections for Embedded Systems», *Embedded Systems Conference Proceedings*. San Francisco: Miller Freeman.

[24] onstantine, L. L. 1994c. «Interfaces for Intermediates», *IEEE Software* 11(4): 96–99.

[25] Constantine, L. L. 1994d. «Graphical Navigation», *Windows Tech Journal*, August.

[26] Constantine, L. L. 1995. *Constantine on Peopleware*. Englewood Cliffs, NJ: Prentice Hall.

[27] Constantine, L. L. 1996. «Usage-Centered Software Engineering: New Models, Methods, and Metrics», In Purvis, M. (ed.) *Software Engineering: Education & Practice*. Los Alamitos, CA: IEEE Computer Society Press.

[28] Constantine, L. L. 1997. «Visual Coherence and Usability: A Cohesion Metric for Assessing the Quality of Dialogue and Screen Designs». *OzCHI'96 Proceedings*. Los Alamitos, CA: IEEE Computer Society Press.

[29] Constantine, L. L. 1998. «Rapid Abstract Prototyping», *Software Development, 6* (11), November.

[30] Constantine, L. L., and Lockwood, L. A. D. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Reading, MA: Addison-Wesley.

[31] Cooper, A. 1995. *About Face: The Essentials of User Interface Design*. Foster City, CA: IDG Books.

[32] DeMarco, T. 1982. *Controlling Software Projects*. New York: Yourdon Press.

[33] DeMarco, T. and T. Lister. 1987. *Peopleware: Productive Projects and Teams*. New York: Dorset House.[1]

[34] Dijkstra, E. W. 1968. «Go To Statement Considered Harmful», *Communications of the ACM*, Vol. 11, No. 3, March, pp. 147–148.

[35] Doyle, M. and M. Strauss. 1982. *How to Make Meetings Work*. New York: Jove.

[36] Finegan, J. 1990. «The Education of Harry Featherstone,» *Inc*. 12 (7): 57–66.

[37] Fisher, R. and W. Ury. 1981. *Getting to Yes*. New York: Houghton Mifflin.

[38] Fisher, R. and Brown. 1988. *Getting Together: Building Relationships As We Negotiate*. New York: Penguin.

[39] Henderson-Sellers, B., L. L. Constantine, and I. M. Graham 1996. «Coupling and Cohesion: Toward a Valid Metrics Suite for Object-Oriented Analysis and Design», To appear in *Object-Oriented Systems*.

[40] Holtzblatt, K. and H. Beyer. 1993. «Making Customer-Centered Design Work for Teams», *Communications of the ACM* 36(10), October.

[41] Humphrey, W. S., T. S. Snyder, and R. R. Willis. 1991. «Software Process Improvement at Hughes Aircraft», *IEEE Software* 8(4): 11–23.

[42] Hyman, R. B. 1993. «Creative Chaos in High-Performance Teams: An Experience Report», *Communications of the ACM* 36(1). October.

[1] Т. ДеМарко и Т. Листер «Человеческий фактор: эффективные проекты и команды». – Пер. с англ. – СПб: Символ-Плюс, I кв. 2004 г.

[43] Jacobson, I., G. Booch, and J. Rumbaugh 2000. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

[44] Jacobson, I., M. Christerson, P. Jonsson, and G. Цvergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, Mass.: Addison-Wesley.

[45] Kantor, D. K. and W. Lehr. 1975. *Inside the Family: Toward a Theory of Family Process*. San Francisco: Jossey-Bass.

[46] Kruchten, P. 2000. *The Rational Unified Process*. Reading, MA: Addison-Wesley.

[47] Larson, C. E. and F. M. J. LaFasto. 1989. *TeamWork*. Beverly Hills: Sage.

[48] Lickert, R. 1989. *New Patterns in Management*. New York: McGraw-Hill.

[49] Lockwood, L. A. D., and L. L. Constantine 1993. «From Events to Objects: The Heresy of Event-Orientation in a World of Objects», *OOPSLA '92: Addendum to the Proceedings*. New York, ACM Press.

[50] Mackenzie, D. D. 1966. «The Philosophy of Conventions», in *Concepts in Program Design*, L. L. Constantine, ed. Cambridge, Mass.: Information & Systems Press.

[51] Newmann, P. G. 1976. «Peopleware in Systems», in *Peopleware in Systems* 15–18. Cleveland, Ohio: Association for Systems Management.

[52] Nielsen, J. 1993. *Usability Engineering*. Boston: Academic Press.

[53] Norman, D. O. 1988. *The Psychology of Everyday Things*. New York: Basic Books.

[54] Page-Jones, M. 1980. *Practical Guide to Structured Systems Design*. New York: Yourdon Press.

[55] Page-Jones, M. 1995. *What Every Programmer Should Know About Object-Oriented Design*. New York: Dorset House.

[56] Page-Jones, M. 2000. *Fundamentals of Object-Oriented Design in UML*. Reading, MA: Addison-Wesley.

[57] Page-Jones, M., L. L. Constantine, and S. J. Weiss. 1990. «Modeling Object-Oriented Systems: A Uniform Object Notation», *Computer Language* 7(1), October.

[58] Plauger, P. J. 1993. *Programming on Purpose II: Essays on Software People*. Englewood Cliffs, N.J.: Prentice Hall.

[59] Priem, R. L. and K. H. Price. 1991. «Process and Outcome Expectations for Dialectical Inquiry, Devil's Advocacy, and Consensus Techniques of Strategic Decision Making», *Group & Organizational Studies* 16(2): 206–25.

[60] Rettig, M. 1990. «The Practical Programmer: Software Teams». *Communications of the ACM* 33(10), October.

[61] Sutherland, I.E. 1968. «SketchPad: A Man-Machine Graphical Communication System», in *Proceedings, AFIPS Spring Joint Computer Conference*. 1963. 23. pp. 329–346.

[62] Thomsett, R. 1990. «Effective Project Teams: A Dilemma, A Model, A Solution», *American Programmer* 3(7/8): 25–35.

[63] van Harmelan, M. (ed.) 2001. *Object Modeling and User Interface Design*. Harlow, England: Addison-Wesley.

[64] Ward, P. 1992. «The Evolution of Structured Analysis. Part II: Maturity and Its Problems», *American Programmer* 5(4): 18–29.

[65] Watzlawick, P., J. H. Beavin, and D. D. Jackson. 1967. *Pragmatics of Human Communication*. New York: Norton.

[66] Weinberg, G. M. and E. L. Schulman. 1974. «Goals and Performance in Computer Programming», *Human Factors* 16(1): 70–77.

[67] Whitchurch, G. G. and L. L. Constantine. 1992. «Systems Theory», in *Sourcebook of Family Theories and Methods: A Contextual Approach*, P. B. Boss et al., eds. New York: Plenum.

[68] Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, N.J.: Prentice Hall.

[69] Wood, J. and D. Silver. 1989. *Joint Application Design*. New York: John Wiley & Sons.

[70] Yourdon, E., and L. L. Constantine 1974. *Structured Design, first printing*. New York: Yourdon, Inc.

[71] Zahniser, R. A. 1990. «Building Software in Groups», *American Programmer* 3(7/8): 50–56.

[72] Zahniser, R. A. 1993. «Design by Walking Around», *Communications of the ACM* 36(10), October.

Алфавитный указатель

N

O

P

R

S

T

U

Г

Д

Е

Ж

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ч

Ш

Э

Ю

Я