ПРОФЕССИОНАЛЬНО

Пит Гудлиф, Робърт Мартин,
Дэвидэлис Стинеллис, Кевлин Хенни и др.

97

ЭТЮДОВ ДЛЯ ПРОГРАММИСТОВ

O'REILLY®

СИМВОЛ®

16 лет вместе
с профессионалами

97 Things Every Programmer Should Know

Collective Wisdom from the Experts

Pete Goodliffe, Robert Martin,
Diomidis Spinellis, Kevlin Henney and others

Edited by Kevlin Henney

O'REILLY®

ПРОФЕССИОНАЛЬНО

97 этюдов для программистов

Опыт ведущих экспертов

Пит Гудлиф, Роберт Мартин,
Диомидис Спинеллис, Кевлин Хенни и др.

под редакцией Кевлина Хенни

СИМВОЛ®

Санкт-Петербург — Москва
2012

Серия «Профессионально»

Пит Гудлиф, Роберт Мартин,
Диомидис Спинеллис, Кевлин Хенни и др.

97 этюдов для программистов
Опыт ведущих экспертов

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Науч. редактор	*А. Долгушин*
Редактор	*М. Зислис*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*

Гудлиф П., Мартин Р., Спинеллис Д., Хенни К. и др.

97 этюдов для программистов. Опыт ведущих экспертов. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 256 с., ил.

ISBN 978-5-93286-198-1

Приобщитесь к мудрости экспертов и запомните то, что должен знать каждый программист, с каким бы языком и на какой платформе он ни работал. 97 кратких и очень полезных советов повысят ваш профессионализм посредством новых подходов к старым проблемам, лучших практик и разумных подсказок, предназначенных для оттачивания мастерства.

Авторы этой книги, очень опытные и признанные в отрасли специалисты, передадут вам практические знания и принципы, полезные для проектов любого типа. Статьи касаются разных тем: от рекомендаций по написанию кода до культуры, от выбора алгоритмов до гибкого программирования, от приемов реализации до профессионализма, от стиля до сущности. Новички смогут познакомиться с фундаментальными положениями, а для профессионалов сборник сможет стать отправной точкой для обсуждений.

ISBN 978-5-93286-198-1
ISBN 978-0-596-80948-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 29.02.2012. Формат 70×90 $^{1}/_{16}$.
Печать офсетная. Объем 16 печ. л.

Отсутствующим друзьям посвящается

Оглавление

Статьи по категориям

Ошибки и исправления

Сборка и развертывание

Правила написания и форматирования кода

Принципы проектирования и техника написания кода

Предметно-ориентированное мышление

Ошибки, их обработка и исключения

Обучение, мастерство и опыт

Ночное и волшебное

Производительность, оптимизация и представление

Профессионализм, мировоззрение и позиция

Языки и парадигмы программирования

Рефакторинг и забота о коде

Повторное использование и повторение кода

Графики, сроки и оценки

Простота

Командная работа и сотрудничество

Тесты, тестирование и тестировщики

Инструменты, автоматизация, среды разработки

Пользователи и заказчики

Предисловие

Новейший компьютер способен лишь с большей скоростью усложнить древнейшую проблему отношений между людьми, и в конечном итоге участнику общения по-прежнему придется решать, что и как говорить.

Эдвард Р. Мэроу (Edward R. Murrow)

Программистам есть, над чем думать. Языки программирования, приемы программирования, среды разработки, стили написания кода, инструменты, процессы разработки, планы работ, совещания, архитектуры программ, шаблоны проектирования, динамика командного взаимодействия, код, технические требования, дефекты, качество кода. И другое. Много чего еще.

Здесь мы находим искусство, ремесло и науку, которые простираются далеко за рамки программы. Деятельность программиста объединяет дискретный мир компьютеров и текучий мир человеческих занятий. Программисты служат связующим звеном между бизнесом с его расплывчатыми договорными истинами и выверенной, бескомпромиссной областью, где царят биты, байты и построенные на их основе пользовательские типы.

Учитывая объемы знаний, работы и разнообразие способов ее выполнения, никакой человек или источник не может претендовать на знание «истинного пути». Поэтому, опираясь на народную мудрость и накопленный опыт, книга «97 этюдов для программистов» предлагает не столько упорядоченную общую картину, сколько пеструю мозаику мнений о том, что должно быть известно каждому программисту. Она касается разных тем: от рекомендаций по написанию кода до культуры, от выбора алгоритмов до гибкого программирования, от приемов реализации до профессионализма, от стиля до сущности.

Отдельные статьи не стыкуются между собой, да и цель ставилась скорее противоположная. Ценность отдельной статьи здесь как раз в том, что она не похожа на другие. А ценность сборника в целом состоит в том, что статьи дополняют, подтверждают одна другую и даже противоречат друг другу. Они не связаны общим сюжетом: читатель сам может оценить материал, поразмышлять над ним и увязать прочитанное, сравнив новое с собственными контекстом, знаниями и опытом.

Лицензионные права

Контакты

На веб-странице книги перечислены найденные ошибки и приводятся дополнительные сведения:

http://www.oreilly.com/catalog/9780596809485/

Сопроводительный сайт, где опубликованы все статьи, биографии авторов и другие данные, находится по адресу:

http://programmer.97things.oreilly.com

Вы также можете следить за новостями и исправлениями книги в Twitter:

http://twitter.com/97TEPSK

Комментарии и технические вопросы, касающиеся этой книги, можно отправить электронной почтой:

bookquestions@oreilly.com

Дополнительная информация о наших книгах, Центрах ресурсов и сети O'Reilly Network приведена на нашем веб-сайте:

http://www.oreilly.com/

Safari® Books Online

Safari Books Online – цифровая библиотека, которая дает возможность быстро находить ответы на ваши вопросы в 7500 технических книг, справочников и видеозаписей.

Подписка Safari дает право читать любую страницу и смотреть любое видео в режиме онлайн. Читайте книги на сотовых телефонах и мобильных устройствах. Получайте доступ к новым изданиям до выхода их из печати. Получайте эксклюзивный доступ к рукописям в процессе работы и отправляйте замечания авторам. Копируйте текст примеров кода, загружайте главы, создавайте закладки и заметки, печатайте страницы – вот лишь некоторые из множества функций, экономящих ваше время.

O'Reilly Media опубликовала эту книгу в Safari Books Online. Чтобы получить полный цифровой доступ к этой книге и книгам схожей тематики, выпущенным

O'Reilly и другими издательствами, оформите бесплатную подписку на *http://my.safaribooksonline.com*.

Благодарности

Проекту «97 этюдов для программистов» прямо или косвенно отдали свое время и знания многие люди. Все они заслуживают благодарности.

Ричард Монсон-Хейфел (Richard Monson-Haefel) – редактор серии «97 Things» и редактор первой книги из этой серии, «97 Things Every Software Architect Should Know»[1], в написании которой я принимал участие. Спасибо Ричарду за идею серии и за ее открытость для потенциальных участников, а также за то, что он так энергично поддерживал мои предложения по данной книге.

Хочу поблагодарить всех тех, кто отдал свое время и силы, участвуя в создании текстов для этого проекта: как тех, чьи статьи опубликованы в этой книге, так и тех, чьи тексты не попали в нее, но опубликованы на веб-сайте. Большое количество и высокое качество материала весьма затруднили процесс окончательного отбора – жестко фиксированное в названии книги число не оставило места для дополнительных статей.

Я также благодарен за дополнительные отзывы, комментарии и предложения, авторами которых были Джованни Аспрони (Giovanni Asproni), Пол Колин Глостер (Paul Colin Gloster) и Микаэль Хунгер (Michael Hunger).

Спасибо O'Reilly за предоставленную этому проекту поддержку – от вики-хостинга, сделавшего книгу реальностью, до обеспечения всех стадий процесса публикации в бумажном виде. Сотрудники O'Reilly, которых я хотел бы особо поблагодарить: Майк Лукидес (Mike Loukides), Лорел Акерман (Laurel Ackerman), Эди Фридман (Edie Freedman), Эд Стивенсон (Ed Stephenson) и Рейчел Монахан (Rachel Monaghan).

Дело не только в том, что текст книги рождался в среде Веб: через Веб проект также приобрел известность и популярность. Спасибо всем, кто распространял сведения о нем через социальные сети, блоги и прочими путями.

Хочу также поблагодарить свою жену Кэролин за то, что привносит порядок в мой хаос, и двух моих сыновей, Стефана и Янника, за то, что часть этого порядка они вновь превращают в хаос.

Надеюсь, что эта книга станет для вас источником информации, открытий и вдохновения.

Приятного чтения!

Кевлин Хенни (Kevlin Henney)

---

[1] Нил Форд, Майкл Найгард, Билл де Ора и др. «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

Будьте благоразумны

Себ Роуз

В любом деле будь благоразумен и думай о последствиях.

Неизвестный

Как бы успокаивающе ни выглядел график работы в начале итерации, в какой-то момент неизбежно возникает нехватка времени. Если приходится разрываться между «сделать правильно» и «сделать быстро», часто возникает соблазн «сделать быстро» с оговоркой, что вы исправите решение позже, когда появится время. Вы совершенно искренне даете обещание именно так и поступить – даете самому себе, команде или заказчику. Но очень часто на следующей итерации возникают уже другие проблемы, которым и приходится посвящать свое внимание. Такую отложенную работу называют *техническим долгом*, и хорошего от него не жди. В своей классификации технических долгов Мартин Фаулер называет такой вид долга *умышленным техническим долгом*, и его не следует путать с *непреднамеренным техническим долгом*.[1]

Технический долг подобен кредиту: в краткосрочной перспективе он выгоден, но по нему приходится выплачивать проценты до полного погашения займа. Срезая углы при написании кода, вы затрудняете как разработку новой функциональности, так и рефакторинг. Это создает благоприятную почву для появления ошибок и нестабильных *тестовых сценариев* (*test cases*). Чем дольше долг существует, тем тяжелее последствия. К тому времени, как дойдут руки внести запланированные исправления, может оказаться, что на основе изначального сомнительного кода уже выстроена целая гора не вполне верных с точки зрения проектирования решений, а это значительно осложнит рефакторинг и исправление этого кода. На самом деле, к решению изначальной проблемы часто возвращаются лишь тогда, когда *уже нет выбора*, кроме как вернуться и все исправить. И зачастую к этому моменту исправление оказывается уже настолько

[1] *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html*

сложным, что вы просто не можете себе позволить потерять так много времени или пойти на подобный риск.

Бывают ситуации, когда приходится идти на создание технического долга, если необходимо уложиться в срок или частично реализовать некую функциональность. Старайтесь не оказываться в таких ситуациях, однако если положение совершенно безвыходное, действуйте. Но (и это увесистое *но*) вы должны вести учет своего технического долга и гасить его как можно скорее, иначе проблемы растут, как снежный ком. И если уж вы пошли на такой компромисс, составьте карточку с заданием (task card) или создайте запись в системе учета дефектов, чтобы не забыть о проблеме.

Если вы планируете погасить свой долг на следующей итерации, потери будут минимальными. На непогашенный долг капают проценты, за которыми нужно постоянно следить, чтобы видеть реальную конечную цену. Это подчеркивает влияние технического долга проекта на его бизнес-стоимость и позволяет разумно расставлять акценты в вопросах погашения такого долга. Способ начисления и отслеживания процентов зависит от конкретного проекта, но отслеживать их должны вы.

Гасите технические долги как можно скорее. Поступать иначе – неблагоразумно.

Применяйте принципы функционального программирования

Эдвард Гарсон

Функциональное программирование недавно снова обратило на себя внимание большинства в сообществе программистов. Отчасти благодаря тому, что *эмерджентные свойства* функциональной парадигмы созвучны решению задач, возникающих в нашей отрасли в связи с ростом значимости многоядерных архитектур. И хотя данное применение, несомненно, важно, однако не оно является главным основанием для моего наставления *познать функциональное программирование*.

Овладев парадигмой функционального программирования, программист может значительно повысить качество кода, создаваемого в других контекстах. Глубокое понимание парадигмы функционального программирования и ее применение на практике помогут вам проектировать системы, обладающие гораздо большей степенью *ссылочной прозрачности* (*referential transparency*).

Ссылочная прозрачность является качеством очень желательным: она предполагает, что функции неизменно дают одинаковые результаты на одинаковых входных данных независимо от места и времени обращения к этим функциям. Вычисление функции, таким образом, слабо зависит от побочных эффектов изменяющегося (mutable) состояния – в идеале не зависит от них вообще.

Один из главных источников дефектов в коде на императивном языке программирования – изменяемые (mutable) переменные. Каждому читателю наверняка приходилось разбираться, почему в каком-либо конкретном случае некоторое значение не соответствовало ожидаемому. Семантика областей видимости может препятствовать появлению таких коварных ошибок или, по крайней мере, значительно сужать возможную область их появления. Но истинной причиной их возникновения может быть сама концепция проектирования такого кода, который беспорядочно полагается на изменяемость (mutability).

И в этом отношении нам определенно не стоит ждать особой помощи от собственной отрасли. Вводные тексты по объектно-ориентированному программированию скрыто пропагандируют подобные конструкции. В них часто приводятся

примеры групп объектов, которые обладают относительно долгим сроком жизни и обмениваются вызовами методов, изменяющих состояние (mutator methods), что небезопасно. Однако грамотное проектирование на основе тестов (test-driven design), особенно если обеспечена «имитация ролей, а не объектов (mock roles, not objects)»,[1] позволяет избавиться от излишеств изменчивости в архитектуре.

В итоге, как правило, получается архитектура с более удачным распределением обязанностей и множеством мелких функций, которые работают с полученными аргументами, а не обращаются напрямую к изменяемым переменным-членам. Дефектов становится меньше, а также упрощается их отладка, ведь легче найти, откуда взялось неверное значение в такой конструкции, чем пытаться выяснить, в каком конкретном контексте появляется ошибочное присваивание. Это *значительно повышает* ссылочную прозрачность, и решительно ничто не может так способствовать глубокому усвоению этих идей, как изучение функционального языка программирования, где такая модель вычислений является нормой.

Конечно, такой подход оптимален не всегда. Например, зачастую в объектно-ориентированных системах лучшие результаты этот стиль дает при разработке модели предметной области (то есть когда сотрудничество объектов служит снижению сложности бизнес-правил), чем при разработке интерфейса пользователя.

Овладейте парадигмой функционального программирования, чтобы разумно применять полученные знания в других областях. Взять хотя бы ваши иерархии объектов – они станут просто светиться качеством ссылочной прозрачности и окажутся значительно ближе к своим функциональным аналогам, чем можно было бы предположить. На самом деле, некоторые даже высказывают мнение, что в своем высшем проявлении функциональное программирование и объектно-ориентированный подход оказываются *лишь отражениями друг друга*, своего рода вычислительными инь и ян.

[1] *http://www.jmock.org/oopsla2004.pdf*

Выясните, как поступит пользователь (и вы – не пользователь)

Жиль Колборн

Мы все склонны полагать, что другие люди рассуждают так же, как мы. Но это не так. В психологии это называется *эффектом ложного согласия*. Если люди думают или поступают иначе, чем мы, мы часто (подсознательно) считаем их в чем-то неполноценными.

Этот эффект объясняет, почему программисту так трудно поставить себя на место пользователя. Пользователи рассуждают иначе, чем программисты. Прежде всего, они значительно меньше времени проводят за компьютером. Они не знают и не хотят знать, как работает компьютер. Это означает, что пользователи не могут прибегать к арсеналу приемов решения проблем, которым в совершенстве овладели программисты. Им не знакомы шаблоны и визуальные приметы, которыми пользуются программисты при работе с интерфейсами, для навигации по ним и для освоения интерфейсов.

Чтобы понять образ мыслей пользователя, лучше всего понаблюдать за ним. Предложите ему выполнить задание с помощью программы, аналогичной той, которую вы разрабатываете. Задача должна быть реальной. «Сложить числа в колонке» – неплохо, но еще лучше: «подсчитать собственные расходы за последний месяц». Не следует брать слишком конкретные задачи, например «Вы не могли бы выделить эти ячейки в таблице и ввести ниже формулу суммирования?» – в этом вопросе содержится слишком явная подсказка. Предложите пользователю рассказывать о своих действиях по ходу работы. Не перебивайте его. Не пытайтесь помочь. Спрашивайте себя: «Почему он делает так?» и «Почему она не делает этого?».

Прежде всего, вы обнаружите, что есть ряд действий, которые все пользователи выполняют схожим образом. Они пытаются выполнять задачи в одном и том же порядке и делают одинаковые ошибки в одних и тех же местах. Такое базовое поведение нужно класть в основу проектирования. Сравните данный подход с тем, что обычно происходит на встречах по проектированию системы, когда собравшиеся прислушиваются к вопросам вроде такого: «А вдруг пользователь захочет...». Так в приложении появляются многочисленные функции и возникает

путаница в отношении того, что нужно пользователям. Если наблюдать за пользователями, такой путаницы не будет.

Вы можете столкнуться с тем, что пользователь где-то застрял. Когда вы сами застреваете, то осматриваетесь по сторонам. Когда застревает пользователь, его область внимания, напротив, сужается. Ему становится сложнее увидеть решение в другой области экрана. Вот одна из причин, почему вспомогательный текст – плохое решение для плохого пользовательского интерфейса. Если необходимо дать инструкции или сопроводительный текст, размещать их нужно в непосредственной близости от проблемной области. Именно сужение поля внимания пользователя служит причиной тому, что всплывающие подсказки более полезны, чем встроенная справка.

Обычно пользователи находят способ с грехом пополам довести дело до конца. Если они находят какой-то путь к цели, то будут идти по нему и впредь, каким бы запутанным он ни был. Лучше дать им один действительно очевидный способ решения задачи вместо двух или трех неочевидных (но более быстрых).

Вы также обнаружите большое расхождение между тем, чего пользователь, по его словам, хочет, и тем, что он делает в действительности. Это тревожный факт, ведь обычно требования пользователей собираются путем их опроса. Вот почему лучший способ собирать требования – наблюдать за пользователями. Час наблюдений за пользователями даст вам больше информации, чем день гаданий о том, что им нужно.

Автоматизируйте свой стандарт форматирования кода

Филип ван Лаенен

Вероятно, вы тоже через это проходили. В начале проекта у всех полно благих замыслов – назовем их «новопроектными обещаниями»[1]. Довольно часто многие из этих обещаний фиксируются документально. Обещания, связанные с кодом, попадают в стандарт оформления кода данного проекта. На первом собрании проекта ведущий разработчик оглашает этот документ, и в идеале все соглашаются старательно соблюдать предложенные требования. Однако по ходу работы над проектом все эти благие намерения одно за другим забываются. Когда проект наконец завершен, код выглядит весьма запутанно, и, похоже, никто не понимает, как так получилось.

Когда же все пошло наперекосяк? Вполне вероятно, что как раз на том первом собрании. Некоторые его участники были невнимательны. Другие не поняли, в чем смысл стандарта. Хуже того, кое-кто был против предложенного и прямо на собрании затевал против стандарта восстание. И даже те, кто понял и согласился, в какой-то момент работы на проекте были вынуждены под давлением обстоятельств упростить себе жизнь. Ведь хорошо отформатированный код не будет оценен клиентом, которому нужны новые функции в приложении. Кроме того, соблюдение стандарта оформления кода может оказаться весьма утомительным делом, если его не автоматизировать. Попробуйте расставить отступы в плохо написанном классе, и вы убедитесь в этом сами.

Но если это так трудно, зачем нам вообще создавать стандарт оформления кода? Одна из целей единообразного форматирования кода – не позволить никому «приватизировать» код путем форматирования его своим особым способом. Также не следует допускать применения разработчиками определенных антипаттернов, чтобы избежать ряда известных ошибок. В целом стандарт оформления должен облегчать работу над проектом и поддерживать постоянную скорость разработки от начала до конца. Отсюда следует, что поддержка стандарта должна быть

---

[1] По аналогии с новогодними обещаниями, которые люди дают себе: заниматься спортом, бросить курить и т. п. – *Прим. ред.*

единогласной: плохо, если один разработчик делает отступы размером в три пробела, а другой – в четыре.

Существует множество инструментов, с помощью которых можно создавать отчеты о качестве кода, а также документировать и поддерживать стандарт форматирования кода, но это только часть решения. Стандарт следует автоматизировать и внедрять принудительно там, где это возможно. Вот некоторые примеры:

- Сделайте форматирование кода частью процедуры сборки, чтобы оно происходило автоматически при каждой компиляции кода.
- Применяйте средства статического анализа кода для поиска нежелательных антипаттернов. Прерывайте сборку при их обнаружении.
- Научитесь настраивать эти инструменты, что поможет находить антипаттерны, специфические для вашего проекта.
- Не просто замеряйте процент покрытия тестами, но и делайте автоматическую оценку результатов. Прерывайте сборку, если процент покрытия тестами недопустимо низкий.

Постарайтесь внедрить эти принципы в отношении всех требований к коду, которые вы считаете важными. Полностью автоматизировать все, что вас беспокоит, вы не сможете. Те аспекты, которые невозможно обнаружить или исправить автоматически, следует включить в дополнительный набор правил – как приложение к автоматизированной части стандарта. Однако вам придется принять тот факт, что у вас и ваших коллег есть возможность соблюдать правила из этого приложения менее строго.

Наконец, стандарт кодирования должен эволюционировать, а не быть высеченным в камне. С течением времени потребности проекта меняются, и то, что казалось разумным в начале, совсем не обязательно останется таковым через несколько месяцев.

Красота – следствие простоты

Йорн Ольмхейм

У Платона есть одно высказывание, которое, как мне кажется, особенно полезно было бы знать и принимать близко к сердцу всем разработчикам программного обеспечения:

> Красота стиля, гармония, изящество и хороший ритм основываются на простоте.

Одно это предложение объединяет ценности, которыми нам, разработчикам, надлежит восхищаться.

Есть ряд вещей, которых мы стремимся достичь в нашем коде:

- Читаемость
- Простота сопровождения
- Скорость разработки
- Неуловимая красота

Платон говорит нам, что все эти качества возможны только благодаря простоте.

Что такое красивый код? Вероятно, это очень субъективный вопрос. Восприятие красоты сильно зависит от личного опыта, как зависит от него наше восприятие чего бы то ни было. Изучавшие искусство иначе воспринимают красоту (по крайней мере, иначе к ней подходят), чем получившие техническое образование. Люди с гуманитарным образованием обычно рассматривают красоту программы, сравнивая ее с произведением искусства, тогда как люди с техническим образованием чаще рассуждают о симметриях и золотом сечении, пытаясь все свести к формулам. По моему опыту, именно на простоте строятся в большинстве своем доводы обеих сторон.

Поразмышляйте об исходном коде всех программ, которые вам доводилось встречать. Если вы никогда не занимались изучением кода, написанного другими людьми, прямо сейчас отложите чтение, найдите какой-нибудь открытый

исходный код и изучите его. Я серьезно! Поищите в Интернете код на вашем любимом языке, написанный известным и признанным специалистом.

Вы вернулись? Хорошо. На чем мы остановились? Ах, да... Я обнаружил, что примеры кода, которые находят отклик в моей душе и кажутся мне красивыми, имеют некоторые общие свойства. Прежде всего, это простота кода. Я считаю, что каким бы сложным ни было приложение или система в целом, отдельные его части должны быть простыми: простые объекты, выполняющие единственную задачу, и в них настолько же простые специализированные методы с хорошо понятными именами. Некоторые считают, что требовать, чтобы методы содержали не более 5–10 строк кода, – это крайность, и в некоторых языках этого очень трудно достичь. Однако мне все же кажется, что такая лаконичность очень желательна.

Вывод таков: красивый код – значит простой код. Все составные части просты, решают простые задачи и связаны простыми отношениями с другими частями системы. Благодаря этому мы можем облегчить сопровождение кода своих систем, а понятный, простой, легко тестируемый код обеспечит высокую скорость разработки на протяжении всего времени существования системы.

Простота – источник и непременный атрибут красоты.

Прежде чем приступать к рефакторингу

Раджит Ammanamty

Рано или поздно каждому программисту приходится выполнять рефакторинг существующего кода. Но прежде чем броситься в бой, поразмыслите о нескольких вещах, которые могут сберечь вам и коллегам уйму времени (и уберечь от головной боли):

- *Лучше всего начинать рефакторинг с оценки состояния существующего в проекте кода и написанных для него тестов.* Так вы сможете выяснить достоинства и недостатки кода в его текущем состоянии, чтобы сохранить его сильные стороны и избежать уже сделанных ошибок. Каждому кажется, что его система будет лучше, чем нынешняя... до тех пор, пока не выяснится, что новый код не лучше, а может, даже хуже, чем предыдущая версия, – и все потому, что мы не стали учиться на ошибках, допущенных в старой системе.
- *Сопротивляйтесь желанию переписать все заново.* Лучше всего повторно использовать как можно больше кода. Каким бы уродливым он ни казался, этот код уже протестировали, прорецензировали и все прочее. Выкинуть старый код, а особенно если он использовался в рабочей системе, – значит выкинуть месяцы (или годы) работы над протестированным и проверенным в боях кодом, который может содержать неведомые вам обходные решения или исправления дефектов. Если не учесть этого, в новом коде могут проявиться те же загадочные ошибки, которые уже были исправлены в старом коде. В результате вы потеряете массу времени и сил, а также знания, копившиеся годами.
- *Множество мелких изменений лучше, чем одно масштабное.* Внося небольшие изменения, легче оценить их воздействие на систему при помощи стандартных каналов обратной связи, таких, например, как тестирование. Грустно видеть, как после внесенного изменения «падает» добрая сотня модульных тестов. Вызванные подобными результатами раздражение и нервозность могут спровоцировать вас на принятие сомнительных решений. Значительно легче справляться, если в каждый момент времени «падают» лишь один-два теста.

- *После каждой итерации разработки важно убедиться, что все имеющиеся тесты успешно отрабатывают.* Если имеющиеся тесты не покрывают внесенные вами изменения, создайте новые тесты. Не выбрасывайте тесты из старого кода бездумно. Внешне может казаться, что некоторые из них не применимы к новой архитектуре системы, но будет совершенно не лишним потратить время и разобраться, с какой целью был создан конкретный тест.
- *Личные предпочтения и самолюбие оставьте в стороне.* Зачем чинить то, что и так работает? Если стиль или организация кода противоречат вашим вкусам, это не является достаточной причиной для рефакторинга. Равно как и ваша уверенность, что вы сможете написать код лучше, чем предыдущий программист.
- *Появление новой технологии – недостаточно веская причина для проведения рефакторинга.* Очень плохо, когда за рефакторинг берутся только потому, что имеющийся код на годы отстал от крутых новейших технологий, и нам кажется, что новый язык или платформа позволят решить задачу намного элегантнее. Код вашей системы лучше всего оставить в покое, за исключением случаев, когда анализ затрат и результатов показывает, что новый язык или платформа могут дать существенный выигрыш в функциональности, простоте сопровождения или производительности.
- *Помните, что люди ошибаются.* Новая структура кода не всегда гарантирует, что новый код будет лучше предыдущего или хотя бы того же качества. Мне приходилось быть свидетелем и участником нескольких провалившихся попыток реорганизации. Приятного было мало, но ведь людям свойственно ошибаться.

Осторожно: общий код

Уди Дахан

Мой первый проект в компании. Я только что защитил диплом, мне не терпится проявить себя, так что я допоздна засиживаюсь на работе и тщательно изучаю существующий код. Работая над своей первой задачей, я применяю все, чему меня учили: комментарии, журналы событий, выделение общего кода в библиотеки (где это возможно), все дела. Мне кажется, что я полностью готов к рецензированию кода, но коллеги словно окатывают меня холодным душем: я получаю за повторное использование кода!

Как так? Во время моей учебы в колледже повторное использование кода превозносилось в числе лучших практик разработки программ. Все прочитанные мною статьи и учебники, наставлявшие меня опытные профессиональные программисты – неужели все они ошибались?

Оказывается, я упустил из виду нечто очень важное.

Контекст.

Тот факт, что две напрямую не связанные части системы пользуются одной и той же логикой, означает меньше, чем мне казалось. До того как я поместил общий код в одну библиотеку, эти части работали независимо одна от другой. Они могли независимо развиваться. Логика каждой из них могла измениться в соответствии с меняющимися требованиями бизнес-среды. Те четыре строчки похожего кода возникли случайно – временная аномалия, совпадение. Ну, то есть это было совпадением, пока за дело не взялся я.

Созданные мною библиотеки общего кода связали шнурки ботинок в один клубок. Любое движение в первой предметной области требовало синхронизации со второй предметной областью. Пока функции были независимы, стоимость их сопровождения была пренебрежимо мала, но как только появилась общая библиотека, объемы необходимого тестирования выросли на порядок.

Сократив абсолютное число строк кода системы, я вместе с тем увеличил количество зависимостей. Контекст этих зависимостей крайне важен: находись они в одном месте, общий код можно было бы оправдать и извлечь из него пользу. Но если не держать эти зависимости под контролем, они запустят свои щупальца в более важные вопросы функционирования системы, даже если сам по себе код с виду безупречен.

Коварство таких ошибок – в правильности основополагающих идей. Подобные приемы применять в должном контексте полезно, а вот в неверном контексте от них больше убытков, чем пользы. Теперь, если я работаю с кодом существующего проекта и не знаю, где могут использоваться различные его части, я гораздо осторожнее подхожу к совместному использованию кода.

Осторожнее с общим кодом. Изучите контекст. И лишь затем действуйте.

Правило бойскаута

Роберт Мартин, известный также как «Дядюшка Боб»

У бойскаутов есть правило: «Оставь после себя лагерь чище, чем он был, когда ты пришел». Если на земле валяется мусор, ты убираешь его, даже если намусорили другие. Ты намеренно улучшаешь условия существования для следующей группы, которая придет в лагерь. (А исходно это правило, установленное основателем скаутского движения Робертом Стивенсоном Смитом Бейден-Пауэллом, звучало так: «Постарайся, чтобы этот мир стал лучше, чем до того, как ты в него пришел».)

Представьте, что мы следуем похожему правилу для нашего кода: «Всегда сохраняй модуль в репозиторий в лучшем состоянии, чем он был, когда ты его оттуда загрузил». Кто бы ни написал этот модуль изначально, что если потратить хоть немного сил, чтобы улучшить его? К чему это может привести?

Думаю, следуй мы все этому простому правилу, нам больше не пришлось бы видеть неумолимую деградацию наших программных систем. Напротив, они становились бы все лучше по мере своего развития. На смену отдельным людям, беспокоящимся лишь о собственных фрагментах работы, пришли бы целые команды, заботящиеся о системах в целом.

Мне кажется, такое правило не слишком сложно в исполнении. Необязательно доводить до совершенства каждый модуль, который вы возвращаете в репозиторий. Просто сделайте его *чуть лучше*, чем он был, когда попал к вам в руки. Естественно, это означает, что, расширяя модули собственным кодом, вы создаете чистый код. Кроме того, нужно навести порядок хотя бы еще в одном месте, прежде чем сохранять модуль. Достаточно лишь дать какой-то переменной более удачное имя или разбить длинную функцию на две более коротких. Можно устранить циклическую зависимость или добавить интерфейс, устраняющий зависимость между политикой и реализацией.

Честно говоря, мне кажется, что это обычные правила приличия – как мыть руки после туалета или выбрасывать мусор в мусорное ведро, а не на пол. Скажем прямо, оставлять беспорядок в коде должно быть столь же социально неприемлемо, как сорить на улице. Такого попросту не следует делать.

Здесь скрыто больше, чем кажется. Одно дело – следить за порядком в собственном коде, и совсем другое – следить за порядком в коде всей команды. Команды помогают одна другой и подчищают код одна за другой. Каждая следует правилу бойскаута, потому что оно приносит пользу всем, а не только одной конкретной команде.

Прежде чем пенять на других, проверь собственный код

Аллан Келли

Разработчику — любому из нас! — часто бывает трудно признать, что его код не работает. Это кажется настолько неправдоподобным, что мы скорее готовы допустить наличие ошибки в компиляторе.

В действительности, очень и очень редко код оказывается неработоспособным из-за ошибки в компиляторе, интерпретаторе, операционной системе, сервере приложений, базе данных, менеджере памяти или любом другом элементе системного программного обеспечения. Да, там встречаются ошибки, но гораздо реже, чем нам хотелось бы думать.

Однажды я действительно столкнулся с ошибкой в компиляторе (удаление переменной цикла при оптимизации), но во всех остальных случаях мои претензии к компилятору или операционной системе оказывались беспочвенными. Я тратил массу своего времени, времени службы поддержки и времени начальства, а в результате оказывался в неловком положении, когда обнаруживалось, что ошибка – моя собственная.

Когда применяемые в проекте средства разработки проверены временем, широко используются и входят в многочисленные технологические цепочки, нет особых оснований сомневаться в их качестве. Конечно, если это одна из ранних версий инструмента, или им пользуются лишь несколько человек в мире, или это редко загружаемый проект с открытым исходным кодом и номером версии 0.1, вполне можно заподозрить этот инструмент. (Точно так же можно с подозрением отнестись к альфа-версии коммерческого инструмента.)

Учитывая, насколько редки ошибки в компиляторах, гораздо выгоднее тратить время и силы на поиск ошибок в собственном коде, а не пытаться доказать, что компилятор ошибается. Тут действуют все обычные соображения по отладке: изолировать проблему, поставить заглушки вместо вызовов, окружить проблемный участок проверками; проверить выполнение соглашений по вызовам, общие библиотеки и номера версий; описать проблему коллеге; выяснить, не поврежден ли стек, и установить соответствие типов переменных; попробовать выполнить

код на разных компьютерах и в разных конфигурациях сборки, например отладочной и окончательной (release).

Подвергайте сомнению собственные допущения и допущения других людей. Допущения в основе работы инструментов разных производителей могут не совпадать; то же верно и для разных инструментов одного и того же производителя.

Если коллега сообщает об ошибке, которую вы не можете воспроизвести, подойдите и посмотрите, как она у него возникает. Его действия или последовательность их выполнения, возможно, никогда не приходили вам в голову.

Моя личная система такова: если я не могу обнаружить ошибку и начинаю грешить на компилятор, пришло время проверить целостность стека. Это особенно полезно, когда при добавлении кода трассировки локализация проблемы меняется.

Многопоточность – еще один источник ошибок, из-за которых программисты начинают орать на компьютер и раньше срока седеют. Все советы писать простой код многократно увеличиваются в цене для многопоточных систем. При поиске таких ошибок трудно системно полагаться на отладку и модульное тестирование, поэтому простота конструкции приобретает первостепенное значение.

Итак, прежде чем забрасывать обвинениями компилятор, вспомните совет Шерлока Холмса – «Если исключить все невозможное, то, что останется, и будет истиной, какой бы неправдоподобной она ни казалась» – и следуйте ему, а не совету Дирка Джентли[1]: «Если исключить все неправдоподобное, то, что останется, и будет истиной, какой бы невозможной она ни казалась».

[1] Дирк Джентли – персонаж Дугласа Адамса, который, как и Шерлок Холмс, занимался детективными расследованиями и держал собственное детективное агентство. – *Прим. ред.*

Тщательно выбирайте инструменты

Джованни Аспрони

Современные приложения крайне редко создают «с чистого листа». Их собирают из уже существующих кубиков – компонентов, библиотек и фреймворков, и тому есть ряд веских причин:

- Объемы, сложность и изощренность приложений растут, а времени на их создание отводится все меньше. Выгоднее тратить время и интеллект разработчиков на код бизнес-логики, чем на код инфраструктуры приложения.
- В широко используемых компонентах и фреймворках меньше шансов столкнуться с ошибками, чем в разработанных самостоятельно.
- Высококачественный инструментарий доступен бесплатно в сети Интернет, благодаря чему снижаются затраты на разработку и упрощается поиск заинтересованных разработчиков с нужным опытом.
- Создание и сопровождение программного обеспечения требует большого объема человеческого труда, поэтому бывает дешевле купить готовые продукты, чем создавать их.

Однако правильный выбор ингредиентов для нового приложения может оказаться сложной задачей, требующей взвешенного подхода. Делая свой выбор, нужно учитывать несколько факторов:

- Каждый инструмент рассчитан на применение в определенном контексте, в который входят инфраструктура, модель управления, модель данных, коммуникационные протоколы и тому подобное, вследствие чего может возникать *несоответствие* инструментов *архитектуре* приложения. Такое несоответствие приводит к необходимости применять «грязные» методы и обходные маршруты при решении задач, что неоправданно усложняет код.
- У каждого инструмента собственный график выхода новых версий, и обновление какого-либо из них может стать чрезвычайно сложной и затратной по времени задачей, поскольку новая функциональность, архитектурные изменения и даже исправления, связанные с ошибками, иногда приводят к несо-

вместимости с другими инструментами. Чем больше в проекте инструментов, тем острее может становиться эта проблема.

- Есть инструменты, требующие серьезной настройки, часто в виде одного или нескольких конфигурационных файлов XML, количество и размер которых может быстро выйти из-под контроля. В итоге приложение выглядит так, будто оно написано на XML, и лишь вдобавок к этому имеется кучка строк кода на одном из языков программирования. Сложность конфигурирования затрудняет сопровождение и дальнейшее развитие приложения.
- Может возникнуть жесткая привязка к определенному производителю (vendor lock-in), когда код, интенсивно использующий его продукты, вдруг оказывается ограничен такими параметрами, как простота сопровождения, производительность, возможность развития, цена и другими.
- Если планируется использовать бесплатное программное обеспечение, может выясниться, что на практике оно не такое уж и бесплатное. Возможно, придется оплачивать коммерческую поддержку этого программного продукта, а она может оказаться недешевой.
- Нужно учитывать условия лицензионных соглашений, даже если это бесплатное программное обеспечение. Например, в некоторых компаниях не допускается использование программного обеспечения с лицензией GNU по причине его вирусной природы, в том смысле, что такая лицензия позволяет распространять новый продукт только совместно с его исходным кодом.

Моя личная стратегия смягчения этих проблем состоит в том, чтобы начинать с малого – лишь с тех инструментов, которые абсолютно необходимы. Как правило, на старте самое главное – устранить необходимость программирования инфраструктуры низкого уровня (и сопутствующих проблем). Скажем, в случае работы над распределенным приложением это достигается посредством использования связующего программного обеспечения (middleware) вместо работы с сокетами напрямую. Затем при необходимости можно добавить другие инструменты. Кроме того, я стараюсь отделить внешние инструменты от объектов моей предметной области с помощью интерфейсов и слоев. Это позволяет с минимальными потерями заменить какой-либо инструмент, если понадобится. Положительный побочный эффект такого подхода – как правило, на выходе я получаю приложение меньшего размера и с меньшим количеством внешних инструментов, чем изначально предполагалось.

Пишите код на языке предметной области

Дэн Норт

Представьте себе текст двух программ. В одной вы встречаете такое:

```
if (portfolioIdsByTraderId.get(trader.getId())
    containsKey(portfolio.getId())) {...}
```

Вы чешете затылок, пытаясь осмыслить, что делает этот код. Похоже, он получает идентификатор объекта трейдера, и по этому идентификатору находится ассоциативный массив в, э-э-э... очевидно, ассоциативном массиве ассоциативных массивов; затем проверяется, есть ли в этом внутреннем массиве еще один идентификатор – из объекта портфеля. Вы снова чешете затылок. Ищете объявление переменной `portfolioIdsByTraderId` и обнаруживаете следующее:

```
Map<int, Map<int, int>> portfolioIdsByTraderId;
```

Понемногу вы начинаете понимать, что это как-то связано с наличием у трейдера доступа к определенному портфелю. Конечно, вы найдете такой же фрагмент поиска – а скорее похожий, но немного отличающийся код, – когда в другом месте понадобится узнать, есть ли у трейдера доступ к некоторому портфелю.

В тексте другой программы вы видите:

```
if (trader.canView(portfolio)) {...}
```

Никаких головоломок. Вам не нужно знать, как объект трейдера определяет доступность портфеля. Где-то в недрах программы, наверное, все же зарыт ассоциативный массив ассоциативных массивов. Но это заботы объекта `trader`, а не ваши.

Внимание, вопрос. Над кодом какой из программ вы предпочли бы работать?

Давным-давно у нас были только самые элементарные структуры данных: биты, байты и символы (на самом деле, те же байты, но мы делали вид, что это буквы и специальные символы). С десятичными числами выходило посложнее, ведь счисление по основанию 10 плохо вписывается в двоичную систему, так что у нас было несколько размеров для чисел с плавающей запятой. Затем появились

массивы и строки (по сути, разновидность массивов). Потом в нашем распоряжении оказались стеки и очереди, хеши, связные списки, списки с пропусками и масса других замечательных структур данных, *которых нет в реальном мире*. Термин «компьютерная наука» тогда означал в основном трудоемкое отображение реального мира на наши ограниченные структуры данных. Настоящие гуру могут даже вспомнить, как именно удавалось решать задачу.

Затем появились пользовательские типы! Ладно, это ни для кого не новость, но они несколько меняют правила игры. Если в вашей предметной области есть такие понятия, как «трейдер» и «портфель», вы можете моделировать их с помощью типов, назначив типам такие имена, как `Trader` и `Portfolio`. Но, что еще важнее, *и отношения между типами* можно моделировать через термины из той же предметной области.

Если вы не используете в коде термины предметной области, значит вы формируете подразумеваемое (читай: секретное) правило, что *вот эта* переменная типа `int` в этом месте обозначает трейдера, а *вон то* `int` в том месте обозначает портфель. (И лучше их не путать!) А если вы реализуете некоторое бизнес-правило («некоторым трейдерам нельзя просматривать некоторые портфели – это незаконно») с помощью нетривиального алгоритма в коде, – например поиска существования значения в ассоциативном массиве, – вы вряд ли облегчаете жизнь ребятам, которые будут проводить аудит и проверку на соответствие законодательству.

Следующему программисту, который будет работать с этим кодом, ваше тайное знание может быть недоступно, так почему не описать все явно? Применение одного ключа для поиска другого, используемого в проверке существования, не слишком очевидная штука. Можно ли рассчитывать, что кто-нибудь догадается, что таким образом реализуются бизнес-правила, препятствующие конфликту интересов?

Явное применение понятий предметной области в коде дает возможность другим программистам понять его назначение со значительно меньшими усилиями, чем при попытках сопоставить алгоритм с тем, что им известно о предметной области. Кроме того, при совершенствовании модели предметной области, которое происходит по мере расширения ваших знаний о ней, вам будет легче дорабатывать код. Если правильно организовать инкапсуляцию, велики шансы, что правило будет располагаться в одном-единственном месте, и вы сможете менять его так, что никакой вызывающий код этого не заметит.

Программист, который спустя несколько месяцев продолжит работу над вашим кодом, будет вам благодарен. И этим программистом можете оказаться вы сами.

Код – это проектирование

Райан Браш

Представьте себе, вы утром просыпаетесь и узнаете, что в строительной промышленности произошел эпохальный прорыв. Теперь миллионы дешевых и невероятно быстрых роботов умеют создавать различные материалы буквально из воздуха, почти не расходуя энергии, и сами себя чинят. Но это еще не все: если есть четкие чертежи, роботы построят по ним здание без всякого вмешательства человека, и стоимость этой работы будет пренебрежимо мала.

Можно представить себе, как это преобразит строительную промышленность, но какие изменения произойдут на более высоком уровне? Как поведут себя архитекторы и проектировщики, когда стоимость строительства станет пренебрежимо мала? Сегодня дорогостоящему строительству обязательно предшествует создание и тщательное тестирование физических и компьютерных моделей. Станем ли мы так утруждать себя, если строительство будет фактически бесплатным? Что за проблема, если здание развалится? Найдем, в чем ошибка, и наши чудо-роботы построят нам новое.

Возможны и другие последствия. С уходом в прошлое моделей не доведенные до конца проекты зданий будут развиваться по мере того, как здание раз за разом строится заново, и проектировщики вносят в него улучшения, имея в виду желаемую конечную цель. Стороннему наблюдателю трудно будет отличить незаконченный проект здания от сданного в эксплуатацию объекта.

Наша способность предсказывать сроки работ уйдет в небытие. Стоимость строительства легче рассчитать, чем стоимость проектирования: мы примерно знаем, сколько стоит установка балки и сколько балок нам нужно. Поскольку доля предсказуемых задач стремится к нулю, преобладающее значение станет иметь трудно предсказуемое время проектирования. Результаты получаются быстрее, но планирование работ становится ненадежным.

Конечно, конкуренция останется действующим фактором. Когда стоимость строительства ничтожна, преимущество на рынке получает компания, способная быстро выполнять проектирование. Основным интересом инженерных контор, таким

образом, станет ускоренное проектирование. Неизбежно произойдет так, что кто-то, не обладая глубоким пониманием проекта, увидит непроверенный вариант, осознает преимущества выхода на рынок с опережением конкурентов и скажет: «Ладно, и так сойдет».

Некоторые проекты, связанные с жизнью и здоровьем людей, будут прорабатываться тщательнее, но во многих случаях покупатели привыкают терпеть неприятности, которые им несет незавершенное проектирование. Ведь компании всегда могут послать своих чудо-роботов и «залатать» проданные ими бракованные здания или автомобили. Все это приводит нас к неожиданному заключению: нашей единственной отправной точкой было резкое снижение стоимости строительства, а в результате мы *получили снижение качества*.

Не стоит удивляться, что рассказанная история стала реальностью в программной индустрии. Если мы согласимся, что основу кода составляет проектирование – творческий, а не механический процесс, – это объясняет *кризис программной отрасли*. У нас *кризис проектирования*: потребность в качественных, проверенных архитектурных решениях для приложений превышает наши способности их создавать. Обстоятельства вынуждают работать на основе незавершенных проектов.

К счастью, эта же модель содержит подсказки, как улучшить положение. Физическое моделирование равносильно автоматизированному тестированию: архитектура приложения не может считаться завершенной, пока она не подверглась суровой проверке набором тестов. Чтобы сделать такие тесты более эффективными, мы находим способы справляться с гигантским числом состояний крупных систем. Совершенствование языков и практик проектирования дает нам надежду. Наконец, есть неоспоримый факт: выдающиеся проекты зданий создаются выдающимися проектировщиками, посвятившими себя овладению своим мастерством. С написанием кода все точно так же.

Важность форматирования кода

Стив Фримен

В незапамятные времена я работал над проектом на языке COBOL, в котором всем участникам запрещалось изменять размер отступа, если не было необходимости изменить код. Все потому, что однажды кто-то что-то сломал – строка кода переползла на следующую и попала в специальные колонки в начале строки. Запрет действовал, даже если форматирование кода вводило в заблуждение, – а такое случалось, – так что приходилось очень внимательно читать код, ведь доверять ему было нельзя. Уверен, убытки от этой политики были гигантскими, потому что она тормозила работу программистов.

Исследователи показали, что у программиста отнимает больше времени перемещение по коду и его чтение (чтобы *найти то место*, которое нужно изменить), чем собственно набор кода, поэтому желательно оптимизировать эти операции. Вот три способа это сделать.

Возможность быстрого просмотра

Зрение человека прекрасно приспособлено для поиска в потоке нужного незначимого (пережиток тех времен, когда нам приходилось бдительно следить, не *появился* ли в саванне лев). Стало быть, я могу облегчить себе жизнь тем, что стандартизирую и уберу на задний план все не связанное с предметной областью – всю «случайную сложность», вносимую большинством коммерческих языков. Если код с одинаковым поведением и выглядит одинаково, моя система восприятия поможет мне быстро находить отличия. Поэтому я также соблюдаю соглашения о размещении членов класса внутри компилируемого файла: константы, поля, открытые методы, закрытые методы.

Выразительная верстка

Все мы научились не жалеть времени на выбор подходящих имен – согласитесь, это приближает наш код к выразительному описанию выполняемых действий и отличает его от простого перечисления шагов. Верстка кода – другая составляющая такой выразительности. Прежде всего необходимо, чтобы вся команда разработки согласилась использовать программу автоматического

форматирования для основных конструкций. Собственные поправки в форматирование кода я могу вносить вручную во время кодирования. Если не возникает острых разногласий, команда быстро приходит к общему стилю, «доведенному вручную». Средство автоматического форматирования не в состоянии понять мои намерения (мне ли не знать, я когда-то сам написал такую программу), а мне более важно, чтобы переносы строк и группировка строк отражали смысл кода, а не синтаксис языка. (Кевин Макгвайр[1] вылечил мою рабскую зависимость от средств автоматического форматирования кода.)

Компактный формат

Чем больше кода умещается на экране, тем больше кода я вижу без разрыва контекста, возникающего при прокрутке текста на экране и при переключении между файлами. Тем меньше информации о контексте мне нужно держать в голове. Длинные комментарии к процедурам и обилие пробелов имели смысл во времена восьмибуквенных имен файлов и построчных принтеров, но сегодня я работаю в интегрированной среде разработки с поддержкой цветной подсветки синтаксиса и перекрестных ссылок. Теперь меня ограничивает разрешение экрана, и каждый его пиксел должен работать таким образом, чтобы облегчить мне понимание кода. Я хочу, чтобы форматирование помогало мне понимать код, и не более того.

Мой друг (непрограммист) однажды заметил, что код похож на стихи. У меня возникает такое же ощущение при виде действительно хорошего кода: каждый фрагмент текста имеет свое значение и помогает мне понять замысел автора. К сожалению, написание кода не слывет таким же романтичным занятием, как сочинение стихотворений.

[1] Кевин Макгвайр Kevin McGuire) – в свое время один из ведущих разработчиков Eclipse, интегрированной среды разработки для Java. – *Прим. перев.*

Рецензирование кода

Маттиас Карлссон

Проводить рецензирование кода (code review) необходимо. Почему? *Оно повышает качество кода и снижает относительную долю дефектов*. Но вы, возможно, неверно представляете себе, почему так происходит.

Многие программисты неприязненно относятся к рецензированию, что бывает связано с их неудачным личным опытом. Мне встречались организации, где весь код проходил формальное рецензирование, прежде чем он мог попасть в систему для коммерческого использования. Часто рецензирование проводит архитектор или ведущий разработчик – практика, которую можно назвать «*архитектор проверяет все*». Это записано в руководстве по процессу разработки программного обеспечения компании, и программисты обязаны подчиняться.

Возможно, в некоторых организациях действительно необходим такой строгий и формальный процесс, но таких меньшинство. В большинстве же организаций подобный подход контрпродуктивен. Авторы рецензируемого кода словно предстают перед комиссией по досрочному освобождению. Рецензирующим требуется успевать и читать код, и отслеживать все особенности эволюционирующей системы: они могут быстро стать узким местом всего процесса, так что процесс вскоре деградирует.

Рецензирование кода должно быть не просто методом исправления ошибок в коде, а средством *распространения знаний* и установления общих основ написания кода. Дав другим программистам познакомиться со своим кодом, вы тем самым делаете каждого совладельцем кода. Пусть любой из участников команды *пройдется по коду* вместе с остальными. Не нужно искать ошибки; при рецензировании кода нужно стараться изучить его и понять, как он работает.

Во время рецензировании кода будьте доброжелательны. Комментарии должны быть *конструктивными*, а *не колкими*. Назначьте *роли* для проведения рецензирования, чтобы не получилось, что на оценку кода влияют отношения старшинства в команде. Например, один рецензент может сосредоточиться на документации, другой – на исключениях, а третий – рассмотреть функциональность.

При таком подходе нагрузка по рецензированию более равномерно распределится между членами команды.

Договоритесь, что определенный день недели будет регулярным *днем рецензирования кода*. Выделите для этого мероприятия пару часов. Каждый раз меняйте автора рецензируемого кода по кругу. Кроме того, не забывайте на каждом собрании по рецензированию менять роли участников рецензирования. *Вовлекайте в рецензирование новичков*. Несмотря на малый опыт, они могут дать вам новую точку зрения благодаря своим свежим университетским знаниям. *Привлекайте экспертов* – они обладают опытом и знаниями. Они быстрее и точнее укажут на код, чреватый ошибками. Рецензирование кода будет проходить легче, если соглашения команды по написанию кода проверяются автоматизированным инструментом. В этом случае на собрании по рецензированию никогда не придется обсуждать форматирование кода.

Пожалуй, в главной мере успех рецензирования определяется тем, будет ли оно *интересно* людям. Рецензирование ориентировано на участвующих в нем людей. Если собрание по рецензированию проходит в неприятной или скучной атмосфере, трудно будет кого-либо мотивировать. Проводите *рецензирование кода в неформальной обстановке*, и пусть главной задачей мероприятия станет распространение знаний среди членов команды. Оставьте в стороне сарказм, а вместо него принесите тортик или бутерброды.

Пиши код с умом

Йехиль Кимхи

Попытки доказать корректность программного обеспечения вручную приводят к формальному доказательству, которое длиннее самого кода и содержит ошибки с большей вероятностью, чем сам код. Желательно применять автоматизированные средства, но это не всегда возможно. Ниже описывается срединный путь: полуформальное доказательство корректности.

Метод основан на разделении исследуемого кода на короткие фрагменты размером от одной строки, которая может содержать вызов функции, до блоков длиной не более 10 строк и обсуждении их корректности. Доказательство должно оказаться достаточно убедительным для вашего коллеги, играющего роль «адвоката дьявола».

Фрагменты следует выбирать таким образом, чтобы в конечной точке блока *состояние программы* (а именно счетчик адреса команд и значения всех «живых» объектов) удовлетворяло простому в описании свойству, а функциональность этого фрагмента (преобразование состояния) легко описывалась в виде одной независимой задачи. Соблюдение предложенных правил упрощает ведение доказательства. Такие свойства конечной точки фрагмента обобщают понятия *предусловий* и *постусловий* для функций, а также *инвариантов* для циклов и классов (в отношении экземпляров классов). Необходимо стремиться, чтобы фрагменты как можно меньше зависели друг от друга, что облегчает доказательство и очень пригодится, если предполагается изменять эти фрагменты.

Многие хорошо известные (хотя и, видимо, реже применяемые) и имеющие статус «качественных» практики написания кода также облегчают проведение доказательств. Таким образом, одно лишь *намерение* провести в будущем доказательство корректности своего кода способствует улучшению его стиля и структуры. Не стоит удивляться, что большинство подобных практик проверяется статическими анализаторами кода:

- Избегайте операторов *goto*, потому что они создают сильную зависимость между фрагментами, разнесенными в коде.

- Избегайте изменяемых глобальных переменных, потому что они делают зависимыми между собой все фрагменты, в которых используются.
- Область видимости каждой переменной должна быть наименьшей из возможных. Например, локальный объект можно объявить непосредственно перед первым использованием.
- Делайте объекты *неизменяемыми* (*immutable*), где это возможно.
- Улучшайте читаемость кода посредством пробелов – как горизонтальных, так и вертикальных. Например, выравнивайте отступы для родственных структур и разделяйте фрагменты кода пустыми строками.
- Пишите самодокументируемый код, выбирая содержательные (но достаточно короткие) имена для объектов, функций, типов и т. д.
- Если фрагмент оказывается вложенным, превратите его в функцию.
- Каждая функция должна решать единственную задачу и быть короткой. *Ограничение длины функции 24 строками*, введенное много лет назад, по-прежнему действует. Размер и разрешение экрана по сравнению с 60-ми годами прошлого века увеличились, но человеческие возможности восприятия остались прежними.
- У функции не должно быть много аргументов (хорошая практика – не более четырех). Это не ограничивает объем передаваемых функции данных: объединение родственных аргументов в одном объекте локализует *инварианты объекта*, что упрощает доказательство в плане проверки согласованности и состояний объектов.
- В общем случае каждая единица кода, начиная с фрагмента и заканчивая целой библиотекой, должна иметь *ограниченный интерфейс*. Сокращение потока информации упрощает доказательство. Это означает, что следует избегать методов, возвращающих *внутреннее состояние* (*getters*). Нужно не запрашивать у объекта информацию для обработки, а требовать, чтобы он выполнил работу с той информацией, которая у него уже есть. Иными словами, *инкапсуляция – это ограниченные интерфейсы, и только они*.
- Чтобы сохранить *инварианты* класса, следует избегать *методов, присваивающих значения* (*setters*). Они часто влекут нарушение инвариантов, определяющих состояния объекта.

Доказательство корректности кода, как и его обсуждение, позволит вам лучше в нем разобраться. Сообщайте о своих открытиях – это пойдет всем на пользу.

Комментарий о комментариях

Кэл Эванс

На моем первом занятии по программированию в колледже преподаватель выдал нам по два бланка для составления текста программы на BASIC. На доске он написал задание: «Составить программу для ввода и вычисления среднего из 10 результатов в боулинге». Затем преподаватель вышел из комнаты. Трудна ли эта задача? Не помню своего решения, но, кажется, там был цикл `FOR/NEXT` и не более 15 строк кода.

В каждом бланке для кода программы – молодым людям, читающим этот текст, поясняю, что мы тогда писали код от руки, прежде чем ввести его в компьютер – было около 70 строк. Мне было совершенно непонятно, почему преподаватель выдал нам по два бланка. Так как почерк у меня всегда был отвратительный, я воспользовался вторым бланком, чтобы аккуратно переписать свой код, надеясь заработать пару очков за стиль.

К большому моему удивлению, получив свою работу обратно на следующем занятии, я обнаружил, что оценка за нее была выставлена чуть выше проходной. (Это стало предзнаменованием всего моего последующего обучения в колледже.) Поверх моего тщательно переписанного кода было выведено: «А комментарии?»

И преподаватель, и я понимали, что делает эта программа, но этого было недостаточно. Часть задачи состояла в том, чтобы научить меня следующему: мой код должен быть понятен другому программисту, который придет после меня. Этот урок я помню до сих пор.

Комментарии – это не порок. Они столь же необходимы в программировании, как основные конструкции ветвлений и циклов. В большинстве современных языков есть средства типа javadoc, которые анализируют написанные в определенном формате комментарии и автоматически составляют справочник по API (интерфейсу прикладного программирования). Иметь такой справочник неплохо, но этого совершенно недостаточно. Код должен содержать пояснения о своем предполагаемом назначении. Когда вы пишете код по древнему принципу «если это было трудно написать, то и читать должно быть не легче», то оказываете мед-

вежью услугу своему клиенту, своему работодателю, своим коллегам, а в будущем и себе.

С другой стороны, не нужно слишком увлекаться комментированием. Следите, чтобы ваши комментарии проясняли код, а не осложняли его чтение. Вставьте в код уместные комментарии, из которых будет ясно, что этот код должен делать. Комментарии в заголовке должны дать любому программисту достаточно информации, чтобы использовать ваш код, не читая его, а комментарии в коде должны помочь тому разработчику, который будет исправлять или расширять код.

На одной моей работе у меня возникло несогласие с проектным решением, принятым «наверху». С язвительной иронией, свойственной молодым программистам, я поместил текст почтового сообщения, содержавшего указания «сверху», в блок комментариев в заголовке файла. Однако оказалось, что менеджеры этой компании лично просматривали код, попадавший в репозиторий. Так я впервые познакомился с термином *шаг, стоивший дальнейшей карьеры*.

Комментируйте только то, о чем не скажет код

Кевлин Хенни

Расхождение между теорией и практикой на практике больше, чем в теории. Это наблюдение определенно применимо к комментариям. В теории общая идея комментирования кода выглядит достойно: дать читателю детальное объяснение происходящего. Что может быть полезнее, чем давать полезное? А вот на практике комментарии часто вредят. Как и любой вид писательского творчества, написание комментариев требует мастерства. Это мастерство в значительной мере включает в себя понимание того, когда комментарии писать не нужно.

Если код написан с нарушениями синтаксиса, то компиляторы, интерпретаторы и другие средства разработки обязательно воспротивятся. Если код некорректен с функциональной точки зрения, большая часть ошибок выявится в результате рецензирования, статического анализа, тестирования и боевого применения на коммерческом предприятии. А что с комментариями? В книге «The Elements of Programming Style»[1] (Computing McGraw-Hill) Керниган и Плоджер замечают, что «неверный комментарий имеет нулевое или отрицательное значение». И все же такие негодные комментарии успешно приживаются в коде на зависть ошибкам всех видов. Они постоянно отвлекают внимание и дезинформируют. Они служат незаметным, но постоянно действующим тормозом мышления программиста.

Что можно сказать о комментариях, которые формально не являются ошибочными, но не повышают ценности кода? Такие комментарии – просто шум. Иногда комментарии лишь повторяют уже сказанное в коде на естественном языке, то есть попугайничают, не сообщая читателю ничего нового; такое повторение не придает коду ни веса, ни правильности. Закомментированный код не выполняется, поэтому он бесполезен как при чтении кода, так и при его выполнении. Кроме того, он очень быстро устаревает. Комментарии относительно номеров версий и блоки закомментированного кода – это попытки решить вопросы контроля

[1] Керниган Б. и Плоджер Ф. «Элементы стиля программирования». – Пер. с англ. – Радио и связь, 1984.

версий и истории кода. Такие вопросы решаются (и гораздо эффективнее) с помощью систем управления версиями.

Засилье в коде бессодержательных и неправильных комментариев провоцирует программистов попросту игнорировать все комментарии, пропуская их при чтении либо выключая их отображение. Программисты – люди изобретательные, и найдут способы обойти все, что покажется им вредоносным: свернут комментарии, изменят цветовую схему так, чтобы комментарии были одного цвета с фоном, или удалят комментарии специально написанным сценарием. Чтобы спасти код от такого неуместного приложения творческих способностей программистов и чтобы снизить риск того, что кто-то пропустит действительно ценные комментарии, следует считать комментарии частью кода. Каждый комментарий должен иметь какую-то ценность для читателя, иначе это просто мусор, который нужно убрать или переработать.

Какой же комментарий можно считать ценным? Только такой комментарий, который сообщает то, чего не говорит и не может сказать код. Если комментарий лишь разъясняет то, что должен самостоятельно сказать фрагмент кода, это указывает на необходимость изменить структуру кода или принятые соглашения по написанию кода, чтобы код говорил за себя сам. Чем комментировать недостаточно точные имена методов и классов, лучше их переименовать. Чем комментировать блоки в длинных функциях, выделите их в маленькие самостоятельные функции, названия которых будут отражать назначения этих блоков. Старайтесь сообщать максимум информации посредством кода. Если вы не можете описать все, что хотелось бы, с помощью одного лишь кода, возможно, тут будет уместен комментарий. Комментируйте то, что *не способен* сказать код, а не просто то, чего код не говорит.

Непрерывное обучение

Клинт Шэнк

Мы живем в интересные времена. Разработка распределена по всему миру, и, как выясняется, очень многие способны выполнять вашу работу. Чтобы сохранить конкурентоспособность на рынке рабочей силы, нужно непрерывно учиться. Иначе вы превратитесь в динозавра, застрявшего на своем рабочем месте, пока в один прекрасный день не окажется, что вы стали не нужны, или что вашу работу отдали туда, где ее готовы делать дешевле.

Как же решать эту задачу? Одни работодатели не скупятся и организуют обучение, развивающее уже нанятых программистов. Другие вообще не могут позволить себе выделить на это время или средства. Самое надежное – самому позаботиться о своем образовании.

Вот перечень способов продолжать учиться. Многое из перечисленного можно бесплатно получить в Интернете:

- Читайте книги, журналы, блоги, ленты Twitter и веб-сайты. Если вы хотите глубже освоить какой-то предмет, можно зарегистрироваться в списке рассылки или группе новостей.
- Если вы хотите по-настоящему изучить новую технологию, необходима практика: напишите немного кода.
- Старайтесь работать с наставником, ведь если вы самый продвинутый специалист компании, это может помешать вашему образованию. Конечно, чему-то научиться можно у любого человека, но гораздо большему вы научитесь у более толкового или более опытного коллеги. Если не можете найти наставника, подумайте о переходе на другое место работы.
- Используйте виртуальных наставников. Ищите в Интернете авторов или разработчиков, которые действительно вам интересны, и читайте все, что они пишут. Подпишитесь на их блоги.
- Изучите программную среду и библиотеки, которыми пользуетесь. Поняв, как работает определенный инструмент, вы сможете более эффективно его применять. Если это инструменты с открытым исходным кодом, вам крупно повезло.

Пройдитесь по коду в отладчике, чтобы узнать, как он устроен. Вы сможете увидеть код, который написали и прошерстили действительно толковые люди.

- Сделав ошибку, разобравшись с дефектом или столкнувшись с проблемой, постарайтесь до конца разобраться в происшедшем. Вполне возможно, что кто-то уже сталкивался с такой проблемой и описал ее в Сети. Google вам в помощь.
- Хороший способ изучить какой-либо предмет – это учить ему других или рассказывать о нем. Если вам предстоит выступать перед другими людьми и отвечать на их вопросы, это даст вам серьезную мотивацию изучить предмет. Попробуйте организовать небольшой семинар для коллег за обедом либо выступить перед специализированной группой пользователей (user group) или на местной конференции.
- Запишитесь в группу изучения какой-либо технологии (типа сообщества для обсуждения вопросов применения паттернов проектирования – patterns community) или сами организуйте такую группу. Это может быть группа изучения языка, технологии или предмета, который вас интересует.
- Участвуйте в конференциях. Если нет возможности поехать на конференцию, в Интернете можно найти бесплатные видеозаписи выступлений, которые публикуют организаторы многих конференций.
- Долго ехать на работу? Слушайте подкасты.
- Вы пользуетесь средствами статистического анализа кода? Читаете предупреждения, выдаваемые вашей IDE? Разберитесь в сообщениях этих программ и причинах их появления.
- Следуйте советам книги «The Pragmatic Programmer: From Journeyman to Master»[1] и каждый год изучайте какой-нибудь новый язык. Или хотя бы новую технологию, или инструмент. Такое горизонтальное расширение знаний даст новые идеи для той связки технологий, которую вы используете сейчас.
- Новые знания не обязательно должны ограничиваться технологиями. Глубже изучите свою предметную область; это позволит лучше понимать технические требования и поможет решать бизнес-задачи. Еще один полезный вариант – изучить способы повышения личной продуктивности.
- Снова поступите в университет.

Было бы замечательно, если бы мы могли, как Нео в «Матрице», просто записать необходимую информацию в свой мозг. Но это невозможно, а потому получение знаний требует времени. Необязательно посвящать учению каждую свободную минуту. Отвести для него немного времени, скажем раз в неделю, – уже лучше, чем ничего. Существует жизнь (как минимум должна существовать) и вне работы.

Технологии меняются быстро. Не отставайте.

---

[1] Э. Хант, Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». – Пер. с англ. – Лори, 2009.

Удобство – не атрибут качества

Грегор Хоп

О важности и сложности проектирования хороших API (интерфейсов прикладного программирования) сказано много. Трудно все сделать правильно с первого раза и еще труднее изменить что-либо в пути; это похоже на воспитание детей. Опытные программисты уже поняли, что хороший API предлагает одинаковый уровень абстракции для всех методов, обладает единообразием и симметрией, а также образует словарь для выразительного языка. Увы, знать принципы – это одно, а следовать им на практике – другое. Вы же знаете, что есть сладкое вредно.

Но перейдем от слов к делу и разберем конкретную «стратегию» проектирования API, которая встречается мне постоянно: проектировать API так, чтобы он был удобным. Как правило, все начинается с одного из следующих «озарений»:

- Почему клиентские классы должны выполнять два вызова, чтобы выполнить одно действие?
- Зачем создавать еще один метод, если он делает почти то же самое, что уже существующий? Добавлю простой `switch`.
- Слушайте, это элементарно: если строка второго параметра оканчивается на «.txt», метод автоматически понимает, что первый параметр является именем файла, так что не нужно создавать два метода.

Намерения благие, но приведенные решения чреваты снижением читаемости кода, использующего ваш API. Следующий вызов метода

```
parser.processNodes(text, false);
```

несет смысловую нагрузку только для того, кто знает, как метод реализован, либо прочел документацию. Этот метод создавался скорее для удобства автора, а не пользователя: «Я не хочу заставлять программиста делать два вызова» на практике означало «Мне не хотелось писать два метода». В принципе нет ничего плохого, если удобство используется как средство против монотонности, громоздкости и неуклюжести. Но если вдуматься, противоядием для этих симптомов служат эффективность, элегантность, последовательность – и не обязательно удобст-

во. API предполагают сокрытие внутренней сложности системы, поэтому вполне резонно ожидать, что проектирование хорошего API требует некоторых усилий. Вполне возможно, что удобнее написать один большой метод, а не тщательно продуманный набор операций, но каким из вариантов проще пользоваться?

В таких ситуациях более удачные архитектурные решения могут основываться на метафоре API как естественного языка. API должен предлагать выразительный язык, обеспечивающий вышележащему уровню словарь, которого достаточно, чтобы задавать полезные вопросы и получать на них ответы. Это не значит, что каждому возможному вопросу будет сопоставлен лишь один метод или глагол. Обширный словарь позволяет передавать оттенки смысла. Например, мы предпочитаем говорить `run` (бежать), а не `walk(true)` (идти), хотя можно рассматривать эти действия как одну и ту же операцию, выполняемую с разной скоростью. Последовательный и хорошо продуманный словарь API способствует выразительности и прозрачности кода более высокого уровня. А что еще важнее – словарь, допускающий сочетания слов, дает возможность другим программистам использовать API способами, которые вы не предвидели, – и это действительно большое удобство для его пользователей! Когда у вас в очередной раз возникнет соблазн сложить в один метод API сразу несколько операций, вспомните, что слова `ПриберикомнатуНеШумиИСделайДомашнееЗадание` нет в словарях, хотя оно пришлось бы весьма кстати для такой популярной операции.

Развертывание приложения: раннее и регулярное

Стив Берчук

Отладку процедуры развертывания (deployment) и установки часто откладывают до этапа завершения проекта. В некоторых проектах создание средств установки возлагается на выпускающего продукт инженера, который воспринимает эту задачу как «неизбежное зло». Промежуточные демонстрации приложения проводятся в специально подготовленной среде, чтобы все работало, как надо. В результате команда не получает опыта работы с процессом и средой развертывания до того момента, когда времени для внесения изменений уже может не остаться.

Процедура развертывания или установки – первое, что видит заказчик, и если она проста, это первый шаг к надежной (или хотя бы простой в отладке) производственной среде. Развертываемое программное обеспечение – это то, чем будет пользоваться клиент. Если вы не обеспечите правильное развертывание приложения, у ваших клиентов появятся вопросы еще до того, как они приступят к плотной работе с вашими программами.

Начиная проект с процедуры установки, вы получаете время на ее совершенствование по ходу цикла разработки продукта и возможность вносить изменения в код приложения с целью облегчить его установку. Периодический запуск и тестирование процедуры установки в чистой среде позволит также проверить, не осталось ли в коде зависимостей от среды разработки или тестирования.

Задвигая развертывание на последнее место, вы можете тем самым усложнить процесс развертывания, ведь вам придется искать обходные маршруты для допущений, сделанных в коде. То, что казалось удачной идеей в IDE, где имеется полный контроль над средой, может значительно усложнить процедуру развертывания. Обо всех компромиссах лучше узнать заранее.

Может казаться, что «способность развернуть приложение» на ранних стадиях не имеет большой ценности для бизнеса по сравнению с возможностью увидеть, как приложение работает на компьютере разработчика. Однако нужно учесть тот простой факт, что реальную ценность для бизнеса представляет лишь конечный

продукт, способный работать в среде клиента. А это в любом случае невозможно на начальном этапе и еще потребует значительной работы. Если вы откладываете создание процедуры развертывания, считая ее тривиальной, следует тем более решить этот вопрос сразу, раз решение настолько дешево. Если же процедура слишком сложная или в ней слишком много неясного, нужно работать с ней так же, как с кодом приложения: экспериментировать, оценивать и переделывать по мере необходимости.

Процедура установки или развертывания критически важна для успешной работы ваших клиентов или вашей сервисной команды, поэтому необходимо постоянно тестировать и совершенствовать ее. Тестирование и совершенствование исходного кода происходят на всем протяжении проекта. Развертывание приложения заслуживает такого же отношения.

Отличайте исключения в бизнес-логике от технических

Дэн Берг Джонссон

Есть, в общем-то, две главные причины возникновения ошибок времени выполнения (runtime errors): технические проблемы, препятствующие работе приложения, и бизнес-логика, препятствующая использованию приложения неверным способом. Большинство современных языков, таких как LISP, Java, Smalltalk и C#, сигнализируют о возникновении ситуаций обоих типов при помощи исключений. Но эти две ситуации настолько различны, что их следует тщательно разделять. Представлять их посредством единой иерархии исключений – не говоря уже об одинаковых классах исключений – значит создавать возможность путаницы в будущем.

Ошибка программирования может породить неразрешимую техническую проблему. Например, если пытаться получить 83-й элемент массива размером 17 элементов, программа сойдет с рельс и сгенерирует исключение. Более тонкий вариант – вызов кода библиотеки с недопустимыми аргументами, приводящий к тому же результату, но внутри библиотеки.

Было бы ошибкой пытаться на месте разрешать все подобные ситуации, возникшие по нашей вине. Вместо этого мы даем возможность исключениям всплыть на самый верхний уровень архитектуры, где некий общий механизм обработки исключений сделает все возможное, чтобы привести систему в безопасное состояние, например откатит транзакцию, отразит событие в файле журнала и известит администратора, а также вежливо сообщит о произошедшем пользователю.

Вариант этой ситуации встречается при обращении к вашей библиотеке, когда вызывающий нарушил контракт метода, например, передав совершенно неприемлемый аргумент или не настроив должным образом связанный объект. Это ситуация того же типа, что с обращением к 83-му элементу из 17: вызывающий должен был проверить, и это ошибка программиста на стороне клиентского кода. Правильная реакция – генерировать техническое исключение.

Иная, но тоже техническая ситуация возникает, когда программа не может продолжить работу из-за проблем с окружением, например из-за недоступности базы

данных. В такой ситуации следует предположить, что инфраструктура сделала все возможное, чтобы решить проблему – попыталась восстановить соединение должное число раз, – и не справилась. Даже если истинная причина в другом, положение для вызывающего кода аналогичное: он мало чем может исправить ситуацию. Поэтому мы сообщаем о ситуации посредством исключения, которое дойдет до универсального механизма обработки исключений.

С другой стороны, случаются ситуации, когда вызов не может быть завершен по логике предметной области. Это исключительная ситуация, иначе говоря, необычная и нежелательная, но в ней нет странности или ошибки программирования (примером может служить попытка снять со счета больше средств, чем на нем находится). Иными словами, такая ситуация является частью контракта, а генерация исключения – это *альтернативный маршрут возврата*, часть модели, о которой клиентский код должен знать и которую должен быть готов обработать. Для таких случаев рекомендуется создать отдельное исключение или отдельную иерархию исключений, чтобы клиентский код мог обработать ситуацию особым образом.

Объединение технических исключений и исключений бизнес-логики в одну иерархию размывает различия между ними и может запутать вызывающего относительно контракта метода, предусловий вызова и ситуаций, которые должны обрабатываться. Разделение этих случаев придает ясности и повышает вероятность того, что технические исключения будут обрабатываться стандартными механизмами каркаса приложения, а исключения предметной области будут рассмотрены и обработаны клиентским кодом.

Больше осознанной практики

Джон Джаггер

Осознанная практика – это не просто выполнение задания. Если на вопрос «Зачем я выполняю задание?» вы отвечаете: «Чтобы выполнить это задание», это не осознанная практика.

Осознанная практика нужна для того, чтобы улучшить ваши способности к выполнению задачи. Цель – повышение мастерства и техники. Осознанная практика – это повторение. Осознанная практика – это решение задачи с целью повышения мастерства в одном или нескольких аспектах задачи. Осознанная практика – это повторы повторений. Вы продвигаетесь медленно, выполняя задачу снова и снова, пока не достигнете желаемого уровня мастерства. Осознанная практика выполняется, чтобы овладеть способами решения задания, а не для того, чтобы его выполнить.

Главная цель коммерческой разработки – конечный продукт, а главная цель осознанной практики – повышение эффективности труда. Это разные вещи. Прикиньте, сколько времени вы тратите на работу над чужим проектом? А сколько времени на работу над собой?

Сколько осознанной тренировки необходимо для приобретения мастерства?

- Питер Норвиг (Peter Norvig) пишет[1], что «возможно, 10000 часов... – это и есть то самое магическое число».
- В книге «Leading Lean Software Development: Results Are not the Point» (Addison-Wesley Professional, 2009) Мэри Поппендик (Mary Poppendieck) пишет, что «разработчикам, достигшим высшего уровня производительности, требуется не менее 10000 часов целенаправленной тренировки, чтобы стать экспертами».

Мастерство растет постепенно, а не возникает скачком после 10-тысячного часа! Однако 10000 часов – это серьезно: примерно 20 часов в неделю в течение 10 лет. Требуется такое упорство, что вы можете усомниться, получится ли из вас эксперт.

---

[1] *http://norvig.com/21-days.html*

Получится. Величие есть по преимуществу вопрос сознательного выбора. Вашего выбора. Исследования последних двадцати лет показывают, что главным фактором приобретения компетенции является время, потраченное на целенаправленную тренировку. Врожденные способности – не главный фактор. Вот что пишет Мэри Поппендик:

> Многие исследователи высших профессиональных достижений сходятся в том, что врожденный талант является всего лишь фиксированной отправной точкой: нужны какие-то минимальные природные способности, чтобы начать заниматься спортом или определенной профессиональной деятельностью. Начиная с этого порогового значения, преуспевают те, кто трудится упорнее всего.

Нет смысла осознанно практиковаться в том, что вы и так умеете делать мастерски. Осознанной тренировкой мы развиваем то, что умеем недостаточно хорошо. Питер Норвиг пишет:

> Ключ [к достижению мастерства] – *осознанная* практика: не просто многократное повторение одного и того же, но смелость взяться за задачу, которая несколько превышает ваши нынешние способности, попытаться ее решить, анализировать эффективность своих действий во время и после работы над решением, а также исправить допущенные ошибки.

А Мэри Поппендик пишет:

> Осознанная практика – это не повторение того, что вы уже умеете; это выбор сложной задачи, попытка заняться тем, в чем вы не вполне компетентны. Нельзя рассчитывать, что это будет приятное времяпрепровождение.

Осознанная практика – это учеба, которая изменяет вас, изменяет ваше поведение. Удачи.

Предметно-ориентированные языки

Микаэль Хунгер

Если прислушаться к разговору экспертов в какой-либо области, будь то игроки в шахматы, воспитатели детского сада или страховые агенты, можно заметить, что их лексика существенно отличается от повседневной. Отчасти такова причина появления предметно-ориентированных языков (domain specific language, DSL): у каждой предметной области есть собственный специализированный словарь для описания явлений, присущих этой области.

Если говорить о программировании, DSL представляют собой выполняемые выражения на языке, присущем предметной области. Выражения языка строятся на ограниченном словаре и грамматике, так что специалисты в данной предметной области способны читать и понимать выражения на этом языке, а в идеале еще и писать собственные. Языки DSL, ориентированные на разработчиков и ученых, существуют уже довольно давно. Достаточно древними примерами могут послужить «малые языки» настроечных файлов UNIX, а также языки на базе мощных макросов LISP.

Обычно DSL делятся на *встроенные* и *независимые*:

Встроенные DSL

Создаются на универсальных языках программирования, синтаксис которых подогнан под структуры естественного языка. Проще всего делать это с языками, предоставляющими широкие возможности для синтаксического украшательства и гибкого форматирования (например, Ruby и Scala), тогда как с другими все сложнее (например, Java). Большинство встроенных DSL – суть обертки существующих API, библиотек и бизнес-логики. Они снижают входной порог применения уже существующей функциональности. Приложения на встроенных DSL можно запускать как обычные приложения. В зависимости от реализации и предметной области они могут использоваться для наполнения структур данных, описания зависимостей, запуска процессов или задач, сообщения с другими системами или проверки корректности вводимых пользователями данных. Синтаксис встроенного DSL ограничен возможностями

базового языка. Существует множество шаблонов – например построитель выражений, цепочка методов, аннотация – для подгонки базового языка к нужному DSL. Если базовый язык не требует перекомпиляции, встроенный DSL при тесном взаимодействии с экспертом в предметной области можно создать достаточно быстро.

Независимые DSL

Представляют собой текстовое или графическое описание языка, причем текстовые DSL встречаются чаще графических. Текстовые выражения могут проходить обработку в цепочке, включающей в себя лексический анализатор, анализатор синтаксиса, преобразователь модели, генераторы и любые другие средства постобработки. Как правило, выражения не независимых DSL преобразуются во внутренние модели, служащие основой для дальнейшей обработки. Полезно определить грамматику (например, в виде РФБН[1]). Грамматика служит отправным пунктом для создания элементов инструментальной цепочки (например, редактора, визуализатора, генератора анализаторов синтаксиса). Для простых DSL может оказаться достаточно анализатора синтаксиса, созданного вручную – например, на основе регулярных выражений. Если требования к анализатору синтаксиса достаточно сложны, созданный вручную анализатор синтаксиса может стать слишком громоздким, и следует обратить взор на такие инструменты для работы с грамматиками и DSL, как openArchitectureWare, ANTLR, SableCC, AndroMDA. Довольно часто независимые DSL определяют в виде диалектов XML, но с чтением в этом случае могут быть сложности, особенно у неспециалистов.

Всегда следует принимать во внимание целевую аудиторию вашего DSL. Из кого она состоит – из разработчиков, менеджеров, клиентов или конечных пользователей? В зависимости от целевой аудитории нужно выбирать технический уровень языка, доступные пользователю функции, инструмент для подсказки по синтаксису (например, IntelliSense), средства ранней валидации, визуализации и представления. Скрывая технические детали, DSL отдает власть пользователям, предоставляя им возможность адаптировать системы к собственным потребностям, не прибегая к помощи разработчиков. DSL позволяет еще и увеличить скорость разработки благодаря распределению задач после того, как создан начальный каркас языка. Язык может развиваться постепенно. Существуют также различные способы перевести существующие языки и грамматики на новый DSL.

[1] Расширенная форма Бэкуса-Наура (Extended Backus-Naur Form, EBNF).– *Прим. ред.*

Не бойтесь что-нибудь сломать

Майк Льюис

Каждый поработавший в нашей отрасли наверняка встречался с проектом, код которого внушал опасения. Части такой системы сильно взаимосвязаны, и изменение кода одной функции почему-то приводит к нарушению работы совсем другой. При добавлении нового модуля приходится ограничиваться минимальными изменениями и, затаив дыхание, ждать последствий. Это все равно что играть в *дженгу* перекрытиями небоскреба – однозначно ведет к катастрофе.

Внесение изменений так изматывает нервы только потому, что система больна. Она нуждается в лечении, иначе ее состояние будет лишь ухудшаться. Вы знаете, в чем пороки системы, но боитесь принять решительные меры. Опытный хирург знает, что необходимо сделать разрезы, чтобы провести операцию, но он знает также, что разрезы временные и потом заживут. Результат операции оправдывает перенесенные страдания, и состояние пациента должно стать лучше, чем до хирургического вмешательства.

Не бойтесь своего кода. Кому какое дело, что код не работает в процессе работы над ним? Именно боязнь перемен и привела ваш проект к нынешнему жалкому состоянию. Потраченное на рефакторинг время многократно окупится в течение жизненного цикла вашего проекта. Да к тому же переработка нездоровой системы сделает всех участников команды специалистами в ее устройстве. Такой опыт нужно ценить, а не жаловаться на него. А вот работа над системой, постоянно вызывающей тошноту, не лучший выбор в жизни.

Переопределите внутренние интерфейсы. Реструктурируйте модули. Реорганизуйте код, полученный путем копирования-вставки. Упростите архитектуру, сократив число зависимостей. Сложность кода можно существенно снизить, устранив из него патологические случаи, а они часто возникают из-за неправильно организованной взаимосвязи между частями системы. Медленно переходите от старой структуры к новой, не забывая в процессе о тестировании. Попытка осуществить рефакторинг «в один заход» может вызвать столько проблем, что возникнет сомнение в целесообразности переработки вообще.

Станьте хирургом, смело удаляющим пораженные ткани во имя исцеления. Такой подход заразителен и вдохновит ваших коллег к проведению давно откладывавшейся зачистки в других проектах. Ведите список «гигиенических» работ, которые, по мнению команды, принесли бы пользу проекту. Убедите руководство в том, что, несмотря на отсутствие видимых результатов, такие работы сокращают издержки и ускоряют выпуск новых версий. Постоянно проявляйте заботу о «здоровье» кода в целом.

Не прикалывайтесь с тестовыми данными

Род Бегби

Был поздний вечер. Я подбирал фиктивные данные, чтобы протестировать верстку страницы, находившейся в работе. Имена пользователей я придумал в честь участников группы «The Clash». Названия компаний? Сгодятся названия песен «Sex Pistols». Теперь нужны какие-нибудь биржевые символы – четырехбуквенные слова, состоящие из заглавных букв.

Я выбрал те самые четырехбуквенные слова[1].

Вроде вполне безобидно. Просто небольшое развлечение для меня или тех разработчиков, которые увидят это завтра, пока я не подключу реальный источник данных.

А на следующее утро руководитель проекта сделал несколько снимков экрана для презентации.

История программирования пестрит множеством подобных воспоминаний. Шалости разработчиков и дизайнеров, которые «никто никогда не увидит», вдруг представали всеобщему обозрению.

Важен не вид утечки, а то, что результаты бывают смертельными для отдельных лиц, команд или компаний. Вот некоторые примеры:

- Во время показа текущей версии клиент щелкает по кнопке, функциональность которой еще не реализована, и получает сообщение: «Придурок, не трогай эту кнопку».
- Программист, сопровождавший старую систему, получил задание добавить диалоговое сообщение об ошибке и решил вывести то, что записывается во внутренний журнал. И вот у пользователей при ошибках появляются сообщения типа «Твою мать, Бэтмен, сбой транзакции БД!».

---

[1] Следовало бы перевести «трехбуквенные» слова, но биржевые символы содержат 4 буквы, и автор имеет в виду соответствующие четырехбуквенные слова на английском. – *Прим. ред.*

- Человек перепутал интерфейс для ввода тестовых данных и интерфейс администрирования рабочей системы. Он вводит «прикольные» данные, и вот покупатели видят на витрине вашего интернет-магазина «Фаллоимитатор в виде Билла Гейтса» ценой 1000000 $.

Если старую поговорку «пока правда обувается, ложь уже обойдет полсвета» перевести на современный язык, то можно сказать, что в наше время такие ляпы прославятся в Twitter, LiveJournal или Facebook до того, как в часовом поясе разработчиков проснется кто-то, способный их исправить.

Вполне возможно, что и ваш исходный код не избежит пристального интереса. Когда в 2004 архивы с исходным кодом Windows 2000 появились в файлообменных сетях, некоторые весельчаки профильтровали их в поисках сквернословия, ругательств и прочих забавных вещей.[1] (Признаюсь, с той поры я иногда прибегаю к комментарию: `// TERRIBLE HORRIBLE NO GOOD VERY BAD HACK` (УЖАСНО ОТВРАТИТЕЛЬНО БЕСПОЛЕЗНО ДРЯННОЙ ТРЮК!)

Короче, когда вы пишете в коде любой текст, будь то комментарии, вывод в журнал, диалоговое окно или тестовые данные, думайте о том, как это будет выглядеть, если текст попадет в открытый доступ. И тогда никому не придется краснеть.

[1] *http://www.kuro5hin.org/story/2004/2/15/71552/7795*

Не проходите мимо ошибки!

Пит Гудлиф

Как-то вечером я шел по улице на встречу с друзьями в баре. Мы давненько не пили вместе пиво, я спешил и думал только о встрече с ними. Поэтому я плохо смотрел себе под ноги. Споткнувшись о край тротуара, я упал плашмя, но решил не обращать на это внимания.

Нога болела, но я же спешил на встречу с друзьями! Я собрался с силами и двинулся дальше. По ходу дела боль усиливалась. Поначалу я думал, что просто проходит болевой шок, но вскоре понял, что дело серьезнее.

И все же я направился в сторону бара. Когда я добрался туда, мне было уже невыносимо больно. Вечер не особенно удался – боль не давала о себе забыть. Утром я пошел к доктору, и выяснилось, что у меня сломана голень. Остановись я, когда стало больно, мне удалось бы избежать многих осложнений, вызванных тем, что я продолжал идти дальше. Предположу, то было мое худшее в жизни утреннее похмелье.

Слишком многие программисты пишут код в духе моего поведения в тот вечер.

Ошибка? Какая еще ошибка? Это просто ерунда. Честно. Можно не обращать на нее внимания. Для создания надежного кода такая стратегия *не годится*. Скажу больше, это попросту лень. (Вредная ее разновидность.) Что бы вы там себе ни воображали по поводу отсутствия ошибок в собственном коде, их всегда нужно проверять и всегда обрабатывать. Обязательно. Поступая иначе, вы не экономите время, но запасаетесь потенциальными проблемами на будущее.

Сообщить об ошибке в коде можно разными способами, например:

- Вернуть в результате работы функции **код ошибки**, который означает, что функция отработала неверно. Код ошибки очень легко игнорировать. В коде ничто не укажет на проблему. Вообще теперь стало принято игнорировать значения, возвращаемые некоторыми стандартными функциями C. Часто ли вы проверяете значение, возвращаемое функцией `printf`?
- `errno` – любопытный маразм языка C, специальная глобальная переменная, которая должна сигнализировать об ошибке. Ее легко игнорировать, ее неудобно применять и она служит источником всяческих неприятностей: что, напри-

мер, делать, если в программе несколько потоков, вызывающих одну и ту же функцию? Одни платформы помогают избежать проблем в таких ситуациях, другие – нет.

- **Исключения** – это более организованный способ уведомлять об ошибках и обрабатывать их, причем способ, встроенный в язык. Игнорировать исключения невозможно. Или все же можно? Мне часто встречался код типа

```
try {
    // ...что-то выполнить...
}
catch (...) {} // игнорировать ошибки
```

Единственное достоинство этой ужасной конструкции – подчеркнуть, что в коде происходит нечто сомнительное с точки зрения морали.

Игнорируя ошибку, закрывая глаза и делая вид, что ничего не произошло, вы подвергаетесь большому риску. Как моя нога сильнее пострадала потому, что я не остановился немедленно, так и движение напролом через красные флажки может привести к очень серьезным проблемам. Проблемы нужно решать при первой же возможности. Не затягивайте выплату долгов.

Если не обрабатывать ошибки, вот что вы получите:

- **Нестабильный код.** Код со множеством восхитительных и трудных для обнаружения ошибок.
- **Небезопасный код.** Взломщики часто пользуются недостатками обработки ошибок для взлома программных систем.
- **Плохая структура.** Если в вашем коде постоянно проявляются ошибки, которые трудно обрабатывать, то, вероятно, у вас неудачный интерфейс. Формулируйте его таким образом, чтобы ошибки становились менее назойливыми, а их обработка не была столь обременительна.

Следует не только проверять все возможные ошибки в коде, но и обнажать все потенциально ошибочные состояния в интерфейсах. Не скрывайте их – нет смысла притворяться, что ваши службы всегда будут работоспособны.

Почему мы не выполняем проверку на ошибки? Вот некоторые стандартные оправдания:

- Обработка ошибок загромождает код, затрудняя его чтение и отслеживание «нормального» потока выполнения.
- Это лишняя работа, а на меня давят сроки сдачи.
- Я уверен, что эта функция никогда не вернет ошибку (`printf` всегда работает, `malloc` всегда выделяет память, а если нет, наши проблемы гораздо серьезнее...)
- Это прототип, нет смысла делать его пригодным для коммерческого применения.

С какими из них вы согласны? Что вы могли бы на них возразить?

Не просто учите язык, поймите его культуру

Андерс Норас

В средней школе мне пришлось изучать иностранный язык. В то время мне казалось, что достаточно хорошего знания английского языка, и потому я благополучно проспал три года уроков французского. Спустя несколько лет я поехал отдыхать в Тунис. Официальный язык этой страны – арабский, но так как это бывшая французская колония, в Тунисе широко распространен французский. По-английски говорят только в местах пребывания туристов. Из-за своего невежества в языках мне пришлось проводить время главным образом близ бассейна за чтением книги «Finnegans Wake» (Поминки по Финнегану), в которой Джеймс Джойс продемонстрировал свое владение как формой, так и языком. Джойс играючи соединил вместе более 40 языков, и чтение книги стало для меня удивительным, хотя и непростым опытом. Это переживание новых средств выражения, сотканных автором из иностранных слов и фраз, осталось со мной на протяжении всей моей карьеры программиста.

В своей новаторской книге «The Pragmatic Programmer: From Journeyman to Master»[1] (Addison-Wesley Professional) Энди Хант и Дэйв Томас рекомендуют каждый год изучать какой-нибудь новый язык программирования. Я попробовал последовать их совету и за годы работы приобрел опыт программирования на многих языках. Самый важный вывод из моих приключений полиглота: для изучения языка недостаточно освоить его синтаксис, нужно понять его культуру.

Можно на любом языке писать, как на Фортране, но, чтобы по-настоящему изучить язык, нужно принять его полностью.

Не ищите себе оправданий, если ваш код C# состоит из длинного метода `Main` и преимущественно *статических* вспомогательных методов. Лучше разберитесь, почему имеет смысл применять классы. Не избегайте малопонятных лямбда-выражений в функциональных языках, заставьте себя пользоваться ими.

---

[1] Э. Хант, Д. Томас «Программист-прагматик. Путь от подмастерья к мастеру». – Пер. с англ. – Лори, 2009.

Овладев приемами нового языка, вы с удивлением обнаружите, что по-новому пользуетесь теми языками, которые знали раньше.

Я научился эффективно использовать делегирование в C# после того, как освоил Ruby; раскрытие всех возможностей обобщений .NET (generics) навело меня на мысли о том, как с большей пользой применять обобщения в Java; после LINQ мне было легко изучать Scala.

Переход с одного языка на другой помогает также лучше понять шаблоны проектирования. Программисты на C обнаруживают, что в C# и Java одинаково широко употребляется шаблон итератора. В Ruby и других динамических языках можно все еще пользоваться шаблоном посетителя (visitor), но ваша реализация не будет похожа на пример из книги Банды Четырех (The Gang of Four).

Одни утверждают, что «Поминки по Финнегану» невозможно читать, другие восторгаются стилистической красотой книги. Чтобы облегчить чтение книги, были сделаны ее моноязычные переводы. Забавно, что первый перевод был французским.

С кодом ситуация во многом аналогична. Если писать код в стиле «Поминок», чтобы в нем было чуть-чуть Python, немного Java и примесь Erlang, проект превращается в месиво. Но если вы изучаете новые языки, чтобы расширить свой кругозор и встретить новые идеи для решения задач разными способами, вы обнаружите, что код, который вы пишете на старом проверенном языке, становится более красивым с каждым новым изученным языком.

Не прибивайте программу гвоздями к стене

Верити Стоб

Однажды я написала опрос-розыгрыш по C++, в котором в шутку предложила такую стратегию обработки исключений:

> Путем размещения в коде многочисленных конструкций `try...catch` иногда удается избежать аварийного завершения программы. О полученном в результате состоянии можно сказать, что «тело прибито в вертикальном положении».

Несмотря на легкомыслие, я всего лишь излагала урок, извлеченный из Его Величества Горького Опыта.

То был базовый класс приложения в нашей самописной библиотеке C++. В коде класса многие годы ковырялись шаловливыми ручонками то один программист, то другой. Класс содержал код обработки исключений, ускользнувших от всех других обработчиков. Взяв пример с Йоссариана из «Уловки-22», мы решили или, скорее, нам показалось (слово «решили» предполагает больше мыслительных усилий, чем ушло на создание этого монстра), что экземпляр этого класса должен жить вечно или умереть в попытках это сделать.

С этой целью мы сплели воедино множество обработчиков исключений. Мы смешали обработку структурированных исключений Windows с собственными исключениями (помните `__try...__except` в C++? Я тоже не помню). Когда неожиданно возникало исключение, мы вызывали методы снова, запихивая в них те же параметры. Когда я вспоминаю это, мне нравится думать, что, создавая вложенный обработчик `try...catch` внутри предложения `catch` другого обработчика, я испытывала смутное подозрение, что как-то случайно съехала с надежного шоссе хорошей практики на ароматную, но нездоровую дорогу к безумию. Впрочем, это я понимаю, скорее, задним умом.

Стоит ли говорить, что, когда возникали проблемы в приложениях, основанных на этом классе, приложения исчезали бесследно, словно жертвы мафии, сброшенные с причала. Не оставалось даже пузырей на воде в качестве подсказки о печальном происшествии, несмотря на наличие подпрограмм аварийной регистрации, отвечавших за протоколирование события. В конце концов – по прошествии

немалого времени – мы критически переоценили свое творение и устыдились. Все это месиво мы заменили маленьким и надежным механизмом генерации отчета. Но тому предшествовал не один десяток критических сбоев приложения.

Я не стала бы отнимать у вас время этой историей, ибо трудно представить, что кто-то еще мог бы соперничать с нашей глупостью, но недавно в сети у меня случилась дискуссия с типом, академический чин которого предполагает большее понимание подобных вопросов. Обсуждался код на Java в удаленных транзакциях. Тип утверждал, что если в коде случился сбой, исключение должно быть перехвачено и блокировано на месте. («И *что же* с ним тогда делать? – спросила я. – Приготовить его на ужин?»)

Он процитировал правило проектировщиков пользовательских интерфейсов: ПОЛЬЗОВАТЕЛЬ НИКОГДА НЕ ДОЛЖЕН ВИДЕТЬ СООБЩЕНИЕ ОБ ИСКЛЮЧИТЕЛЬНОЙ СИТУАЦИИ, как будто это правило решает проблему, напиши его прописными буквами или как-то еще. Интересно, может, это он автор кода для банкоматов, прославившихся своими синими экранами смерти в некоторых чахлых блогах, и не получил ли он душевную травму на всю жизнь.

Во всяком случае, если вам доведется встретиться с ним, кивайте, улыбайтесь и не слушайте, пока будете бочком пробираться к двери.

Не полагайтесь на «автоматические чудеса»

Алан Гриффитс

Если взглянуть на любую деятельность, процесс или дисциплину с достаточно большого расстояния, все вроде бы просто. Менеджеры, у которых нет опыта разработки программ, считают, что работа программиста проста, а программисты, у которых нет опыта руководства, точно так же относятся к работе менеджера.

Программирование – некоторое занятие, на которое тратят некоторую часть своего времени некоторые люди. А самая трудная часть работы – мыслительный процесс – менее всего заметна и менее всего ценится непосвященными. Десятилетиями предпринимаются попытки избавиться от потребности в квалифицированных мыслящих кадрах. Одна из наиболее ранних и памятных попыток – усилия Грэйс Хоппер (Grace Hopper) сделать языки программирования не такими таинственными. Благодаря этому, как предсказывали некоторые, потребность в программистах узкой специализации может исчезнуть. Результат (COBOL) способствовал материальному благополучию многих программистов узкой специализации в последующие десятилетия.

Навязчивая идея, будто разработку программного обеспечения можно упростить, если исключить из нее программирование, представляется совершенно наивной программисту, который понимает, о чем речь. Но ход мыслей, приводящий к этой ошибке, заложен в человеческой природе, и программисты подвержены этой ошибке так же, как и все остальные.

В любом проекте найдется множество вещей, с которыми отдельный программист может не соприкасаться тесно: получение технических требований от пользователей, утверждение бюджета, настройка сервера сборки, развертывание приложения в тестовой или производственной среде, перевод бизнеса на новые процессы или программы и тому подобное.

Если вы не заняты активно в каком-то виде деятельности, возникает подсознательное стремление считать, что он прост и происходит «по какому-то волшебству сам по себе». Пока длится это волшебство, все хорошо. Но когда (обычно уместно

именно слово «когда», а не «если») волшебство прекращается, проект сталкивается с неприятностями.

Мне встречались проекты, в которых разработчики бесплодно тратили недели своего времени, поскольку никто не осознавал, что их работа зависит от того, «правильная» ли версия DLL загружена. Когда начинались периодические сбои, проверялись все возможные причины, пока кто-то не обращал внимание, что загружалась «неверная» версия DLL.

В другом подразделении все шло гладко: проекты завершались вовремя, без отладок до глубокой ночи и исправлений в последний момент. Настолько гладко, что высшее руководство решило, что все работает «само собой» и можно обойтись без руководителя проекта. Через полгода проекты в этом подразделении стали идти, как везде: с задержками, ошибками и непрерывными заплатками.

Не нужно разбираться во всех «волшебствах», благодаря которым работает ваш проект, но понимать их хотя бы частично не повредит, как и ценить тех, кто разбирается в некоторых вещах лучше вас.

Самое главное, если волшебство вдруг перестанет работать, нужно знать, как его возобновить.

Не повторяй свой код

Стив Смит

«Don't Repeat Yourself (DRY)» («не повторяйся») является, вероятно, одним из главнейших принципов программирования. Этот принцип сформулировали Энди Хант и Дэйв Томас в книге «Программист-прагматик», и он лежит в основе многих других широко известных правильных подходов и шаблонов проектирования в разработке программного обеспечения. Разработчик, научившийся распознавать дублирование и умеющий устранять его с помощью надлежащих приемов и подходящей абстракции, способен писать гораздо более понятный код, чем тот, кто постоянно загрязняет приложение ненужными повторениями.

Дублирование кода – это балласт

Каждая строка кода приложения требует сопровождения, и она служит источником возможных ошибок в будущем. Дублирование приводит к ненужному увеличению объема кода, что повышает вероятность появления ошибок и делает систему излишне сложной. Увеличение объема системы из-за дублирования, во-первых, мешает разработчикам полностью разобраться в системе, а во-вторых, не позволяет гарантировать, что изменения, внесенные в одном месте, не потребуется повторить в других местах, где дублируется эта логика. Принцип DRY требует, чтобы «каждый элемент информации имел в системе единственное, однозначное и надежное представление».

Повторение в процессах указывает на необходимость автоматизации

Многие процессы в разработке программного обеспечения многократно повторяются и легко автоматизируются. Принцип DRY применим как к исходному коду приложения, так и в подобных контекстах. Ручное тестирование проходит медленно, подвержено ошибкам и повторять его трудно, поэтому по возможности следует применять *автоматизированные наборы тестов* (*test suites*). Интеграция приложения вручную занимает много времени и подвержена ошибкам, поэтому процесс сборки нужно выполнять как можно чаще – желательно после

каждой записи кода в репозиторий. Там, где есть трудоемкие ручные процессы, которые можно автоматизировать, их нужно автоматизировать и стандартизировать. Цель – обеспечить наличие единственного способа решения задачи и сделать его как можно менее трудоемким.

Повторение в логике указывает на необходимость абстракции

Повторения в логике бывают разных видов. Очень легко обнаруживаются и исправляются случаи копирования и вставки конструкций *if-then* и *switch-case*. Многие шаблоны проектирования явным образом направлены на сокращение или удаление дублирования логики в приложении. Если объект требует выполнения некоторых действий, прежде чем его можно использовать, полезно применить шаблон Abstract Factory или Factory Method. Если у объекта много разных вариантов поведения, полезно организовать их с помощью шаблона Strategy, а не больших структур *if-then*. Фактически само создание шаблонов проектирования является попыткой сократить расходование сил, затрачиваемых на решение стандартных проблем и их обсуждение. Кроме того, применение принципа DRY к структурам, таким как схемы баз данных, приводит к нормализации.

Дело принципа

Другие принципы программного обеспечения также связаны с DRY. Принцип «один и только один раз» (Once and Only Once), который применим только к функциональному поведению кода, можно рассматривать как подмножество DRY. Принцип «открыт/закрыт» (Open/Closed), гласящий, что «элементы программного обеспечения должны быть открыты для расширения, но закрыты для модификации» на практике действует только тогда, когда выполняется принцип DRY. Аналогично на DRY основан известный «принцип единственной ответственности» (Single Responsibility Principle), который требует, чтобы у класса была только одна причина изменения.

Если принцип DRY применять в отношении структуры, логики, процесса или функции, он служит базовым руководством для разработчиков программного обеспечения и способствует созданию более простых, легких в сопровождении и более качественных программ. Но хотя и существуют ситуации, в которых повторение оказывается необходимым для достижения нужной производительности или выполнения других требований (например, денормализация данных в базе данных), применять повторение следует только для решения реальных, а не воображаемых проблем.

Этот код не трогать!

Кэл Эванс

С каждым из нас такое когда-нибудь да случалось. Ваш код загружен на промежуточный (staging) сервер для системного тестирования, и руководитель отдела тестирования сообщает вам, что есть проблема. Вы сразу готовы ответить: «Дайте-ка я быстро все исправлю: я знаю, в чем дело».

Однако в более широком смысле проблема в том, что вы как разработчик считаете, что вам должен быть предоставлен доступ к серверу, где осуществляется тестирование.

В большинстве случаев, если речь идет о веб-разработке, архитектуру можно разбить на следующие части:

- Локальная разработка и модульное тестирование на машине разработчика
- Сервер разработки, где проводится автоматическое или ручное интеграционное тестирование
- Промежуточный (staging) сервер, где команда контроля качества и пользователи осуществляют приемочное тестирование
- Боевой (production) сервер

Да, существуют и другие серверы и сервисы, например для управления исходным кодом или дефектами ПО, но идея понятна. В такой модели разработчик – даже ведущий – ни при каких условиях не должен иметь доступа дальше сервера разработки. Основная разработка происходит на локальной машине программиста с использованием его любимых IDE, виртуальных машин и некоторой полезной для успеха дела черной магии.

После сохранения кода в репозиторий, будь то вручную или автоматически, код должен быть развернут на сервере разработки, где его можно протестировать и при необходимости подправить, чтобы убедиться, что он корректно работает в этой среде. И начиная с этого момента разработчику остается только наблюдать за процессом.

Администратору промежуточного сервера следует упаковать и перенести код на промежуточный сервер, где с ним будет работать группа контроля качества. Подобно тому как разработчики не должны иметь доступ за пределы сервера разработки, группа контроля качества и пользователи не должны выходить за пределы среды тестирования. Если версия готова к приемочному тестированию, собирайте версию и выкатывайте ее: не предлагайте пользователям «по-быстрому глянуть вот на это» на сервере разработки. Помните, за исключением ситуаций «один воин в поле», на сервере есть код и других авторов, которые могут быть не готовы к показу его пользователям. Ответственный за выпуск версий – единственный человек, у которого должен быть доступ к обоим серверам.

Ни в коем случае – ни при каких обстоятельствах – разработчик не должен быть допущен к боевому серверу. Если возникнут проблемы, команда поддержки должна либо справиться с ними сама, либо предложить вам внести исправления. Когда вы сохраните исправление в репозиторий, команда поддержки возьмет «заплатку» оттуда. Кое-какие ужаснейшие осложнения на проектах с моим участием возникали из-за того, что кое-кто (*догадайтесь, кто*) нарушил это правило. Если приложение сломалось, боевой сервер – не место для внесения исправлений.

Инкапсулируйте поведение, а не только состояние

Эйнар Ландре

В теории систем существует концепция изоляции, которая входит в число наиболее полезных, если речь идет о крупных и сложных системных структурах. В индустрии разработки ПО все хорошо понимают ценность изоляции одной сущности внутри другой, иначе говоря, инкапсуляции. В языках программирования для обеспечения изоляции применяются подпрограммы и функции, модули и пакеты, классы и т. д.

Модули и пакеты решают задачи инкапсуляции крупного масштаба, в то время как классы, подпрограммы и функции призваны решать те же задачи на более низком уровне. За годы работы я обнаружил, что из всех видов инкапсуляции тяжелее всего программистам дается инкапсуляция в классах. Нередко встречается класс, единственный метод `main` которого имеет 3000 строк, или же класс, в котором есть только методы `set` и `get` для его элементарных атрибутов. Такие примеры показывают, что разработчики этих классов не вполне освоили объектно-ориентированное мышление и не умеют воспользоваться мощью объектов как моделирующих конструкций. Для тех, кто знаком с терминами POJO (Plain Old Java Object – простой Java-объект в старом стиле) и POCO (Plain Old C# Object или Plain Old CLR Object), замечу: изначально они выражали возврат к основам ООП как парадигмы моделирования – понятным и простым, но не тупым объектам.

Объект инкапсулирует как состояние, так и поведение, причем поведение определяется фактическим состоянием. Возьмем объект «дверь». У него четыре состояния: открыта, закрыта, открывается, закрывается. Он предоставляет две операции: «открыть», «закрыть». В зависимости от состояния операции «открыть» и «закрыть» ведут себя по-разному. Это неотъемлемое свойство объекта делает процесс проектирования концептуально простым. Все сводится к двум простым задачам: назначению и делегированию обязанностей различных объектов, включая и протоколы межобъектного взаимодействия.

Лучше всего показать, как это работает, на примере. Допустим, у нас есть три класса: `Customer` (Покупатель), `Order` (Корзина) и `Item` (Товар). Объект `Customer` – естественный источник правил проверки платежеспособности и определения

максимальной суммы, доступной для списания. Объект `Order` знает, какой `Customer` с ним связан, так что операция `addItem` (добавитьТовар) делегирует фактическую проверку платежеспособности методу `customer.validateCredit(item.price())` (покупатель.проверитьПлатежеспособность(товар.цена())). Если постусловие метода не выполнено, можно сгенерировать исключение и отменить покупку.

Менее опытные в ООП разработчики иногда решают обернуть все бизнес-правила в один объект, который часто называют `OrderManager` (МенеджерЗаказов) или `OrderService` (СлужбаЗаказов). В такой архитектуре объекты `Order`, `Customer` и `Item`, по сути, служат табличными типами. Вся логика выводится из классов и увязывается в одном большом процедурном методе, содержащем множество конструкций *if-then-else*. В методы такого рода легко вкрадываются ошибки, и сопровождать их почти невозможно. Почему? Потому что инкапсуляция нарушена.

Заключение напоследок: не нарушайте инкапсуляцию и используйте мощь своего языка программирования, чтобы ее поддерживать.

Числа с плавающей запятой недействительны

Чак Эллисон

Числа с плавающей запятой не являются «действительными числами» в математическом смысле, хотя в некоторых языках программирования, например в Pascal и Fortran, носят название таковых (real). Действительные числа имеют бесконечную точность, и потому они непрерывны и не подвержены искажениям. Числа с плавающей запятой имеют ограниченную точность, а потому конечны и похожи на «непослушные» целые, так как неравномерно распределены по всему своему диапазону.

Чтобы проиллюстрировать это, попробуйте присвоить 2147483647 (самое большое 32-разрядное целое со знаком) 32-разрядной переменной типа `float` (скажем, `x`), а потом напечатайте его. Вы увидите 2147483648. Теперь напечатайте `x – 64`. Результат – снова 2147483648. Теперь выведите `x – 65`, и вы получите 2147483520! Почему? Потому что промежуток между соседними `float` составляет в этом диапазоне 128, и операции с такими числами округляются до ближайшего числа с плавающей запятой.

В стандарте IEEE числа с плавающей запятой имеют фиксированную точность и записываются как числа по основанию 2 в научной нотации: $1.d_1d_2...d_{p\,1} \times 2^e$, где p – это точность (24 для типа `float`, 53 для типа `double`). Интервал между двумя последовательными числами – величина 2^{1-p+e}, которая хорошо аппроксимируется величиной $\varepsilon|x|$, где ε – *машинный эпсилон* (2^{1-p}).

Зная интервал между соседними числами в окрестности некоторого числа с плавающей запятой, можно избежать классических ошибок в вычислениях. Например, при проведении итеративных вычислений, таких как поиск корня уравнения, нет смысла требовать большей точности, чем способна дать числовая система в окрестности решения. Следите, чтобы затребованная точность не оказалась меньше интервала между числами в окрестности решения, иначе вы войдете в бесконечный цикл.

Поскольку числа с плавающей запятой – лишь приближения действительных чисел, неизбежно наличие небольшой ошибки. Эта ошибка, называемая *ошибкой*

округления, бывает причиной неожиданных результатов. Например, при вычитании очень близких чисел наиболее значимые цифры погашают друг друга, и тогда в результате положение наиболее значимых занимают те цифры, которые были наименее значимыми (и содержали в себе ошибку округления). В сущности, это непоправимо искажает все последующие вычисления. Такое явление называется *размыванием* (*smearing*). Нужно внимательно следить за своими алгоритмами, чтобы предотвратить это *катастрофическое явление*. Для иллюстрации возьмем решение уравнения $x^2 - 100000x + 1 = 0$ по формуле корней квадратного уравнения. Поскольку операнды выражения $-b + sqrt(b^2 - 4)$ почти равны по величине, можно вместо этого вычислить корень $r_1 = -b - sqrt(b^2 - 4)$, а затем получить $r_2 = 1/r_1$, так как в квадратном уравнении $ax^2 + bx + c = 0$ корни удовлетворяют соотношению $r_1 r_2 = c/a$.

Размывание может происходить и более скрытым образом. Допустим, что библиотека в коде наивно вычисляет e^x по формуле $1 + x + x^2/2 + x^3/3! + ...$ Этот способ отлично работает для положительных x, но посмотрите, что произойдет, если x будет большим отрицательным числом. Члены с четными степенями окажутся большими положительными числами, а вычитание значений нечетных степеней почти не скажется на результате. Проблема в данном случае в том, что округление в больших положительных членах происходит в гораздо более значимых позициях, чем у правильного решения. Результат отклоняется к плюс бесконечности! Проблема решается просто: для отрицательных x используйте формулу $e^x = 1/e^{|x|}$.

Нечего и говорить, что числа с плавающей запятой нельзя использовать в финансовых приложениях – для них в таких языках, как Python и C#, есть классы десятичных чисел. Числа с плавающей запятой предназначены для эффективных научных расчетов. Но эффективность бесполезна, если теряется точность, поэтому помните, откуда идут ошибки округления, и пишите соответствующий код!

Удовлетворяйте свое честолюбие через Open Source

Ричард Монсон-Хейфел

Как правило, программы, которые вы пишете на работе, не удовлетворяют ваши самые честолюбивые мечты программиста. Может статься, вы разрабатываете ПО для огромной страховой компании, а хотели бы работать в Google, Apple, Microsoft или начать собственный стартап, который совершит следующую революцию. Вы никогда не придете к своей цели, разрабатывая программы для систем, которые вам не интересны.

К счастью, для вашей проблемы есть решение: open source, то есть проекты с открытым исходным кодом. В мире тысячи проектов с открытым исходным кодом, причем многие весьма активные, в которых программист может получить любой опыт, какой душа пожелает. Если вас влечет идея разработки операционной системы, выберите для себя какой-нибудь из десятка имеющихся проектов операционных систем. Если вы хотите работать над музыкальными программами, приложениями для анимации, криптографии, робототехники, играми для ПК, массовыми онлайн-играми, программами для мобильных телефонов или заниматься чем-то еще, вы почти наверняка найдете по крайней мере один действующий проект с открытым исходным кодом в соответствующей области.

Разумеется, за все нужно платить. Вам придется пожертвовать своим личным временем, ведь едва ли ваш работодатель позволит вам заниматься разработкой видеоигры с открытым исходным кодом в рабочее время. Кроме того, участие в открытых проектах очень редко приносит доход: некоторые зарабатывают на этом, но большинство – нет. Вы должны быть готовы отказаться от части развлечений (это не смертельно, если вы станете меньше играть в видеоигры или смотреть телевизор). Чем больше труда вы будете вкладывать в проект с открытым исходным кодом, тем быстрее осознаете, где лежат ваши истинные амбиции как программиста. Необходимо также учитывать условия вашего контракта с работодателем – иногда условия контракта ограничивают участие в посторонних проектах даже в личное время. Кроме того, нужно следить за тем, чтобы не нарушать законы об интеллектуальной собственности в отношении авторских прав, патентов, торговых знаков и коммерческой тайны.

Открытые проекты дают невероятные возможности для мотивированного программиста. Во-первых, вы можете увидеть, каким способом кто-то другой реализует интересные вам вещи: очень многому можно научиться, читая код, написанный другими людьми. Во-вторых, вы можете привнести в проект свой код и свои идеи; не все ваши блестящие идеи будут приняты, но некоторые пройдут. Да и просто решение задач проекта и создание кода позволят вам узнать что-то новое. В-третьих, вы познакомитесь с замечательными людьми, которые настолько же увлечены выбранным вами проектом, насколько и вы. Эта дружба через open source может продлиться до конца жизни. В-четвертых, если вы компетентны в своем деле, то сможете внести свой практический опыт в технологию, которая вам нравится.

Начать работать в открытых проектах довольно просто. Существует огромный объем документации по инструментам, которые вам понадобятся (системам управления исходным кодом, редакторам, языкам программирования, системам сборки и т. д.). Выберите проект, с которого хотите начать, и изучите инструменты, которые в нем используются. Документация о собственно проектах обычно бывает скудной, но это едва ли имеет большое значение, потому что лучше всего учиться, самостоятельно исследуя код. Для начала можно предложить свою помощь в составлении документации. Либо вызваться написать код для тестирования. Это может выглядеть не столь заманчиво, но написание тестового кода для чужих программ позволяет учиться гораздо быстрее, чем практически все прочие виды программистской деятельности. Пишите тестовый код, действительно хороший тестовый код. Ищите дефекты, предлагайте исправления, заводите друзей, работайте над проектом, который вам нравится, и удовлетворяйте свои честолюбивые программистские желания.

Золотое правило проектирования API

Майкл Фезерс

Проектировать интерфейсы прикладного программирования (API) сложно, особенно в общем случае. Разрабатывая API, у которого будут сотни или тысячи пользователей, нужно думать о том, что со временем его, скорее всего, придется изменять, и эти изменения способны нарушить работу клиентского кода. Кроме того, нужно учесть возможное воздействие на ваш API со стороны его пользователей. Если класс API вызывает собственный метод, следует помнить, что пользователь способен создать подкласс вашего класса и переопределить этот метод, а это может привести к катастрофе. И вы не сможете изменить этот метод, потому что некоторые ваши пользователи придали ему другой смысл. Вы попадаете в зависимость от своих пользователей в том, что касается выбора внутренней реализации.

Разработчики API решают эту проблему разными способами, хотя проще всего заблокировать свой API. Скажем, в Java весьма соблазнительной может стать идея навешивать на большинство классов и методов модификатор `final`. В C#, в свою очередь, классы и методы можно объявить как `sealed`. В любом языке можно попробовать представить API с помощью синглтона или воспользоваться фабрикой *статических* методов для защиты от тех, кто попытается переопределить поведение и использовать ваш код способами, которые ограничат ваши возможности в будущем. Все это кажется разумным, но так ли это в действительности?

За последние 10 лет мы постепенно пришли к пониманию того, что модульное тестирование – крайне важный элемент практики, но этот урок еще не везде усвоен в нашей отрасли. Свидетельств тому предостаточно, и они вокруг нас. Возьмите произвольный непротестированный класс, использующий API стороннего разработчика, и попробуйте написать для него модульные тесты. Скорее всего, у вас возникнут проблемы. Выяснится, что код, использующий API, приклеен к нему намертво. Невозможно эмулировать классы API так, чтобы можно было понять, как взаимодействует с ними ваш код, или обеспечить возврат значений для тестирования.

Положение со временем улучшится, но только если при проектировании API мы станем рассматривать тестирование как реальный сценарий использования. К сожалению, это несколько сложнее, чем просто тестировать наш код. Здесь уместно вспомнить **золотое правило проектирования API**: *недостаточно написать тесты для разрабатываемого API; вы должны написать модульные тесты для кода, использующего ваш API*. Следуя этому правилу, вы на собственном опыте узнаете, какие трудности ожидают ваших пользователей, когда они попытаются протестировать свой код самостоятельно.

Нет какого-то единого решения, облегчающего разработчикам тестирование кода на основе вашего API. Ключевые слова `static`, `final` и `sealed` по сути своей не являются плохими конструкциями. Иногда они бывают полезны. Но важно помнить о проблеме тестирования, а для этого вы должны испытать ее на себе. Как только это сделано, ее можно решать так же, как любую другую проблему проектирования.

Миф о гуру

Райан Браш

Каждому, кто достаточно давно работает в компьютерной отрасли, приходилось слышать вопросы типа:

> У меня генерируется исключение *XYZ*. Не знаете ли, в чем проблема?

Задающие подобные вопросы редко утруждаются тем, чтобы показать трассировку стека, журнал приложения или дать какой-либо контекст, способствующий пониманию проблемы. По-видимому, они считают, что вы мыслите в какой-то другой плоскости, где решения открываются вам без всякого анализа фактов. Они считают вас гуру.

Мы ожидаем подобных вопросов от людей, не знакомых с программным обеспечением, от тех, кому работа системы кажется практически волшебством. Меня беспокоит, что с этим мы сталкиваемся и в сообществе программистов. Аналогичные вопросы возникают при проектировании программ, например: «Я пишу приложение для управления складом. Следует ли мне использовать оптимистическую блокировку?». Ирония состоит в том, что, как правило, спрашивающий лучше подготовлен к тому, чтобы ответить на вопрос, чем тот, кому он задан. Вопрошающий, вероятнее всего, знает контекст, знает требования и способен прочесть материалы о преимуществах и недостатках тех или иных стратегий. И при этом от вас ждут разумного ответа безо всякого контекста. Они ждут чуда.

Пора отрасли разработки попрощаться с мифом о гуру. «Гуру» – обычные люди. Они применяют логику и систематически анализируют проблемы так же, как все мы. Они обращаются к приемам быстрого принятия решений и интуиции. Возьмите лучшего программиста, которого вы когда-либо встречали: было время, когда этот человек меньше знал о программировании, чем вы сейчас. Если какой-то человек кажется вам гуру, то лишь благодаря тому, что он годы посвятил учебе и совершенствованию своего мыслительного процесса. «Гуру» – это просто умный человек с неутолимым любопытством.

Конечно, природные способности людей сильно различаются. Уровень сообразительности, знаний и продуктивности многих живущих на свете хакеров для меня

недостижим. Несмотря на это, разрушение мифа о гуру принесет пользу. Например, если я работаю с тем, кто умнее меня, я должен буду проделать рутинную работу и снабдить этого человека достаточным объемом контекста, зная который, он сможет эффективно применить свое мастерство. Избавление от мифа гуру также означает разрушение иллюзорного барьера на пути к совершенствованию. Вместо этого волшебного барьера я буду видеть непрерывный маршрут своего развития.

Наконец, одной из главных помех в развитии программного проекта являются толковые люди, умышленно распространяющие миф о гуру. Они могут делать это из честолюбия или в расчете повысить собственную значимость в глазах клиентов или работодателей. Ирония в том, что такое поведение может делать толковых людей менее ценными, поскольку они не способствуют профессиональному росту своих коллег. Нам не нужны гуру. Нам нужны специалисты, готовые развивать других специалистов в своей отрасли. Места хватит всем.

Тяжелый труд не оправдывает себя

Олве Маудал

Каждому программисту предстоит убедиться, что напряженная работа зачастую не оправдывается. Можно вводить себя или своих коллег в заблуждение, будто, задерживаясь в офисе после работы, вы делаете значительный вклад в проект. Но на самом деле, работая меньше, можно достичь большего – иногда ощутимо большего. Если вы стараетесь сосредоточенно и «продуктивно» работать более 30 часов в неделю, вы, скорее всего, перерабатываете. Стоит подумать о том, чтобы уменьшить свою рабочую нагрузку, благодаря чему вы сможете работать эффективнее и сделаете больше.

Может показаться, что мое утверждение противоречит здравому смыслу и вообще спорно. Однако оно является прямым следствием того факта, что программирование и вообще разработка программного обеспечения требуют непрерывного расширения знаний. По ходу работы над проектом вы станете лучше понимать предметную область и, можно надеяться, найдете более эффективные способы достижения цели. Чтобы не заниматься напрасной работой, необходимо выделять время на изучение результатов того, что делаете, размышлять над увиденным и соответствующим образом корректировать свое поведение.

Профессиональное программирование обычно мало напоминает бег на дистанцию в несколько километров, где в конце асфальтированной дороги видна цель. Большинство программных проектов можно скорее сравнить со спортивным ориентированием. На марафонскую дистанцию. В темноте. И вместо карты местности лишь набросок от руки. Если вы сорветесь с места и побежите изо всех сил в одном направлении, это может произвести на кого-то впечатление, но таким способом вы едва ли добьетесь успеха. Вы должны двигаться в ровном темпе и корректировать свой курс по мере того, как становится понятнее, где находитесь и куда направляетесь.

Кроме того, нужно постоянно расширять свои знания о разработке программного обеспечения в целом и приемах программирования в частности. Полезно читать книги, участвовать в конференциях, общаться с другими профессионалами, экспериментировать с новыми технологиями и осваивать мощные инструменты,

упрощающие работу. Профессиональный программист должен постоянно следить за прогрессом в своей отрасли – так же как нейрохирург или летчик должны это делать в своей. Вы должны посвящать свободное время (вечера, выходные и праздники) самообразованию, поэтому нельзя тратить свободные вечера, выходные и праздники на сверхурочную работу над текущим проектом. Вы же не ждете, что нейрохирурги будут оперировать по 60 часов в неделю, а летчики по 60 часов в неделю пилотировать? Конечно, нет. Подготовка и образование – важнейшая часть их профессии.

Сосредоточьтесь на своем проекте, постарайтесь отыскать для него как можно больше интересных решений, совершенствуйте мастерство, обдумывайте свои действия и корректируйте поведение. Не позорьте себя и нашу профессию, действуя как белка в колесе. Профессиональный программист должен знать, что пытаться сосредоточенно и «продуктивно» работать по 60 часов в неделю – дело неразумное. Действуйте профессионально: готовьтесь, осуществляйте, наблюдайте, обдумывайте и корректируйте.

Как пользоваться системой отслеживания ошибок

Мэтт Доар

Как бы вы их ни называли – *баги*, *дефекты* или даже *побочные эффекты проектирования*, – избавиться от них полностью невозможно. Чтобы проект успешно двигался вперед, очень важно уметь правильно составить отчет об ошибке, а также знать, на что обращать в нем внимание.

В хорошем отчете об ошибке должны быть описаны три вещи:

- Как воспроизвести ошибку – максимально точно – и как часто при этом проявляет себя ошибка.
- Что должно было произойти – как вам это видится.
- Что фактически происходит – хотя бы те данные, которые вы смогли зафиксировать.

Объем и качество предоставленной информации в такой же мере характеризует составителя отчета, как и саму ошибку. Короткий злой отчет («Эта функция – отстой!») мало что сообщает разработчикам помимо того, что у вас было плохое настроение. Отчет, содержащий подробные сведения о контексте происшедшего, облегчает воспроизведение ошибки и вызывает уважение у всей команды, даже если ошибка задерживает выпуск версии.

Отчет об ошибке похож на беседу, и всю ее, с самого начала, может видеть каждый. Не перекладывайте вину на других и не отрицайте само существование ошибки. Лучше попросите дать дополнительную информацию или подумайте, что вы могли упустить.

Изменение состояния ошибки, например с *открыта* на *закрыта*, является публичным заявлением вашего мнения об ошибке. Не жалейте времени, чтобы сразу объяснить, почему посчитали возможным закрыть данную ошибку. Так вы избавите себя в будущем от долгих и утомительных объяснений с недовольными менеджерами и клиентами. Изменение приоритета ошибки также является публичным заявлением, и если ошибка кажется тривиальной лично вам, для кого-то другого она может оказаться поводом прекратить пользоваться продуктом.

Не перегружайте поля отчета информацией, необходимой лично вам. Пометка «ВАЖНО:» в заголовке отчета, возможно, облегчит вам сортировку результатов определенного отчета об ошибках, но в конце концов и другие начнут копировать эту пометку, причем обязательно с опечатками. А может быть, ее потребуется удалить и применить для другого отчета. Лучше использовать новое значение или новое поле и описать, как оно используется, чтобы другим не пришлось повторяться.

Сделайте так, чтобы каждый знал, над какими именно ошибками должна работать команда. Обычно для этих целей применяется общедоступный запрос с очевидным наименованием. Убедитесь, что этот запрос одинаков для всех, а если необходимо изменить его, обязательно известите команду о том, что план работы меняется.

Наконец, помните, что обнаруженная ошибка *не является* стандартной единицей измерения труда, точно так же как число строк кода – точной оценкой потраченных усилий.

Улучшайте код, удаляя его

Пит Гудлиф

Меньше – значит больше. Это избитая короткая максима, но иногда она действительно верна.

За последние несколько недель в числе проведенных мною улучшений нашего кода было удаление некоторых его фрагментов.

Мы создавали проект, следуя принципам экстремального программирования, в том числе YAGNI (You Aren't Gonna Need It – Вам это не понадобится). Но человеческая природа несовершенна, и в некоторых местах мы допустили промахи.

Я заметил, что нашему продукту требовалось неоправданно много времени, чтобы выполнять некоторые задачи – простые задачи, которые должны были бы выполняться почти мгновенно. Все потому, что мы излишне усложнили их реализацию – разного рода фенечками и бантиками, которые фактически не требовались, но в какой-то момент казались полезными.

Итак, я упростил код, повысил производительность продукта и уменьшил уровень глобальной энтропии кода, и все благодаря удалению из кода проекта лишних функций. К счастью, мои модульные тесты помогли мне убедиться, что в результате своих действий я ничего не сломал.

Вот такой простой и весьма приятный опыт.

Но откуда же возник этот ненужный код? Почему один из программистов вдруг решил написать лишнего и почему это не вскрылось во время рецензирования (code review) или парного программирования? Почти наверняка дело обстояло так:

- Эти дополнительные функции были интересны программисту, и он с удовольствием их написал. (*Совет: Пишите полезный код, а не прикольный код.*)
- Кто-то решил, что это может понадобиться в будущем, так почему не написать код сразу. (*Совет: Это противоречит YAGNI. Не пишите прямо сейчас то, что прямо сейчас не нужно.*)

- «Излишество» не выглядело столь уж большим, и проще было сходу реализовать эту функцию, чем обращаться к клиенту и выяснять, нужна ли она ему. (*Совет: Писать и сопровождать лишний код всегда дольше. Да и клиент – славный малый. Маленький кусочек лишнего кода со временем разрастается, как снежный ком, в огромный кусок работы по сопровождению.)*
- Чтобы оправдать дополнительную функцию, программист придумал новые требования, которых не было ни в документации, ни на обсуждениях. Фактически это поддельные технические требования. (*Совет: Требования к системе устанавливаются не программистами, а заказчиком.*)

Так над чем вы сейчас работаете? Это действительно нужно?

Установи меня!

Маркус Бэйкер

Мне нисколько не интересна ваша программа.

У меня полно своих проблем и длиннющий список задач. На ваш сайт я зашел лишь благодаря непроверенным слухам, будто ваша программа решит все мои проблемы. Уж простите мне мое недоверие.

Если исследования движения взгляда верны, я уже прочел заголовок и шарю по странице в поисках синей подчеркнутой ссылки *Загрузить сейчас*. Кстати, если я зашел на эту страницу из браузера, работающего под Linux, и мой IP принадлежит Великобритании, то можно предположить, что мне нужна версия для Linux с зеркала в Европе, так что об этом, пожалуйста, не спрашивайте. Я так понимаю, что диалоговое окно загрузки файла откроется сразу, так что я отправлю эту штуку в свою папку для загруженных файлов и продолжу чтение.

Выполняя любое действие, человек анализирует соотношение затрат и выгоды. Если ваша программа хоть на секунду упадет ниже приемлемого для меня качества, я тут же ее выброшу и найду что-то другое. Немедленное вознаграждение – вот в чем сила.

Первый барьер – *установка*. Думаете, невелика беда? Так загляните в свою папку загруженных файлов. Полно файлов *.tar* и *.zip*, верно? Какую часть из них вы распаковали? Сколько установили? У меня, например, лишь третья часть из них служит чем-то, кроме балласта для жесткого диска.

Даже если я хочу получить обслуживание на дому, я не желаю, чтобы вы входили в мой дом без приглашения. Прежде чем набрать команду `install`, я бы хотел точно знать, где вы собираетесь размещать свои данные. Это мой компьютер, и я хочу по возможности соблюдать на нем порядок. Я хочу также иметь возможность удалить вашу программу в ту же секунду, как я разочаруюсь в ней. Заподозрив, что это невозможно, я просто не буду ее устанавливать. Моя машина работает стабильно, и я хочу, чтобы так было и впредь.

Если у вашей программы графический интерфейс, я хотел бы выполнить какую-нибудь простую задачу и увидеть результат. «Мастера» тут не помогут, потому

что они делают какие-то вещи, которых я не понимаю. Весьма вероятно, что я захочу прочесть или записать файл. Я не хочу создавать проект, импортировать каталоги или сообщать вам свой электронный адрес. Если все работает, переходим к учебнику (tutorial).

Если ваше программное обеспечение – библиотека, я продолжаю читать вашу веб-страницу в поисках краткого руководства для начинающих. Мне нужно что-то вроде «Hello, World» в пяти незамысловатых строках кода, и чтобы результат был точно таким, как описывает ваш веб-сайт. Никаких огромных файлов XML или шаблонов, которые нужно заполнить, – лишь один файл с исходным кодом. Не забывайте, я загрузил еще и продукт вашего конкурента. Ага, того самого, который на всех форумах твердит, насколько его продукт лучше вашего. Если все работает, переходим к учебнику.

У вас ведь есть учебник, да? На понятном мне языке?

И если в учебнике говорится о моей проблеме, я приободряюсь. Теперь, когда я читаю о том, что я смогу делать, мне становится интересно и даже увлекательно. Я откидываюсь в кресле и прихлебываю чай – я ведь уже говорил, что живу в Великобритании? Теперь я поиграю с вашими примерами и попробую научиться пользоваться вашим творением. Если оно решит мои проблемы, я пошлю вам благодарственное письмо. Я буду также посылать сообщения об ошибках, когда приложение падает, и предложения по новым функциям. Я даже расскажу своим друзьям, что ваша программа лучше других, даже если не пробовал программы ваших конкурентов. И все потому, что вы с таким вниманием отнеслись к моим первым робким шагам.

И как только я мог усомниться в вас?

Межпроцессная коммуникация влияет на время отклика приложения

Рэнди Стэффорд

Время отклика имеет критическое значение для эргономики программ. Мало что так раздражает, как ожидание ответа программной системы, особенно если взаимодействие с ней состоит из повторяющихся циклов воздействия и отклика[1]. Возникает ощущение, будто программа крадет у вас время и снижает продуктивность. Однако причины замедленного отклика программисты осознают не всегда, особенно в современных приложениях. В литературе, посвященной контролю производительности, по-прежнему много внимания уделяется структурам данных и алгоритмам, то есть вопросам, способным оказывать влияние на производительность в некоторых случаях, но едва ли в современных многоуровневых приложениях корпоративного уровня.

Мой опыт показывает, что когда в таких приложениях возникает проблема производительности, поиски ее решения следует начинать не с изучения структур данных и алгоритмов. Время отклика больше всего зависит от количества взаимодействий между удаленными процессами (interprocess communications, IPC), осуществляемых в ответ на воздействие. Да, встречаются и другие локальные узкие места, но число взаимодействий между удаленными процессами обычно имеет решающее значение. Каждое взаимодействие между удаленными процессами добавляет некоторую ненулевую задержку в общее время отклика, и эти отдельные добавки суммируются, особенно если возникают последовательно.

Каноническим примером служит пульсирующая нагрузка в приложениях, использующих объектно-реляционное отображение (ORM). Пульсирующая нагрузка описывает последовательное выполнение множественных обращений к базе данных для чтения данных, необходимых для построения графа объектов

---

[1] Автор использует в статье биологические термины «раздражитель» (stimulus) и «реакция» (response) в применении к программному обеспечению. В данной статье под термином «воздействие» понимается взаимодействие пользователя с системой («раздражитель»), которое запускает определенные ответные действия в системе («отклик»). – *Прим. науч. ред.*

(см. шаблон Lazy Load[1] в книге Мартина Фаулера «Patterns of Enterprise Application Architecture»[2], Addison-Wesley Professional). Когда клиентом базы данных является сервер приложений промежуточного уровня (middle-tier), который компонует веб-страницу, обращения к базе данных обычно происходят последовательно в едином потоке. Их периоды ожидания суммируются, образуя суммарное время отклика. Даже если каждое обращение к базе данных длится всего 10 мс, страница, требующая 1000 обращений (что не редкость), появится с задержкой не менее чем в 10 секунд. Другими примерами могут служить обращение к веб-службам, HTTP-запросы веб-браузера, вызов распределенных объектов, шаблон связи «запрос-ответ» и взаимодействие с таблицей данных по специальным сетевым протоколам. Чем больше удаленных IPC требуется для ответа на воздействие, тем большим будет время отклика.

Существует ряд относительно очевидных и широко известных стратегий сокращения количества взаимодействий между удаленными процессами в расчете на одно воздействие. Одна из таких стратегий заключается в применении принципа бережливости – мы можем оптимизировать интерфейс между процессами, чтобы передавались только те данные, которые нужны непосредственно, а объем взаимодействия был минимален. Другая стратегия – как можно более широкое распараллеливание связей между процессами, чтобы общее время отклика определялось в основном IPC с наибольшей задержкой. Третья стратегия – кэшировать результаты предшествующих IPC, чтобы в будущем вместо IPC использовать обращение к локальному кэшу.

Проектируя приложение, следите за количеством взаимодействий между процессами, происходящих в ответ на каждое воздействие. Анализируя приложения с низкой производительностью, я часто сталкивался с тем, что отношение количества IPC к воздействию составляет 1000:1 и больше. Сокращение этого отношения путем кэширования, распараллеливания или других приемов даст значительно больший эффект, чем изменение структуры данных или модификация алгоритма сортировки.

---

[1] *http://martinfowler.com/eaaCatalog/lazyLoad.html.*

[2] Мартин Фаулер др. «Шаблоны корпоративных приложений». – Пер. с англ. – Вильямс, 2010.

Сборка должна быть чистой

Йоханнес Бродуолл

Приходилось ли вам видеть список предупреждений компилятора (warnings) размером с очерк на тему, как не стоит писать код, и думать при этом: «Конечно, с этим надо что-то делать... только сейчас у меня нет на это времени»? И наоборот, случалось ли вам увидеть единственное предупреждение, появившееся во время компиляции, и тут же исправить его?

Когда я начинаю новый проект с нуля, нет никаких предупреждений, нет беспорядка, нет проблем. Но объем кода растет, и, если не принять меры, не исключено, что беспорядок, хлам, предупреждения и проблемы начнут постепенно накапливаться. В большом потоке «шума» значительно тяжелее отыскать действительно важное предупреждение среди сотен других, которые мне не интересны.

Чтобы предупреждения снова стали полезными, я стараюсь придерживаться политики полной недопустимости предупреждений во время сборки. Даже если предупреждение несущественно, я его устраняю. Если оно не критично, но все же относится к делу, я исправляю код. Если компилятор сообщает об опасности исключения с нулевым указателем, я исправляю источник этой опасности, даже если «знаю», что в реальной обстановке эта проблема никогда не возникнет. Если встроенная в код документация (Javadoc или ее аналог) ссылается на параметры, которые удалены или переименованы, я исправляю документацию.

Если мне не интересно предупреждение и оно совсем несущественное, я советуюсь с командой, не изменить ли нам политику выдачи предупреждений. Например, я считаю, что документирование параметров и возвращаемого значения метода во многих случаях не приносит никакой пользы, и нет нужды выводить предупреждение об их отсутствии. Или вот еще: при переходе на новую версию языка программирования могут появиться предупреждения в коде – там, где их раньше не было. Например, когда в Java 5 появились обобщения (generics), старый код, не обозначавший типы параметров обобщений, запестрил предупреждениями. Мне не нужны такие назойливые предупреждения (пока, во всяком случае). Предупреждения, не согласованные с реальностью, бесполезны.

Обеспечив неизменно чистую сборку, я уже не должен при рассмотрении каждого предупреждения решать, существенно оно или нет. Проигнорировать что-либо – это тоже работа мысли, а я стараюсь избавляться от всей лишней мыслительной работы. Благодаря чистой сборке мне легче передать свою работу другому человеку. Если я оставлю все эти предупреждения, кому-то другому придется разбираться с ними и решать, какие из них важны, а какие нет. А еще вероятнее, что этот новый человек просто проигнорирует все предупреждения, включая существенные.

Предупреждения во время сборки полезны. Нужно лишь избавиться от бесполезного шума, чтобы начать извлекать пользу. Не откладывайте это до «генеральной уборки» кода. Если что-то начинает мозолить глаза, разберитесь с этим сразу же. Нужно либо исправить источник предупреждения, либо подавить его вывод, либо изменить политику вывода предупреждений вашей среды разработки. Чистая сборка нужна не только для проверки ошибок компиляции и «упавших» тестов. Предупреждения – важная и неизбежная часть поддержания гигиены кода.

Умей пользоваться утилитами командной строки

Кэрролл Робинсон

Сегодня многие средства разработки программного обеспечения поставляются в виде *интегрированных сред разработки* (IDE). Помимо двух популярных примеров – Visual Studio от Microsoft и Eclipse от сообщества открытых проектов – существует и множество других. В пользу IDE можно сказать многое. Ими легко пользоваться и они избавляют программиста от необходимости вникать во множество мелких деталей, включая процесс сборки.

Однако легкость использования имеет свой недостаток. Обычно инструментом легко пользоваться, когда он принимает решения за программиста и автоматически проделывает большой объем работы за кулисами. Поэтому если в качестве среды программирования вы используете только IDE, вполне возможно, вы никогда до конца не поймете, что фактически делают ваши инструменты. Жмете на кнопку, творится волшебство, и в папке вашего проекта появляется исполняемый файл.

Работая с инструментами в командной строке, вы гораздо больше узнаете о том, что делают ваши инструменты во время сборки проекта. Составление собственных *make*-файлов поможет осмыслить все этапы сборки исполняемого файла (компиляция, ассемблирование, компоновка и т. д.). Эксперименты с многочисленными ключами командных строк этих инструментов – ценная и познавательная практика. Начать работу с инструментами командной строки для сборки можно со средств командной строки с открытым исходным кодом, таких как GCC, или тех, что поставляются в составе вашей коммерческой IDE. В конце концов, правильно спроектированная IDE – это всего лишь графический интерфейс к набору инструментов командной строки.

Так вы обретете лучшее понимание процедуры сборки. Кроме того, определенные задачи в командной строке выполняются проще и эффективнее, чем в IDE. Например, возможности поиска и замены таких утилит, как *grep* и *sed*, зачастую превышают аналогичные функции IDE. Инструменты командной строки ориентированы на сценарное выполнение задач, что позволяет автоматизировать, например, выполнение ежедневной сборки в заданное время, создание нескольких

версий проекта и прогон наборов тестов. В IDE автоматизация такого рода может быть затруднена (или вообще невозможна), потому что параметры сборки обычно указываются через диалоговые окна GUI, а процедура сборки запускается щелчком мыши. Если вы никогда не выходили за рамки IDE, то, возможно, не догадываетесь о возможности автоматизации таких задач.

Минуточку. Разве IDE существуют не для того, чтобы облегчить разработку и повысить продуктивность программиста? Что ж, все так. Я вовсе не предлагаю прекратить пользоваться IDE. Просто вам следует заглянуть «под капот» и понять, какие задачи IDE выполняет за вас. И наилучший способ это сделать – научиться применять инструменты командной строки. После этого, вернувшись к своей IDE, вы гораздо лучше будете понимать, что она за вас делает и как можно управлять процессом сборки. С другой стороны, освоив работу с инструментами командной строки и познав их мощь и гибкость, вы, возможно, предпочтете пользоваться командной строкой вместо IDE.

Как следует изучи более двух языков программирования

Рассел Уиндер

Психология программирования: давно уже известно, что профессионализм программиста непосредственно зависит от количества различных парадигм программирования, которыми он владеет – не просто что-то слышал и знает о них, но умеет реально использовать их в работе.

Каждый программист начинает с какого-то одного языка. Этот язык оказывает преобладающее влияние на то, как программист видит программное обеспечение. Но каким бы долгим ни был опыт работы программиста с этим языком, если он работает только с ним одним, то и будет знать только этот язык. Мышление программиста, знающего лишь один язык, ограничено возможностями этого языка.

Программист, изучающий второй язык, встретится с трудностями, особенно если вычислительная модель второго языка отличается от первого. C, Pascal, Fortran – все они основаны на одной вычислительной модели. Переход с Fortran на C вызывает некоторые трудности, но их не так много. Переход с C или Fortran на C++ или Ada сопровождается фундаментальными трудностями с пониманием поведения программ. Переход с C++ на Haskell означает существенные изменения, а потому существенные трудности. Переход с языка C на Prolog станет весьма существенным испытанием.

Можно перечислить некоторые парадигмы вычислений: процедурная, объектно-ориентированная, функциональная, логическая, парадигма потоков данных и т. д. Переход между этими парадигмами создает наибольшие трудности.

Чем полезны такие трудности? Здесь играют роль наши представления о реализациях алгоритмов, а также идиомах и шаблонах этих реализаций. В частности, взаимное обогащение идеями лежит в основе достижения мастерства. Идиомы решения задач, применимые в одном языке, могут оказаться недоступны в другом. Пытаясь перенести идиомы из одного языка в другой, мы начинаем лучше понимать оба эти языка и задачу, которую решаем.

Взаимное обогащение при использовании разных языков программирования дает мощнейшие эффекты. Пожалуй, наиболее очевидным из них является все растущее применение декларативных режимов описания в системах, реализованных посредством императивных языков. Любой знаток функционального программирования может легко применить декларативный подход, даже при работе с таким языком, как C. Применение декларативных методов обычно приводит к созданию более коротких и понятных программ. Скажем, C++ определенно выступает за такой подход, обеспечивая всестороннюю поддержку обобщенного (generic) программирования, в котором декларативный режим описания является почти обязательным.

Как следствие всего сказанного выше каждому программисту надлежит уметь хорошо программировать хотя бы в двух разных парадигмах, а в идеале хотя бы в перечисленных пяти. Программист всегда должен стремиться к освоению новых языков, предпочтительно с незнакомыми ему парадигмами. Даже если в своей повседневной работе он неизменно использует один и тот же язык программирования, нельзя недооценивать то более искусное применение этого языка, которое станет возможным благодаря идеям из других парадигм. Работодателям следует это учесть и закладывать в бюджет обучение сотрудников языкам, которые в данное время на проектах не используются, как средство научить работников более искусно применять те языки, которые реально используются.

Неделя начального курса – это хорошо, но не достаточно для изучения нового языка. Чтобы приобрести рабочие навыки применения языка, обычно нужно потратить несколько месяцев хотя бы факультативно. Важно практическое освоение идиом, а не просто изучение синтаксиса и вычислительной модели.

Знай свою IDE

Хейнц Кабуц

В восьмидесятые годы среда программирования, как правило, не сильно отличалась от текстового редактора с наворотами – в лучшем случае. Это сегодня мы воспринимаем подсветку синтаксиса как нечто само собой разумеющееся, а в то время она была роскошью, доступной далеко не каждому. Средства форматирования кода существовали в виде внешних инструментов, применение которых корректировало расстановку пробелов. Отладчики тоже «жили» отдельно как программы для пошагового выполнения кода, и для работы с ними требовалось знать массу загадочных сочетаний клавиш.

В девяностые годы компании начали осознавать потенциал прибыли от более удобных и полезных инструментов разработки. Интегрированная среда разработки (Integrated Development Environment, IDE) объединила уже предлагавшиеся ранее функции редактирования с компилятором, отладчиком, средствами форматирования и другими инструментами. Как раз в это время стали популярны меню и мышь, а значит, отпала надобность заучивать разработчикам сложные комбинации клавиш для работы со своим редактором. Достаточно было выбрать команду из меню.

В двадцать первом веке IDE настолько распространились, что некоторые компании, нацеленные на доли рынка в других областях, раздают их бесплатно. Современная IDE предлагает множество восхитительных функций. Мне особенно нравится автоматический рефакторинг, в частности функция *Extract Method*, позволяющая выделить фрагмент кода и сделать из него метод. Средства рефакторинга найдут все параметры, которые нужно передать методу, благодаря чему становится чрезвычайно просто модифицировать код. Моя IDE даже найдет другие фрагменты кода, которые можно заменить вызовом этого метода, и спросит у меня, следует ли это сделать.

Другая замечательная возможность современных IDE – способность принуждать к соблюдению стиля, принятого в компании. Например, в Java некоторые программисты стали объявлять все параметры `final` (на мой взгляд, это пустая трата времени). Тем не менее раз такое правило установлено, мне достаточно задать

его в настройках IDE, и я стану получать предупреждения для всех параметров, которые не объявлены как `final`. С помощью правил стиля можно также искать возможные ошибки, такие как проверка равенства автоматически упакованных (autoboxed) объектов посредством ссылочной семантики, как в случае использования оператора `==` для примитивов, упакованных в соответствующие объекты.

К сожалению, современные IDE не требуют, чтобы мы прилагали усилия к освоению этих самых IDE. Когда я начал программировать на C под UNIX, мне пришлось потратить немало времени, чтобы научиться работать в редакторе *vi*, что обусловлено его кривой обучения. Но потраченное на старте время сторицей окупилось с годами. Даже черновик этой статьи набран в *vi*. У современных IDE кривая обучения такая, что мы никогда не выходим за пределы базовых приемов работы с ними.

Первое, что я делаю при изучении IDE, – запоминаю управляющие сочетания клавиш. Когда я набираю код, пальцы лежат на клавиатуре, и нажатие *Ctrl+Shift+I* позволяет встроить переменную (операция рефакторинга *Inline Variable*), не нарушая рабочего потока, тогда как навигация по меню указателем мыши отвлекла бы меня. Такие отвлечения создают ненужные переключения контекста и значительно снижают мою продуктивность, если я пытаюсь делать все «ленивым» образом. То же справедливо в отношении владения клавиатурой: освойте печать вслепую, и вы не пожалеете о потраченном времени.

Наконец, у программистов есть проверенные временем конвейерные UNIX-утилиты, позволяющие манипулировать кодом различными способами. Например, если при рецензировании кода я замечаю, что программисты назвали многие классы одинаково, я легко могу обнаружить эти повторения с помощью утилит *find*, *sed*, *sort*, *uniq* и *grep*, например:

```
find . -name "*.java" | sed 's/.*\///' | sort | uniq -c | grep -v "^ *1 " | sort -r
```

Мы ожидаем, что посетивший нас сантехник умеет пользоваться паяльной лампой. Давайте же потратим немного времени и поучимся более эффективно работать со своими IDE.

Знай свои возможности

Грег Колвин

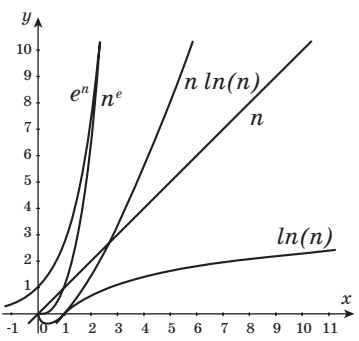
Нужно знать предел своих возможностей.

«Грязный Гарри»

Ваши ресурсы ограничены. Отведенные на выполнение работы время и деньги определены конкретно, в том числе время и деньги, необходимые для поддержания ваших знаний, навыков и инструментов на современном уровне. Существует предел интенсивности, скорости, изобретательности и длительности для вашей работы. Возможности ваших инструментов ограничены. Мощность машин, для которых вы пишете программы, ограничена. Поэтому вы должны учитывать пределы своих ресурсов.

Как учесть эти границы? Нужно знать себя, своих сотрудников, свой бюджет и свою технику. Инженеру-программисту нужно знать пространственно-временную сложность своих структур данных и алгоритмов, архитектуру и показатели производительности своих систем. Ваша задача – создать оптимальное сочетание программного обеспечения и систем.

Пространственная и временная сложность задаются в виде функции $O(f(n))$, где n равно размеру входных данных. Эта функция определяет асимптотическое поведение памяти или времени для n, стремящегося к бесконечности. Важные классы сложности для $f(n)$ – это $ln(n)$, n, $n\ ln(n)$, n^e и e^n. Как ясно видно из графиков этих функций, когда n растет, $O(ln(n))$ становится гораздо меньше $O(n)$ и $O(n\ ln(n))$, а те, в свою очередь, становятся гораздо меньше $O(n^e)$ и $O(e^n)$. В формулировке Шона Пэрента (Sean Parent): для практически достижимых n все классы сложности близки к функциям констант, линейным либо бесконечным.



Анализ сложности осуществляется в терминах некой абстрактной машины, но программы работают

на реальных компьютерах. Современные компьютерные системы образуют целые иерархии физических и виртуальных машин, включающие библиотеки времени выполнения для языков программирования, операционные системы, процессоры, кэш-память, оперативную память, жесткие диски и сети. В приведенной таблице показаны пределы времени произвольного доступа к данным и пределы емкости памяти для типичного сервера, подключенного к сети.

	Время доступа	Емкость
Регистр	<1 нс	64 бит
Уровень кэша		64 байт
Кэш L1	1 нс	64 килобайт
Кэш L2	4 нс	8 мегабайт
ОЗУ	20 нс	32 гигабайт
Диск	10 мс	10 терабайт
Локальная сеть	20 мс	>1 петабайт
Интернет	100 мс	>1 зеттабайт

Заметим, что вариативность памяти и скорости составляет несколько порядков. Для компенсации различий на всех уровнях системы интенсивно применяется кэширование и упреждающий просмотр, но они действенны только тогда, когда доступ предсказуем. Если часто происходят кэш-промахи, система будет тормозить. Например, случайное чтение каждого байта на жестком диске может занять 32 года. Даже случайное чтение каждого байта оперативной памяти может занять 11 минут. Случайный доступ непредсказуем. А что предсказуемо? Зависит от системы, но обычно выигрыш приносят повторный доступ к недавно считанным данным и последовательный доступ к элементам данных.

Алгоритмы и структуры данных различаются эффективностью использования кэша. Например:

- Линейный поиск эффективно использует упреждающий просмотр, но требует $O(n)$ сравнений.
- Двоичный поиск в отсортированном массиве требует всего $O(log(n))$ сравнений.
- Поиск по дереву ван Эмде Боаса (van Emde Boas) имеет сложность $O(log(n))$ и нечувствителен к кэшу.

Что выбрать? Для окончательного анализа нужны измерения. В таблице ниже показано время поиска в массивах 64-разрядных целых чисел с помощью этих трех методов. На моем компьютере:

- Линейный поиск составляет конкуренцию другим методам на малых массивах, но проигрывает экспоненциально на больших.
- Поиск ван Эмде Боаса побеждает без вариантов благодаря схеме предсказуемого доступа.

Время поиска (нс)			
8	50	90	40
64	180	150	70
512	1200	230	100
4096	17000	320	160
	Линейный	**Двоичный**	**ван Эмде Боас**

Каждый сам решает, что для него лучше.

— Punch

Знай, что сохранишь в репозиторий

Дэн Берг Джонссон

Я похлопал трех программистов по плечу и поинтересовался, чем они заняты. «Я провожу рефакторинг этих методов», – был ответ первого. «Я добавляю кое-какие параметры в эту веб-операцию», – отвечал второй. Третий сказал: «Я работаю над этим сценарием использования».

Может показаться, что первые двое были поглощены деталями своей работы, и только третий видел картину шире, и его подход лучше. Я поинтересовался, когда и что они собираются поместить в репозиторий, и тут картина резко изменилась. Первые два вполне ясно представляли, какие это будут файлы, и собирались закончить работу примерно в течение часа. Третий сказал: «Предполагаю, что закончу через несколько дней. Наверное, я добавлю некоторые классы и кое-что модифицирую службы».

Дело не в том, что два программиста не обладали цельной картиной происходящего. Они просто выбрали задачи, которые, по их мнению, вели в нужном направлении и могли быть выполнены за пару часов. Покончив с этими задачами, они выберут новую функцию или рефакторинг для работы. Таким образом, они писали свой код, исходя из четко обозначенных задач и имея небольшую, но реалистичную цель.

Третий программист оказался неспособен провести декомпозицию задачи и работал сразу по всем направлениям. Он не имел представления о том, во что это выльется, и в принципе занимался рискованной работой в надежде, что в какой-то момент у него появится код для записи в репозиторий. Вероятнее всего, написанный на старте этого долгого предприятия код оказался малопригодным для того решения, которое получилось в итоге.

Как поступили бы первые два программиста, потребуй их задачи более двух часов работы каждая? Поняв, что не рассчитали свои силы, они, скорее всего, отказались бы от внесенных изменений, выбрали задачи помельче и начали все сначала. Продолжай они работу, произошла бы расфокусировка, а в репозиторий

попал бы сомнительный код. Лучше отбросить сделанные изменения, но сохранить понимание сути.

Третий программист, возможно, продолжил бы свои гадания и в отчаянии старался бы сшить из своих заплаток код, который можно сохранить в репозиторий. В конце концов, как же можно выбрасывать внесенные изменения – ведь это будет означать, что вы напрасно трудились! К сожалению, если не выбрасывать такой код, в репозиторий попадает слегка странноватый код, не решающий определенную задачу.

Бывают моменты, когда даже программисты, которые ориентированы на частое сохранение кода в репозиторий, не могут найти ничего полезного, с чем бы они могли справиться за пару часов. Тогда они переходят в режим написания ненадежного кода и балуются с кодом, но, конечно, выкидывают свои изменения, когда некое озарение возвращает их на путь истинный. Даже такие бессмысленные, на первый взгляд, периоды работы имеют цель: лучше изучить код, чтобы суметь выявить задачу, решение которой принесет немедленную пользу.

Определите, что сдадите в репозиторий на сей раз. Если не удается завершить задачу, удалите свои изменения и определите новую задачу на основе приобретенного понимания. Займитесь рискованными экспериментами, если это необходимо, но следите за тем, чтобы случайно и незаметно не соскользнуть в режим написания сомнительного кода. Не помещайте в репозиторий плоды рискованных экспериментов.

Место для больших наборов взаимосвязанных данных – в базе данных

Диомидис Спинеллис

Если ваше приложение должно обрабатывать большой долговечный набор взаимосвязанных элементов данных, можете, не раздумывая, поместить его в реляционную базу данных. В прошлом РСУБД было мало, они были сложными, дорогими в содержании и неуклюжими животными. Эти времена миновали. Сегодня найти РСУБД нетрудно: вполне возможно, что на вашей машине уже установлена РСУБД или даже две. Некоторые весьма продвинутые РСУБД, такие как MySQL и PostgreSQL, доступны в исходном коде, поэтому вопрос о затратах на их покупку больше не стоит. Более того, так называемые «встраиваемые системы баз данных» можно подключать к вашему приложению в виде библиотек, почти не требующих настройки и администрирования; к таким относятся две интересные открытые СУБД, SQLite и HSQLDB. Подобные системы крайне эффективны.

Если данные вашего приложения не помещаются в ОЗУ системы, индексированная таблица РСУБД будет работать на несколько порядков быстрее, чем ассоциативные массивы библиотеки языка, где узким местом станет загрузка страниц виртуальной памяти. Возможности современных баз данных легко наращиваются по мере роста ваших потребностей. При надлежащем подходе вы сможете масштабировать встроенную базу данных до более крупной базы данных, если это понадобится. Затем вы сможете перейти с бесплатной базы с открытым кодом на коммерческую систему, предлагающую более развитую поддержку или возможности.

Изучив SQL достаточно хорошо, вы будете с удовольствием создавать приложения, построенные вокруг баз данных. После того как нормализованные данные сохранены в базе, можно легко получать фактическую информацию об этих данных с помощью прозрачно сформулированных запросов SQL; не нужно писать для этого сложный код. Аналогичным образом в такие наборы данных можно вносить сложные изменения при помощи единственной команды SQL. Для разовых модификаций, например при необходимости изменить способ организации хранимых данных, не нужно даже писать код – достаточно запустить приложе-

ние прямого SQL-доступа к базе данных. Этот интерфейс позволит также экспериментировать с запросами в обход цикла редактирования-компиляции обычных языков программирования.

Другое преимущество кода, ориентированного на использование РСУБД, – это поддержка связей между элементами данных. Можно описать *ограничения* (*constraints*), обеспечивающие целостность данных, что позволит избежать риска появления висячих ссылок, если вы забудете обновить свои данные в случае с пограничными условиями. Например, ограничение можно сформулировать так, что при удалении пользователя будут удалены и все сообщения, отправленные этим пользователем.

Можно также в любой момент создавать быстрые связи между хранящимися в базе записями путем простой генерации индекса. Не требуется проводить дорогостоящий и обширный рефакторинг полей класса. Кроме того, когда код построен на основе базы данных, к этим данным могут безопасно обращаться другие приложения. Благодаря этому легко добавить в ваше приложение параллельную обработку и написать отдельные его части с использованием наиболее подходящих языков или платформ. Например, можно написать XML-сервер веб-приложения на Java, некоторые сценарии для аудита – на Ruby, а интерфейс визуализации – с помощью Processing[1].

Наконец, следует помнить, что РСУБД приложит максимум усилий, чтобы оптимизировать ваши SQL-запросы, благодаря чему вы сможете сконцентрироваться на функциональности своего приложения, а не на тонкой настройке алгоритмов. Более развитые базы данных способны даже использовать преимущества многоядерных процессоров, причем совершенно без вашего участия. По мере совершенствования технологий будет расти и производительность вашего приложения.

[1] *http://www.processing.org/*

Учите иностранные языки

Клаус Маркардт

Программистам нужно общаться. И общаться много.

В жизни программиста бывают периоды, когда общение в основном происходит с компьютером – точнее, с выполняемыми на нем программами. Это общение основано на выражении идей в понятном машине виде. Возможность превращать идеи в реальность посредством программ и практически без использования физического вещества по-прежнему вызывает восторг.

Программист должен бегло владеть языком машины – реальной или виртуальной – и абстракциями, которые можно связать с этим языком с помощью инструментов разработчика. Важно знать много разных абстракций, иначе некоторые идеи очень трудно выразить. Хороший программист должен уметь выходить за пределы повседневной рутины и осознавать, что существуют и другие языки, более выразительные для других задач. В какой-то момент эти знания обязательно окупятся.

Программистам нужно общаться не только с машинами, но и со своими коллегами. Современный крупный проект – это больше социальное предприятие, чем просто приложение искусства программирования. В нем требуется понять и выразить больше, чем позволяют доступные машинам абстракции. Лучшие из известных мне программистов обычно очень хорошо владеют родным языком, а часто также и другими языками. И это важно не только для общения: умение хорошо говорить на каком-либо языке означает способность ясно мыслить, без чего невозможно абстрагировать задачу. А это тоже часть программирования.

Общаться приходится не только с машиной, самим собой и коллегами, но и с многими другими связанными с проектом лицами, которые могут не иметь технической подготовки. Они заняты тестированием и контролем качества, развертыванием приложений или маркетингом и продажей. Иногда это конечные пользователи в каком-то офисе (или в магазине, или у себя дома). Необходимо их понимать и знать, с какими проблемами они сталкиваются, а это почти невозможно, если вы не умеете говорить с ними на одном языке – языке их мира, их предметной

области. Когда вам кажется, что разговор с ними прошел удачно, они, возможно, так не считают.

Если вы общаетесь с бухгалтерами, нужно примерно представлять, что такое «учет затрат по местам их возникновения», «вложенный капитал», «чистые активы» и т. п. Если вы беседуете со специалистами по маркетингу или юристами, их жаргон и язык (а значит, миропонимание) должны быть в какой-то мере известны и вам. Всеми этими языками, специфическими для предметной области проекта, должен владеть кто-то из проекта – в идеале программисты. В конечном счете программисты отвечают за практическую реализацию идей посредством компьютеров.

И конечно, жизнь состоит не только из программных проектов. Как заметил Карл Великий, *знать второй язык – значит иметь вторую душу*. Вы оцените пользу знания иностранных языков, когда придется общаться со знакомыми за рамками отрасли программной разработки. Поймете, когда лучше слушать, чем говорить. Узнаете, что большая часть общения происходит без слов.

О чем нельзя говорить, о том следует молчать.

Людвиг Витгенштейн

Учитесь делать оценки

Джованни Аспрони

Будучи программистом, вы должны уметь предоставлять своим менеджерам, коллегам и пользователям оценки предстоящей вам работы, чтобы у них было достаточно точное представление о времени, стоимости, технологиях и других ресурсах, необходимых им для достижения своих целей.

Для проведения надежной оценки необходимо владеть некоторыми приемами. Однако сначала следует разобраться, что собой представляют оценки и для чего они могут быть использованы – как ни странно, многие разработчики и менеджеры плохо в этом разбираются.

Вот типичный диалог между менеджером проекта и программистом:

Менеджер: Можешь оценить, сколько тебе нужно времени, чтобы разработать функцию *X*?

Программист: Месяц.

Менеджер: Это слишком долго! У нас есть всего неделя.

Программист: Мне нужно хотя бы три.

Менеджер: Больше двух я тебе дать не могу.

Программист: По рукам!

В итоге программист предлагает «оценку», приемлемую для менеджера. Но поскольку она как бы сделана программистом, менеджер будет считать, что программист несет за нее ответственность. Для понимания того, что неправильно в этом диалоге, нам нужны три определения: оценки, цели и обязательства:

- Оценка – это приблизительный подсчет или суждение относительно значения, числа, количества или протяженности чего-либо. Это определение предполагает, что оценка является фактической мерой, основанной на надежных данных и прежнем опыте; мечты и пожелания должны быть исключены при ее расчете. Это определение предполагает также, что оценка приблизительна и не может быть дана точно, например, оценка продолжительности разработки не может составить 234,14 дня.

- Цель – это описание бизнес-задачи, которую требуется решить, например «система должна поддерживать одновременную работу не менее 400 пользователей».
- Обязательство – это обещание обеспечить указанную функциональность с определенным уровнем качества к определенному сроку или событию. Например, «функция поиска будет доступна в следующей версии продукта».

Оценки, цели и обязательства не зависят друг от друга, но цели и обязательства должны основываться на надежных оценках. Как отмечает Стив Макконнелл[1], «главная задача оценок в программировании – не предсказание результата проекта, а определение реалистичности целей проекта и возможности их достижения при правильном управлении». Таким образом, полученные оценки должны обеспечить возможность надлежащего планирования и управления проектом, что позволит заинтересованным участникам проекта брать обязательства исходя из реалистичных целей.

В приведенном выше диалоге менеджер в действительности требовал от программиста не оценки, а принятия обязательства в отношении невысказанной цели, которая была в голове этого менеджера. Когда вам в следующий раз предложат сделать оценку, убедитесь, что все стороны хорошо представляют, о чем идет речь, и тогда шансы на успех ваших проектов вырастут. А вот теперь пора освоить несколько приемов...

[1] Стив Макконнелл «Профессиональная разработка программного обеспечения». – Пер. с англ. – СПб.: Символ-Плюс, 2006.

Научитесь говорить «Hello, World»

Томас Гест

Пол Ли, под ником `leep`, более известный под прозвищем Хоппи, слыл местным экспертом по вопросам программирования. Мне потребовалась помощь. Я подошел к его рабочему столу и спросил, не посмотрит ли он вместе со мной кое-какой код.

«Конечно, – сказал Хоппи, – бери стул». Я осторожно придвинулся, стараясь не опрокинуть пирамиду пустых банок из-под колы, громоздившуюся рядом с ним.

«Что за код?»

«В функции в файле», – сказал я.

«Ладно, посмотрим на эту функцию». Хоппи отодвинул в сторону экземпляр «K&R»[1] и придвинул ко мне клавиатуру.

«А где IDE?» Выяснилось, что у Хоппи не было IDE, лишь некий редактор, в котором я не умел работать. Он забрал клавиатуру обратно. Несколько нажатий на клавиши, и перед нами предстал файл – довольно большой файл, а затем и функция – довольно большая функция. Он листал ее, пока не добрался до условно выполняемого блока, о котором я хотел спросить.

«Что здесь произойдет при отрицательном `x`? – спросил я. – Здесь явно ошибка».

Все то утро я пытался заставить `x` принять отрицательное значение, но большая функция в большом файле была частью большого проекта, и меня совершенно измотала необходимость повторно компилировать и повторно запускать свои эксперименты. Может быть, такой эксперт, как Хоппи, просто сообщит мне верный ответ?

Хоппи признался, что он не вполне уверен в результатах. К моему удивлению, он не потянулся за «K&R». Вместо этого он скопировал блок кода в новый буфер редактора, заново расставил отступы и обернул его в функцию. После этого он

---

[1] Имеется в виду классическая книга Кернигана и Ричи «Язык программирования C». – *Прим. ред.*

написал функцию `main`, выполнявшую бесконечный цикл, в котором она предлагала пользователю ввести значение, передавала его функции и выводила результат. Он сохранил буфер в виде нового файла *tryit.c*.

Все это я мог бы сделать и сам, разве что не так быстро. Но следующий его шаг был очень прост, и в то время такой способ работы был мне совершенно незнаком:

```
$ cc tryit.c && ./a.out
```

Надо же! Его программа, придуманная всего за несколько минут до того, работала полным ходом. Мы опробовали несколько значений, и мои подозрения подтвердились (хоть в чем-то я оказался прав!), и *лишь затем* он дополнительно сверился с соответствующим разделом «K&R». Я поблагодарил Хоппи и ушел, снова стараясь не обрушить его пирамиду банок из-под колы.

Вернувшись на рабочее место, я закрыл свою IDE. Я так привык работать над большими проектами в рамках больших продуктов, что стал думать, будто именно так и должен работать. А ведь компьютер, предназначенный для решения самых глобальных проблем, может решать и мелкие задачи. Я открыл текстовый редактор и набрал:

```
#include <stdio.h>
int main()
{
    printf("Hello, World\n");
    return 0;
}
```

Пусть ваш проект говорит сам за себя

Дэниэл Линднер

Скорее всего, в вашем проекте имеется система управления версиями. Весьма вероятно также, что она подключена к серверу непрерывной интеграции, который проверяет корректность проекта с помощью автоматизированных тестов. Это замечательно.

Можно подключить средства статического анализа кода к серверу непрерывной интеграции и получать метрики кода. Эти метрики сообщают специфические характеристики вашего кода и их изменения во времени. Когда введены метрики кода, всегда имеется красная черта, которую нельзя пересекать. Допустим, что вначале у вас покрыто тестами 20% кода, и вы не хотите, чтобы эта величина опускалась ниже 15%. Непрерывная интеграция позволяет следить за всеми этими числами, но все равно вам придется регулярно проверять их значения. Было бы хорошо, если бы проект самостоятельно выполнял эту работу и извещал вас в случае ухудшения ситуации.

Вам нужно дать своему проекту возможность заговорить. Например, с помощью электронной почты или мгновенного обмена сообщениями. Так вы сможете информировать разработчиков о последних ухудшениях или улучшениях показателей. Но еще более эффективно – материализовать проект в офисе посредством *оконечного устройства обратной связи* (*extreme feedback device*, *XFD*).

Идея XFD состоит в управлении физическим устройством – например лампой, миниатюрным фонтаном, роботом или даже подключенной к USB пусковой ракетной установкой – на основе результатов автоматического анализа. Когда нарушаются допустимые границы, устройство сообщает об этом, изменяя свое состояние. Если это лампа, она загорится, яркая и беспристрастная. Невозможно пропустить это сообщение, даже если вы уже идете к двери, направляясь домой.

В зависимости от типа устройства обратной связи вы услышите звук «ломающейся» сборки, увидите красные сигналы некачественного кода или даже почувствуете дурной запах кода. Устройства могут быть установлены в разных офисах, если разработка ведется распределенно. Можно поставить уличный светофор

в офисе менеджера проекта и сигнализировать с его помощью, в каком состоянии находится работа. Менеджер проекта это оценит.

Выбор подходящего вам устройства определяется вашими творческими способностями. Если культура проекта «гиковская», можете попробовать снабдить талисман своей команды радиоуправляемыми игрушками. Если вам нравится более профессиональный подход, попробуйте воспользоваться элегантными дизайнерскими лампами. Поищите вдохновения в Интернете. Все, что подключается к сети или имеет пульт управления, может быть использовано как устройство обратной связи.

Оконечное устройство обратной связи как бы дает голос вашему проекту. Теперь проект физически живет вместе с разработчиками. Он жалуется на них или хвалит их в соответствии с установленными командой правилами. Такое очеловечивание можно развить далее при помощи программы синтеза речи и пары динамиков. Теперь ваш проект действительно говорит сам за себя.

Компоновщик не таит в себе никаких чудес

Уолтер Брайт

Удручающе часто (со мной это снова случилось как раз перед написанием этого текста) встречается следующий взгляд программистов на процесс превращения исходного кода на компилируемом языке в статически скомпонованный выполняемый модуль:

1. Отредактировать исходный код.
2. Скомпилировать исходный код в объектные файлы.
3. Происходит волшебство.
4. Запустить исполняемый файл.

Шаг 3 – это, конечно, *компоновка* (*linking*). Почему же я говорю такие возмутительные вещи? Я занимаюсь технической поддержкой уже не первый десяток лет, и ко мне регулярно приходят с одними и теми же проблемами:

1. Компоновщик сообщает, что `def` определен более одного раза.
2. Компоновщик сообщает, что символ `abc` не найден (unresolved).
3. Почему мой исполняемый файл такой большой?

Затем следует вопрос «Что мне теперь делать?» с вкраплениями слов «вроде бы» и «как-то там», и все это в атмосфере полнейшей озадаченности. Эти «вроде бы» и «как-то там» свидетельствуют о том, что процесс компоновки воспринимается как некое волшебство, понятное только колдунам и чародеям. Процесс компиляции не приводит к формулировкам такого рода, то есть программисты в целом понимают, как работают компиляторы или хотя бы в чем их назначение.

Компоновщик – это тупая, приземленная и прямолинейная программа. Ее задача – склеить область кода и область данных объектных файлов, соединить ссылки на символы с их определениями, выбросить неразрешенные символы из библиотеки и записать исполняемый файл. Все! Никаких чудес и магии! То, что написание компоновщика является утомительным трудом, обычно связано

с декодированием и генерацией файлов, формат которых бывает безобразно сложным, но суть компоновщика от этого не меняется.

Итак, допустим, что компоновщик сообщает вам, что `def` определен более одного раза. Во многих языках программирования, включая C, C++ и D, существуют объявления и определения. Объявления обычно помещаются в файлы заголовков, например:

```
extern int iii;
```

что генерирует внешнюю ссылку на символ `iii`. Напротив, определение фактически отводит память для хранения символа, обычно находится в файле реализации и выглядит примерно так:

```
int iii = 3;
```

Сколько определений может существовать для каждого символа? Как в фильме «Горец», «в живых останется только один». Что произойдет, если определение `iii` обнаружится более чем в одном файле реализации?

```
// Файл a.c
int iii = 3;
// Файл b.c
double iii(int x) { return 3.7; }
```

Компоновщик сообщит о неоднократном определении `iii`.

Определение может быть только одно, более того, оно должно быть обязательно. Если `iii` появляется только в объявлении, но для него нет определения, компоновщик сообщит о том, что символ `iii` не найден.

Чтобы выяснить, почему у исполняемого модуля такой размер, взгляните на файл карты (map), который компоновщик может вывести по запросу. Этот файл содержит перечень всех символов в исполняемом модуле с их адресами. Из него вы узнаете, какие модули были взяты из библиотек и их размеры. Теперь станет ясно, почему так раздулась ваша программа. Часто обнаруживаются библиотечные модули, о подключении которых вы и не догадывались. Чтобы разобраться, временно удалите подозрительный модуль из библиотеки и повторите компоновку заново. Ошибка «неопределенный символ» поможет узнать, кто обращается к этому модулю.

Хотя не всегда очевидно, почему компоновщик выводит то или иное сообщение, в компоновщиках нет ничего волшебного. Механика проста, а вот в конкретных деталях приходится разбираться каждый раз.

Долговечность временных решений

Клаус Маркардт

Почему мы создаем временные решения?

Обычно виной тому срочная задача. Бывает, что это внутренняя задача разработчиков – создать недостающий инструмент для цепи разработки. Бывает, что задача внешняя, ориентированная на пользователей, например обходной путь для восполнения отсутствующей функциональности.

В большинстве систем и команд можно найти модуль, который каким-то образом выделяется в системе. Считается, что это черновой вариант, и его нужно будет впоследствии переделать, потому что он не соответствует стандартам и правилам, по которым живет остальной код. Вам обязательно придется услышать жалобы разработчиков по этому поводу. Причины появления такого кода бывают разными, но основная причина появления на свет промежуточных решений – это их полезность.

Однако промежуточные решения обладают инертностью (или моментом инерции, если вам так больше нравится). Временное решение уже есть – крайне полезное и всеми принятое, – так что нет острой необходимости создавать что-то взамен. Решая, какие действия более других повысят ценность продукта, заинтересованный участник проекта найдет множество задач, приоритет которых будет выше, чем доведение до ума временного решения. Почему? Потому что временное решение уже есть, оно работает, и оно принято. Единственный его видимый недостаток – оно не соответствует выбранным стандартам и правилам, но, за исключением некоторых нишевых продуктов, это слабый аргумент в пользу изменений.

Вот так и остается временное решение. Навсегда.

А если временное решение станет источником проблем, маловероятно, что при починке будет также поставлена задача привести его в соответствие с принятыми стандартами качества. Что же делать? Быстрое промежуточное изменение временного решения часто устраняет проблему и всех устраивает. У него те же достоинства, что и у первоначального временного решения, просто оно... новее.

Представляет ли это проблему?

Все зависит от проекта и вашего личного интереса в поддержании стандарта качества готового кода. Если в системе слишком много временных решений, ее энтропия или внутренняя сложность возрастает, а удобство сопровождения снижается. Однако в первую очередь, возможно, следует задавать иной вопрос. Не забывайте, что мы обсуждаем решение. Возможно, вы и сами не рады такому решению – как вряд ли рад ему кто-то другой, – но стимул перерабатывать решение слаб.

Так что же можно сделать, если это для нас проблема?

1. Прежде всего, избегать временных решений.
2. Изменить факторы, влияющие на решение руководителя проекта.
3. Оставить все, как есть.

Рассмотрим эти варианты более подробно:

1. Во многих случаях отказ от временного решения неприемлем. Есть реальная задача, которую необходимо решить, и оказывается, что стандарты слишком жесткие для этого. Можно попробовать изменить стандарты – достойное, хотя и трудоемкое свершение, – но на это уйдет время, а проблему нужно решать прямо сейчас.
2. Эти факторы коренятся в культуре проекта, а она противится насильственным изменениям. Успеха можно достичь лишь в очень маленьких проектах – особенно когда работаешь один – и если удастся навести порядок, не спрашивая разрешения. Успех возможен и тогда, когда в коде проекта такой беспорядок, что это заметно тормозит работу, и все согласны, что нужно потратить некоторое время на наведение порядка.
3. Если предыдущий вариант не действует, положение дел сохраняется автоматически.

Среди множества созданных вами решений некоторые будут временными и в большинстве своем полезными. Лучший способ избавиться от временных решений – это сделать их избыточными, предложив более элегантные и полезные решения. И да пребудет с вами душевный покой, чтобы принять то, что вы не можете изменить, и силы изменить то, что можете, и мудрость, чтобы отличить одно от другого.

Интерфейсы должно быть легко использовать правильно и трудно – неправильно

Скотт Мейерс

Одна из наиболее распространенных задач в разработке программного обеспечения – это спецификация интерфейса. Интерфейсы существуют на высшем уровне абстракции (интерфейсы пользователя), на низшем (интерфейсы функций) и на промежуточных уровнях (интерфейсы классов, библиотек и т. д.). Независимо от того, чем вы заняты – согласовываете с конечными пользователями их будущее взаимодействие с системой, сотрудничаете с разработчиками, разрабатывая спецификацию API, или объявляете закрытые функции класса, – проектирование интерфейса составляет важную часть вашей работы. Если вы справитесь с ней хорошо, пользоваться вашими интерфейсами будет сплошное удовольствие, а производительность пользователей возрастет. Если вы справитесь с задачей плохо, ваши интерфейсы станут источником разочарований и ошибок.

Хорошие интерфейсы обладают следующими свойствами:

Их легко использовать правильно

Пользователи хорошо спроектированного интерфейса почти всегда используют его правильно, потому что таков для этого интерфейса путь наименьшего сопротивления. Если это графический интерфейс пользователя, они почти всегда щелкают по нужному значку, кнопке или пункту меню, потому что это действие оказывается наиболее очевидным и простым. Если это интерфейс прикладного программирования, они почти всегда передают вызовам правильные параметры с правильными значениями, делая то, что кажется наиболее естественным. Если интерфейс таков, что им легко пользоваться правильно, *все работает само*.

Их трудно использовать неправильно

Хорошие интерфейсы учитывают, какие ошибки бывают у пользователей, и мешают их делать, а в идеале вообще этого не позволяют. Например, графический интерфейс пользователя может сделать неактивными или скрыть команды, не имеющие смысла в текущем контексте, а интерфейс прикладного

программирования может решить проблему порядка аргументов, разрешив передавать параметры в любой последовательности.

Хороший подход к проектированию интерфейсов, которыми легко пользоваться правильно, – практиковаться в работе с ними до их создания. Создайте макет графического интерфейса пользователя (например, на доске с маркерами или на основе разложенных на столе листков для заметок) и поиграйте с макетом, прежде чем писать код. Напишите обращения к API, прежде чем объявлять функции. Разберите стандартные сценарии применения и определите, какого поведения ожидаете от интерфейса. По каким элементам в конечном итоге хотелось бы щелкнуть? Какие параметры в итоге хотелось бы передавать? Простые в работе интерфейсы естественны, потому что позволяют делать именно то, что вам нужно. Такие интерфейсы чаще удаются, если разрабатывать их с точки зрения пользователя. (Это одна из сильных сторон разработки через тестирование, TDD.)

Чтобы затруднить некорректное использование интерфейса, нужны две вещи. Во-первых, следует предугадывать, какие ошибки могут делать пользователи, и находить способы их предотвращать. Во-вторых, следует понаблюдать за ошибками, которые допускают первые пользователи предварительной версии интерфейса, и модифицировать интерфейс – да-да, модифицировать интерфейс! – с целью воспрепятствовать таким ошибкам. Лучший способ предотвратить некорректное использование – это сделать его невозможным. Если пользователи упорно стараются отменить неотменяемое действие, сделайте это действие отменяемым. Если они постоянно передают интерфейсу прикладного программирования неверное значение, постарайтесь модифицировать этот API, чтобы принимать те значения, которые хотят передать пользователи.

А самое главное, помните, что интерфейсы существуют для удобства их пользователей, а не их создателей.

Пусть невидимое станет более видимым

Джон Джаггер

Во многих отношениях невидимость справедливо поощряется как принцип разработки качественного программного обеспечения. В нашем профессиональном языке немало метафор невидимости, таких как прозрачность механизма и инкапсуляция. Программное обеспечение и процесс его разработки могут, если перефразировать Дугласа Адамса (Douglas Adams), оказаться *в основном невидимыми*[1]:

- Исходный код не обладает врожденным присутствием или врожденным поведением и не подчиняется физическим законам. Его можно увидеть, когда он загружен в редактор, но закройте редактор – и он исчезнет. Если долго над этим думать, то начинаешь сомневаться, существует ли он вообще – как дерево, которое упало, но некому было это услышать.
- Работающее приложение обнаруживает свое присутствие и поведение, но ничего не сообщает об исходном коде, на основе которого создано. Домашняя страница Google приятно лаконична, но за кулисами, несомненно, скрыто немало серьезного.
- Если вы сделали 90% работы и безнадежно увязли в отладке остальных 10%, то, видимо, нельзя считать, что сделано 90%? Исправление ошибок – это не движение вперед. Вам не платят за отладку. Отладка – пустая трата времени. Хорошо, если пустая трата времени будет более заметна, чтобы можно было видеть ее реальную цену и, прежде всего, постараться не допускать этого.
- Если кажется, что ваш проект идет по графику, а через неделю обнаруживается, что он опаздывает на полгода, то у вас есть проблемы, главная из которых, возможно, не в том, что он опаздывает на полгода, а в существовании

[1] Отсылка к книге «Mostly Harmless» («В основном безвредна», АСТ, 2003) – пятой, заключительной части серии книг «The Hitchhiker's Guide to the Galaxy» («Автостопом по галактике. Путеводитель»). Словосочетание «в основном безвредна» в книгах серии является полной энциклопедической статьей о планете Земля. – *Прим. ред.*

невидимых силовых полей, способных скрыть полугодовую задержку! Отсутствие видимого прогресса – то же самое, что отсутствие прогресса вообще.

Невидимость таит в себе опасность. Рассуждение становится яснее, когда есть конкретный предмет для обдумывания. Легче управлять вещами, которые можно видеть, и видеть в непрерывном изменении:

- При написании модульных тестов вы узнаете, насколько легко проводить модульное тестирование для конкретного модуля кода. Модульное тестирование выявляет присутствие (или отсутствие) качеств, которые желательны для кода, такие *как слабая связанность* (*coupling*) и *сильная связность* (*cohesion*).
- Прогон модульных тестов демонстрирует, как ведет себя код. Он позволяет обнаружить присутствие (или отсутствие) характеристик времени выполнения, желательных для приложения, например устойчивости и корректности.
- С помощью доски и карточек можно сделать прогресс наглядным и конкретным. Можно увидеть, что задачи находятся в состоянии *Не начата*, *В процессе* и *Завершена*, и при этом не придется заходить в неочевидную систему управления проектом и не придется упрашивать программистов составлять фиктивные отчеты о состоянии проекта.
- Итеративная разработка повышает наглядность прогресса (или его отсутствия), поскольку чаще фиксируются факты того, что разработка ведется. Создание программного обеспечения, готового к выпуску, отражает реальное положение вещей, в отличие от оценок.

Лучше всего разрабатывать программы, имея многочисленные наглядные показатели. Наглядность дает уверенность в том, что прогресс является реальным, а не вымышленным; спланированным, а не непреднамеренным; воспроизводимым, а не случайным.

Передача сообщений улучшает масштабируемость параллельных систем

Рассел Уиндер

Уже на первых лекциях по информатике программистов учат, что конкурентные вычисления – и в особенности параллельные как особый подвид конкурентных – вещь трудная, и что лишь лучшим дается надежда справиться с этой задачей, и что даже лучшие не справляются. При этом неизменно уделяется большое внимание потокам, семафорам, мониторам и трудностям организации потоковой безопасности при одновременном доступе к переменным.

Сложных проблем здесь действительно много, и решать их бывает очень трудно. Но в чем корень проблем? Общая память. Практически все проблемы конкурентных вычислений, о которых постоянно приходится слышать, касаются общей памяти с изменяемыми данными: состояние гонки (race conditions), взаимная блокировка (deadlock), активная блокировка (livelock) и т. п. Кажется, ответ очевиден: забудьте о конкурентности либо держитесь подальше от общей памяти!

Отказ от конкурентности почти наверняка не вариант. Количество ядер в компьютерах возрастает чуть ли не каждый квартал, поэтому достижение настоящего параллелизма становится все важнее. Мы не можем больше полагаться на непрерывный рост тактовой частоты процессоров как основу производительности приложений. Производительность может вырасти только за счет параллелизма. Конечно, можно не заботиться о производительности приложений, но едва ли это понравится пользователям.

Так можно ли отказаться от общей памяти? Определенно, да.

Вместо потоков и общей памяти можно воспользоваться процессами и передачей сообщений. Под *процессом* здесь понимается защищенное независимое состояние исполняющегося кода, а не обязательно процесс операционной системы. Такие языки, как Erlang (а до него occam), показали, что процессы – весьма удачный механизм программирования конкурентных и параллельных систем. В таких системах меньше проблем синхронизации, чем в многопоточных системах с общей памятью. Кроме того, существует формальная модель взаимодействующих

последовательных процессов (Communicating Sequential Processes, CSP), которую можно применять при разработке таких систем.

Можно пойти дальше и организовать вычисления в виде *системы, управляемой потоком данных* (*dataflow system*). В такой системе нет явно запрограммированного потока управления. Вместо этого создается направленный граф операторов, соединенных путями передачи данных, а затем в систему подаются данные. Контроль вычислений осуществляется по готовности данных внутри системы. И никаких проблем синхронизации.

При этом для системной разработки применяются главным образом такие языки, как C, C++, Java, Python и Groovy, о которых программистам говорят, что они служат для разработки многопоточных систем с общей памятью. Как же быть? Решение в том, чтобы использовать – или создавать, если их не существует, – библиотеки и среды, которые предлагают схемы процессов и пересылки сообщений, полностью исключающие применение общей изменяемой памяти.

В итоге отказ от общей памяти в пользу передачи сообщений станет, скорее всего, наиболее удачным методом реализации систем, эффективно использующих параллелизм, получивший сегодня повсеместную прописку в компьютерном железе. Занимательно выходит – процессы как единица конкурентного исполнения появились раньше потоков, но в будущем, по-видимому, потоки станут использоваться для реализации процессов.

Послание потомкам

Линда Райзинг

Возможно, дело в том, что в большинстве своем программисты — умные люди, но за многие годы моего преподавания и тесной совместной работы с ними у меня сложилось впечатление, будто сложность задач, над которыми они бьются, оправдывает для них создание решений столь же сложных (возможно, и для них самих спустя несколько месяцев после написания кода) для понимания и сопровождения.

Помню один случай с Джо, слушателем моего курса по структурам данных, который пришел показать мне результат своего труда.

– Держу пари, вы не догадаетесь, что делает этот код! – радостно воскликнул он.

– Ты прав, – согласилась я, не слишком вглядываясь в его текст и думая, как донести до него важную мысль. – Уверена, ты хорошо потрудился над этим примером. Боюсь, правда, ты упустил нечто важное. Скажи, Джо, у тебя есть младший брат?

– Да, конечно! Его зовут Фил, и он слушает ваш вводный курс. Он тоже учится программировать! – гордо объявил Джо.

– Это замечательно, – отвечала я. – Интересно, сможет ли он понять этот код?

– Ни за что, – сказал Джо, – это сложная штука!

– Давай предположим, – продолжила я, – что это реальный рабочий код и что через несколько лет Филу предложат работу по внесению изменений в этот код. Что ты сделал для Фила?

Джо моргал, глядя на меня.

– Мы знаем, что Фил – толковый парень, верно?

Джо кивнул.

– И не хочу хвастаться, но я тоже довольно толковая!

Джо ухмыльнулся.

– Итак, мне нелегко понять, что ты тут сделал, и твоему очень способному младшему брату тоже. Скорее всего, придется поломать над этим голову. В таком случае, что можно сказать о написанном тобой коде?

Как мне показалось, Джо увидел свой код в новом свете.

– Представим себе дело так, – сказала я, стараясь как можно лучше играть роль доброго наставника. – Каждая строка твоего кода – это послание человеку будущего, которым может оказаться твой младший брат. Попробуй объяснить этому умному человеку, как решить эту трудную задачу. Так ли ты видишь это будущее? Что этот умный программист увидит твой код и воскликнет: «Ничего себе! Как здорово! Мне совершенно понятно, что здесь происходит, и я поражен элегантностью – нет, красотой – этого кода. Надо немедленно показать его коллегам по команде. Это же шедевр!»

– Джо, можешь ли ты написать код, который решает эту задачу, но притом прекрасен, как песня? Да, как запавшая в память мелодия. Я думаю, что тот, кто сумел найти такое сложное решение, как предложенное тобой сегодня, может также написать что-нибудь красивое. Гм-м... Не начать ли мне выставлять оценки за красоту? Как ты считаешь, Джо?

Джо забрал свою работу и посмотрел на меня. Легкая улыбка пробежала по его лицу.

– Я все понял, профессор. Пойду улучшать мир для Фила. Спасибо.

Упущенные возможности применения полиморфизма

Кирк Пеппердин

Полиморфизм – одна из грандиозных идей, лежащих в фундаменте ООП. Это слово, заимствованное из греческого языка, означает множество (*poly*) форм (*morph*). В контексте программирования полиморфизм означает многообразие форм некоторого метода или класса объектов. Но полиморфизм – это не просто альтернативные реализации. Уместное применение полиморфизма создает миниатюрные локализованные контексты исполнения и позволяет обойтись без громоздких блоков *if-then-else*. Находясь в контексте, мы можем напрямую выполнять нужные действия, тогда как, находясь вне этого контекста, мы вынуждены сначала воссоздать его и лишь затем выполнять нужные действия. Аккуратное использование альтернативных реализаций позволяет выделить контекст, а значит, решить ту же задачу через меньший объем более удобочитаемого кода. Лучше всего продемонстрировать это на примере. Возьмем следующий код для (нереально) простой корзины покупок:

```
public class ShoppingCart {
    private ArrayList<Item> cart = new ArrayList<Item>();
    public void add(Item item) { cart.add(item); }
    public Item takeNext() { return cart.remove(0); }
    public boolean isEmpty() { return cart.isEmpty(); }
}
```

Допустим, некоторые товары в нашем интернет-магазине можно скачать из сети Интернет, а другие требуют доставки. Создадим другой класс, который поддерживает эти операции:

```
public class Shipping {
    public boolean ship(Item item, SurfaceAddress address) { ... }
    public boolean ship(Item item, EMailAddress address) { ... }
}
```

Когда клиент рассчитался, нужно доставить покупки:

```
while (!cart.isEmpty()) {
    shipping.ship(cart.takeNext(), ???);
}
```

Параметр *???* – это не какой-то очередной оператор Элвиса[1]; это неразрешенный вопрос о том, как нужно доставить товар – электронной или обычной почтой. Контекста для ответа на этот вопрос уже нет. Можно было сохранить метод доставки в виде `boolean` или `enum`, а затем использовать *if-then-else*, чтобы заполнить значение недостающего параметра. А другое решение – создать два класса, расширяющих `Item`. Назовем их `DownloadableItem` и `SurfaceItem`. Теперь напишем немного кода. Я сделаю из `Item` интерфейс, который поддерживает единственный метод `ship` (доставить). Чтобы доставить содержимое корзины, вызываем `item.ship(shipper)`. Оба класса, `DownloadableItem` и `SurfaceItem`, реализуют `ship`:

```
public class DownloadableItem implements Item {
    public boolean ship(Shipping shipper, Customer customer) {
        shipper.ship(this, customer.getEmailAddress());
    }
}
public class SurfaceItem implements Item {
    public boolean ship(Shipping shipper, Customer customer) {
        shipper.ship(this, customer.getSurfaceAddress());
    }
}
```

В этом примере мы делегировали ответственность за `Shipping` (доставку) каждому `Item` (товару). Поскольку каждый товар знает, как его следует доставлять, такая организация позволяет справиться с доставкой, не прибегая к *if-then-else*. Этот код также демонстрирует применение двух шаблонов проектирования, которые часто хорошо сочетаются между собой: Command и Double Dispatch. Эффективное применение этих шаблонов основано на правильном использовании полиморфизма. При выполнении этих условий число блоков *if-then-else* сокращается.

В некоторых ситуациях гораздо практичнее использовать не полиморфизм, а *if-then-else*, однако при написании кода в стиле полиморфизма он имеет меньший объем, более удобочитаем и менее хрупок. Количество упущенных возможностей несложно подсчитать – оно совпадает с числом операторов *if-then-else* в коде.

---

[1] Оператор Элвиса – это оператор `?:` в Groovy. Он появился в версии 1.5 языка и получил название благодаря сходству с фирменной прической Элвиса. – *Прим. ред.*

Невероятно, но факт: тестировщики – ваши друзья

Берк Хафнагель

Сами себя они могут называть *контролем качества* или *обеспечением качества*, но многие программисты зовут их просто *напастью*. Мой опыт показывает, что у программистов часто враждебные отношения с теми, кто тестирует их программы. «Они слишком придирчивы» или «Они хотят, чтобы все было идеально» – вот обычные жалобы. Знакомо, да?

Не знаю, по какой причине, но у меня всегда был другой взгляд на работу тестировщиков. Может быть потому, что «тестировщиком» на моей первой работе была секретарь фирмы. Весьма приятная дама Маргарет занималась делопроизводством и пыталась научить пару молодых программистов профессиональному поведению в присутствии клиентов. Она также обладала даром в считанные секунды обнаруживать любой дефект программы, даже самый малозаметный.

В то время я работал над программой, которую написал бухгалтер, считавший себя программистом. Естественно, с ней были большие проблемы. Когда мне казалось, что я исправил какой-то кусок, Маргарет пыталась поработать с ним, и чаще всего после нескольких нажатий на клавиши оказывалось, что программа неправильно работает, но уже каким-то новым образом. Временами это приводило в отчаяние и вызывало неловкость, но Маргарет была настолько приятным человеком, что мне никогда не приходило в голову обвинить ее в моем жалком положении. Наконец, настал день, когда Маргарет смогла без проблем запустить программу, создать счет-фактуру, распечатать ее и выйти из программы. Я испытал восторг. Более того, когда мы установили программу на машине клиента, все заработало правильно. Клиент не столкнулся с проблемами, потому что сначала Маргарет помогла мне обнаружить и исправить эти проблемы.

Вот почему я говорю, что тестировщики – ваши друзья. Может казаться, будто тестировщики портят вашу репутацию, сообщая о малозначительных проблемах. Но когда клиент в восторге от программы, потому что ему не портят жизнь все эти «досадные мелочи», которые группа контроля качества заставила вас исправить, вы на высоте. Мысль понятна?

Представьте себе такую ситуацию: вы тестируете программу, использующую «сногсшибательные алгоритмы искусственного интеллекта» для нахождения и исправления проблем с конкурентным доступом. Вы запускаете программу и обнаруживаете, что на заставке слово «интеллект» написано с ошибкой. Несколько дурное предзнаменование, но ведь это лишь опечатка, верно? Затем выясняется, что экран настройки предлагает флажки (checkboxes) вместо переключателей (radiobuttons), а некоторые сочетания клавиш не работают. Все это мелочи, но по мере их накопления вы начинаете задумываться, что за люди создавали программу. Если они неспособны справиться с простыми вещами, каковы шансы, что их искусственный интеллект действительно сумеет найти и исправить такие замысловатые вещи, как проблемы конкурентного доступа?

Возможно, эти гении настолько погрузились в задачи разработки искусственного интеллекта, что не обратили внимания на мелочевку, а «придирчивых тестировщиков» у них не было, так что вам приходится сталкиваться с этой самой мелочевкой. Однако в результате вы начинаете сомневаться в компетентности этих программистов.

Следовательно, как ни странно, тестировщики, полные решимости найти все мелкие недостатки вашего кода, в действительности – ваши друзья.

Один бинарный файл

Стив Фримен

Мне приходилось встречать проекты, где при сборке переписывалась часть кода, чтобы для каждой среды исполнения генерировался собственный бинарный файл. Такой подход всегда излишне усложняет вещи и создает риск появления несовместимых версий при каждой установке. Как минимум при этом собирается несколько почти идентичных экземпляров программы, каждый из которых предназначен для установки в соответствующей ему среде. Возникает слишком много подвижных частей, а значит, больше возможностей для ошибки.

Однажды я работал в команде, где после каждого изменения свойства нужно было сохранять код и проводить полный цикл сборки, поэтому тестировщики простаивали всякий раз, как находилась малейшая неточность (я уже говорил, что к тому же проект собирался невероятно долго?). Работал я и в такой команде, где системные администраторы требовали полной пересборки программы при вводе ее в эксплуатацию (с помощью наших же сценариев сборки), так что невозможно было гарантировать, что в эксплуатацию попадала та версия, которая прошла тестирование. И так далее.

Правило простое: *создавать единственный бинарный файл, который можно точно идентифицировать и провести через все этапы конвейера выпуска продукта*. Все специфические особенности среды исполнения должны оставаться *частью среды*. Например, их можно хранить в контейнере с компонентами (component container), в заранее согласованном файле или в определенных папках.

Если во время сборки вашего проекта производятся манипуляции с кодом или настройки целевой среды хранятся в самом коде, значит, приложение было спроектировано недостаточно хорошо: ключевые функции приложения не отделены от функций, определяемых платформой. Или еще хуже: команда знает, как нужно поступить, но не считает внесение нужных изменений достаточно приоритетной задачей.

Бывают, конечно, исключения: иногда приходится делать сборку для нескольких вариантов целевой среды, в которых ограничения по ресурсам существенно

разнятся. Однако это не относится к тем из нас (и таких большинство), кто создает приложения типа «отправить данные из базы данных на экран и обратно». Другой вариант – работа с плохо написанным унаследованным (legacy) кодом, в котором навести порядок сразу слишком тяжело. В таких случаях следует двигаться постепенно, но начинайте это движение как можно раньше.

И еще одно: храните информацию о среде выполнения в системе управления версиями, как и код. Нет ничего хуже, чем испортить конфигурацию среды и не иметь возможности узнать, какие в нее были внесены изменения. Настройки среды должны храниться в отдельном репозитории, так как они меняются с другой скоростью и по другим причинам, чем код. Некоторые команды используют для этого распределенные системы управления версиями (например, bazaar и git), поскольку в них проще сохранять в репозиторий изменения, сделанные в производственной (production) среде – а они неизбежно случаются.

Правду скажет только код

Петер Зоммерлад

В конечном счете семантика программы определяется работающим кодом. Если он есть у вас только в виде бинарного файла, его будет непросто прочесть! Однако исходный код, как правило, доступен, если это ваша собственная программа, типичная коммерческая разработка, проект с открытым исходным кодом или программа на динамически интерпретируемом языке. При чтении исходного кода смысл программы должен быть очевиден. Можно с уверенностью узнать, что делает программа, только глядя в исходный код. Даже самое точное описание технических требований не скажет всей правды: в нем содержится не детальное описание того, что фактически делает программа, а общие пожелания составителя требований. Документ с архитектурой может содержать описание планируемой архитектуры, но в нем не будут описаны нужные детали реализации. Эти документы могут устареть в сравнении с текущей реализацией... или просто потеряться. Быть может, их даже и не писали. Возможно, единственное, что осталось, – это исходный код.

Учтя все сказанное, задайте себе вопрос, насколько понятно ваш код может рассказать вам или другому программисту, что он делает.

Вы можете сказать: «О, так в моих комментариях есть все, что нужно знать». Но учтите, что комментарии – это не работающий код. Они могут вводить в заблуждение так же, как любая другая документация. Традиционно считалось, что комментарии – безусловно хорошая практика, поэтому некоторые программисты без всяких раздумий пишут обширные комментарии, даже повторяя и разъясняя в них факты, и без того очевидные из собственно кода. Это плохой способ сделать код понятным.

Если ваш код нуждается в комментариях, попробуйте провести рефакторинг так, чтобы они стали не нужны. Пространные комментарии загромождают экран, а ваша IDE может автоматически их скрывать. Если нужно пояснить изменение, сделайте это не в коде, а с помощью сообщения при сохранении кода в систему управления версиями.

Что можно сделать, чтобы ваш код действительно говорил правду и как можно яснее? Старайтесь выбирать хорошие имена. Структурируйте код с учетом сильносвязанной (cohesive) функциональности, что также облегчает выбор имен. Уменьшите (decouple) связанность кода, чтобы достичь ортогональности. Напишите автоматизированные тесты, раскрывающие запланированное поведение, и проверьте интерфейсы. Безжалостно переделывайте код, если найдете способ написать проще и лучше. Старайтесь, чтобы ваш код был как можно проще для чтения и понимания.

Относитесь к своему коду как к любому другому творческому тексту – стихотворению, эссе, записи в открытом блоге или важному электронному письму. Тщательно формулируйте то, что хотите выразить, чтобы код делал то, для чего предназначен, и как можно проще сообщал, что он делает, – так, чтобы ваши намерения были ясны, когда их нельзя уже будет узнать у вас самого.

Помните, что полезным кодом пользуются гораздо дольше, чем предполагают его авторы. Те, кто будут сопровождать ваш код, останутся вам благодарны. А если вы сами занимаетесь сопровождением, и код, с которым вы работаете, не спешит раскрыть свои тайны, как можно раньше начните применять вышеупомянутые принципы. Придайте здравый смысл коду, чтобы сохранить собственный здравый рассудок.

Возьмите сборку (и ее рефакторинг) на себя

Стив Берчук

Не столь уж редки случаи, когда команды, в целом дисциплинированно соблюдающие хорошие практики написания кода, пренебрежительно относятся к сценариям сборки. Их считают либо малозначительными, либо настолько сложными, что обслуживать их может только секта специалистов по выпуску продукта (release engineers). Если сценарии сборки сложны в сопровождении, содержат дублирование и ошибки, это приводит к проблемам того же масштаба, что и плохо спроектированный код.

Почему ответственные и грамотные разработчики считают сборку проекта некой второстепенной работой? Одно из объяснений – сценарии сборки часто пишут на ином языке, чем исходный код. Второе – сценарии сборки не являются «кодом». Такие объяснения противоречивы, ведь большинство разработчиков с удовольствием изучает новые языки, а именно в результате сборки появляются исполняемые модули, которые разработчики и конечные пользователи будут тестировать и запускать. Код бесполезен, если из него не собирается исполняемый модуль, а ведь именно сборка определяет компонентную архитектуру приложения. Сборка – важная часть процесса разработки, и решения в области сборки способны упрощать и сам код, и процесс его написания.

Если в сценариях сборки используются неверные идиомы, такие сценарии тяжело сопровождать и, что хуже, тяжело улучшать. Стоит потратить некоторое время, чтобы разобраться, как правильно вносить изменения. Если приложение собирается с неверными версиями зависимых библиотек или во время сборки заданы неверные параметры конфигурации, это может вызвать ошибки в самом приложении.

Традиционно тестирование всегда возлагалось на группу контроля качества. Сейчас мы понимаем, что тестирование в процессе написания кода – необходимое условие для получения предсказуемого результата. Аналогично и владельцем процесса сборки должна быть команда разработчиков.

Понимание процесса сборки может упростить весь жизненный цикл разработки и сократить издержки. Если процесс сборки легко осмыслить и применить, это дает возможность новому разработчику быстро и легко включиться в работу. Если конфигурацию приложения автоматизировать в рамках процесса сборки, это поможет гарантировать получение одинаковых результатов, когда над проектом работает несколько человек, и избежать реплик в духе «А у меня все работает». Многие инструменты сборки генерируют отчеты о качестве кода, заблаговременно вскрывающие потенциальные проблемы. Потратив время и научившись самостоятельно управлять процессом сборки, вы облегчите жизнь себе и всем остальным участникам вашей команды. Вы сможете сосредоточиться на разработке новой функциональности, что принесет пользу клиентам и сделает вашу работу более приятной.

Изучите процесс сборки достаточно хорошо, чтобы знать, когда и как изменять его. Сценарии сборки – это тоже код. Они слишком важны, чтобы доверить их кому-то другому, хотя бы по той причине, что приложение не закончено, пока оно не собрано. Задача программирования не завершена, пока мы не поставили работающее приложение пользователю.

Программируйте парами и входите в поток

Гудни Хаукнес, Кари Россланд и Анн Кэтрин Гэгнат

Представьте себе, что вы совершенно поглощены своей работой: сосредоточены, увлечены и заняты. Вы потеряли счет времени. Вы счастливы. Вы в состоянии потока. В масштабах всей команды разработчиков трудно достичь и поддерживать состояние потока из-за многочисленных помех, отвлечений и прочих препятствий, которые легко могут его нарушить.

Если вы уже участвовали в парном программировании, то, вероятно, знаете, как оно способствует достижению состояния потока. Если нет, то мы хотим поделиться своим опытом, чтобы побудить вас заняться парным программированием немедленно! Чтобы парное программирование было успешным, требуются некоторые усилия со стороны отдельных участников команды и всей команды в целом.

Будучи частью команды, проявляйте терпение по отношению к менее опытным разработчикам. Преодолейте свой страх перед более опытными разработчиками. Осознайте, что все люди разные, и научитесь это ценить. Помните о сильных и слабых качествах – как своих, так и других членов команды. Вас может удивить, сколь многому способны научить коллеги.

Применяйте парное программирование для распространения навыков и знаний среди всех участников проекта. Задачи нужно решать парами и часто производить ротацию участников пар и выполняемых каждой парой задач. Сообща установите правило для такой ротации. Если нужно, откажитесь от этого правила или подправьте его. Наш опыт показывает, что не обязательно доводить задачу до конца, прежде чем передать ее следующей паре. Может показаться, будто это неразумно, однако на практике мы обнаружили, что это эффективно.

Существует множество ситуаций, когда состояние потока может быть нарушено, но парное программирование позволяет его сохранить:

- *Снижается роль «фактора грузовика»*. Вот слегка пугающий мысленный эксперимент: сколько членов вашей команды должно попасть под грузовик, чтобы стал невозможным выпуск конечного продукта? Иными словами, насколько

выпуск вашего конечного продукта зависит от определенных участников команды? Являются ли знания привилегией или находятся в общем доступе? Если вы осуществляете ротацию задач между парами, всегда найдется кто-то еще, обладающий знаниями, необходимыми для завершения работы. На состояние потока вашей команды «фактор грузовика» не повлияет.

- *Эффективно решаются проблемы*. Если вы программируете в паре и сталкиваетесь со сложной проблемой, вам всегда есть с кем ее обсудить. В таком диалоге варианты решения найдутся скорее, чем если вы будете биться над проблемой в одиночестве. В результате ротации задач внутри команды ваше решение будет повторно рассмотрено и критически оценено следующей парой, поэтому не столь важно, если ваше первоначальное решение окажется неоптимальным.
- *Плавно проходит интеграция*. Если ваша текущая задача требует обращения к другому фрагменту кода, можно надеяться, что названия методов, документы и тесты достаточно содержательны, чтобы получить представление о том, что этот код делает. Если это не так, то, работая в паре с автором этого фрагмента кода, вы сможете лучше понять и быстрее интегрировать его в свой код. Кроме того, в процессе обсуждения вы получите возможность улучшить именование, документацию и способы тестирования.
- *Можно безболезненно прерываться*. Если кто-то подошел к вам с вопросом, или звонит телефон, или нужно срочно ответить на письмо, или нужно принять участие в совещании, ваш партнер по парному программированию может продолжить работу над кодом. Когда вы вернетесь, ваш напарник все еще будет в состоянии потока, и вы сможете быстро наверстать упущенное и присоединиться к нему.
- *Новые участники команды быстро вливаются в проект*. Если в команде применяется парное программирование и правильно организована ротация пар и задач, новички быстро знакомятся как с кодом, так и с другими членами команды.

Поток дает невероятную продуктивность. Но это состояние легко утратить. Старайтесь всеми силами войти в рабочий поток, а затем, когда это получилось, удерживайтесь в нем!

Предпочитайте примитивам предметно-ориентированные типы данных

Эйнар Ландре

23 сентября 1999 года космический аппарат Mars Climate Orbiter стоимостью 327,6 миллионов долларов потерялся при выходе на орбиту Марса из-за программной ошибки на Земле. Ошибку впоследствии окрестили *смешением единиц измерений* (*metric mix-up*). Программное обеспечение наземной станции производило расчеты в фунтах силы, а космический аппарат ожидал указаний в ньютонах[1], в результате чего наземная станция недооценила мощность ускорителей аппарата в 4,45 раза.

Это один из многих примеров отказов программного обеспечения, которых можно было избежать благодаря более строгой и предметно-ориентированной типизации. Это также наглядная демонстрация назначения многих возможностей языка Ada, спроектированного преимущественно для создания встраиваемого отказоустойчивого программного обеспечения. В Ada применяется строгая типизация со статической проверкой как примитивных, так и определенных пользователем типов:

```
type Velocity_In_Knots is new Float range 0.0 .. 500.00;
type Distance_In_Nautical_Miles is new Float range 0.0 .. 3000.00;
Velocity: Velocity_In_Knots;
Distance: Distance_In_Nautical_Miles;
Some_Number: Float;
Some_Number:= Distance + Velocity; -- Компилятор отловит здесь
                                   ошибочное использование типов.
```

Разработчики приложений, сбои в которых менее критичны, также могут выиграть от более широкого применения предметно-ориентированной типизации. Предметно-ориентированные типы можно использовать вместо имеющихся в языках программирования и библиотеках базовых типов данных, таких как строки и числа с плавающей запятой. В Java, C++, Python и других современных языках абстрактный тип данных известен как `class`. Применение таких классов,

[1] 1 ньютон равен 0,224808943 фунта силы. – *Прим. ред.*

как `Velocity_In_Knots` (скорость в узлах) и `Distance_In_Nautical_Miles` (расстояние в морских милях) значительно повышает качество кода:

- Такой код легче читать, поскольку он выражает понятия предметной области, а не просто описывает строки (`String`) или действительные числа (`Float`).
- Такой код легче тестировать, потому что он инкапсулирует поведение, которое легко проверить.
- Такой код облегчает повторное использование в разных приложениях или системах.

Этот подход в равной мере пригоден для использования в языках как со статической, так и с динамической типизацией. Единственное отличие в том, что в языках со статической типизацией разработчик получает некоторую помощь от компилятора, тогда как при динамической типизации больше приходится полагаться на модульное тестирование. Могут различаться стили проверки, но подход и стиль выражения одинаковы.

Мораль: начинайте пробовать предметно-ориентированные типы с целью повышения качества разрабатываемых программ.

Предотвращайте появление ошибок

Жиль Колборн

Сообщения об ошибках – наиболее ответственный вид взаимодействия между пользователем и системой. Они возникают, когда общение пользователя с системой находится на грани разрыва.

Проще всего считать, будто ошибки возникают по вине пользователей в результате ввода неверных данных. Но ошибки, которые люди совершают, предсказуемы и происходят системно. Поэтому можно «отлаживать» взаимодействие между пользователем и системой так же, как вы отлаживаете взаимодействие между другими компонентами системы.

Допустим, пользователь должен ввести дату в определенном диапазоне. Чем позволять ему ввести любую дату, не лучше ли предоставить средство вроде списка или календаря, которое покажет только допустимые даты? Это исключит всякую возможность ввода даты за пределами разрешенного диапазона.

Другая распространенная проблема – ошибки форматирования. Например, если пользователь видит текстовое поле для даты и вводит однозначно трактуемую дату «29 июля 2012», неправильно будет забраковать ее только потому, что данные имеют не тот формат, который предпочитаете вы (например, «ММ/ДД/ГГГГ»). Еще хуже отклонить дату «29 / 07 / 2012» только из-за лишних пробелов; такие проблемы пользователям сложнее всего осознать, ведь им кажется, что дата имеет верный формат.

Ошибка возникает потому, что нам проще отклонить данные, чем разбирать три или четыре формата даты, получивших широкое распространение. Подобные мелкие ошибки раздражают пользователя, так что ему сложно сосредоточиться, и он делает все новые ошибки. Чтобы избежать этого, с пониманием отнеситесь к желанию пользователя вводить именно информацию, а не данные.

Другой способ избежать ошибки форматирования – предложить пользователю подсказку, например, с помощью метки в поле ввода, которая показывает нужный формат («ДД/ММ/ГГГГ»). Другой способ подсказать – разделить поле на три части по два, два и четыре символа.

Подсказки – это не то же самое, что инструкции: подсказки ненавязчивы и лаконичны, а инструкции многословны. Подсказки появляются в момент взаимодействия, а инструкции – до этого момента. Подсказки подчеркивают контекст, а инструкции диктуют поведение.

Обычно инструкции малоэффективны в предотвращении ошибок. Пользователи склонны считать, что интерфейсы должны действовать согласно их прежнему опыту («Любому должно быть понятно, что означает *29 июля 2012*!»). Поэтому инструкции никто не читает. Подсказки уводят пользователей от совершения ошибок.

Еще один способ избежать ошибок – предлагать значения по умолчанию. Например, пользователи, как правило, вводят значения, которые соответствуют датам *сегодня*, *завтра*, *мой день рождения*, *срок сдачи моего проекта* или *дате, указанной в этой же форме в прошлый раз*. В зависимости от контекста одна из этих дат вполне может оказаться хорошим вариантом для значения по умолчанию.

Какова бы ни была причина ошибки, совершенной пользователем, система должна ошибки прощать. Этому можно содействовать, обеспечив возможность многоуровневой отмены (*undo*) всех выполненных операций – в особенности тех, которые могут удалить или изменить данные пользователя.

Запись каждой операции отмены в файл журнала и его последующий анализ могут пролить свет на то, какие особенности интерфейса заставляют пользователей бессознательно совершать ошибки, например постоянно нажимать «не на ту» кнопку. Часто оказывается, что такие ошибки вызваны неудачными подсказками, которые вводят пользователя в заблуждение, или непродуманным порядком выполнения действий, который можно перепроектировать, чтобы предотвратить появление ошибок.

Какой бы подход вы ни избрали, большинство ошибок являются системными и возникают из взаимного недопонимания между пользователем и приложением. Если вы поймете, как пользователи думают, как они воспринимают информацию, как принимают решения и вводят данные, тогда вы сможете отладить взаимодействие между вашими программами и вашими пользователями.

Профессиональный программист

Роберт Мартин (Дядюшка Боб)

Кого можно считать профессиональным программистом?

Самая главная черта профессионального программиста – *личная ответственность*. Профессиональные программисты отвечают за свою карьеру, свои оценки, свои обязательства по срокам, свои ошибки и свое мастерство. Профессиональный программист не перекладывает эту ответственность на других.

- *Профессионал сам отвечает за свою карьеру.* Чтение и обучение – ваша ответственность. Быть в курсе последних достижений отрасли и технологий – ваша ответственность. Слишком часто программисты считают, что обучать их – задача работодателя. Извините, это совершенно неверно. Как вы думаете, врачи тоже так считают? А юристы? Нет, они учатся в свое свободное время и на собственные средства. Они проводят значительную часть свободного времени, изучая журналы и решения суда. Они поддерживают свой профессиональный уровень. Так должны поступать и мы. Отношения между вами и работодателем хорошо описаны в вашем контракте. Если коротко, работодатель обещает вам платить, а вы обещаете хорошо работать.
- *Профессионал отвечает за написанный код.* Он не выпускает код, если *не уверен* в его работе. Задумайтесь на минуту. Как вы можете считать себя профессионалом, если готовы выпустить код, в котором у вас нет уверенности? Профессиональные программисты ожидают, что отдел контроля качества *ничего* не найдет в их коде, потому что они *выпускают его не раньше, чем тщательно протестируют*. Конечно, группа контроля качества что-нибудь да обнаружит, ведь идеальных людей нет. Но как профессионалы мы должны *стремиться* к тому, чтобы контролю качества ничего не досталось.
- *Профессионал – это командный игрок.* Он отвечает за результат всей команды, а не только за свою работу. Профессионалы помогают друг другу, учат друг друга, учатся друг у друга и даже прикрывают друг друга при необходимости. Когда один участник команды спотыкается, другие вступаются за него, зная, что когда-нибудь им самим понадобится прикрытие.

- *Профессионал не приемлет длинных списков дефектов*. Огромный список дефектов – это признак неряшливой работы. Системы, в которых система управления дефектами содержит тысячи записей, – это трагедия беспечности. Более того, в большинстве проектов сама потребность в автоматизированной системе управления дефектами есть симптом беспечности. Только очень крупные системы могут иметь такое большое количество ошибок, что для управления ими требуется автоматизация.
- *Профессионал поддерживает порядок*. Он гордится своим мастерством. Его код понятен, хорошо структурирован и легко читается. Профессионалы следуют оговоренным стандартам и лучшим практикам. Они никогда, *ни при каких обстоятельствах* не работают впопыхах. Представьте себе, что вы можете покинуть собственное тело и наблюдать, как врач делает вам операцию на открытом сердце. У этого врача есть крайний (в буквальном смысле) срок завершения работы. Он должен закончить операцию до того, как аппарат искусственного сердца повредит слишком много кровяных клеток в вашем организме. Как, по-вашему, он должен себя вести? Вы бы хотели, чтобы он вел себя как типичный программист, пишущий код в спешке и беспорядке? Хотите ли вы услышать от него: «Как-нибудь потом все это исправлю»? Или все же он должен тщательно придерживаться правил своей науки, рассчитывать время и быть уверенным, что избранный им подход – лучший из доступных ему? Чего хотите вы – беспорядка или профессионализма?

Профессионалы обладают чувством ответственности. Они отвечают за собственную карьеру. Они отвечают за правильную работу своего кода. Они отвечают за уровень своего мастерства. Они не отказываются от своих принципов, когда на них давят сроки. На самом деле, когда давление растет, профессионалы еще крепче держатся за тот порядок, который считают правильным.

Держите все в системе управления версиями

Диомидис Спинеллис

Храните все, что касается любых ваших проектов, в системе управления версиями. Необходимые для этого ресурсы уже имеются: бесплатные инструменты типа Subversion, Git, Mercurial и CVS, вдоволь дискового пространства, дешевые и мощные серверы, повсеместный доступ в Интернет и даже службы хостинга проектов. После того как вы установили систему управления версиями, сохранить ваши труды в репозиторий очень просто: достаточно лишь выполнить соответствующую команду в чистом каталоге с кодом. А освоить нужно всего две новые основные операции: *запись* (*commit*) в репозиторий изменений, сделанных вами в коде, и *обновление* (*update*) вашей рабочей версии проекта до той, которая находится в репозитории.

После того как проект помещен в систему управления версиями, можно без труда просмотреть его историю, узнать, кто написал каждый фрагмент кода, и обратиться к конкретной версии файла или проекта с помощью уникального идентификатора. А что еще важнее – теперь вы можете делать рискованные изменения в коде, и больше не нужно оставлять закомментированный код на случай, если он потребуется в будущем. Ведь старая версия надежно хранится в репозитории. Можно (и нужно) *помечать* (*tag*) стабильные версии понятными вам именами, чтобы потом быстро получить именно ту версию, которая работает у вашего клиента. Можно создавать отдельные *ветви* (*branches*) и разрабатывать их параллельно: в большинстве проектов есть активно разрабатываемая ветвь и одна или несколько ветвей более ранних версий, для которых осуществляется активная поддержка.

Система управления версиями минимизирует трения между разработчиками. Когда программисты работают над независимыми частями программного обеспечения, их интеграция проходит «на ура». Когда они одновременно изменяют одни и те же файлы, система сообщит об этом и позволит разрешить конфликты. Можно настроить систему так, чтобы она оповещала всех разработчиков о каждом внесенном изменении, что даст каждому общее представление о ходе развития проекта.

Организуя работу над проектом, не жадничайте: поместите в систему управления версиями *все, что относится к проекту*. Помимо исходного кода занесите в репозиторий документацию, инструменты, сценарии для сборки, описания тестовых сценариев, графический материал и даже библиотеки. Когда весь проект надежно помещен в репозиторий (для которого регулярно делается резервная копия), возможный ущерб от потери диска или данных становится минимальным. Чтобы начать разработку на новой машине, достаточно получить копию (check out) проекта из репозитория. Это упрощает распространение, сборку и тестирование кода на разных платформах: на любой машине единственная команда обновления гарантирует вам загрузку последней версии программного обеспечения.

После того как вы оцените прелести работы с системой управления версиями, присмотритесь к следующим правилам, которые сделают вашу работу и работу вашей команды еще более эффективной:

- Сохраняйте каждое логическое изменение в виде отдельной операции. Если вы объедините большую кучу изменений в одну запись (commit), вам будет трудно разделить их впоследствии. Это особенно важно, когда проводится рефакторинг или изменение стиля в рамках всего проекта, что может легко скрыть другие модификации.
- Сопровождайте каждое изменение поясняющим сообщением. Как минимум кратко опишите, что вы изменили. И если вам требуется сохранить на будущее причины сделанных изменений, лучшего места не найти.
- Наконец, не стоит сохранять в репозиторий такой код, который ломает сборку проекта, иначе вы быстро навлечете на себя недовольство других участников проекта.

Жизнь с системой управления версиями слишком приятна, чтобы портить ее ошибками, которых легко избежать.

Брось мышь и медленно отойди от клавиатуры

Берк Хафнагель

Вы уже несколько часов работаете над какой-то неподдающейся задачей, а решения все не видно. Вы встаете, чтобы размять ноги, или направляетесь к автомату по продаже напитков, а на обратном пути ответ вдруг становится очевиден.

Случалось ли с вами такое? Приходилось ли вам задумываться, почему так происходит? Все дело в том, что, когда вы пишете код, активна логическая часть вашего мозга, а творческая отключена. Она никак не сможет себя проявить, пока логическая сторона не сделает перерыв в работе.

Вот вам пример из жизни. Я причесывал кое-какой старый код и наткнулся на «занятный» метод. Он предназначался для проверки правильности формата времени в строке вида *hh:mm:ss xx*, где *hh* – это часы, *mm* – минуты, *ss* – секунды, а *xx* принимает значение *AM* или *PM*.

Метод содержал следующий код для преобразования двух символов (представляющих час) в число и проверки, что час находится в заданном диапазоне:

```
try {
    Integer.parseInt(time.substring(0, 2));
} catch (Exception x) {
    return false;
}
if (Integer.parseInt(time.substring(0, 2)) > 12) {
    return false;
}
```

Тот же самый код появлялся еще дважды с соответствующими изменениями в смещении символов и в значении верхней границы, чтобы проверить правильность минут и секунд. Заканчивался метод следующими строками, проверяющими *AM* и *PM*:

```
if (!time.substring(9, 11).equals("AM") &
    !time.substring(9, 11).equals("PM")) {
    return false;
}
```

Если ни одно из этого ряда условий не оказывалось ложным (при этом возвращается `false`), метод возвращал `true`.

Если приведенный код кажется слишком многословным и трудным для понимания, не волнуйтесь. Мне тоже так показалось, и я решил, что нашел код, который стоит подчистить. Я переработал его и написал несколько модульных тестов, чтобы проверить, по-прежнему ли правильно работает новый код.

Закончив работу, я остался доволен результатами. Новый вариант легко читался, был вдвое меньшего размера и точнее, поскольку прежний код проверял только верхнюю границу часов, минут и секунд.

Когда я собирался на работу на следующий день, меня посетила идея: а почему бы не проверить правильность строки с помощью регулярного выражения? Через несколько минут у меня была рабочая реализация, состоящая всего из одной строки кода:

```
public static boolean validateTime(String time) {
    return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");
}
```

Смысл не в том, что в итоге мне удалось заменить 30 с лишним строк кода одной, а в том, что пока я не отошел от компьютера, мне казалось, что мой первый вариант был лучшим решением задачи.

Поэтому, когда вы в следующий раз столкнетесь с неподатливой задачей, сделайте себе одолжение. По-настоящему разобравшись в сути проблемы, займитесь чем-то, что включит творческую часть вашего мозга: нарисуйте схему проблемы на бумаге, послушайте музыку или просто выйдите из дома. Иногда лучшее, что вы можете сделать, чтобы решить задачу, – это бросить мышь и отойти от клавиатуры.

Читайте код

Карианне Берг

Мы, программисты, странные создания. Мы любим писать код. Но что касается чтения кода, мы обычно сторонимся этого дела. В конце концов, писать код гораздо увлекательней, а читать код трудно – иногда почти невозможно. Особенно тяжело читать код, написанный другими. Не всегда из-за того, что он плохо написан, но потому, что другой человек думает и решает задачи иначе, чем вы. А вам не приходило в голову, что чтение чужого кода может помочь улучшить ваш собственный код?

Когда вы в следующий раз будете читать какой-нибудь код, остановитесь и задумайтесь. Трудно его читать или легко? Если трудно, то почему? Он плохо отформатирован? Система именования непоследовательна или нелогична? Несколько задач смешались в одном фрагменте кода? Возможно, выбранный язык затрудняет чтение кода. Старайтесь учиться на чужих ошибках, чтобы не повторять их в своем коде. Вас могут ожидать сюрпризы. Например, приемы разрыва зависимостей могут быть полезны для введения слабого связывания, но они могут также затруднить чтение кода. А код, который одни считают *элегантным*, другие могут назвать *нечитаемым*.

Если код читается легко, обратите на него внимание и посмотрите, нельзя ли чему-то научиться у этого кода. Возможно, в нем применяется шаблон проектирования, который вам не знаком или который вы пытались реализовать ранее. Или он содержит методы более короткие и более точно именованные, чем ваши. В некоторых проектах с открытым исходным кодом можно встретить массу примеров великолепного, понятного кода, тогда как в других вы столкнетесь с прямо противоположным! Скачайте немного кода из репозиториев таких проектов и поизучайте его.

Чтение собственного старого кода из проекта, над которым вы больше не работаете, тоже может оказаться поучительным опытом. Начните с какого-нибудь самого старого кода и постепенно продвигайтесь к коду, который пишете сегодня. Возможно, вы обнаружите, что читать старый код совсем не так легко, как было в то время, когда вы его писали. Ваш ранний код может вызвать у вас определенное

смущение, типа того, которое испытываешь, когда тебе рассказывают о том, что ты говорил накануне вечером, выпивая в пабе. Посмотрите, как с годами росло ваше мастерство; это может стать весьма ободряющим открытием. Выясните, какие части кода тяжело читать, и подумайте, продолжаете ли вы писать код в том же стиле сегодня.

Итак, когда вы в следующий раз почувствуете необходимость улучшить свое мастерство программирования, не беритесь за книги. Читайте код.

Читайте гуманитарные книги

Кейт Брэйтуэйт

Во всех проектах, кроме самых маленьких, люди работают с другими людьми. Во всех исследовательских областях, кроме самых абстрактных, люди пишут программы для людей, чтобы помочь им достичь определенную цель. Люди пишут программы вместе с людьми и для людей. Этот бизнес строится вокруг людей. К сожалению, программистов обучают тому, что мало помогает им в общении с людьми, для которых и с которыми они работают. К счастью, существует целая область знаний, которая может в этом помочь.

К примеру, Людвиг Витгенштейн (Ludwig Wittgenstein) в своей книге «Philosophical Investigations»[1] (Wiley-Blackwell) и других работах весьма убедительно утверждает, что ни один человеческий язык не является – и не может являться – форматом сериализации для передачи идей или образов из головы одного человека в голову другого. Нам следует помнить об этом уже тогда, когда мы «собираем требования». Витгенштейн также показывает, что наша способность понимать друг друга связана не с общими определениями, а с единым опытом и образом жизни. Возможно, по этой причине у программистов, глубоко вникающих в предметную область, все получается лучше, чем у тех, кто держится от нее в стороне.

Лакофф (Lakoff) и Джонсон (Johnson) в своей книге «Metaphors We Live By»[2] (University of Chicago Press) утверждают, что язык в значительной степени метафоричен и что метафоры языка позволяют нам заглянуть в собственное восприятие мира. Даже такой на первый взгляд конкретный термин, как *денежный поток* (*cash flow*), употребляемый нами в разговоре о финансах, может быть понят метафорично: «деньги – это жидкость». А как эта метафора влияет на наши представления о системах, оперирующих деньгами? Или другой пример: мы говорим об уровнях в стеке протоколов, где есть протоколы высокоуровневые,

---

1 Людвиг Витгенштейн «Философские исследования», АСТ, 2011.

2 Лакофф Дж. и Джонсон М. «Метафоры, которыми мы живем», ЛКИ, 2008.

а есть низкоуровневые. Это сильная метафора: пользователь – «вверху», а технологии «внизу». Метафора вскрывает наше представление о структуре системы, которую мы строим. Она также может означать привычку мыслить шаблонно, которую мы время от времени можем с пользой для себя ломать.

Мартин Хайдеггер (Martin Heidegger) внимательно изучал пути познания человеком инструментов через опыт их применения. Программисты создают и используют инструменты, мы думаем об инструментах, создаем инструменты, модифицируем и переделываем их. Инструменты – это объекты нашего интереса. Но для их пользователей, как показывает Хайдеггер в своей книге «Being and Time»[1] (Harper Perennial), инструмент – невидимая сущность, постижение которой происходит только во время ее применения. Для пользователей инструменты становятся объектами интереса только тогда, когда они не работают. Об этой разнице стоит помнить, когда разговор идет об удобстве работы.

Элеонора Рош (Eleanor Rosch) опровергла модель категоризации Аристотеля, упорядочивающую наше понимание мира. Когда мы, программисты, спрашиваем пользователей, чего они хотят от системы, мы обычно стараемся получить определения, основывающиеся на предикатах. Нам это очень удобно. Термы в предикатах легко превращаются в атрибуты класса или колонки таблицы. Подобные категории являются точными, непересекающимися и логически верными. Но, к сожалению, как показывает Рош в «Natural Categories»[2] (Естественные категории) и более поздних своих работах, люди представляют себе мир иными способами. Представление людьми мира основано на примерах. Некоторые из таких примеров, так называемые прототипы, лучше других, и поэтому основанные на них категории размыты, пересекаются одна с другой, могут обладать насыщенной внутренней структурой. Пока мы не перестанем настаивать на ответах в духе системы категорий Аристотеля, мы не сможем задать пользователям правильные вопросы об их мире, и нам будет трудно добиться необходимого общего понимания.

---

1 Мартин Хайдеггер «Время и бытие», Наука, 2007.

2 «Cognitive Psychology» (Когнитивная психология) 4: 328-50 (1973).

Почаще изобретайте колесо

Джейсон П. Сэйдж

Пользуйтесь готовыми решениями – глупо снова изобретать колесо...

Приходилось вам слышать подобный совет? Ну, еще бы! Каждый разработчик или студент часто слышит такие высказывания. Но почему? Почему на повторное изобретение колеса так косо смотрят? Потому что существующий код, как правило, работает. Он уже в какой-то форме прошел контроль качества и строгое тестирование и теперь успешно применяется. Кроме того, время и силы, потраченные на изобретение нового решения, едва ли окупятся в той же мере, как использование готового продукта или кода. Стоит ли браться за изобретение колеса? Зачем? В каких случаях?

Вероятно, вы знакомы с публикациями о шаблонах в разработке или с книгами, посвященными проектированию программного обеспечения. Какой бы замечательной ни была предлагаемая в них информация, они часто вгоняют в сон. В какой мере просмотр фильма о парусном спорте отличается от самостоятельного хождения под парусом, в такой же мере использование готового кода отличается от процесса, когда вы разрабатываете собственное приложение с нуля, тестируете его, ломаете, чините и постепенно улучшаете его.

Изобретение колеса – это не просто упражнение в правильном размещении конструкций кода: оно требует глубокого понимания внутреннего устройства различных готовых компонентов. Вы знаете, как работают менеджеры памяти? Виртуальные страницы? Можете сами их реализовать? А как насчет двунаправленных связных списков? Классов динамических массивов? Клиентов ODBC? Можете написать графический интерфейс пользователя, который работает так же, как тот, что вы сейчас используете, и который вам нравится? Можете сами создать виджет для браузера? Знаете, когда нужно писать систему с мультиплексированием, а когда – многопоточную? Как решить, какая база данных лучше – в файле или в памяти?

Большинству разработчиков никогда не приходилось создавать такие базовые программные компоненты, а потому они лишь поверхностно разбираются в работе этих «колес». В результате все подобные модули и библиотеки считаются таинственным черным ящиком, который каким-то образом работает. Если вы видите только поверхность воды, этого недостаточно для понимания опасностей, таящихся в ее глубине. Если вы ничего не знаете о создании более низкоуровневого программного обеспечения, это ограничивает вашу способность создавать действительно выдающиеся продукты.

Попытаться изобрести колесо и потерпеть неудачу полезнее успеха с первой же попытки. Уроки, полученные методом проб и ошибок, имеют эмоциональную составляющую, которую простое чтение технической книги не способно создать!

Запоминать факты и читать умные книги важно, но в становлении выдающегося программиста приобретение опыта имеет такое же значение, как накопление фактов. Изобретение колеса так же важно для образования и мастерства программиста, как поднятие тяжестей для культуриста.

Не поддавайтесь очарованию шаблона Singleton

Сэм Сааристе

Шаблон Singleton решает многие ваши проблемы. Вы знаете, что вам нужен единственный экземпляр. Вы получаете гарантию, что этот экземпляр будет инициализирован перед использованием. Архитектура остается простой благодаря наличию глобальной точки доступа. Все прекрасно. Ну что может не понравиться в этом классическом шаблоне проектирования?

Если подумать, то весьма многое. Как ни соблазнительно применение синглтонов, они, как показывает опыт, приносят больше вреда, чем пользы. Они затрудняют тестирование и усложняют сопровождение. К сожалению, это понимание не столь распространено, как хотелось бы, и синглтоны сохраняют свое обаяние для множества программистов. Хотя есть основания задуматься, так ли они хороши:

- *Ограничение на число экземпляров класса часто иллюзорно*. Во многих случаях утверждение, что в будущем не понадобятся дополнительные экземпляры, ничем не подкреплено. Домыслы в основе архитектуры приложения обязательно приведут к неприятностям в будущем. Технические требования меняются. Хорошая архитектура это учитывает. Синглтоны – нет.
- *Синглтоны создают неявные зависимости между концептуально независимыми модулями кода*. Беда в том, что, во-первых, эти зависимости незаметны, а во-вторых, создают ненужные связи между модулями. Этот запашок в коде становится острее, когда вы пытаетесь писать модульные тесты, основанные на слабом связывании и возможности выборочно применять реализации-макеты вместо настоящих. Синглтоны не дают осуществлять такое простое моделирование.
- *Синглтоны неявно хранят состояние, что опять-таки препятствует модульному тестированию*. Модульное тестирование предполагает, что тесты независимы друг от друга, благодаря чему их можно выполнять в любом порядке, а программу можно возвращать в известное состояние перед выполнением каждого модульного теста. Как только появляются синглтоны с изменяемым (mutable) состоянием, обеспечить такие условия может оказаться

затруднительно. Кроме того, такое глобально доступное долгоживущее состояние затрудняет интерпретацию кода человеком, особенно в многопоточной среде.

- *Многопоточность создает дополнительные капканы в использовании шаблона синглтона*. Поскольку простая блокировка доступа не очень эффективна, получила распространение так называемая блокировка с двойной проверкой (DCLP). К несчастью, иногда это просто другая разновидность рокового влечения. Оказалось, что во многих языках DCLP не является потоково-безопасной, и даже в тех, где она является потоково-безопасной, сохраняются возможности для ее неправильной работы.

Избавление от синглтонов может стать в окончательном итоге сложной задачей:

- *Явное уничтожение объектов синглтонов не поддерживается*. В отдельных контекстах это может оказаться проблемой, например в архитектуре с подключаемыми модулями, где модуль можно безопасно выгрузить, только если все его объекты удалены из памяти.
- *При завершении программы порядок неявной зачистки синглтонов не определен*. Это может вызвать проблемы в приложениях, содержащих синглтоны с взаимными зависимостями. При завершении таких приложений одни синглтоны могут продолжать обращаться к другим, которые к тому моменту уже уничтожены.

Некоторые из перечисленных недостатков можно преодолеть с помощью специальных механизмов. Однако за это приходится расплачиваться усложнением кода, чего удалось бы избежать, если бы в проекте использовались иные подходы к архитектуре.

Поэтому ограничьте использование шаблона Singleton теми классами, для которых действительно не должно никогда создаваться более одного экземпляра. Не стоит пользоваться глобальной точкой входа в синглтон в произвольных участках кода. Прямое обращение к синглтону должно происходить лишь в нескольких четко определенных местах и быть доступным коду в целом только через узкий интерфейс. Весь остальной код не знает, как реализован интерфейс – через синглтон или какой-то другой класс, – а потому не зависит от реализации. В результате всего этого разрушаются связи, мешающие модульному тестированию, и облегчается сопровождение. Так что надеюсь, что, когда вы в следующий раз решите реализовать синглтон или к нему обратиться, вы дважды подумаете, стоит ли это делать.

Путь к повышению эффективности программ заминирован грязным кодом

Кирк Пеппердин

Как правило, настройка производительности системы требует изменения исходного кода. Когда необходимо изменить код, каждый его фрагмент, слишком сложный или сильно связанный с другими, оказывается «бомбой грязного кода», способной свести на нет все ваши усилия. Первой жертвой грязного кода становится график работ. Если движение вперед происходит равномерно, легко предсказать, когда закончится работа. Но неожиданные столкновения с грязным кодом делают весьма затруднительным разумное планирование.

Допустим, вы обнаружили место, где теряется производительность. В этом случае обычно пытаются снизить сложность алгоритма, создающего недопустимую нагрузку. Вы сообщаете своему менеджеру, что исправление займет у вас, скажем, три-четыре часа. Работая над исправлениями, вы обнаруживаете, что перестал работать зависимый участок кода. Родственные фрагменты кода часто связаны друг с другом, что вызвано объективной необходимостью, и такое нарушение работы, скорее всего, предполагалось и учитывалось в оценке времени. Но что если исправление этой зависимости приведет к нарушению работы и других зависимых частей? Более того, чем дальше эти зависимости находятся от исходной точки, тем менее вероятно, что вы их обнаружите и учтете в своей оценке. Внезапно начальная оценка раздувается от трех-четырех часов до трех-четырех недель. Часто одна подобная неожиданность увеличивает сроки сразу на 1–2 дня. Не столь уж редко «небольшой» рефакторинг затягивается на несколько месяцев. В таких ситуациях ущерб, нанесенный доверию к действующей команде и ее «политическому капиталу», может быть тяжелым и даже смертельным. Вот если бы у нас был инструмент, позволяющий обнаружить и оценить такой риск...

На самом деле, существует много способов измерить и проконтролировать степень и глубину связанности и сложности нашего кода. С помощью программных метрик можно посчитать встречаемость в коде определенных характеристик. Эти количественные показатели коррелируют с качеством кода. Отметим две метрики из тех, которые оценивают связанность кода: *число входов* (*fan-in*) и *число выходов* (*fan-out*). Например, для классов fan-out определяется как количест-

во классов, к которым прямо или косвенно обращается оцениваемый класс. Это можно представить себе как количество всех классов, которые нужно скомпилировать, прежде чем можно начать компилировать ваш класс. С другой стороны, fan-in – это количество классов, которые зависят от данного класса. Зная fan-out и fan-in, можно по формуле $I = fo / (fi + fo)$ рассчитать коэффициент нестабильности. Чем ближе I к нулю для конкретного класса или модуля, тем он стабильнее. Когда значение I близко к 1, модуль становится нестабильным. Изменение кода стабильных модулей сопряжено с меньшим риском, тогда как в нестабильных модулях более вероятно наличие бомб грязного кода. Задача рефакторинга – приблизить I к нулю.

При использовании метрик нужно помнить, что это всего лишь эмпирические правила. Исходя из чистой математики мы увидим, что увеличивая fi, не изменяя при этом fo, мы приближаем I к нулю. Однако слишком большая величина fan-in имеет и обратную сторону: такие классы труднее изменить, не нарушая работу классов, зависимых от них. Кроме того, не решая проблемы fan-out, вы на самом деле не снижаете свои риски, поэтому нужно соблюдать некоторое равновесие.

Недостатком программных метрик является то, что огромное количество цифр, выдаваемых инструментами для снятия метрик, может произвести устрашающее впечатление на непосвященных. Тем не менее программные метрики способны стать мощным средством в борьбе за чистый код. Они помогают обнаружить и ликвидировать бомбы грязного кода, составляющие серьезный риск для операций по повышению производительности.

Простота достигается сокращением

Пол У. Гомер

«Делай заново...», – сказал мне начальник, твердо удерживая пальцем клавишу Delete. С привычной тоской я смотрел на экран, где безвозвратно исчезал мой код – строка за строкой.

Мой начальник Стефан не отличался особой красноречивостью, но определял плохой код с первого взгляда. И он хорошо знал, что с ним нужно делать.

Я поступил на ту работу в качестве человека, изучающего программирование, – с запасами энергии и энтузиазма, но без малейшего понятия, как писать код. Я пребывал в ужасном заблуждении, будто любые проблемы решаются путем добавления еще одной переменной в подобающем месте либо путем добавления еще одной строки кода. В плохие дни мой код деградировал – его логика не совершенствовалась, более того, он становился все пространнее, сложнее и неустойчивее.

Желание решить вопрос посредством минимальных изменений в блоке кода – даже если они ужасны – вполне естественно, особенно при недостатке времени. Большинство программистов сохраняет плохой код, опасаясь, что, если начать все заново, потребуется гораздо больше усилий, чем если просто вернуться к началу. Это бывает верно для почти рабочего кода, но встречается и такой код, которому уже ничто не поможет.

Слишком много времени впустую тратится на попытки спасти плохую работу. Если что-то начинает высасывать ресурсы, от этого следует избавиться. И побыстрее.

Не хочу сказать, что так уж легко расстаться с набранным текстом, выбранными именами, форматированием. Реакция моего начальника была излишне жесткой, однако она заставляла меня переосмысливать свой код во время второй (а иногда и третьей) попытки. Тем не менее лучший способ исправить плохой код – приготовиться безжалостно его перерабатывать, переносить туда-сюда или удалять.

Код должен быть простым. Количество переменных, функций, объявлений и прочих синтаксических элементов языка должно быть минимальным. Лишние

строки, лишние переменные... *все* лишнее должно немедленно уничтожаться. А того, что осталось, должно быть минимально достаточно, чтобы выполнить задание, реализовать алгоритм или осуществить вычисления. Все прочее – бесполезный, ненужный шум, появившийся по ошибке. Он запутывает алгоритм работы и скрывает действительно важные вещи в коде.

Конечно, если этого окажется недостаточно, удалите вообще все и начните сначала. Такое «рисование по памяти» часто позволяет избавиться от ненужного хлама.

Принцип единственной ответственности

Роберт Мартин (Дядюшка Боб)

Вот один из наиболее фундаментальных принципов качественного проектирования:

> Собрать вместе те вещи, которые изменяются по одной и той же причине, и разделить те, которые изменяются по разным причинам.

Этот принцип иначе известен как *принцип единственной ответственности*, или SRP (single responsibility principle). Если коротко, он гласит, что причина изменения любой подсистемы, класса и даже функции должна быть ровно одна. Классический пример – класс с методами для работы с бизнес-правилами, отчетами и базами данных:

```
public class Employee {
public Money calculatePay() ...
public String reportHours() ...
public void save() ...
}
```

Кое-кто из программистов считает, что сочетание этих трех функций в одном классе вполне уместно. В конце концов, классы и должны собирать вместе функции, работающие с одними и теми же переменными. Однако проблема в том, что все три функции изменяются по совершенно разным причинам. Функция `calculatePay` (рассчитать зарплату) изменяется вместе с бизнес-правилами расчета заплаты. Функция `reportHours` (отчитаться о часах) изменяется, когда требуется другой формат отчета. Функция `save` (сохранить) изменяется, когда администратор базы данных меняет схему базы данных. В совокупности эти три причины делают `Employee` очень неустойчивым классом. Он будет меняться по *каждой* из этих причин. И что еще важнее, эти изменения затронут любые классы, которые зависят от `Employee`.

В хорошей архитектуре система разделена на компоненты, которые можно развернуть независимо. Независимость развертывания означает, что изменение одного компонента не требует повторного развертывания других. Но если `Employee`

интенсивно используется многими классами в других компонентах, каждое изменение `Employee` может требовать повторного развертывания других компонентов, что сводит на нет основное преимущество компонентного подхода к проектированию (или SOA, если вам нравится это более модное название). Следующее простое разделение решает проблему:

```
public class Employee {
    public Money calculatePay() ...
}
public class EmployeeReporter {
    public String reportHours(Employee e) ...
}
public class EmployeeRepository {
    public void save(Employee e) ...
}
```

Каждый класс можно поместить в отдельный компонент. Или, выражаясь точнее, все классы, создающие отчеты, поместить в компонент отчетов. Все классы, связанные с базами данных, можно поместить в компонент хранилища. А все бизнес-правила поместить в компонент бизнес-правил.

Внимательный читатель заметит, что в приведенном решении остаются зависимости. Другие классы по-прежнему зависят от `Employee`. Поэтому если `Employee` изменится, вполне возможно, что эти классы придется заново скомпилировать и развернуть. Таким образом, `Employee` невозможно модифицировать, а потом развернуть независимо. Однако другие классы можно модифицировать и независимо развернуть. Никакая их модификация не требует перекомпиляции и повторного развертывания остальных классов. Даже `Employee` можно независимо развернуть, если тщательно применить *принцип инверсии зависимости* (DIP, dependency inversion principle), но это уже тема для другой книги.[1]

Продуманное применение принципа единственной ответственности, то есть разделение тех сущностей, которые изменяются по разным причинам, является одной из основ создания архитектур со структурой независимо развертываемых компонентов.

[1] *http://www.amazon.com/dp/0135974445/*

Сначала скажите «да»

Алекс Миллер

Недавно я был в продуктовом магазине и обыскался там эдамаме, зеленых соевых бобов (я лишь приблизительно догадывался, что это какие-то овощи). Я не знал, где мне искать этот продукт: в овощном отделе, в отделе замороженных продуктов или на полках с консервами? Наконец я сдался и обратился за помощью к сотруднице магазина. Она тоже не знала!

Сотрудница магазина могла отреагировать на мою просьбу по-разному. Она могла дать мне понять, что только дурак не знает, где это искать, или отделаться туманными намеками, или даже просто сказать, что у них нет такого товара. Но она посчитала возможным найти решение и помочь покупателю. Позвав других сотрудников, она уже через пару минут отвела меня к нужному товару. Эдамаме оказались в отделе замороженных продуктов.

В данной ситуации сотрудница выслушала просьбу и стала исходить из того, что проблема решится, и просьба будет удовлетворена. Она начала с того, что сказала *да* вместо *нет*.

Когда мне впервые пришлось занять должность технического руководителя, мне казалось, что главное в моей работе – защищать мои прекрасные программы от потока нелепых требований, исходившего от менеджеров по продукту и бизнес-аналитиков. В большинстве разговоров я рассматривал любую просьбу как атаку, требующую защитной реакции, а не положительного ответа.

В какой-то момент на меня снизошло прозрение, и я увидел, что возможен другой подход к работе, который отличается от моего всего лишь начальным *да* вместо начального *нет*. Я пришел к убеждению, что начинать разговор с *положительного* ответа – важнейшая часть работы технического руководителя.

Эта простая перемена радикально изменила мой подход к работе. Оказалось, существует много способов говорить. Когда слышишь: «Послушай, это приложение станет в сто раз круче, если мы сделаем все окна круглыми и прозрачными!», можно отвергнуть это предложение как нелепое. Но обычно лучше сначала спросить: «А почему?». Часто существует реальный и убедительный довод, почему

этот человек требует круглые прозрачные окна. Например, ведутся переговоры с новым крупным клиентом, у которого комиссия по стандартизации требует эти самые круглые прозрачные окна.

Обычно, ознакомившись с контекстом просьбы, обнаруживаешь, что открываются новые возможности. Зачастую просьба может быть выполнена в рамках существующего продукта каким-то другим способом, что позволяет ответить *да*, нисколько не напрягаясь: «Собственно говоря, вы можете загрузить в пользовательских настройках «шкурку» (skin) с круглыми прозрачными окнами и активизировать ее».

Иногда у людей появляются идеи, которые кажутся несовместимыми с вашим видением продукта. Я обнаружил, что в этом случае обычно полезно обратиться с вопросом «почему» к самому себе. Порой сама попытка объяснить себе причину помогает понять ошибочность первой реакции. Если это не так, то можно облегчить ситуацию, включив в обсуждение других ответственных лиц. Помните, что цель всего этого – сказать *да* другому человеку и попытаться сделать так, чтобы все получилось, и не только ради него, но ради себя и всей своей команды.

Если вы сможете убедительно объяснить, почему предложенная функция не совместима с существующим продуктом, это даст возможность конструктивно обсудить, нужный ли продукт вы создаете. Независимо от результатов обсуждения каждый участник более четко осознает, чем продукт является, а чем не является.

Начинать с *да* – значит работать вместе со своими коллегами, а не против них.

Шаг назад. Теперь автоматизируй, автоматизируй, автоматизируй...

Кэй Хорстман

Я работал с программистами, которые в ответ на просьбу посчитать количество строк кода в модуле открывали файлы в текстовом процессоре и пользовались функцией «счетчик строк». То же самое они повторяли через неделю. И еще через неделю. Это было ужасно.

Я участвовал в проекте с неуклюжей процедурой развертывания: требовалось подписать код, скопировать подписанный код на сервер, выполнить множество щелчков мышью. Кто-то автоматизировал процедуру, и этот сценарий запускался сотни раз во время финального тестирования – гораздо чаще, чем ожидалось. Это было замечательно.

Так почему же люди делают одну и ту же работу многократно, вместо того чтобы остановиться и потратить время на ее автоматизацию?

Распространенное заблуждение №1: автоматизация нужна только для тестирования

> Да, автоматизация тестирования – это классно, но зачем останавливаться на этом? Любой проект изобилует повторяющимися задачами: управление версиями, компиляция, сборка JAR-файлов, генерация документации, развертывание системы и составление отчетов. Для многих из перечисленных задач эффективнее использовать сценарий, а не указатель мыши. Выполнение рутинных задач ускоряется и становится надежным.

Распространенное заблуждение №2: у меня есть IDE, поэтому мне не нужна автоматизация

> Приходилось ли вам участвовать в спорах с коллегами на тему «А на моей машине все компилируется (все загружается из репозитория, все тесты проходят)»? Современные IDE предлагают тысячи настроек, и фактически невозможно гарантировать, что у всех участников команды настройки будут одинаковыми. Системы автоматизации сборки, такие как Ant или Autotools, обеспечивают контроль и повторяемость.

Распространенное заблуждение №3: для автоматизации придется изучать экзотические инструменты

Можно успешно построить систему автоматизации на базе хорошего языка командной оболочки (например, bash или PowerShell). Если требуется взаимодействовать с веб-сайтами, воспользуйтесь такими инструментами, как iMacros или Selenium.

Распространенное заблуждение №4: я не могу автоматизировать эту задачу, потому что не смогу работать с файлами этого формата

Если ваш процесс требует работы с документами Word, электронными таблицами или графикой, автоматизация действительно может стать сложной проблемой. Но так ли вам необходимы эти форматы? А нельзя ли использовать обычный текст? Значения, разделяемые запятой? XML? Средства генерации графики из текстовых файлов? Часто достаточно немного изменить процесс, чтобы получить хорошие результаты и резко сократить рутинные операции.

Распространенное заблуждение №5: у меня нет времени со всем этим разбираться

Чтобы начать, вам не обязательно досконально изучать bash или Ant. Учитесь на ходу. Когда вы видите задачу, которую можно и нужно автоматизировать, изучите свои инструменты лишь настолько, насколько требуется в данный момент. И занимайтесь этим в самом начале проекта, когда обычно легче найти время. После первого удачного опыта вы (и ваш начальник) убедитесь, что автоматизация окупает потраченные усилия.

Пользуйтесь инструментами для анализа кода

Сара Маунт

Важность тестирования вколачивают в головы разработчиков программного обеспечения, когда они делают свои первые шаги на этом поприще. Широко распространившиеся в последние годы модульное тестирование, разработка на основе тестирования и методы гибкого программирования свидетельствуют о росте интереса к максимально эффективному использованию тестирования на всех стадиях цикла разработки. Однако тестирование – лишь один из многочисленных инструментов, с помощью которых можно повысить качество кода.

В те далекие времена, когда язык C еще был молод, процессорное время и память всех видов обходились очень дорого. В первых компиляторах C это учитывалось, поэтому количество проходов по коду сокращалось за счет отказа от некоторых видов семантического анализа. В результате компилятор проверял лишь небольшую часть тех ошибок, которые можно было обнаружить на этапе компиляции.

Чтобы компенсировать этот недостаток, Стивен Джонсон написал инструмент под названием *lint* – для прочесывания кода на предмет блох, – и реализовал в нем некоторые методы статического анализа, исключенные из компилятора C. Однако средства статического анализа прославились тем, что давали большое количество ложных срабатываний и часто предупреждали о нарушении стилистических правил, следовать которым необязательно.

Сегодняшний пейзаж языков, компиляторов и средств статического анализа весьма отличается от прежнего. Память и процессорное время стали относительно дешевыми, поэтому компиляторы могут позволить себе проверку большего числа ошибок. Почти для каждого языка существует хотя бы один инструмент, выявляющий нарушения стилистических правил, стандартные ошибки и иногда трудные для обнаружения ошибки типа возможного разыменования нулевого указателя. Более сложные инструменты, такие как Splint для C или Pylint для Python, допускают настройку, то есть возможность выбрать ошибки и предупреждения, о которых будет сообщать инструмент, с помощью файла конфигурации, параметров командной строки или настроек IDE. Splint даже позволяет аннотировать код комментариями, поясняющими работу программы.

Если эти средства не помогают, и вам приходится искать простые ошибки или нарушения стандартов, которые не обнаруживают ваш компилятор, IDE или инструмент типа lint, можно написать собственный статический анализатор. Это не так трудно, как может показаться. Большинство языков, особенно *динамических*, в составе стандартной библиотеки предлагают абстрактное синтаксическое дерево и инструменты компилирования. Вам не помешает заглянуть в дальние уголки стандартных библиотек, используемых разработчиками языка, с которым вы работаете, так как там встречаются сокровища, полезные для статического анализа и динамического тестирования. Так, в стандартной библиотеке Python есть дизассемблер, который сообщит вам, какой байт-код использовался для генерации некоторого скомпилированного кода или объекта. Многим покажется, будто это какой-то загадочный инструмент для команды разработчиков python-dev, разрабатывающей компилятор Python, но для повседневной работы он удивительно полезен. Например, с помощью этой библиотеки можно дизассемблировать последнюю трассировку стека и точно выяснить, какая команда байт-кода сгенерировала последнее необработанное исключение.

Так что не ограничивайте свой контроль качества одним тестированием – используйте инструменты для анализа и не бойтесь создавать свои собственные.

Тестируйте требуемое, а не случайное поведение

Кевлин Хенни

Типичное заблуждение при тестировании – воображать, что тестировать необходимо именно то, что делает реализация. На первый взгляд в этом не видно ничего дурного. Однако если сформулировать эту проблему иначе, она становится понятнее: частой ошибкой при тестировании является привязка тестов к особенностям реализации, тогда как эти особенности являются случайными и не имеют отношения к требуемой функциональности.

Когда тесты жестко связаны с особенностями реализации, изменения в реализации, которые в действительности совместимы с требуемым поведением, могут привести к отказу тестов. Из-за этого будут возникать ложные сообщения об ошибках. Обычно программисты реагируют на это исправлением теста или исправлением кода. Принятие ложного срабатывания за истинное часто есть следствие страха, неуверенности и сомнений. Это равносильно принятию случайного поведения за требуемое. При исправлении теста программист либо изменяет тест так, чтобы он проверял требуемое поведение (что хорошо), либо привязывает тест к новой реализации (что плохо). Тесты должны быть достаточно точными, но они должны быть и правильными.

Например, требования к тройственному сравнению, такому как `String.compareTo` в Java или `strcmp` в C, таковы: результат должен быть отрицательным, если левая часть меньше правой, положительным, если левая часть больше правой, и нулем, если они равны. Такого рода сравнение используется во многих API, включая функцию сравнения для `qsort` в C и `compareTo` в интерфейсе Java `Comparable`. Обычно в реализациях для обозначения *меньше* и *больше* используются конкретные значения `-1` и `+1`, в связи с чем программисты иногда ошибочно предполагают, что именно эти значения и выражают действительное требование, и пишут тесты, основанные на этом предположении.

Похожая проблема возникает, когда тесты жестко привязаны к количеству пробелов, наличию определенных слов и другим аспектам форматирования и представления текста, которые являются особенностью реализации. Наличие пробелов не должно влиять на результат, если только вы не пишете, скажем, XML-ге-

нератор с настройкой форматирования. Аналогичным образом жестко фиксированное расположение кнопок и меток на элементах управления пользовательского интерфейса уменьшает возможность в будущем менять эти несущественные детали. Мелкие изменения в реализации и несущественные изменения в форматировании внезапно приводят к тому, что проект не собирается.

Слишком конкретные тесты часто создают проблему при модульном тестировании по принципу белого ящика. Тесты белого ящика используют структуру кода, чтобы определить, какие тесты нужны. Типичная ошибка при таком тестировании – создание тестов, которые в конечном итоге проверяют, что код делает то, что он делает. Простое подтверждение того, что и так очевидно из кода, не имеет никакой ценности и приводит к возникновению ложного чувства прогресса и надежности.

Чтобы быть эффективными, тесты должны проверять обязательства по контракту, а не бессмысленно повторять реализацию. Они должны рассматривать тестируемые модули как черные ящики, описывая контракты интерфейса в исполняемой форме. Поэтому следите, чтобы проверяемое поведение совпадало с требуемым поведением.

Тестируйте точно и конкретно

Кевлин Хенни

При тестировании модуля кода важно проверять его требуемое и существенное поведение, а не случайное поведение конкретной реализации. Но из этого не следует, что тесты могут быть неопределенными. Тесты должны быть точными *и* конкретными.

В качестве иллюстрации можно взять тестирование проверенных и надежных классических процедур сортировки. Программисты не так часто пишут код алгоритмов сортировки, и все же каждый считает, будто знает, каким должен быть результат. Ведь сортировка так хорошо всем знакома. Однако это поверхностное знакомство может помешать обнаружить определенные неверные допущения.

Если спросить у программиста, что он собирается тестировать, в подавляющем большинстве случаев он ответит что-то вроде: «Нужно проверить, что результат будет отсортированной последовательностью элементов». Это правда, но не вся правда. Если программисту предложить уточнить проверяемое условие, то многие добавят, что результирующая последовательность должна быть той же длины, что и исходная. Верно, но и этого мало. Возьмем, например, последовательность:

```
3 1 4 1 5 9
```

Последовательность, приведенная ниже, удовлетворяет постусловию сортировки в неубывающем порядке и условию равенства длин исходной и результирующей последовательностей:

```
3 3 3 3 3 3
```

Хотя результат соответствует спецификации, едва ли это то, что мы хотели получить! Пример взят из реального производственного кода (к счастью, ошибка была обнаружена до его выпуска). В этом случае опечатка или секундная потеря концентрации привела к тому, что сложный алгоритм всего лишь заполнял результат первым элементом исходного массива.

Полное постусловие состоит в том, что результат отсортирован *и* содержит перестановку исходных значений. Оно накладывает правильные ограничения на требуемое поведение. То, что результат имеет одинаковую длину с входными данными, очевидно и не требует повторения.

Даже такая формулировка постусловия не гарантирует, что это хороший тест. Хороший тест должен легко читаться. Он должен быть достаточно понятным и простым, чтобы сразу можно было увидеть, корректен он или нет. Если у вас нет готового кода для проверки того, что последовательность отсортирована и что одна последовательность содержит перестановку значений другой, не исключено, что тестирующий код окажется сложнее, чем тестируемый. Как заметил Тони Хоар (Tony Hoare):

> Есть два способа конструировать программное обеспечение: можно сделать его таким простым, чтобы отсутствие дефектов было очевидно, а можно сделать таким сложным, что в нем не будет очевидных дефектов.

Использование конкретных примеров устраняет такую случайно внесенную сложность и возможность случайности вообще. Пусть, например, дана последовательность:

```
3 1 4 1 5 9
```

Результат сортировки таков:

```
1 1 3 4 5 9
```

Никакой другой результат не подойдет. Другого ответа быть не может.

Конкретные примеры помогают проиллюстрировать общее поведение доступным и однозначным способом. Результатом добавления элемента в пустую коллекцию будет не только то, что она станет непустой: в коллекции должен появиться один элемент, причем он будет равен добавленному. Два или более элементов тоже означают, что коллекция не пуста, но это ошибка. Один элемент в коллекции, но с другим значением – тоже ошибка. Результатом добавления строки в таблицу является не просто то, что в ней становится на одну строку больше; следует проверить и то, что по ключу этой строки можно найти ее в таблице. И так далее.

Описывая поведение, тесты должны быть не только правильными – они должны быть точными.

Тестируйте во сне (и по выходным)

Раджит Ammanamty

Успокойтесь. Я имею в виду не оффшорные центры разработки программного обеспечения, не сверхурочную работу по выходным и не ночные смены. Я просто хочу обратить ваше внимание на то, какая огромная вычислительная мощь находится в нашем распоряжении. Точнее, как мало мы ее используем, чтобы хоть немного облегчить жизнь программиста. Вы постоянно сталкиваетесь с нехваткой вычислительной мощности в течение рабочего дня? Если так, то чем заняты ваши серверы тестирования в нерабочее время? Очень часто тестовые серверы простаивают ночью и по выходным. Вы можете использовать этот резерв.

- *Был ли за вами грешок сохранить изменения в репозиторий, не прогнав все тесты?* Одна из главных причин, по которой программисты не прогоняют наборы тестов перед сохранением изменений в репозиторий, – тестирование выполняется слишком долго. Когда горят сроки и растет давление, программист вполне естественно начинает срезать углы. Одно из решений этой проблемы – разбить набор тестов хотя бы на два профиля. Создайте меньший обязательный профиль тестов, который быстро отрабатывает и который можно будет запускать перед каждым сохранением. Все остальные профили (и обязательный на всякий случай тоже) можно автоматизировать и запускать по ночам, чтобы иметь к утру готовый результат.
- *У вас была возможность проверить стабильность работы вашего продукта?* Для выявления утечек памяти и других проблем со стабильностью критически важно проводить тесты, которые выполняются часами и сутками. Их редко запускают в дневное время, потому что они отнимают время и ресурсы. Зато можно автоматически выполнять нагрузочное тестирование в ночное время и по выходным. С 6 вечера пятницы до 6 утра понедельника у вас есть 60 часов, которые можно занять тестированием.
- *Удается ли вам получить доступ к среде для тестирования производительности в удобное для вас время?* Мне приходилось видеть, как команды ругаются одна с другой, выбивая себе доступ к среде для тестирования производительности. Мало кому удается получить в достаточном количестве удобное

время для тестирования в рабочие часы, хотя по окончании рабочего дня серверы практически простаивают. В то же время серверы и сеть не так сильно загружены ночью и по выходным. Это идеальное время для выполнения тестов на производительность и получения надежных результатов.

- *Не слишком ли много у вас разных конфигураций, чтобы тестировать их вручную?* Часто продукт рассчитан на работу на нескольких платформах. Например, 32-разрядные и 64-разрядные версии для Linux, Solaris и Windows или просто для нескольких версий одной операционной системы. Дело осложняется еще и тем, что современные приложения допускают применение множества транспортных механизмов и протоколов (HTTP, AMQP, SOAP, CORBA и т. д.). Проверка всех возможных сочетаний требует очень много времени и чаще всего выполняется ближе к выпуску продукта в силу ограниченности ресурсов. Увы, некоторые ужасные ошибки обнаруживаются лишь на этой стадии, когда уже слишком поздно.

Запуск автоматизированных тестов по ночам или на выходных позволит чаще проверять все эти сочетания. Стоит чуть пораскинуть мозгами и приложить некоторые познания в написании сценариев, и вы сможете создать с помощью планировщика *cron* несколько заданий, которые будут запускать тесты по ночам и на выходных. Существует также множество полезных инструментов для тестирования. В некоторых организациях созданы даже серверные сетки, распределяющие серверы между разными отделами и командами с целью более эффективного использования ресурсов. Если в вашей организации есть такие средства, вы можете отдать тесты на выполнение в ночное время или на выходных.

Тестирование – это инженерная строгость в разработке программного обеспечения

Нил Форд

Разработчики программного обеспечения обожают использовать вымученные метафоры, пытаясь рассказать о своей работе родственникам, супругам и прочим далеким от техники людям. Часто мы ссылаемся на такие области, как строительство мостов, или другие «строгие» инженерные дисциплины. Но все эти метафоры быстро разваливаются на части, стоит их шатнуть чуть сильнее. Оказывается, что разработка программного обеспечения во многих ключевых аспектах *отличается* от «строгих» инженерных дисциплин.

В сравнении с «реальной» инженерией разработка программ находится примерно на том уровне, где были строители мостов в далеком прошлом. В те дни стандартный подход был такой: сначала построить мост, а потом пустить по нему тяжелую повозку. Если выдержит, значит, мост хороший. Если нет – что ж, возвращаемся к чертежной доске. За последние несколько тысяч лет инженеры развили математику и физику до такой степени, что отпала необходимость строить объект, чтобы понять, как он работает, – для поиска надежных решений строительство уже не требуется. Ничего подобного в программировании нет и, вероятно, не будет, потому что программное обеспечение имеет очень существенные отличия. Классическое исследование разницы между программной и обычной инженерией провел Джек Ривс (Jack Reeves) в статье «What is Software Design?»[1], опубликованной в «C++ Journal» в 1992 году. Статья, написанная почти два десятилетия назад, и сегодня на удивление верна. Ривс в своем сравнении нарисовал мрачную картину, но в 1992 году еще не было одной вещи: серьезного и повсеместного подхода к тестированию программного обеспечения.

Тестировать «реальные» объекты тяжело, потому что, прежде чем тестировать, их нужно построить, а это отбивает охоту возводить рискованную постройку только для того, чтобы посмотреть, что из этого получится. Но в отрасли программного обеспечения «строительство» обходится смехотворно дешево. Мы создали целую экосистему инструментов, с помощью которых его легко осуществить:

---

[1] *http://www.developerdotstar.com/mag/articles/reeves_design.html*

модульное тестирование, объекты-макеты, средства тестирования и многое другое. Другие инженеры были бы безумно рады возможности что-то построить и протестировать в реальных условиях. Мы, разработчики программного обеспечения, должны принять тестирование в качестве главного (но не единственного) механизма верификации для программ. Не нужно ждать появления своего рода «высшей математики» для программирования, потому что в нашем распоряжении уже есть инструменты, гарантирующие хорошую инженерную практику. В этом смысле у нас есть оружие против менеджеров, которые говорят нам, что «нет времени для тестирования». Строитель моста никогда не услышит от своего начальника: «Не трать время на расчет прочности этого здания – мы не укладываемся в график». Если признать, что тестирование – это реальный способ добиться воспроизводимости и качества в отрасли ПО, это позволит нам, разработчикам, отвергать доводы против тестирования как безответственные с профессиональной точки зрения.

Тестирование точно так же требует времени, как его требует и расчет прочности моста. Оба процесса служат гарантии качества конечного продукта. Разработчикам программного обеспечения пора взять на себя ответственность за то, что они производят. Одного тестирования недостаточно, но оно необходимо. Тестирование *и есть* инженерная строгость в разработке программного обеспечения.

Думайте состояниями

Никлас Нильссон

У тех, кто живет в реальном мире, странные представления о том, что такое состояние. Сегодня утром я заехал в местный магазин, чтобы подготовиться к очередному дню переработки кофеина в код. Я предпочитаю это делать за чашечкой латте, а поскольку я не нашел в продаже молока, то обратился к сотруднице магазина.

«У нас абсолютно, ну совершенно абсолютно закончились запасы молока».

Программисту странно слышать такое высказывание. Либо у вас закончилось молоко, либо нет. Не существует каких-то промежуточных степеней отсутствия молока. Возможно, мне пытались сообщить, что молока не будет целую неделю, но результат тот же: придется весь день пить эспрессо.

В большинстве практических ситуаций такое легкомысленное отношение людей к состояниям не вызывает никаких проблем. К несчастью, многие программисты тоже имеют весьма смутное представление о состояниях, а это уже проблема.

Рассмотрим простой интернет-магазин, который принимает к оплате только кредитные карты и не выписывает клиентам счета. При этом класс `Order` (заказ) содержит такой метод:

```
public boolean isComplete() {
    return isPaid() && hasShipped();
}
```

Разумно, так? Так вот, хоть это выражение и выделено в аккуратный метод, а не разбросано по всему коду путем копирования и вставки, его вообще не должно существовать. А то, что оно существует, указывает на проблему. Какую? Нельзя доставить заказ, прежде чем он оплачен. Поэтому `hasShipped` не может вернуть `true`, пока `isPaid` не вернет `true`, а в этом случае часть выражения избыточна. Возможно, вам все же нужен метод `isComplete` для ясности кода, но тогда он должен выглядеть как-то так:

```
public boolean isComplete() {
    return hasShipped();
}
```

В своей работе я постоянно сталкиваюсь и с недостающими, и с избыточными проверками. Я привел крошечный пример, но если добавить сюда отмену заказа и возврат денег, дело усложняется, и потребность в правильной обработке состояний возрастает. В данном случае заказ может находиться только в одном из трех разных состояний:

- **В процессе**: можно добавлять и удалять товары. Нельзя осуществлять доставку.
- **Оплачен**: нельзя добавлять и удалять товары. Можно доставлять.
- **Доставлен**: конец. Больше никаких изменений.

Наличие этих состояний важно, и перед выполнением операций следует проверять, что вы в нужном состоянии, а также проверять, что из текущего состояния вы перейдете в допустимое. Короче, нужно защищать объекты тщательно и в правильных местах.

Но как начать думать состояниями? Выделить выражения в осмысленные методы – очень хорошее начало, но это лишь начало. Главное – понимать конечные автоматы. Знаю, у вас сохранились неприятные воспоминания о них из курса информатики, но оставьте это в прошлом. Конечные автоматы – это не особенно сложно. Рисуйте их, тогда их легче понять и обсуждать. Запускайте свой код в тестовом режиме, чтобы разделить допустимые и недопустимые состояния, а также переходы между ними, и поддерживать их корректность. Изучите шаблон State. Освоившись с ним, почитайте о контрактном программировании (design by contract). Оно помогает обеспечить допустимость состояния путем проверки данных и самого объекта на входе и выходе из каждого открытого метода.

Если состояние некорректно, значит, в коде ошибка, и вы рискуете потерять данные, если не прервете выполнение. Если вам кажется, что проверки состояний замусоривают код, научитесь скрывать их с помощью специальных инструментов, генерации кода, вплетений (weaving) или аспектов. Независимо от того, какой метод вы выберете, подход на основе состояний сделает ваш код более простым и надежным.

Одна голова хорошо, но две – часто лучше

Эдриан Уайбл

Программирование требует вдумчивости, а вдумчивость может существовать только в уединении. Такой стереотип присущ многим программистам.

Но стиль «одинокого волка» в программировании уже давно отступает перед командным подходом, который, смею утверждать, улучшает качество работы, повышает производительность и позволяет разработчикам получать большее удовольствие от работы. Новый подход заставляет разработчиков работать друг с другом теснее, а также работать с теми, кто не участвует в разработке, – системными и бизнес-аналитиками, специалистами в области контроля качества и пользователями.

Что это означает для разработчиков? Уже недостаточно быть экспертом в области технологии программирования. Вы должны научиться эффективно работать с другими людьми.

Сотрудничество на работе – это не игра в вопросы и ответы и не долгие совещания. Сотрудничество – это засучить рукава вдвоем с коллегой и приступить к трудной задаче.

Я большой поклонник парного программирования. Можно назвать это «экстремальным сотрудничеством». Когда я работаю в паре, мое мастерство программиста растет. Если я слабее своего партнера в предметной области или в какой-либо технологии, то просто учусь на его опыте. Когда я сильнее в каком-то аспекте, то начинаю лучше понимать, что я знаю и чего не знаю, поскольку мне приходится давать объяснения. В любом случае мы оба вносим свой вклад и учимся друг у друга.

Работая в паре, мы привносим свой коллективный опыт – как опыт в предметной области, так и технический – для решения стоящей перед нами задачи. Вместе мы способны предложить свое уникальное видение и опыт, что позволяет нам решать задачи эффективно и рационально. Даже при наличии значительной разницы в уровне знаний предметной области или в технологических вопросах, более опытный партнер все равно чему-то учится у второго – ну, скажем,

узнаёт о новых «горячих клавишах» либо встречается с новым инструментом или библиотекой. Для менее опытного партнера такая работа – замечательный способ «набрать скорость».

Парное программирование популярно у сторонников гибкой разработки, хотя и не только у них. Иногда противники парной работы интересуются: «А почему я должен платить двум программистам за выполнение работы одного?» Конечно, не должны. Но дело в том, что работа в паре повышает качество, улучшает понимание предметной области, технологий и приемов работы (скажем, неочевидных приемов работы с интегрированной средой разработки, IDE), а также уменьшает отрицательное влияние лотерейного риска (когда один из ваших специалистов-разработчиков выигрывает в лотерею и увольняется на следующий же день).

Какова долгосрочная выгода от того, что вы узнаете о новой «горячей клавише»? Какой мерой мы измерим улучшение качества продукта, достигнутое работой в паре? Какой мерой измерить пользу от того, что ваш партнер не дал вам зайти в тупик в решении сложной проблемы? Одно исследование свидетельствует о приросте эффективности и скорости на 40%.[1] А как оценить уменьшение «лотерейного риска»? Большинство плюсов работы в паре трудно измерить.

Кто должен работать в паре и с кем? Если вы новичок в команде, то важно, чтобы вашим напарником оказался опытный специалист. Столь же важно, чтобы он обладал хорошими навыками общения и наставничества. Если у вас недостаточно знаний в предметной области, работайте в паре с тем, кто ее хорошо знает.

Если нет уверенности, экспериментируйте: сотрудничайте с коллегами. Создавайте пары для решения интересных сложных проблем. Посмотрите, что у вас получится. Попробуйте несколько раз.

---

[1] J. T. Nosek «The Case for Collaborative Programming», *Communications of the ACM*, March 1998.

Две ошибки могут гасить одна другую (и тогда их трудно исправлять)

Аллан Келли

Код никогда не лжет, но может быть внутренне противоречивым. Иногда противоречия вызывают недоумение: как это вообще может работать?

В своем интервью[1] Аллан Кламп (Allan Klumpp), ведущий разработчик программного обеспечения для лунного модуля Apollo, раскрыл тот факт, что ПО управления двигателями содержало дефект, из-за которого спускаемый модуль должен был вести себя неустойчиво. Однако в программе была еще одна ошибка, компенсировавшая первую, и при посадке Apollo 11 и 12 на Луну это ПО успешно использовалось, прежде чем ошибки были обнаружены и исправлены.

Рассмотрим функцию, которая возвращает код завершения. Допустим, она возвращает false, когда должна была бы вернуть true. Теперь представим, что в вызывающей функции не реализована проверка возвращаемого значения. Все работает прекрасно, пока однажды кто-то не обнаружит отсутствие проверки и не вставит ее.

Или рассмотрим приложение, которое хранит состояние в виде документа XML. Допустим, что один из узлов некорректно записывается как `TimeToLive` (время жизни) вместо `TimeToDie` (время смерти), как следовало бы, если верить документации. Все будет хорошо, пока код записи и код чтения содержат одну и ту же ошибку. Но исправьте ее в одном месте или добавьте новое приложение, читающее тот же документ, и симметрия рухнет, как и весь код.

Когда в коде два дефекта, а отказ на вид только один, может стать бесполезным сам методический подход к исправлению ошибок. Получив сообщение об ошибке, разработчик обнаруживает дефект, исправляет его и проводит тестирование. Возникает тот же самый отказ, но уже в силу действия второго дефекта. Тогда отменяется первое исправление, код снова исследуется, и обнаруживается второй дефект, который и исправляется. Но первый дефект снова на месте, снова возникает тот же отказ, и тогда откатывается второе исправление. Процесс повторяется, но

---

[1] *http://www.netjeff.com/humor/item.cgi?file=ApolloComputer*

теперь разработчик отказался от двух исправлений и пытается найти третье, чего ему никогда не удастся.

Взаимодействие двух дефектов в коде, проявляющих себя одинаковым сбоем, не только затрудняет решение проблемы, но и заводит разработчиков в тупик, где они обнаруживают, что их ранние решения проблемы были правильными.

Такое случается не только с кодом: проблемы встречаются в документах, содержащих технические требования. И они способны распространяться, подобно вирусу, из одного места в другое. Ошибка в коде компенсирует ошибку в письменной спецификации.

Вирус может поразить и человека: пользователи обнаруживают, что когда программа говорит «левая», она имеет в виду «правая», и подстраивают под нее свои действия. Они даже сообщают о проблеме новым пользователям: «Запомни, когда приложение говорит, что нужно щелкнуть левой кнопкой, это значит, что нужно щелкнуть правой». Стоит исправить ошибку, и пользователям придется переучиваться.

Одиночные ошибки, как правило, легко обнаруживаются и исправляются. Проблемы возникают в случае множества ошибок, требующих множества исправлений. Частично это вызвано тем, что простые проблемы легко исправлять, и потому их обычно не откладывают, а более сложные проблемы копятся до лучших времен.

Нет простого ответа на вопрос, как решать проблемы, возникающие в связи с родственными дефектами. Нужно помнить об их существовании, иметь ясную голову и готовность при необходимости рассмотреть все возможности.

Написание кода в духе Убунту для друзей

Аслам Хан

Очень часто мы программируем в изоляции, и наши программы отражают как нашу личную интерпретацию проблемы, так и очень личное ее решение. Мы можем работать в команде, но и тогда мы изолированы как команда. Мы легко забываем, что код, созданный в такой изоляции, будут выполнять, использовать и расширять другие люди. Легко упустить из вида социальную сторону программирования. Создание программ – одновременно и техническое, и социальное занятие. Нам следует чаще оглядываться вокруг, чтобы понять, что мы работаем не изолированно и что мы несем общую ответственность за возможный успех не только группы разработчиков, но и каждого человека вокруг нас.

Можно написать код высокого качества в отрыве от реальности, полностью уйдя в себя. С одной стороны, это эгоцентричный подход (*эго* не в смысле высокомерия, а в смысле личности). Это философия Дзен, и в момент создания программы в этом случае действительно существуете только вы. Я всегда стараюсь жить в текущем моменте, поскольку это помогает приблизиться к лучшему качеству, но при этом я живу в *своем* текущем моменте. А как же быть с текущим моментом моей команды? Мой момент и момент моей команды – совпадают ли они?

На языке зулу философия Убунту определяется как «Умунту нгумунту нгабанту», что в первом приближении можно перевести так: «Личность – это личность через (другие) личности». Я становлюсь лучше, поскольку ты делаешь меня лучше своими добрыми поступками. Но, с другой стороны, вы хуже делаете свое дело, если я плохо делаю свое. Для разработчиков это сводится к тому, что «разработчик – это разработчик через (других) разработчиков». Если опуститься до «железа», то «код – это код через (другой) код».

Качество кода, который пишу я, влияет на качество кода, который пишете вы. А что если мой код низкого качества? Даже если вы напишете очень чистый код, там, где вы будете пользоваться моим кодом, качество вашего кода упадет примерно до уровня моего кода. Можно применять множество шаблонов и приемов, чтобы ограничить ущерб, но полностью от него уже не избавиться. Я заставил

вас делать больше, чем требовалось, просто потому, что не думал о вас, когда жил в своем моменте.

Я могу считать свой код чистым, но все же можно сделать его еще лучше, придерживаясь духа Убунту. Как выглядит код в духе Убунту? Он выглядит, как хороший, ясный код. И речь идет даже не о коде как артефакте. Все дело в акте создания этого артефакта. Программирование для своих друзей в духе Убунту поможет вашей команде жить, согласуясь с собственными ценностями, и укреплять свои принципы. Следующий человек, который каким-либо образом прикоснется к вашему коду, станет лучше как личность и как разработчик.

Дзен – это дело личное. Убунту – это Дзен для группы людей. Крайне редко мы пишем программы исключительно для самих себя.

Утилиты UNIX – ваши друзья

Диомидис Спинеллис

Если бы, отправляясь в изгнание на необитаемый остров, я должен был выбирать между IDE и набором инструментов UNIX, я бы, не колеблясь, выбрал утилиты UNIX. Вот причины, по которым следует овладеть искусством работы с утилитами UNIX.

Во-первых, IDE ориентированы на конкретные языки, а утилиты UNIX могут работать с любым материалом в текстовом виде. В современных условиях разработки, когда новые языки и нотации появляются каждый год, затраты на изучение работы с утилитами UNIX окупятся многократно.

Кроме того, IDE предлагают только те команды, которые решили реализовать их создатели, тогда как средствами UNIX можно выполнить любую мыслимую задачу. Их можно представить себе как классические (до появления Bionicle) блоки Lego: вы создаете собственные команды, просто комбинируя маленькие, но универсальные утилиты UNIX. К примеру, следующая последовательность представляет собой текстовую реализацию анализа сигнатуры по Каннингему – последовательность использования точки с запятой, фигурных скобок и кавычек в любом файле может многое сказать о его содержимом:

```
for i in *.java; do
    echo -n "$i: "
    sed 's/[^"{};]//g' $i | tr -d '\n'
    echo
done
```

Далее, каждая изученная вами операция IDE специфична для данной задачи, например добавление нового шага в конфигурацию отладочной сборки проекта. Напротив, более глубокое изучение утилит UNIX повышает вашу эффективность при решении любых задач. Например, я применил утилиту *sed*, использованную в приведенной выше последовательности команд, чтобы преобразовать процедуру сборки проекта для кросскомпиляции на многопроцессорных архитектурах.

Утилиты UNIX были разработаны в ту эпоху, когда многопользовательские компьютеры располагали ОЗУ размером в 128 Кбайт. При их создании была проявлена такая изобретательность, что сейчас они могут очень эффективно обрабатывать огромные наборы данных. Большинство утилит представляет собой фильтры, обрабатывающие по одной строке за раз, а потому нет верхней границы объема данных, которые они могут обработать. Вы хотите узнать, сколько редакций хранится в дампе английской Википедии, имеющем размер полтерабайта? Простой вызов команды

```
grep '<revision>' | wc -l
```

без труда даст вам ответ. Если какая-то последовательность команд покажется вам полезной в общем случае, можно создать на ее основе сценарий интерпретатора команд, применив в нем некоторые исключительно мощные программные конструкции, такие как направление данных в циклы и условные операторы. Еще более впечатляет, что конвейерное выполнение команд UNIX (как в предыдущем примере) естественным образом распределяет нагрузку по нескольким конвейерам обработки современных многоядерных процессоров.

Принцип «прекрасное в малом» и открытость реализаций утилит UNIX делают их доступными повсеместно, даже на платформах с ограниченными ресурсами, таких как медиа-проигрыватель, подключаемый к телевизору, или маршрутизатор DSL. Такие устройства едва ли располагают мощным графическим интерфейсом пользователя, но часто содержат приложение BusyBox, предлагающее наиболее часто используемые утилиты. Если вы занимаетесь разработкой под Windows, среда Cygwin предложит вам все мыслимые утилиты UNIX как в виде исполняемых файлов, так и в виде исходного кода.

Наконец, если вас не удовлетворяют существующие утилиты UNIX, вы можете легко расширить их набор. Напишите программу, которая решает нужную вам задачу, на любом понравившемся вам языке, соблюдая следующие простые правила: программа должна выполнять одну единственную задачу, читать данные со стандартного ввода в виде строк текста и печатать результат на стандартный вывод без всяких заголовков и прочих украшательств. Параметры, влияющие на функционирование утилиты, задаются в командной строке. Следуйте этим правилам, и тогда «Земля – твое, мой мальчик, достоянье».

Правильно выбирайте алгоритмы и структуры данных

Ян Кристиаан ван Винкель

Крупный банк с большим количеством филиалов пожаловался, что купленные для кассиров новые компьютеры работают слишком медленно. Это было еще до повсеместного использования интернет-банкинга, и банкоматы тоже не были так широко распространены, как сейчас. Люди ходили в банк гораздо чаще, а из-за медленно работающих компьютеров выстраивались очереди. Через какое-то время банк пригрозил разорвать контракт с поставщиком.

Поставщик направил в банк специалиста по анализу и настройке производительности, чтобы выяснить, в чем причина задержек. Тот быстро нашел на терминале программу, съедавшую почти всю производительность процессора. Посредством инструмента профилирования он нашел в этой программе функцию, виновную в происходящем. Ее исходный код выглядел так:

```
for (i=0; i<strlen(s); ++i) {
    if (... s[i] ...) ...
}
```

При этом строка s обычно содержала несколько тысяч символов. Код (который был написан в самом банке) быстро изменили, и с тех пор банковские кассиры зажили счастливо...

Почему программисту не пришло в голову ничего умнее, чем написать код с квадратичной сложностью без всякой необходимости?

При каждом вызове `strlen` программа просматривает каждый из нескольких тысяч символов строки, пока не будет обнаружен завершающий ее нулевой символ. Строка при этом не меняется. Заранее вычислив ее длину, программист избавился бы от тысяч вызовов `strlen` (и миллионов итераций цикла):

```
n=strlen(s);
for (i=0; i<n; ++i) {
    if (... s[i] ...) ...
}
```

Всем известен принцип «сделай так, чтобы код работал, потом сделай так, чтобы он работал быстро», который направлен против оптимизаций на микроуровне. Но по приведенному примеру можно решить, что программист исполнил макиавеллевское адажио «сначала сделай так, чтобы код работал медленно».

Подобная бездумность встречается, и нередко. И дело не только в том, что не нужно «заново изобретать колесо». Иногда молодые программисты бросаются, не особо раздумывая, писать код и вдруг «изобретают» пузырьковую сортировку. При этом они, случается, еще и хвастают этим.

Обратной стороной выбора правильного алгоритма является выбор структуры данных. Этот выбор может серьезно повлиять на скорость работы: если для хранения коллекции в миллион объектов, по которой нужно выполнять поиск, вы используете связный список – а не структуру с хешированием данных либо двоичное дерево, – пользователю найдется что сказать насчет вашего умения программировать.

Программисты должны не изобретать колесо, а по возможности использовать имеющиеся библиотеки. Но, чтобы избежать таких проблем, как у вышеупомянутого банка, они должны также обладать знаниями об алгоритмах и возможностях их масштабирования. Что делает современные текстовые редакторы такими же медлительными, как старые программы 1980-х типа WordStar – только ли навороченный интерфейс? Многие говорят, что в программировании важнейшее значение имеет повторное использование кода. Но прежде всего программист должен знать, когда, что и как использовать повторно. Для этого ему нужно знать предметную область, а также алгоритмы и структуры данных.

Хороший программист должен также знать, когда стоит использовать плохой алгоритм. Например, если предметная область такова, что в ней не может быть больше пяти элементов (скажем, количество костей в покере на костях), вы понимаете, что сортировать придется не более пяти элементов. В таком случае пузырьковая сортировка действительно может оказаться наиболее разумным выбором. На каждой улице бывает праздник.

Поэтому прочтите несколько хороших книг – и хорошенько в них разберитесь. А если вы глубоко изучите «Искусство программирования» Дональда Кнута, вам может даже повезти: найдите у автора ошибку, и вы получите от него чек на один шестнадцатеричный доллар ($2.56).

Многословный журнал лишит вас сна

Йоханнес Бродуолл

Когда я встречаю систему, которая достаточно давно разрабатывается или функционирует, первым признаком реальных неприятностей всегда оказывается «грязный» журнал. Вы знаете, о чем я говорю: это когда переход по ссылке при нормальной работе с веб-страницей приводит к целому потоку сообщений, записываемых в единственный журнал системы. Слишком большое количество записей в журнале может быть столь же бесполезным, как и их полное отсутствие.

Если ваши системы похожи на мои, то когда заканчивается ваша работа, начинается работа других людей. После завершения разработки система будет долго и успешно обслуживать клиентов (если вам повезет). Как вы узнаете о проблемах, если система в эксплуатации, и что вы будете с ними делать?

Возможно, кто-то осуществляет мониторинг системы вместо вас, а возможно, это делаете вы. В любом случае мониторинг, вероятно, включает в себя просмотр журналов. Если что-то случилось, и вас будят среди ночи, желательно, чтобы причина была веской. Если моя система при смерти, я должен знать об этом. Но если она просто икнула, я бы предпочел, чтобы мой сладкий сон не нарушали.

Во многих системах первым признаком неприятностей служит запись сообщения в какой-нибудь журнал. Обычно это журнал регистрации ошибок. Так что окажите себе услугу: организуйте процесс таким образом, чтобы с первого же дня разработки, как только что-то появляется в журнале ошибок, вас будили среди ночи телефонным звонком. Если во время системного тестирования вы сможете смоделировать нагрузку на свою систему, чистый журнал ошибок, скорее всего, засвидетельствует надежность вашей системы, а «грязный» послужит первым сигналом тревоги.

Распределенные системы создают дополнительный уровень сложности. Вы должны решить, как действовать в случае отказа внешних зависимостей. Если система сильно распределена, это может происходить часто. Учтите это обстоятельство при выборе политики ведения журнала.

В целом, лучшим свидетельством того, что все в порядке, служит регулярное появление сообщений низкого приоритета. Меня устраивает, когда на каждое существенное событие в приложении появляется примерно одно сообщение уровня INFO.

Слишком подробный журнал говорит о том, что систему трудно контролировать во время эксплуатации. Если вы исходите из того, что журнал ошибок должен оставаться пустым, вам будет гораздо легче разобраться в проблеме, когда в нем что-то все-таки появится.

WET размазывает узкие места производительности

Кирк Пеппердин

Значимость принципа DRY (Don't Repeat Yourself – не повторяйся) состоит в том, что он формализует следующую идею: каждый элемент знаний в системе должен иметь единственное представление. Иными словами, знание должно ограничиваться единственной реализацией. Полную противоположность DRY представляет WET (Write Every Time – пиши каждый раз). Наш код можно назвать «сырым» (WET), когда знание представлено в коде одновременно несколькими способами. Скрытое влияние DRY и WET на производительность становится понятным после рассмотрения их многочисленных эффектов на конкретном примере.

Рассмотрим некоторую функциональность нашей системы (назовем ее X), являющуюся узким местом для процессора. Допустим, что функция X потребляет 30% мощности процессора. Теперь предположим, что у функции X есть 10 различных реализаций. В среднем каждая реализация потребляет 3% процессорного времени. Поскольку такой уровень использования процессора не вызывает беспокойства, при беглом анализе можно не заметить, что эта функция создает узкое место. Но допустим, что мы каким-то образом выяснили, что функция X – узкое место. Тогда ставится задача найти и исправить каждую реализацию. Для WET у нас есть 10 разных реализаций, которые нужно найти и исправить. В случае DRY мы сразу увидим загрузку процессора в 30%, а размер кода для исправления будет в 10 раз меньше. К тому же не потребуется выискивать все многочисленные реализации.

Есть один сценарий, в котором мы часто нарушаем DRY, а именно при работе с коллекциями. Стандартный прием реализации запроса заключается в проходе по коллекции и применении запроса к каждому ее элементу:

```
public class UsageExample {
    private ArrayList<Customer> allCustomers = new ArrayList<Customer>();
    // ...
    public ArrayList<Customer> findCustomersThatSpendAtLeast(Money amount) {
        ArrayList<Customer> customersOfInterest = new ArrayList<Customer>();
        for (Customer customer: allCustomers) {
```

```
            if (customer.spendsAtLeast(amount))
                customersOfInterest.add(customer);
        }
        return customersOfInterest;
    }
}
```

Сделав эту коллекцию напрямую доступной клиентам, мы нарушили принцип инкапсуляции. Это не только снижает потенциал рефакторинга, но и заставляет пользователей кода нарушать DRY, поскольку каждому из них придется заново реализовывать потенциально идентичный запрос. Такой ситуации легко избежать, если убрать открытые коллекции из API. В данном примере можно ввести новый предметно-ориентированный тип коллекции с именем `CustomerList`. Этот класс семантически лучше согласован с предметной областью. Естественным образом он станет местом, в котором содержатся все наши запросы.

Наличие этого нового типа-коллекции позволит также легко выяснить, являются ли эти запросы узким местом в смысле производительности. Включив запросы в класс, мы устраняем необходимость открывать клиентам варианты представления, такие как `ArrayList`. Это дает нам свободу для последующего изменения реализаций без нарушения контрактов с клиентами:

```
public class CustomerList {
    private ArrayList<Customer> customers = new ArrayList<Customer>();
    private SortedList<Customer> customersSortedBySpendingLevel =
            new SortedList<Customer>();
    // ...
    public CustomerList findCustomersThatSpendAtLeast(Money amount) {
        return new CustomerList(
                    customersSortedBySpendingLevel.elementsLargerThan(amount));
    }
}
public class UsageExample {
    public static void main(String[] args) {
        CustomerList customers = new CustomerList();
        // ...
        CustomerList customersOfInterest =
                     customers.findCustomersThatSpendAtLeast(someMinimalAmount);
        // ...
    }
}
```

В этом примере следование принципу DRY позволило нам ввести измененную систему индексирования, в которой используется `SortedList`, а ключом служит объем трат наших покупателей. И, если абстрагироваться от данного примера, намного важнее, что следование DRY помогло найти и исправить узкое место. Пиши мы код по принципу WET, сделать это было бы труднее.

Когда программисты и тестировщики сотрудничают

Джанет Грегори

Когда тестировщики и программисты начинают сотрудничать, происходят чудеса. Меньше времени уходит на игру в пинг-понг дефектами в системе отслеживания дефектов. Меньше времени тратится на обсуждение того, является ли поведение ошибкой или новой функцией, а больше – на разработку качественного программного обеспечения, отвечающего ожиданиям заказчиков. Существует много возможностей наладить сотрудничество еще до того, как начнется написание кода.

Тестировщики могут помочь заказчикам написать приемочные тесты на языке их предметной области с помощью таких инструментов, как Fit (Framework for Integrated Test). Если передать эти тесты программистам перед тем, как они начнут писать код, те смогут применить практику *разработки на основе приемочного тестирования* (*acceptance test-driven development, ATDD*). Программисты пишут фреймворки для прогона тестов, а потом код с проверкой на прохождение этих тестов. Далее эти тесты входят в набор регрессионных тестов. При такой организации сотрудничества функциональное тестирование проходит быстро, и остается больше времени на экспериментальное тестирование граничных условий или сценариев более изощренных рабочих процессов.

Можно пойти еще дальше. Будучи тестировщиком, я могу изложить свои соображения по тестированию еще до того, как программисты начнут писать код новой функции. Если я интересуюсь соображениями программистов, они почти всегда предоставляют мне сведения, позволяющие мне добиться лучшего покрытия кода или сократить затраты времени на ненужные тесты. Часто нам удавалось предотвратить появление дефектов за счет того, что тесты проясняют многие первоначальные идеи. Например, в одном из проектов, где я участвовала, я передала программистам Fit-тесты, которые показывали, какие результаты ожидаются при поиске по маске. Программист до этого твердо намеревался реализовать только поиск по конкретным словам. Нам удалось пообщаться с заказчиком, и мы смогли договориться о правильной интерпретации поиска до того,

как начать писать код. В результате мы предотвратили возникновение дефекта и сэкономили всем уйму времени.

Программисты могут сотрудничать с тестировщиками и в деле автоматизации. Они знакомы с хорошей практикой написания кода и способны помочь тестировщикам создать надежный комплекс автоматизированных тестов, что послужит интересам всей команды. Мне часто приходилось видеть, как проекты по автоматизации тестирования завершались неудачей из-за неумелого проектирования тестов. Либо тесты пытаются проверить слишком многое, либо тестировщики недостаточно разбираются в технологии, чтобы сделать тесты независимыми от кода. Тестировщики часто создают узкие места, поэтому полезно, когда программисты работают с ними вместе над такими задачами, как автоматизация. Определив вместе с тестировщиками, что можно протестировать на раннем этапе – возможно, с помощью какого-нибудь простого инструмента, – программисты получают дополнительный канал обратной связи, что в конечном итоге позволяет им создать более качественный код.

Если тестировщики перестанут думать лишь о том, как бы им сломать программу или найти ошибки в коде разработчиков, то и программисты перестанут считать, что тестировщики стараются только «достать» их, и будут более склонны к сотрудничеству. Когда программисты понимают, что они отвечают за качество своего кода, легкость его тестирования становится естественным дополнительным качеством, и команда может совместно автоматизировать больше регрессионных тестов. Таково чудо успешной групповой работы.

Пишите код так, как будто вам предстоит сопровождать его всю оставшуюся жизнь

Юрий Зубарев

Можно задать вопрос о том, что должен знать и уметь любой программист, 97 разным людям и получить 97 разных ответов. Это может одновременно ошеломить и напугать. Все советы хороши, все принципы здравы, все истории убедительны, но с чего начать? И, что еще более важно, как, однажды начав применять лучшие практики, оставаться на должном уровне и сделать их составной частью своей практики программирования?

Я думаю, что ответ – в вашем настрое или просто в вашем подходе. Если вам безразличны ваши коллеги-разработчики, тестировщики, менеджеры, сотрудники отделов продаж и маркетинга, а также конечные пользователи, у вас не возникнет побуждения вести, к примеру, разработку на основе тестов или писать понятные комментарии в коде. Думаю, что есть простой способ изменить свое отношение и развить в себе стремление выпускать продукты самого лучшего качества:

> *Пишите код так, как будто вам предстоит сопровождать его всю оставшуюся жизнь.*

Вот и все. Если вы примете эту мысль, начнут происходить удивительные вещи. Согласившись с тем, что любой из ваших прежних или нынешних работодателей имеет право позвонить вам среди ночи и попросить объяснить, на чем основаны решения, сделанные вами при написании некоего метода `fooBar`, вы начнете свой путь к мастерству в программировании. Вам самим захочется придумывать лучшие имена для переменных и методов. Вы постараетесь не допускать, чтобы блоки кода состояли из сотен строк. Вы будете искать, изучать и применять шаблоны проектирования. Вы станете писать комментарии, тестировать код и непрерывно осуществлять его рефакторинг. Поддерживать весь написанный вами код в течение всей оставшейся жизни? Эта задача будет становиться все более грандиозной. У вас просто не останется иного выбора, кроме как работать лучше, изобретательнее и эффективнее.

Если вдуматься, то код, написанный вами много лет назад, продолжает влиять на вашу карьеру, нравится вам это или нет. Каждый метод, класс или модуль,

который вы спроектировали или написали, хранит отпечаток ваших знаний, отношения к работе, упорства, профессионализма, степени вовлеченности и уровня удовольствия. Люди формируют о вас свое мнение на основе кода, который они видят. Если это мнение постоянно оказывается отрицательным, ваша карьера не будет развиваться так быстро, как вам хотелось бы. Каждая строка вашего кода должна быть на благо вашей карьере, ваших клиентов и ваших пользователей – пишите код так, как если бы вам пришлось сопровождать его всю оставшуюся жизнь.

Пишите маленькие функции на основе примеров

Кейт Брэйтуэйт

Мы хотим писать правильный код и иметь на руках свидетельство его правильности. В обоих случаях будет полезным принять во внимание «размер» функции. Не в смысле объема кода, который реализует функцию – хотя и это интересно, – а как размер математической функции, которую демонстрирует наш код.

Например, в игре *го* есть положение, называемое *атари*, в котором фишки игрока могут быть захвачены противником: фишка с двумя и более свободными соседними клетками (называемыми *степенями свободы*), не находится в положении атари. Подсчитать количество степеней свободы у фишки бывает нелегко, но, когда оно известно, определить атари легко. Можно для начала написать такую функцию:

```
boolean atari(int libertyCount)
    libertyCount < 2
```

Здесь спрятано больше, чем кажется на первый взгляд. Математическую функцию можно рассматривать как множество – некоторое подмножество декартова произведения области определения (здесь это `int`) и области принимаемых значений (здесь – `boolean`). Будь эти множества одинакового размера, как в Java, в множестве `int×boolean` было бы `2L*(Integer.MAX_VALUE+(-1L*Integer.MIN_VALUE)+1L)`, или 8589934592 элементов. Половина из них принадлежит подмножеству, являющемуся нашей функцией, поэтому для полного доказательства корректности нашей функции нужно проверить около $4{,}3 \times 109$ случаев.

На этом и основывается утверждение, что тестами нельзя доказать отсутствие дефектов. Тесты способны продемонстрировать, что функциональность реализована. Но проблема размера сохраняется.

Выход подсказывает предметная область. Природа *го* такова, что число степеней свободы фишки является не любым целым числом, а одним из чисел {1,2,3,4}. Поэтому можно написать другой вариант кода:

```
LibertyCount = {1,2,3,4}
boolean atari(LibertyCount libertyCount)
    libertyCount == 1
```

Это уже гораздо легче поддается обработке: вычисляемая функция – это множество, в котором максимум восемь элементов. Фактически проверка четырех случаев может дать полную уверенность, что функция корректна. Это одна из причин, почему при написании программ лучше использовать типы, тесно связанные с предметной областью, а не встроенные типы языка. Использование предметно-ориентированных типов часто позволяет значительно уменьшить размер функций. Один из способов выяснить, какими должны быть эти типы, – это найти примеры и определить термины предметной области до того, как писать функцию.

Тесты пишутся для людей

Джерард Мезарос

Вы покрываете автоматизированными тестами весь готовый код или его фрагменты. Поздравляем! Пишете сначала тесты, а потом код? Еще лучше! Уже благодаря этому вас можно причислить к программистам, практикующим передовые подходы в разработке программного обеспечения. Но хороши ли ваши тесты? Как это определить? Один из способов – спросить себя: «А для кого я пишу эти тесты?». Если ответом будет «я пишу их для себя, чтобы сократить затраты на исправление ошибок» или «для компилятора, чтобы их можно было выполнить», то вполне возможно, что вы пишете не самые лучшие тесты. Так для кого же нужно писать тесты? Для того, кто будет пытаться понять ваш код.

Хорошие тесты играют роль документации для тестируемого ими кода. Они описывают, как работает код. Для каждого сценария использования тесты делают следующее:

- Описывают контекст, начальную точку или предусловия, которые должны удовлетворяться.
- Иллюстрируют, как вызывается приложение.
- Описывают ожидаемые результаты или постусловия, которые необходимо проверить.

В различных сценариях использования варианты этих действий будут слегка различаться. Тот, кто попытается понять ваш код, должен иметь возможность посмотреть на несколько тестов и, сравнив эти три части рассматриваемых тестов, суметь определить, что заставляет программу в разных случаях работать по-разному. Каждый тест должен ясно иллюстрировать причинно-следственные связи между этими тремя частями.

Отсюда следует, что невидимая часть теста столь же важна, как и видимая. Обилие кода в тесте отвлекает внимание читателя на несущественные мелочи. По возможности скрывайте такие мелочи в понятных вызовах методов, и в этом вам здорово поможет прием рефакторинга Extract Method (Извлечь метод). И постарайтесь дать каждому тесту выразительное название, чтобы оно описывало кон-

кретный сценарий использования, а читающему тесты человеку не пришлось анализировать каждый тест в попытке понять, в чем заключаются различия между сценариями. Во всех случаях имя класса теста и имя метода класса должны содержать хотя бы начальную точку и способ вызова приложения. Это позволит простым прочтением имен методов проверить покрытие кода тестами. В имена методов тестов полезно бывает включить ожидаемые результаты, если только это не сделает имена слишком длинными для чтения или восприятия.

Тестировать сами тесты – тоже хорошая идея. Можно проверить, смогут ли они обнаружить ошибки, которые должны находить, если ввести эти ошибки в производственный (production) код (разумеется, в локальную копию кода, которую вы потом выкинете). Проверьте, что тесты выдают полезные и осмысленные отчеты. Нужно также проверить, что ваши тесты говорят языком, понятным тому, кто разбирается с вашим кодом. Это можно сделать, только если дать прочесть тесты программисту, который не знаком с вашим кодом, и выслушать его впечатления. Если ему что-то непонятно, не связывайте это с нехваткой умственных способностей. Более вероятно, что это вы не сумели написать ясный код. (Попробуйте поменяться ролями и почитать его тесты!)

Нужно заботиться о коде

Пит Гудлиф

Не нужно быть Шерлоком Холмсом, чтобы понять, что хорошие программисты пишут хороший код. Ну, а плохие – нет. Они создают уродливые вещи, которые всем остальным приходится доводить до ума. Но вы-то хотите писать хороший код, правда? Тогда вам нужно стремиться стать хорошим программистом.

Хороший код не возникает сам по себе из ничего. Его появление не вызвано благоприятным расположением планет. Чтобы сделать код хорошим, нужно над ним работать, и немало. Вы создадите хороший код только тогда, когда действительно к этому стремитесь.

Хорошее программирование не является результатом одной лишь технической компетентности. Я встречал очень умных программистов, которые способны создавать сильные и впечатляющие алгоритмы, в совершенстве знают стандарты своего языка и пишут при этом совершенно отвратительный код. Его тяжело читать, с ним тяжело работать и его тяжело модифицировать. Я встречал и программистов с более скромными способностями, которые тяготеют к очень простому коду, но пишут элегантные и выразительные программы, работать с которыми одно удовольствие.

Опыт работы в отрасли разработки программного обеспечения привел меня к заключению, что на практике разница между просто компетентными программистами и выдающимися программистами заключается в одном: это *отношение к работе*. Хорошее программирование требует профессионального подхода и стремления написать как можно лучший код с учетом ограничений, накладываемых окружающей действительностью и требованиями отрасли.

Код, ведущий в ад, вымощен благими намерениями. Чтобы стать отличным программистом, нужно отказаться от благих намерений и действительно проявить *заботу* о коде – воспитывать в себе позитивный взгляд на написание кода и вырабатывать здоровое отношение к работе. Выдающийся код есть плод тщательных усилий искусных мастеров, а не поделка, небрежно сляпанная программистом, или таинственное создание самопровозглашенных гуру от кодирования.

Вы хотите писать хороший код. Вы хотите быть хорошим программистом. Тогда вы должны стремиться к следующему:

- Какими бы ни были условия работы, вы отказываетесь наскоро писать код, который *предположительно* решает задачу. Вы стремитесь писать красивый код, корректность которого очевидна (и доказывается хорошо написанными тестами).
- Вы пишете код, доступный для *понимания* (другие программисты могут быстро разобраться в нем и продолжить работу), легкий в *сопровождении* (вы или другие программисты легко сможете модифицировать его в будущем) и *верный* (вы сделали все возможное, чтобы показать, что вы действительно решили задачу, а не просто создали видимость работающей программы).
- Вы хорошо ладите с другими программистами. Программист не должен быть отшельником. Редкий программист работает в одиночку: большинство трудится в составе команды программистов, будь то в рамках компании или проекта с открытым исходным кодом. Вы учитываете особенности других программистов и пишете код так, чтобы они могли его прочесть. Вы стремитесь помочь команде создать как можно лучший продукт, а не показать, какой вы умный.
- Когда к вам в руки попадает код, вы стараетесь, чтобы после вас он стал немного лучше (лучше организован, лучше протестирован, более понятен...).
- Вы любите код и программирование, поэтому вы постоянно изучаете новые языки, идиомы и новые приемы. Но применяете их только тогда, когда это уместно.

К счастью, вы читаете эти советы потому, что действительно любите код. Вам это интересно. Это ваше увлечение. Получайте удовольствие от программирования. Радуйтесь, написав код для решения сложной задачи. Пишите программы, которыми можно гордиться.

Ваши заказчики имеют в виду не то, что говорят

Нэйт Джексон

Я еще не встречал заказчика, который не был бы рад возможности рассказать мне, что ему нужно – обычно до мельчайших подробностей. Проблема в том, что заказчики не всегда рассказывают всю правду. В целом заказчики не лгут, но они говорят на своем языке заказчиков, а не на языке разработчиков. У них свой словарь и свой контекст. Они опускают важные детали. Они говорят так, будто вы тоже проработали в их компании лет двадцать. А осложняется это все тем, что на самом деле заказчики часто сами не знают, что им нужно! У одних есть понимание общей картины, но они редко в состоянии толково выразить свое представление. У других общее представление может быть менее ярким, но они знают, чего им не нужно. Так как же можно разработать программный проект для того, кто не способен правдиво рассказать, что именно ему нужно? Выход достаточно прост. Нужно больше взаимодействовать с заказчиком.

Начинайте задавать своим заказчикам вопросы как можно раньше, и задавайте вопросы чаще. Не стоит повторять их же словами то, что они сказали о своих пожеланиях. Помните: они имели в виду не то, что сказали вам. Я часто провожу такую проверку: в разговоре с заказчиком заменяю термины заказчика своими собственными и наблюдаю за его реакцией. Вы не поверите, как часто термин *заказчик* имеет смысл, совершенно отличный от термина *клиент*. Тем не менее человек, который объясняет вам, что он хочет видеть в программном продукте, считает эти слова взаимозаменяемыми и уверен, что вы понимаете, какой свой статус он имеет в виду в конкретный момент. А вас это дезориентирует, и качество вашей программы страдает.

Обсуждайте рабочие темы со своими заказчиками бесчисленное количество раз до тех пор, пока не придете к выводу, что вам действительно понятно, что им требуется. Попробуйте вместе с ними переформулировать задачу два-три раза. Обсуждайте с ними то, что происходит непосредственно перед выполнением задачи, и то, что следует за выполнением задачи, чтобы лучше понять контекст. Если есть возможность, обсудите ту же тему с разными людьми в разное время. Рассказы почти всегда будут разными, что выявит отдельные, но связанные факты.

Рассказывая об одном и том же, два человека часто противоречат друг другу. Лучше всего разобраться с этими расхождениями, прежде чем приступать к сверхсложной задаче разработки.

Используйте в беседах наглядные средства. Это могут быть такие простые вещи, как доска во время совещания или простой макет на стадии проектирования, либо такие сложные, как действующий прототип. Общеизвестно, что использование наглядных средств во время обсуждения помогает удержать внимание и закрепить информацию в памяти. Пользуйтесь этим обстоятельством, чтобы обеспечить успех своего проекта.

В прежнюю эпоху своей жизни мне довелось поработать «мультимедийным программистом» в команде, выпускавшей гламурные продукты. Одна наша заказчица очень подробно описала, как она представляет себе внешний вид проекта. Цветовая схема, обсуждавшаяся на проектных совещаниях, предполагала проведение презентации на черном фоне. Мы думали, что все оговорено. Целые команды дизайнеров принялись выдавать сотни многослойных графических файлов. Уйму времени мы убили на формирование конечного продукта. В тот день, когда мы показали клиентке плоды своего труда, нас ждала ошеломительная новость. Увидев продукт и цвет фона, она произнесла: «Когда я говорила „черный“, я имела в виду „белый“». Как видите, «черное» не всегда значит «черное».

Авторы

Алан Гриффитс (Alan Griffiths)

Алан Гриффитс прошел через многие модные поветрия в процессах разработки, технологиях и языках программирования. За эти годы он создал работоспособные программы и процессы разработки для многих организаций, печатался в ряде журналов, выступал на конференциях и подружился со многими людьми. Твердо убежденный в том, что здравый смысл – редкий и ценный рыночный товар, Алан сейчас работает независимым консультантом в собственной компании Octopull Limited.

«Не полагайтесь на “автоматические чудеса”», стр. 78

Алекс Миллер (Alex Miller)

Алекс Миллер – технический руководитель и инженер в Terracotta, Inc., специализирующейся на разработке решений кластеризации с открытым исходным кодом на Java. До Terracotta Алекс работал в BEA Systems над линейкой продуктов AquaLogic и был главным архитектором в MetaMatrix. В сферу его интересов попадают Java, конкурентные вычисления, распределенные системы, языки запросов и проектирование программного обеспечения.

Алекс с удовольствием ведет свой блог *http://tech.puredanger.com*. Вместе с другими разработчиками Terracotta он написал выпущенное в 2008 году «The Definitive Guide to Terracotta» (Полное руководство по Terracotta), Apress. Алекс часто выступает на встречах пользователей и конференциях и является одним из основателей конференции Strange Loop в Сент-Льюисе (*http://thestrangeloop.com*).

«Сначала скажите “да”», стр. 174

Аллан Келли (Allan Kelly)

Аллан Келли – профессионал в области разработки программного обеспечения, но сейчас работает в роли менеджера команды разработки. Он помогает командам разработчиков повышать эффективность своего труда и внедрять методы гибкого программирования. Аллан работает в Лондоне, предлагает мелким и крупным компаниям услуги по обучению, коучингу и консультированию.

Аллан часто публикуется в журналах, появляется на конференциях и является соавтором «Changing Software Development: Learning to Be Agile» (John Wiley & Sons). У Келли степень бакалавра по информатике и MBA по администрированию. В настоящее время он пишет книгу о шаблонах бизнес-стратегий для компаний, выпускающих программное обеспечение. Подробнее узнать о нем можно на сайте *http://www.allankelly.net*.

«Прежде чем пенять на других, проверь собственный код», стр. 38

«Две ошибки могут гасить одна другую (и тогда их трудно исправлять)», стр. 192

Андерс Норас (Anders Norås)

Андерс Норас – закаленный разработчик программного обеспечения и оратор. «Энтерпрайзность» EJB подтолкнула его к Microsoft .NET в 2002 году. Он быстро прославился в сообществе Microsoft, поскольку опыт работы с Java дал ему хорошую фору. В 2006 он восстановил свои отношения с бывшей любовью – Java, и сегодня он – полиглот, отбирающий лучшие решения обеих платформ для создания более качественных программ. Андерс – основатель проекта Quaere и участник нескольких проектов с открытым исходным кодом. Он выступает на многих конференциях и встречах групп пользователей, где обычно показывает мало слайдов и много кода. Андерс живет в Норвегии, где работает главным технологическим евангелистом в Objectware. Ведет свой блог по адресу *http://andersnoras.com*.

«Не просто учите язык, поймите его культуру», стр. 74

Анн Кэтрин Гэгнат (Ann Katrin Gagnat)

Анн Кэтрин Гэгнат четыре года работает с Java и занимает должность системного разработчика в Steria AS в Норвегии. Профессионально интересуется гибкой разработкой, шаблонами и написанием легко читаемого кода.

«Программируйте парами и входите в поток», стр. 148

Аслам Хан (Aslam Khan)

Аслам Хан больше половины своей жизни занимался созданием программного обеспечения. Он по-прежнему верит, что истина заключается в исполняемом коде, но трезво уравновешивает эту веру другим своим базовым принципом: люди важнее компиляторов. Как архитектор программного обеспечения и инструктор Аслам занимается тем, что помогает командам проектировать и выпускать хорошее программное обеспечение, не упуская при этом возможности развлекаться и заводить хороших друзей. Аслам входит в команду factor10 и работает редактором в сообществе архитекторов на DZone. Адрес его блога: *http://aslamkhan.net*.

«Написание кода в духе Убунту для друзей», стр. 194

Берк Хафнагель (Burk Hufnagel)

С 1978 года Берк Хафнагель как архитектор и разработчик программного обеспечения занимается созданием правильных пользовательских интерфейсов. Проведя большую часть своей жизни за проектированием и созданием программ, Берк выработал привычку к тому, чтобы разрабатывать практичные решения сложных задач. Он – библиофил и технофил, интересуется эзотерической тематикой.

Берк был одним из соавторов книги «97 Things Every Software Architect Should Know»[1], O'Reilly. Он выступал в 2008 году на конференции JavaOne с речью, посвященной улучшению условий работы пользователя, и на региональных конференциях Международной ассоциации программных архитекторов в 2007 и 2009 годах. Он также написал статью для библиотеки IASA Skills Library на тему «незаметного слона», а именно связи между опытом взаимодействия пользователя и качеством проектирования пользовательского интерфейса.

«Невероятно, но факт: тестировщики – ваши друзья», стр. 140
«Брось мышь и медленно отойди от клавиатуры», стр. 158

Верити Стоб (Verity Stob)

Верити Стоб – это псевдоним программистки, живущей в Лондоне. Хотя она демонстрирует знание C++ и обычных интерпретируемых языков с фигурными скобками, а также проектирует и пишет код для различных платформ, пожалуй, наиболее удачны и безвредны ее программы для Windows, написанные на CodeGear Delphi.

Свыше 20 лет Верити писала якобы занимательные статьи и колонки для разных журналов, газет и веб-сайтов, включая легендарный (т. е. давно почивший)

[1] Сборник «97 этюдов для архитекторов программных систем», Символ-Плюс, 2010.

«.EXE Magazine», разбивающий общие представления (т. е. почивший несколько позднее) «Dr. Dobb's Journal» и непристойный (т. е. действительно приносящий доход) «The Register». В 2005 году она опубликовала собрание своих трудов под названием «The Best of Verity Stob» (Apress), достигнув тем самым главной цели своей жизни – дважды получить гонорар за одну и ту же работу.

Верити считает посвященную ей в Википедии статью карикатурой на краткость.

«Не прибивайте программу гвоздями к стене», стр. 76

Грег Колвин (Greg Colvin)

Грег Колвин успешно занимается «кодежом» с 1972 года. В свободное от написания кода и чтения специальной литературы время он гуляет со своей собакой по пляжу или играет блюз в местных кабачках.

«Знай свои возможности», стр. 112

Грегор Хоп (Gregor Hohpe)

Грегор Хоп – инженер-программист, работающий в Google. Известность приобрел благодаря своим идеям насчет асинхронной передачи сообщений и сервисно-ориентированной архитектуры, которыми он делится в ряде публикаций, включая фундаментальный труд «Enterprise Integration Patterns»[1] (Addison-Wesley Professional). Подробнее о его работе можно узнать на сайте *http://www.eaipatterns.com*.

«Удобство – не атрибут качества», стр. 58

Гудни Хаукнес (Gudny Hauknes)

Гудни Хаукнес – ведущий разработчик программного обеспечения в норвежском отделении консультативной фирмы Steria. Закончив в 1987 году Норвежский Технологический университет (NTH/NTNU), она занимала различные должности в системных разработках, управлении проектами и контроле качества.

Особенно ее интересуют способы организации спокойной совместной работы, приятной и эффективной, а также, разумеется, создание качественного программного обеспечения.

«Программируйте парами и входите в поток», стр. 148

---

[1] Грегор Хоп, Бобби Вульф «Шаблоны интеграции корпоративных приложений». – Пер. с англ. – Вильямс, 2007.

Диомидис Спинеллис (Diomidis Spinellis)

Диомидис Спинеллис – профессор факультета науки и технологии управления в Университете экономики и бизнеса города Афины, Греция. Он ведет исследования в области разработки программного обеспечения, компьютерной безопасности и языков программирования. Автор двух отмеченных наградами книг из серии «Open Source Perspective»: «Code Reading: The Open Source Perspective»[1] и «Code Quality» (обе изданы в Addison-Wesley Professional), а также десятков научных статей. Его последняя работа – сборник «Beautiful Architecture»[2] (O'Reilly). Диомидис входит в редколлегию IEEE Software и ведет постоянную колонку «Tools of the Trade». Он участвует в разработке FreeBSD, а также UMLGraph и других программных пакетов, библиотек и инструментов с открытым исходным кодом. Получил степень магистра в области разработки ПО и доктора информатики в Имперском колледже Лондона. Диомидис – старший член ACM и IEEE, а также член Usenix Association.

«Место для больших наборов взаимосвязанных данных – в базе данных», стр. 116
«Держите все в системе управления версиями», стр. 156
«Утилиты UNIX – ваши друзья», стр. 196

Джанет Грегори (Janet Gregory)

Соавтор «Agile Testing: A Practical Guide for Agile Testers and Teams» (Addison-Wesley Professional), Джанет Грегори – консультант, помогающий командам создавать качественные системы с помощью методов гибкого программирования. Джанет живет и работает в Канаде, и ее главная страсть – внедрение гибких методов создания качественного программного обеспечения. В качестве инструктора и тестировщика она помогала компаниям внедрять практики гибкого программирования и успешно перевела несколько команд традиционного тестирования в мир гибкого программирования. Ее цель – помочь бизнес-пользователям и тестировщикам осознать свою роль в проектах с гибким программированием. Джанет читает курсы на тему гибкого тестирования и часто выступает на международных конференциях, посвященных гибкому программированию и тестированию. Подробности см. по адресу *http://janetgregory.ca*.

«Когда программисты и тестировщики сотрудничают», стр. 204

---

[1] Диомидис Спинеллис «Анализ программного кода на примере проектов Open Source». – Пер. с англ. – Вильямс, 2004.

[2] Диомидис Спинеллис, Георгиос Гусиос «Идеальная архитектура. Ведущие специалисты о красоте программных архитектур». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

Джейсон П. Сэйдж (Jason P. Sage)

Джейсон П. Сэйдж – консультант по компьютерам и владелец предприятия. Его основные интересы – системное проектирование, интеграция, управление отношениями с клиентами (CRM), оригинальное серверное программное обеспечение, обработка данных и программы для трехмерной графики. Джейсон – настоящий энтузиаст программирования; он отправился в свое путешествие в 1981 году в возрасте 10 лет при помощи Timex Sinclair с двумя килобайтами памяти и кассетным магнитофоном. За прошедшее время он написал множество разнообразных программ, начиная с игр и кончая операционной системой для программного обеспечения управления складскими запасами в одной из крупнейших компаний страны, торгующей продовольственными товарами. Он часто появляется в сетевых форумах, помогая коллегам-программистам и учащимся разного возраста.

«Почаще изобретайте колесо», стр. 164

Джерард Мезарос (Gerard Meszaros)

Джерард Мезарос – независимый консультант, инструктор и преподаватель в области программного обеспечения с 25-летним опытом создания программ и почти 10-летним опытом применения методов гибкого программирования, таких как Scrum, eXtreme Programming и Lean. Он регулярно выступает на таких конференциях по разработке и тестированию программного обеспечения, как OOPSLA, Agile200x и STAR. Написал книгу «xUnit Test Patterns: Refactoring Test Code»[1] (Addison-Wesley) и ведет сайт *http://xunitpatterns.com*.

«Тесты пишутся для людей», стр. 210

Джованни Аспрони (Giovanni Asproni)

Джованни Аспрони – независимый контрагент и консультант, проживающий в Великобритании. Несмотря на то, что он часто работает архитектором, руководителем команды, преподавателем или наставником, в душе он остается программистом, который предпочитает простой код. Он регулярно выступает на конференциях, входит в комитет конференции London XPDay и руководит конференцией ACCU. Джованни является членом ACCU, AgileAlliance, ACM и IEEE.

«Тщательно выбирайте инструменты», стр. 40

«Учитесь делать оценки», стр. 120

---

[1] Джерард Мезарос «Шаблоны тестирования xUnit. Рефакторинг кода тестов». – Пер. с англ. – Вильямс, 2009.

Джон Джаггер (Jon Jagger)

Джон Джаггер – независимый консультант/инструктор/программист/наставник/энтузиаст, специализирующийся на гибком программировании (люди и процесс), разработке на основе тестов, UML, проектировании, анализе, объектно-ориентированном программировании и языках с фигурными скобками (C#, C, C++, Java). Он член британской комиссии по C, бывший член британской комиссии по C++, а также член и главный эксперт (PUKE – Principal UK Expert) британской комиссии по ECMA-стандартизации C#.

Джон также придумал игру «Average Time To Green», опубликовал множество статей в Интернете и журналах и выступил соавтором двух книг: «Microsoft® Visual C#® .NET Step by Step» (Microsoft Press) и «C# Annotated Standard» (Morgan Kaufmann).

Джон женат на прелестной Натали, он счастливый отец Элли, Пенни и Патрика. Безумно увлекается рыбалкой в проточной воде.

«Больше осознанной практики», стр. 64
«Пусть невидимое станет более видимым», стр. 132

Дэн Берг Джонссон (Dan Bergh Johnsson)

Дэн Берг Джонссон – ведущий консультант, партнер и официальный представитель Omegapoint AB. Энтузиаст проектирования на основе предметной области, давний поклонник гибкого программирования, наследник традиций искусства в программировании и «школы разработки OOPSLA». Один из основателей шведской группы предметно-ориентированного проектирования DDD Sverige, регулярно участвует в работе *http://domaindrivendesign.org/* и часто выступает на международных конференциях. О своей любви к профессии он рассказывает в блоге «Dear Junior: Letters to a Junior Programmer» по адресу *http://dearjunior.blogspot.com*.

«Отличайте исключения в бизнес-логике от технических», стр. 62
«Знай, что сохранишь в репозиторий», стр. 114

Дэн Норт (Dan North)

Дэн Норт пишет программы и консультирует команды по методам гибкого и упрощенного программирования. Он ставит во главу угла хорошее отношение к людям и написание простых, практичных программ. Он также считает, что большинство проблем, с которыми сталкиваются команды, вызвано плохо налаженным общением – как, впрочем, и все остальные проблемы тоже.

Поэтому он уделяет такое внимание «правильному выбору слов» и так увлечен разработкой на основе поведения, проблемами обмена информацией и методами обучения. Дэн работает в IT-отрасли с момента окончания учебы в 1991 году и иногда пишет в блоге *http://dannorth.net*.

«Пишите код на языке предметной области», стр. 42

Дэниэл Линднер (Daniel Lindner)

Дэниэл Линднер разрабатывал программное обеспечение в течение 15 лет – как ради денег, так и за интерес (проекты с открытым исходным кодом). Является одним из основателей компании по разработке программного обеспечения в Карлсруэ (Германия) и читает лекции по разработке программного обеспечения. Он также участвует в общественной жизни.

«Пусть ваш проект говорит сам за себя», стр. 124

Жиль Колборн (Giles Colborne)

Жиль Колборн в течение двух десятилетий занимался вопросами эргономики в British Aerospace, Institute of Physics Publishing и Euro RSCG. За это время он провел сотни часов, наблюдая за пользователями в лабораторных и реальных условиях. В 2004 году он основал cxpartners – проектную компанию, ориентированную на пользователя, которая исследует поведение пользователя и проектирует пользовательские интерфейсы для заказчиков со всего света, включая Nokia, Marriott и eBay.

Колборн был президентом британской ассоциации профессионалов в области юзабилити в течение 2003–2007 годов и работал в Британском институте стандартизации, создавая стандарты и руководящие документы по универсальности доступа.

«Выясните, как поступит пользователь (и вы – не пользователь)», стр. 26
«Предотвращайте появление ошибок», стр. 152

Йехиль Кимхи (Yechiel Kimchi)

Йехиль Кимхи – математик (докторская степень Еврейского университета в Иерусалиме за работу в теории множеств), исследователь в области информатики (более 10 лет преподает на факультете информатики в Технионе, Израиль) и разработчик программного обеспечения – более 15 лет работает на большие «хай-тековские» компании и как консультант в собственной небольшой фирме.

Начав писать на C, а потом перейдя на C++, он интересуется ООП и способами разработки программ, которые были бы одновременно корректными, легкими в сопровождении и эффективными. Помимо этого он разработал эвристики для эффективного решения NP-сложных задач, но своим величайшим достижением считает влияние, оказанное на техническое образование тысяч израильских инженеров-программистов.

«Пиши код с умом», стр. 50

Йорн Ольмхейм (Jørn Ølmheim)

Йорн Ольмхейм профессионально занимается программированием уже более 10 лет: поработал и разработчиком, и архитектором, и автором/докладчиком. В настоящее время работает в Statoil, создавая программное обеспечение для ряда исследовательских проектов – в основном с использованием Java, Ruby и Python, а иногда добавляя Fortran и C/C++, если требуются высокоскоростные вычисления. В числе его главных интересов – практика гибкого программирования с упором на мастерство разработчика, языки программирования и автономные системы.

В свободное время Йорн увлекается лыжами, горным велосипедом и общением со своими близкими.

«Красота – следствие простоты», стр. 30

Йоханнес Бродуолл (Johannes Brodwall)

Йоханнес Бродуолл – ведущий научный специалист норвежского отделения консультативной компании Steria. Он любит взглянуть на проект с общей точки зрения, чтобы понять, как сочетание различных дисциплин и технологий может (и может ли) представлять ценность для пользователей программных систем. Он активно участвует в работе сообщества гибкой разработки в Осло. Больше всего времени отнимают у него Oslo Extreme Programming Meetup и ежегодные конференции Smidig 200x (*smidig* на норвежском означает «agile» – гибкий). Он регулярно выступает на мероприятиях, проводимых в районе Осло, и часто пишет о разработке программного обеспечения в своем блоге по адресу *http://johannesbrodwall.com*.

«Сборка должна быть чистой», стр. 104

«Многословный журнал лишит вас сна», стр. 200

Кари Россланд (Kari Røssland)

Кари Россланд – разработчик программного обеспечения в норвежском отделении консультативной компании Steria. За три года, прошедшие после получения степени магистра информатики в NTNU (Тронхейм, Норвегия), Кари приняла участие в нескольких проектах. Ее особенно интересуют гибкая разработка и приятное и эффективное сотрудничество между участниками программных проектов.

«Программируйте парами и входите в поток», стр. 148

Карианне Берг (Karianne Berg)

Карианне Берг получила диплом магистра в Университете Бергена, Норвегия, и в данное время работает в норвежской консультационной фирме Objectware. Ей нравится помогать людям достигать успеха в разработке; она участвует в организации конференций ROOTS и Smidig, а также Oslo XP Meetup. Карианне выступала на нескольких конференциях, и последний раз ее видели на Smidig 2009. Основные сферы ее интересов включают гибкую разработку, шаблоны и фреймворк Spring.

«Читайте код», стр. 160

Кевлин Хенни (Kevlin Henney)

Кевлин Хенни – независимый консультант и инструктор. В основном он занимается шаблонами и архитектурой, приемами программирования и языками, процессами и практикой разработки. Вел колонки в разных журналах и сетевых изданиях, включая «The Register», «Better Software», «Java Report», «CUJ» и «C++ Report». Кевлин – соавтор двух книг серии «Pattern-Oriented Software Architecture» (Архитектура ПО, ориентированная на шаблоны): «A Pattern Language for Distributed Computing» и «On Patterns and Pattern Languages» (Wiley). Он также участвовал в написании книги «97 Things Every Software Architect Should Know»[1].

«Комментируйте только то, о чем не скажет код», стр. 54

«Тестируйте требуемое, а не случайное поведение», стр. 180

«Тестируйте точно и конкретно», стр. 182

[1] Сборник «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

Кейт Брэйтуэйт (Keith Braithwaite)

Кейт Брэйтуэйт – один из главных консультантов Zuhlke. Он также руководит в этой организации Центром практики гибкого программирования. Эта группа осуществляет обучение, инструктирование, наставничество, системное программирование и прямолинейную разработку с целью усиления способностей клиентских команд. Кейт занимался сопровождением компиляторов, моделированием сетей GSM и портированием систем спутниковой навигации для стартапов, промышленных компаний и глобальных сервисных организаций. Он зарабатывал деньги написанием кода на C, C++, Java, Python и Smalltalk. Кейт все более сосредоточивается на использовании «проверенных примеров» или «автоматизированных тестов» как эффективных инструментов для сбора и анализа технических требований, системного проектирования и управления проектами.

Его блог см. по адресу *http://peripateticaxiom.blogspot.com*; его презентации на конференциях можно найти здесь: *http://www.keithbraithwaite.demon.co.uk/professional/presentations/*.

«Читайте гуманитарные книги», стр. 162

«Пишите маленькие функции на основе примеров», стр. 208

Кирк Пеппердин (Kirk Pepperdine)

Кирк Пеппердин работает независимым консультантом и предлагает услуги, связанные с оптимизацией производительности кода на Java. Прежде чем углубиться в Java, Кирк разрабатывал и доводил до ума системы, написанные на C/C++, Smalltalk и ряде других языков. Кирк написал много статей и выступал на ряде конференций, посвященных настройке производительности. Он способствовал превращению *http://www.javaperformancetuning.com* в ресурс, посвященный информации и рекомендациям по настройке производительности.

«Упущенные возможности применения полиморфизма», стр. 138

«Путь к повышению эффективности программ заминирован грязным кодом», стр. 168

«WET размазывает узкие места производительности», стр. 202

Клаус Маркардт (Klaus Marquardt)

Опыт Клауса Маркардта в разработке программного обеспечения включает в себя системы жизнеобеспечения, международные проекты, фреймворки и линейки продуктов, а также гибкую разработку на режимных объектах. Он описал ряд диагнозов и терапевтических мер для программных систем исходя из собственного интереса к вза-

имному влиянию технологий, людей, процессов и организации; их можно найти на сайте *http://www.sustainable-architecture.eu*. Кроме того, Клаус любит создавать шаблоны, вести на конференциях встречи, где исследуются новые возможности, и интересоваться в жизни чем-то еще помимо программирования.

«Учите иностранные языки», стр. 118

«Долговечность временных решений», стр. 128

Клинт Шэнк (Clint Shank)

Клинт Шэнк – разработчик программного обеспечения, консультант и наставник в Sphere of Influence, Inc. – компании, которая лидирует в проектных инновациях, применяя нестандартные подходы для создания невероятных программ, потрясающих во всех смыслах. Обычно он консультирует по вопросам проектирования и конструирования приложений промышленного масштаба.

Особенно его привлекают практики гибкого программирования, такие как непрерывная интеграция и разработка на основе тестирования, языки программирования Java, Groovy, Ruby и Scala, фреймворки Spring и Hibernate, а также проектирование и архитектура приложений в целом.

Клинт ведет блог по адресу *http://clintshank.javadevelopersjournal.com*, он один из авторов сборника статей «97 Things Every Software Architect Should Know»[1].

«Непрерывное обучение», стр. 56

Кэй Хорстман (Cay Horstmann)

Кэй Хорстман вырос в северной Германии и учился в Университете им. Кристиана Альбрехтса в Киле – портовом городе на Балтийском море. Получил степень магистра информатики в Сиракузском университете и доктора математики в Университете штата Мичиган в Энн-Арбор. Четыре года Кэй выступал в роли вице-президента и технического директора интернет-стартапа, разросшегося с трех человек в крошечном офисе до открытой акционерной компании. Сейчас он преподает информатику в Университете Сан-Хосе. Располагая массой свободного времени, Кэй пишет книги и статьи, посвященные языку Java и обучению информатике.

«Шаг назад. Теперь автоматизируй, автоматизируй, автоматизируй...», стр. 176

---

[1] Сборник «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

Кэл Эванс (Cal Evans)

Кэл Эванс – директор экспертного центра PHP (PCE, PHP Center of Expertise) компании Ibuildings. Он программировал на различных языках более 25 лет. Пишет книги и журнальные статьи, посвященные нескольким языкам программирования. Кэл – американец, но сейчас живет в Утрехте (Голландия), где выступает, публикуется, пишет код и участвует в работе глобального сообщества PHP. У него есть блог: *http://blog.calevans.com*.

«Комментарий о комментариях», стр. 52
«Этот код не трогать!», стр. 82

Кэрролл Робинсон (Carroll Robinson)

Кэрролл Робинсон – разработчик прошивок встраиваемых систем с практическим опытом порядка 20 лет. Он писал микропрограммы на C и ассемблере для самых разных процессоров (в том числе 8051, 80x86, 68k, ARM7 и C2000), применявшихся в медицинском оборудовании, лабораторных приборах и системах беспроводной связи. Кэрроллу приходилось создавать и приложения на C++, Java и Python. Он предпочитает пользоваться инструментами с открытым исходным кодом (GCC, GAS, GDB) на различных платформах Linux и создал несколько встроенных Linux-систем.

Кэрролл закончил магистратуру Университета Case Western Reserve (Кливленд, Огайо) по специальности компьютерная инженерия.

«Умей пользоваться утилитами командной строки», стр. 106

Линда Райзинг (Linda Rising)

Линда Райзинг получила докторскую степень в Университете штата Аризона, а ее резюме включает и преподавание в университете, и работу в ряде промышленных отраслей. Линду знают во многих странах благодаря ее выступлениям, посвященным шаблонам, ретроспективам, гибкой разработке и процессам перемен. Она является автором множества статей и четырех книг, последнюю из которых, «Fearless Change: Patterns for Introducing New Ideas» (Addison-Wesley), написала в соавторстве с Мэри Линн Маннс (Mary Lynn Manns).

«Послание потомкам», стр. 136

Майк Льюис (Mike Lewis)

Майк Льюис работает сейчас инженером-программистом в Lutron Electronics, а в свободное время – независимым консультантом по программному обеспечению. Его более чем 10-летний опыт программирования помогает ему создавать элегантные и интуитивно понятные программные решения. Он пропагандирует совершенствование процессов и страстно стремится улучшать пользовательские интерфейсы везде, где только возможно.

У Майка степени бакалавра и магистра по разработке ПО, полученные в Рочестерском технологическом институте. Сейчас он живет в Аллентауне, штат Пенсильвания, у границы Нью-Йорка и Филадельфии.

«Не бойтесь что-нибудь сломать», стр. 68

Майкл Фезерс (Michael Feathers)

Майкл Фезерс – консультант в Object Mentor International. Он занят работой с разными командами по всему миру, их обучением и наставлением. Майкл разработал CppUnit, первый порт JUnit на C++, и FitCpp, портированную на C++ среду интеграционного тестирования Fit. Майкл – автор книги «Working Effectively with Legacy Code»[1] (Prentice Hall).

«Золотое правило проектирования API», стр. 90

Маркус Бэйкер (Marcus Baker)

Маркус Бэйкер обожает заниматься программированием и поражается, что за это ему еще платят деньги. Его обожание распространяется на телефонию, анализ данных, робототехнику и веб-разработку. Время от времени он пишет статьи или ведет колонки, организует группы пользователей и конференции. В настоящее время он, однако, занят уходом за детьми.

«Установи меня!», стр. 100

[1] Майкл Фезерс «Эффективная работа с унаследованным кодом». – Пер. с англ. – Вильямс, 2009.

Маттиас Карлссон (Mattias Karlsson)

Маттиас Карлссон большую часть времени занимается разработкой программного обеспечения для финансового сектора, а также руководит группой пользователей Java (JUG, Java User Group) в Стокгольме (Швеция). Маттиас занимается объектно-ориентированными разработками с 1993 года. С годами он приобрел опыт работы в разных качествах, в том числе разработчика, архитектора, руководителя команды, инструктора, менеджера и преподавателя. Во всех этих ролях его способность вдохновлять и мотивировать своих сотрудников получила высокую оценку. Группа JUG ежегодно проводит от шести до восьми представительных совещаний с числом участников более 200. Маттиас также выступил одним из организаторов Jfokus, крупнейшей ежегодной конференции по Java в Стокгольме.

В свободное время Маттиас играет с детьми или катается на мотоцикле, а также помогает строить жилье для неимущих в рамках организации Habitat for Humanity. Маттиас также поддерживает организацию взаимного микрокредитования Kiva. Узнайте, как присоединиться к его стараниям улучшить мир, на *http://www.kiva.org/team/jug*.

«Рецензирование кода», стр. 48

Микаэль Хунгер (Michael Hunger)

Микаэль Хунгер увлекся программированием еще в детские годы в Восточной Германии. Особенно интересуется людьми, которые разрабатывают программное обеспечение, мастерством программирования, языками программирования и совершенствованием кода. Ему нравится заниматься инструктированием и внутрипроектными разработками в качестве независимого консультанта («евангелист разработки лучших программ» – *http://jexp.de*), но в его жизни есть и другие любимые проекты.

Половину жизни он отдает семье (у Микаэля трое детей), давней зависимости от текстовой многопользовательской игры в подземелье (MUD MorgenGrauen), чтению книг, когда это возможно, работе в собственной кофейне «die-buchbar», где есть мастерская для печати на разных материалах, и всяким поделкам с Lego® или без него. Другая половина занята работой с одними языками программирования и изучением других, прослушиванием IT-подкастов (особенно Software Engineering Radio; *http://se-radio.net/*), участием в интересных и амбициозных проектах типа qi4j, созданием DSL (jequel, squill и xmldsl), обширным рефакторингом и участием в написании и рецензировании книг. Недавно он стал выступать на конференциях.

«Предметно-ориентированные языки», стр. 66

Мэтт Доар (Matt Doar)

Мэтт Доар работает консультантом по инструментам программирования, таким как системы управления версиями (CVS, Subversion), системы сборки (make, Scons) и системы отслеживания ошибок (Bugzilla, JIRA). Большая часть его клиентов – небольшие стартапы в Кремниевой долине. Мэтт – автор вышедшей в O'Reilly книги «Practical Development Environments».

«Как пользоваться системой отслеживания ошибок», стр. 96

Никлас Нильссон (Niclas Nilsson)

Никлас Нильссон – наставник по разработке программного обеспечения, консультант, преподаватель и писатель, глубоко увлеченный профессией и влюбленный в красивые архитектурные решения. Он стал разработчиком в 1992 году. Никлас по собственному опыту знает, какое огромное значение для разработки ПО может иметь выбор языков, инструментов, способа общения и процессов. Вот почему он любит динамические языки, разработку на основе тестов, генерацию кода, метапрограммирование и методы гибкой разработки. Никлас входит в число создателей factor10 и работает редактором сообщества архитекторов в InfoQ. Его блог находится по адресу *http://niclasnilsson.se*.

«Думайте состояниями», стр. 188

Нил Форд (Neal Ford)

Нил Форд – архитектор программного обеспечения и меметик в ThoughtWorks, глобальной консультативной компании в области IT, уделяющей исключительное внимание сквозной разработке и поставке программного обеспечения. Он проектирует/разрабатывает приложения, обучающие материалы, учебные курсы, видеопрезентации, пишет журнальные статьи, является автором и/или редактором пяти книг[1]. Он также часто выступает на конференциях. Удовлетворить свой жадный интерес к личности Нила вы сможете на сайте *http://www.nealford.com*.

«Тестирование – это инженерная строгость в разработке программного обеспечения», стр. 186

[1] Нил Форд «Продуктивный программист». – Пер. с англ. – СПб.: Символ-Плюс, 2009.

Нэйт Джексон (Nate Jackson)

Нэйт Джексон – старший архитектор программного обеспечения в Буффало, штат Нью-Йорк. Занимается написанием кода с 1979 года, когда у него появился TI-99 и картридж с эмулятором Бэйсика. Следуя собственному совету, он удовлетворил всех своих клиентов – даже ту даму, которая пожелала иметь белый фон.

«Ваши заказчики имеют в виду не то, что говорят», стр. 214

Олве Маудал (Olve Maudal)

Олве Маудал живет в Норвегии. Женат. Двое детей. Убежденный компьютерный гик. В данное время в основном пишет код на C и C++.

В университете Олве изучал разработку ПО и искусственный интеллект. Профессиональную карьеру он начал в нефтяной отрасли в компании, разрабатывавшей системы разведки нефти и газа. Затем несколько лет занимался системами денежных переводов. Сегодня работает в телекоммуникационной компании, создавая системы для эффективной связи между людьми.

Олве – активный член энергичного сообщества гиков в Осло, где, среди прочего, является организатором группы пользователей C++ в Осло. Его блог см. по адресу *http://olvemaudal.wordpress.com*.

«Тяжелый труд не оправдывает себя», стр. 94

Петер Зоммерлад (Peter Sommerlad)

Петер Зоммерлад – профессор и глава Института программного обеспечения в HSR Rapperswil. Петер – соавтор книги «Pattern-Oriented Software Architecture», Volume 1 и «Security Patterns» (обе изданы Wiley). Его долгосрочная задача – сделать программное обеспечение проще благодаря *декрементной разработке*: рефакторингу программ до 10% их размера благодаря лучшей архитектуре, легкости тестирования, качеству и функциональности.

«Правду скажет только код», стр. 144

Пит Гудлиф (Pete Goodliffe)

Пит Гудлиф – разработчик программного обеспечения, оратор и писатель, который никогда долго не задерживается на одной роли в отрасли программного обеспечения. Он писал на многих языках во многих проектах. Он также преподает и обучает программистов и ве-

дет постоянную колонку «Professionalism in Programming» в журнале ACCU «CVu» (*http://accu.org/*).

Написанная Питом популярная книга «Code Craft»[1] (No Starch Press) служит практичным и увлекательным исследованием самого занятия программированием. Пит любит писать превосходный код, в котором отсутствуют ошибки, благодаря чему может больше времени проводить со своими детьми. Он обожает карри и не носит обувь.

«Не проходите мимо ошибки!», стр. 72
«Улучшайте код, удаляя его», стр. 98
«Нужно заботиться о коде», стр. 212

Пол У. Гомер (Paul W. Homer)

Пол У. Гомер – разработчик программного обеспечения, писатель и немного фотограф, который занялся разработкой программ несколько десятилетий назад и с тех пор стремится строить все более сложные системы. Его опыт включает в себя работу в фирме, консультирование и коммерческие разработки в самых разнообразных должностях, включая аналитика, архитектора, программиста, менеджера и даже – как ни глупо – технического директора. Он готов заниматься любым делом, направленным на создание и выпуск систем.

В последние несколько лет он стал уделять больше внимания общению с коллегами-разработчиками, в связи с чем опубликовал книгу, ведет блог и очень много выступает в надежде помочь программной отрасли лучше понять себя и достичь новых высот.

«Простота достигается сокращением», стр. 170

Раджит Аттапатту (Rajith Attapattu)

Раджит Аттапатту – старший инженер-программист в команде MRG Red Hat. Раджит – энтузиаст открытых проектов и участник ряда проектов Apache, в том числе Apache Qpid, Apache Synapse, Apache Tuscany и Apache Axis2. В последнее время он сосредоточился на разработке масштабируемого и надежного программного обеспечения промежуточного уровня для передачи сообщений и вошел в группу AMQP (Advanced Message Queuing Protocol).

Он опубликовал несколько статей и выступил на ряде конференций и встреч групп пользователей, в том числе ApacheCon, Colorado Software Summit и Toronto

[1] Питер Гудлиф «Ремесло программиста. Практика написания хорошего кода». – Пер. с англ. – СПб.: Символ-Плюс, 2009.

JUG. В область научных интересов Раджита входит улучшение масштабируемости и высокой доступности распределенных систем. В свободное время Раджит любит рисовать и играть в крикет.

С ним можно связаться по адресу *rajith@apache.org*, а также найти на *http://rajith.2rlabs.com*.

«Прежде чем приступать к рефакторингу», стр. 32
«Тестируйте во сне (и по выходным)», стр. 184

Райан Браш (Ryan Brush)

Райан Браш – директор и заслуженный инженер (Distinguished Engineer) в Cerner Corporation, где он работает с 1999 года. Его главный интерес – применение технологий в области здравоохранения.

«Код – это проектирование», стр. 44
«Миф о гуру», стр. 92

Рассел Уиндер (Russel Winder)

Рассел Уиндер – партнер в Concertant LLP, предоставляющей услуги аналитики и консультаций по всем проблемам параллельных и конкурентных вычислений и многоядерным системам. Он также выступает в качестве независимого консультанта, автора и преподавателя по программированию, языкам программирования (Java, Groovy и Python), системам управления версиями (Subversion, Bazaar и Git) и фреймворкам сборки (Gant, SCons, Gradle, Ant и Maven). Рассел – автор книги «Developing C++ Software» (Wiley), соавтор книг «Developing Java Software» (Wiley) и «Python for Rookies» (Cengage Learning Business Press).

«Как следует изучи более двух языков программирования», стр. 108
«Передача сообщений улучшает масштабируемость параллельных систем», стр. 134

Ричард Монсон-Хейфел (Richard Monson-Haefel)

Ричард Монсон-Хейфел – независимый разработчик программного обеспечения, соавтор всех пяти изданий «Enterprise JavaBeans»[1] и обоих изданий «Java Message Service» (O'Reilly), а также автор

[1] Ричард Монсон-Хейфел «Enterprise JavaBeans», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2002.

«J2EE Web Services» (Addison-Wesley). Ричард – редактор книги «97 Things Every Software Architect Should Know»[1]. Является одним из основателей проекта OpenEJB (проект с открытым исходным кодом), контейнера EJB для Apache Geronimo, а сейчас консультирует разработки iPhone и Microsoft Surface.

«Удовлетворяйте свое честолюбие через Open Source», стр. 88

Роберт Мартин «Дядюшка Боб» (Robert C. Martin «Uncle Bob»)

Роберт Мартин (Дядюшка Боб) профессионально занимается разработкой программного обеспечения с 1970 года и является основателем и президентом Object Mentor, Inc. в Герни, штат Иллинойс. Object Mentor, Inc. – международная компания, где работают разработчики и менеджеры с большим опытом, которые помогают компаниям доводить их проекты до завершения. Object Mentor предлагает консультации по совершенствованию процессов, обучение, консультации и повышение квалификации в объектно-ориентированном программировании для крупных мировых компаний.

Боб опубликовал десятки статей в различных отраслевых журналах и часто выступает на международных конференциях и выставках. Является автором и редактором многих книг, включая «Designing Object-Oriented C++ Applications Using the Booch Method» (Prentice Hall), «Patterns Languages of Program Design 3» (Addison-Wesley Professional), «More C++ Gems» (Cambridge University Press), «Extreme Programming in Practice» (Addison-Wesley Professional), «Agile Software Development: Principles, Patterns, and Practices»[2], «UML for Java Programmers» и «Clean Code»[3] (все вышли в Prentice Hall).

Будучи одним из лидеров отрасли разработки ПО, Боб в течение трех лет выполнял обязанности главного редактора «C++ Report» и был первым президентом «Agile Alliance».

«Правило бойскаута», стр. 36

«Профессиональный программист», стр. 154

«Принцип единственной ответственности», стр. 172

---

[1] Сборник «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

[2] Роберт К. Мартин, Джеймс В. Ньюкирк, Роберт С. Косс «Быстрая разработка программ. Принципы, примеры, практика». – Пер. с англ. – Вильямс, 2004.

[3] Роберт Мартин «Чистый код. Создание, анализ и рефакторинг». – Пер. с англ. – СПб.: Питер, 2011.

Род Бегби (Rod Begbie)

Род Бегби ведет свой род из Шотландии, но сейчас его сердце покорил город Сан-Франциско.

Днем он работает ведущим проектировщиком в Slide, Inc., а вечером ухаживает за пандами. Ранее он трудился архитектором API в Current TV, скрывался в исследовательской лаборатории Bose Corporation, консультировал Sapient и успешно спасся от (первого) лопнувшего пузыря доткомов в подвале банка, где разрабатывал системы для анализа ежегодных рент с фиксированным доходом, что было интересно настолько же, насколько интересно звучит.

«Не прикалывайтесь с тестовыми данными», стр. 70

Рэнди Стэффорд (Randy Stafford)

Рэнди Стэффорд – профессионал в области программного обеспечения с 20-летним опытом работы в качестве программиста, аналитика, архитектора, менеджера, консультанта и автора/докладчика. Сейчас он работает в Oracle, в команде A-Team, где занимается проектами POC, рецензированием архитектуры и производственными кризисами. Он специализируется на грид-вычислениях, SOA, производительности, HA и JEE/ORM.

Рэнди работал техническим консультантом в Rally Software, главным архитектором в IQNavigator, директором по развитию в SynXis, консультантом в GemStone и Smalltalk, а также специалистом по моделированию в аэрокосмической отрасли и CASE-индустрии. Он соавтор книг: «97 Things Every Software Architect Should Know»[1] (O'Reilly), «Patterns of Enterprise Application Architecture»[2] (Wiley) и «EJB Design Patterns» (Addison-Wesley Professional).

«Межпроцессная коммуникация влияет на время отклика приложения», стр. 102

Сара Маунт (Sarah Mount)

Сара Маунт – старший преподаватель информатики в Университете Вулверхемптона. Она интересуется языками и инструментами программирования, особенно для беспроводных сетей датчиков и других распределенных систем. Сара читала студентам вводный курс

---

[1] Сборник «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

[2] М. Фаулер, Д. Райс, М. Фоммел, Э. Хайет, Р. Ми, Р. Стэффорд «Шаблоны корпоративных приложений». – Пер. с англ. – Вильямс, 2010.

программирования в течение 9 лет и является соавтором учебника «Python for Rookies» (Cengage Learning Business Press).

«Пользуйтесь инструментами для анализа кода», стр. 178

Себ Роуз (Seb Rose)

Себ Роуз – главный инженер-программист в эдинбургской команде Rational DOORS. Он начал программировать в 1980 году, создавая приложения для агентов по недвижимости и адвокатов на компилируемом BASIC для Apple IIe. Окончив Эдинбургский университет в 1987 году, он сначала участвовал в проекте REKURSIV, а потом стал независимым разработчиком. Сегодня его главными профессиональными интересами являются методики гибкого программирования и реанимация «унаследованных» программ.

«Будьте благоразумны», стр. 22

Скотт Мейерс (Scott Meyers)

Скотт Мейерс – автор, преподаватель, оратор и консультант. Его опыт в разработке и научной работе превышает тридцать лет. Он написал десятки журнальных статей, а также книги «Effective C++»[1], «More Effective C++»[2] и «Effective STL»[3] (все вышли в Addison-Wesley Professional). Он также проектировал и контролировал их публикацию в форматах HTML и PDF. Скотт – редактор-консультант издаваемой Addison-Wesley серии «Effective Software Development» и был одним из первых членов консультативного совета сетевого журнала «The C++ Source» (*http://www.artima.com/cppsource*). Он получил докторскую степень по информатике в Университете Брауна. Адрес его сайта *http://www.aristeia.com*.

«Интерфейсы должно быть легко использовать правильно и трудно – неправильно», стр. 130

---

[1] Скотт Мейерс «Эффективное использование C++. 55 верных советов улучшить структуру и код ваших программ». – Пер. с англ. – ДМК Пресс, 2006.

[2] Скотт Мейерс «Эффективное использование C++. 35 новых способов улучшить стиль программирования». – Пер. с англ. – СПб.: Питер, 2006.

[3] Скотт Мейерс «Эффективное использование STL». – Пер. с англ. – СПб.: Питер, 2002.

Стив Берчук (Steve Berczuk)

Стив Берчук – инженер-программист в Humedica, где занимается разработкой интеллектуальных бизнес-приложений для медицины. Разработкой программных приложений он занимается свыше 20 лет. Стив – автор книги «Software Configuration Management Patterns: Effective Teamwork, Practical Integration» (Addison-Wesley Professional). Помимо разработки приложений он любит помогать командам более эффективно организовать свою работу на основе методов гибкого программирования и управления конфигурациями программного обеспечения. Адрес его сайта *http://www.berczuk.com*.

«Развертывание приложения: раннее и регулярное», стр. 60
«Возьмите сборку (и ее рефакторинг) на себя», стр. 146

Стив Смит (Steve Smith)

Стив Смит – разработчик программного обеспечения, оратор, автор и наставник. Он профессионально занимается разработкой программного обеспечения с 1997 года и участвовал в написании нескольких книг, в основном в области ASP.NET. Регулярно выступает на встречах пользователей и таких профессиональных конференциях, как DevConnections и Microsoft TechEd. Стив – бывший капитан инженерных войск армии США и ветеран боевых действий в Ираке, где командовал взводом, занимавшимся обезвреживанием неразорвавшихся боеприпасов и самодельных взрывных устройств. Стив живет в Огайо с женой и двумя детьми и является одним из координаторов группы Hudson Software Craftsmanship.

«Не повторяй свой код», стр. 80

Стив Фримен (Steve Freeman)

Стив Фримен – независимый консультант, специализирующийся на методах гибкой разработки. Он возглавлял, инструктировал и обучал команды во всех уголках мира. Стив – соавтор книги «Growing Object-Oriented Software, Guided by Tests» (Addison-Wesley). В 2006 году Стив получил награду Agile Alliance Gordon Pask. Он участвует в проектах jMock и Hamcrest и создал NMock. Он член-основатель eXtreme Tuesday Club и был руководителем первого London XpDay. Стив выполнял роль организатора и участника многих международных профессиональных конференций. Получил докторскую степень в Кембриджском университете, а еще раньше получил степени в области статистики и музыки. В настоящее время его интересуют проблемы создания хорошего кода и исследование сложности организаций.

«Важность форматирования кода», стр. 46
«Один бинарный файл», стр. 142

Сэм Сааристе (Sam Saariste)

Сэм Сааристе – дипломированный магистр электротехники, занимается профессиональной разработкой программного обеспечения с 1995 года. Сэм отметился в различных предметных областях – от обработки речи в реальном времени в телекоммуникационной сфере до трейдерских приложений в инвестиционных банках. Его любимый язык – C++, он член комиссии BSI C++ с 2005 года. Сэм – поклонник методов гибкой разработки с того момента, как в 2000 году открыл для себя экстремальное программирование. Его заботит качество программного обеспечения, и он уверен, что с помощью «agile» и «lean» можно одновременно достичь как высокого качества, так и высокой продуктивности.

«Не поддавайтесь очарованию шаблона Singleton», стр. 166

Томас Гест (Thomas Guest)

Томас Гест – опытный и энергичный программист. Он предпочитает языки высокого уровня и простые решения. Его тексты опубликованы в ряде сетевых и печатных изданий, а также на его личном сайте *http://www.wordaligned.org*.

«Научитесь говорить «Hello, World»», стр. 122

Уди Дахан (Udi Dahan)

Уди Дахан – «The Software Simplist» (программный знахарь), всемирно известный специалист по архитектуре и проектированию программного обеспечения. MVP (Most Valuable Professional) в области архитектуры решений и связанных систем в течение четырех лет подряд, Уди также входит в число 33 европейских экспертов, признанных международной ассоциацией .NET, является автором и преподавателем в International Association of Software Architects, а также гуру SOA, Web Services и XML, рекомендованным журналом DDJ.

В свободное от консультаций, выступлений и преподавания время Уди руководит разработкой NServiceBus, самой популярной сервисной шиной предприятия на основе .NET с открытым исходным кодом. Его можно найти по адресу *http://www.UdiDahan.com*.

«Осторожно: общий код», стр. 34

Уолтер Брайт (Walter Bright)

Уолтер Брайт – создатель компиляторов, написавший их для C, C++, ECMAScript, ABEL, Java и недавно для языка программирования D. Он также известен как изобретатель игры-стратегии *Empire*.

«Компоновщик не таит в себе никаких чудес», стр. 126

Филип ван Лаенен (Filip van Laenen)

Филип ван Лаенен – ведущий инженер в норвежской софтверной компании Computas AS, которая разрабатывает IT-решения в общественном и частном секторе Норвегии. Он начинал разработчиком в малых и больших командах, и за 10 лет работы в отрасли Филип вырос до ведущего разработчика и руководителя, отвечающего за безопасность и программную инженерию целой компании. В своей профессиональной деятельности он использовал различные языки программирования, включая Smalltalk, Java, Perl, Ruby и PL/SQL. Особый интерес Филип проявляет к безопасности компьютеров и криптографии и несколько лет занимал в Computas должность ответственного за безопасность.

У Филипа степени магистра электроники и магистра информатики, полученные в Католическом университете Левена. Он родом из Фландрии, но переехал в 1997 году в Норвегию и теперь вместе с семьей живет в Колсосе около Осло.

«Автоматизируйте свой стандарт форматирования кода», стр. 28

Хейнц Кабуц (Heinz Kabutz)

Хейнц Кабуц – автор «The Java Specialists' Newsletter» (Бюллетень специалистов по Java), издания, которое читают 50000 специалистов по Java в 120 странах. Большую часть времени он пишет код на Java в качестве подрядчика многочисленных компаний. Кроме того, он читает в компаниях лекции о том, как эффективнее писать программы на Java, используя развитые функции этого языка.

Хейнц входит в сообщество «Java Champions», интервью с ним опубликовано Sun Microsystems (см. *http://java.sun.com/developer/technicalArticles/Interviews/community/kabutz_qa.html*).

«Знай свою IDE», стр. 110

Чак Эллисон (Chuck Allison)

Чак Эллисон – адъюнкт-профессор информатики в Университете Юта Вэлли. Предшествующие два десятилетия он работал инженером-программистом на западе США. Чак активно участвовал в разработке стандарта C++98, был ведущим редактором «C/C++ Users Journal» и вместе с Брюсом Эккелем (Bruce Eckel) участвовал в написании 2-го тома «Thinking in C++». Он также основал «The C++ Source» и является пишущим редактором «Better Software Magzine». Подробнее о Чаке можно узнать на его сайте *http://www.chuckallison.com*.

«Числа с плавающей запятой недействительны», стр. 86

Эдвард Гарсон (Edward Garson)

Увлечение Эдварда Гарсона компьютерами началось с изучения Logo для Apple II. Сегодня он независимый консультант по разработке программного обеспечения, помогает компаниям переходить на методы гибкого программирования.

В число технических интересов Эдварда входят архитектура и проектирование программного обеспечения, языки программирования, а также GNU/Linux. Он энергичный оратор и выступал в Британском компьютерном обществе, Совете архитекторов Microsoft и на различных конференциях. Эдвард участвовал в написании книги «97 Things Every Software Architect Should Know»[1].

Эдвард живет в Монреале с женой и двумя сыновьями. В свободное время любит кататься на лыжах, ходить в горы и путешествовать на велосипеде.

«Применяйте принципы функционального программирования», стр. 24

Эдриан Уайбл (Adrian Wible)

Эдриан сам себе присвоил титул «катализатор разработки ПО». Он работает в ThoughtWorks, Inc. в основном как руководитель проектов, но старается опровергать обвинения в отрыве от реальности тем, что время от времени сам занимается практической разработкой программ. Работая в IBM, он усвоил методику каскадной разработки (Waterfall/SDLC) и внедрял ее в управление проектами, людьми и процессами на протяжении своей более чем 20-летней карьеры в IBM и Dell Computer Corporation.

---

[1] Сборник «97 этюдов для архитекторов программных систем». – Пер. с англ. – СПб.: Символ-Плюс, 2010.

С переходом в 2005 году в ThoughtWorks Эдриан открыл для себя «Манифест гибкой разработки» (Agile Manifesto), а также экстремальное программирование, Scrum и прочее и понял, что от участия в проекте и от руководства проектом *можно* получать удовольствие, восторг и удовлетворение. К прежним подходам он уже не вернулся.

Связаться с Эдрианом можно по адресу *awible@thoughtworks.com*.

«Одна голова хорошо, но две – часто лучше», стр. 190

Эйнар Ландре (Einar Landre)

Эйнар Ландре – практикующий профессионал в области программного обеспечения, проработал 25 лет в качестве разработчика, архитектора, менеджера, консультанта и автора/лектора. Сейчас он работает в службе бизнес-приложений StatoilHydro, где занят разработкой критически важных для бизнеса приложений, рецензированием архитектуры и совершенствованием процессов разработки программного обеспечения.

До поступления в StatoilHydro Эйнар работал в качестве разработчика, консультанта и менеджера, занимаясь проектированием и реализацией коммуникационных протоколов, операционных систем и тестированием программного обеспечения для международной космической станции.

В последние годы Эйнар стал активно участвовать в профессиональном сообществе, написал или выступил соавтором ряда докладов, представленных на OOPSLA и SPE (Society of Petroleum Engineers). Среди его профессиональных интересов объектно-ориентированное программирование, проектирование автономных систем, применение практики системной инженерии, методологии гибкой разработки и лидерство в высокотехнологических организациях.

У Эйнара степень магистра информатики, полученная в Университете Стратклайда, и диплом сертифицированного IEEE профессионального разработчика программного обеспечения (CSDP). Он живет вместе с семьей в Ставангере (Норвегия).

«Инкапсулируйте поведение, а не только состояние», стр. 84

«Предпочитайте примитивам предметно-ориентированные типы данных», стр. 150

Юрий Зубарев (Yuriy Zubarev)

Юрий Зубарев – архитектор программных систем и руководитель команды в YachtWorld.com, подразделении Dominion Enterprises. Его деятельность сосредоточена на интеграции программных систем, технологиях сбора знаний и слежения, а также на повышении технической эффективности и уровня мастерства в его компании. Юрий живет и работает в красивейшем городе Ванкувере в канадской провинции Британская Колумбия. Помимо написания кода он увлекается латиноамериканскими танцами.

«Пишите код так, как будто вам предстоит сопровождать его всю оставшуюся жизнь», стр. 206

Ян Кристиаан ван Винкель (Jan Christiaan «JC» van Winkel)

Ян Кристиаан ван Винкель – инструктор и разработчик учебных курсов для небольшой голландской учебной и консультативной фирмы AT Computing. В его работе преобладают системы UNIX/Linux (системное администрирование, безопасность, анализ эффективности) и языки программирования (в основном C, C++ и Python). Он также представляет Голландию в процессе стандартизации C++. Двенадцать лет «JC» входил в состав руководства голландской группы пользователей UNIX (Netherlands UNIX User's group, NLUUG), шесть из которых занимал пост председателя.

«Правильно выбирайте алгоритмы и структуры данных», стр. 198

Алфавитный указатель

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Издательство "СИМВОЛ-ПЛЮС"

Основано в 1995 году

О нас

Наша специализация – книги компьютерной и деловой тематики. Наши издания – плод сотрудничества известных зарубежных и отечественных авторов, высококлассных переводчиков и компетентных научных редакторов. Среди наших деловых партнеров издательства: O'Reilly, Pearson Education, NewRiders, Addison Wesley, Wiley, McGraw-Hill, No Starch Press, Packt, Dorset House, Apress и другие.

Где купить

Наши книги вы можете купить во всех крупных книжных магазинах России, Украины, Белоруссии и других стран СНГ. Однако по минимальным ценам и оптом они продаются:

Санкт-Петербург:
главный офис издательства –
В.О. 16 линия, д. 7 (м. Василеостровская),
тел. (812) 380-5007

Москва:
московский филиал издательства –
ул. 2-я Магистральная, д. 14В
(м. Полежаевская/Беговая),
тел. (495) 638-5305

Заказ книг

через Интернет http://www.symbol.ru

Бесплатный каталог книг высылается по запросу.

Приглашаем к сотрудничеству

www.symbol.ru

Мы приглашаем к сотрудничеству умных и талантливых авторов, переводчиков и редакторов. За более подробной информацией обращайтесь, пожалуйста, на сайт издательства www.symbol.ru.

Также на нашем сайте вы можете высказать свое мнение и замечания о наших книгах. Ждем ваших писем!