



Роберт Мартин

Идеальный программист

Как стать профессионалом разработки ПО



Robert C. Martin

The Clean Coder: A Code of Conduct for Professional Programmers



Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com



БИБЛИОТЕКА ПРОГРАММИСТА

Роберт Мартин

Идеальный программист

Как стать профессионалом
разработки ПО



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара • Новосибирск
Киев • Харьков • Минск

2012

ББК 32.973.2-018-02
УДК 004.415
М29

Мартин Р.

М29 Идеальный программист. Как стать профессионалом разработки ПО. — СПб.: Питер, 2012. — 224 с.: ил.

ISBN 978-5-459-01044-2

Всех программистов, которые добиваются успеха в мире разработки ПО, отличает один общий признак: они больше всего заботятся о качестве создаваемого программного обеспечения. Это — основа для них. Потому что они являются профессионалами своего дела.

В этой книге легендарный эксперт Роберт Мартин (более известный в сообществе как «Дядюшка Боб»), автор бестселлера «Чистый код», рассказывает о том, что значит «быть профессиональным программистом», описывая методы, инструменты и подходы для разработки «идеального ПО». Книга насыщена практическими советами в отношении всех аспектов программирования: от оценки проекта и написания кода до рефакторинга и тестирования. Эта книга — больше, чем описание методов, она о профессиональном подходе к процессу разработки.

ББК 32.973.2-018-02
УДК 004.415

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0137081073 англ.
ISBN 978-5-459-01044-2

© Prentice Hall, Inc., 2011
© Перевод на русский язык ООО Издательство «Питер», 2012
© Издание на русском языке, оформление
ООО Издательство «Питер», 2012

Краткое содержание

Обязательное вступление.	13
Глава 1. Профессионализм	19
Глава 2. Как сказать «нет»	35
Глава 3. Как сказать «да»	56
Глава 4. Написание кода	67
Глава 5. Разработка через тестирование	87
Глава 6. Тренировка	95
Глава 7. Приемочное тестирование	105
Глава 8. Стратегии тестирования	124
Глава 9. Планирование	132
Глава 10. Оценки	145
Глава 11. Под давлением	159
Глава 12. Сотрудничество	166
Глава 13. Группы и проекты	175
Глава 14. Наставники, ученики и мастерство	180
Приложение. Инструментарий	193
Алфавитный указатель.	211

Содержание

Обязательное вступление 13

От издательства 18

Глава 1. Профессионализм 19

Оборотная сторона профессионализма 20

Ответственность 20

Первое правило: не навреди 23

 Не навреди функциональности 23

 Контроль качества не должен ничего обнаружить 24

 Вы должны быть уверены в том, что ваш код работает 25

 Автоматизированный контроль качества 26

 Не навреди структуре 26

Трудовая этика 28

 Знай свою область 29

 Непрерывное обучение 30

 Тренировка 31

 Совместная работа 32

 Наставничество 32

 Знание предметной области 33

 Понимание интересов работодателя/заказчика 33

 Скромность 33

Глава 2. Как сказать «нет» 35

Антагонистические роли 37

 Как насчет «почему»? 40

Высокие ставки 41

Умение работать в коллективе 42

 Не пытайтесь 44

 Пассивная агрессивность 46

Цена согласия 47

О невозможности хорошего кода 53

Глава 3. Как сказать «да» 56

Язык обещаний	58
Признаки пустых обещаний	59
Признаки серьезных обещаний	59
Выполнение обещания зависит от другого человека Х.	60
Вы не уверены в том, что обещание можно выполнить	61
Вы не справились	62
Резюме	62
Учимся говорить «да»	63
Обратная сторона «попытки»	63
Дисциплинированное принятие обязательств	64
Итоги	66

Глава 4. Написание кода 67

Готовность	68
Ночное программирование	70
Программирование в расстроенных чувствах	70
Зона потока	72
Музыка	73
Помехи	74
Творческий кризис	74
Творческий ввод	75
Отладка	76
Время отладки	79
Выбор темпа	79
Умейте остановиться	80
По дороге домой	80
Душ	80
Отставание от графика	81
Надежда	81
Спешка	81
Сверхурочные	82
Ложная готовность	83
Определение «готовности»	83
Помощь	84
Как помогать другим	84

Как принимать помощь	85
Обучение	86
Глава 5. Разработка через тестирование	87
Вердикт вынесен	89
Три закона TDD	89
Длинный перечень преимуществ	90
Уверенность	90
Снижение плотности дефектов	91
Смелость	91
Документация	92
Архитектура	93
Выбор профессионалов	94
Чем TDD не является	94
Глава 6. Тренировка	95
Азы тренировки	96
Двадцать два нуля	97
Длительность рабочего цикла	98
Додзё программирования	99
Ката	100
Вадза	101
Рандори	102
Расширение кругозора	103
Проекты с открытым кодом	103
Этика тренировки	103
Заключение	104
Глава 7. Приемочное тестирование	105
Передача требований	105
Преждевременная точность	107
Принцип неопределенности	107
Стремление к точности оценки	108
Поздняя неоднозначность	108
Приемочные тесты	110
Что такое «выполнено»?	110
Взаимодействие сторон	113

Автоматизация	113
Дополнительная работа	115
Кто и когда пишет приемочные тесты?	115
Роль разработчика	116
Обсуждение тестов и пассивно-агрессивная позиция	117
Приемочные тесты и модульные тесты	119
Графические интерфейсы и другие сложности	120
Выбор интерфейса для тестирования	121
Непрерывная интеграция	122
Стоп-сигнал	122
Заключение	122

Глава 8. Стратегии тестирования 124

Контроль качества не должен находить дефекты	125
Служба контроля качества — часть команды	125
Пирамида автоматизации тестирования	126
Модульные тесты	126
Компонентные тесты	127
Интеграционные тесты	128
Системные тесты	129
Исследовательские тесты	130
Заключение	130

Глава 9. Планирование 132

Встречи	133
Отказ от участия	134
Уход со встречи	134
Повестка дня и цель	135
Пятиминутка	135
Встречи планирования итераций	136
Ретроспективные встречи по итерациям и демонстрации	136
Споры и разногласия	137
Мана концентрации	138
Перезарядка	139
Физические упражнения	139
Ввод и вывод	140
Помидоры и распределение времени	140

Уклонение от работы	141
Инверсия приоритетов	142
Тупики	142
Грязь, болота и трясины	143
Заключение	144
Глава 10. Оценки	145
Что такое «оценка»?	147
Обязательства	148
Оценка	148
Подразумеваемые обязательства	150
PERT	151
Оценка времени выполнения	154
Широкополосный дельфийский метод	154
Метод быстрого голосования	155
Покер планирования	155
Аффинная оценка	156
Анализ по трем переменным	157
Закон больших чисел	157
Заклучение	157
Глава 11. Под давлением	159
Как избежать давления	161
Обязательства	161
Как сохранить чистоту	162
Дисциплина в кризисных ситуациях	163
Как вести себя в тяжелой ситуации	163
Без паники	163
Взаимодействие	164
Доверяйте своим методам	164
Помощь	164
Заклучение	165
Глава 12. Сотрудничество	166
Программисты и люди	168
Программисты и работодатели	168
Программисты и программисты	171

Принадлежность кода	171
Коллективная принадлежность кода	172
Парное программирование	172
Как работать мозжечком	173
Заключение	174
Глава 13. Группы и проекты	175
Формирование группы	176
«Притертая» группа	176
Созревание	177
Что сначала — группа или проект?	177
Но как управлять такой группой?	178
Дилемма владельца проекта	178
Заключение	179
Глава 14. Наставники, ученики и мастерство	180
Диплом для неподготовленных	181
Обучение	182
Digi-comp I, мой первый компьютер	182
ЕСР-18 в средней школе	183
Нетрадиционное обучение	186
Горький опыт	187
Ученичество	187
Период ученичества	189
Мастер	189
Ремесленник	190
Ученики/интерны	190
Реальность	191
Профессионализм	191
Как убедить людей	192
Заключение	192
Приложение. Инструментарий	193
Инструменты	195
Управление исходным кодом	195
«Корпоративные» системы управления исходным кодом	195

Пессимистическая и оптимистическая блокировка	196
CVS / SVN	197
IDE/редактор	200
Непрерывная сборка	203
Инструменты модульного тестирования	204
Инструменты компонентного тестирования	205
Определение	205
FitNesse	206
Другие инструменты	206
Инструменты интеграционного тестирования	207
UML/MDA	207
Детализация	208
Без изменений и надежд	209
Заключение	210
Алфавитный указатель.....	211

Обязательное вступление

(Не пропускайте, оно вам понадобится!)



Почему вы выбрали эту книгу? Наверное, потому что вы — программист, и вас интересует понятие профессионализма. И правильно! Профессионализм — то, чего так отчаянно не хватает в нашей профессии.

Я тоже программист. Я занимался программированием 42¹ года и за это время повидал многое. Меня увольняли. Меня превозносили до небес. Я побывал руководителем группы, начальником, рядовым работником и даже исполнительным директором. Я работал с выдающимися программистами, и я работал со слизнями². Я занимался разработкой как самых передовых встроенных программных/аппаратных систем, так и корпоративных систем начисления зарплаты. Я программировал на COBOL, FORTRAN, BAL, PDP-8, PDP-11, C, C++, Java, Ruby,

¹ Без паники!

² Технический термин неизвестного происхождения.

Smalltalk и на многих других языках. Я работал с бездарными халявщиками, и я работал с высококвалифицированными профессионалами. Именно последней классификации посвящена эта книга.

На ее страницах я попытаюсь определить, что же это такое — «быть профессиональным программистом». Я опишу те атрибуты и признаки, которые, на мой взгляд, присущи настоящим профессионалам.

Откуда я знаю, что это за атрибуты и признаки? Потому что я познал все это на собственном горьком опыте. Видите ли, когда я поступил на свое первое место работы на должность программиста, никому бы не пришло в голову назвать меня профессионалом.

Это было в 1969 году. Мне тогда было 17 лет. Мой отец убедил местную фирму под названием ASC нанять меня программистом на неполный рабочий день. (Да, мой отец это умеет. Однажды он на моих глазах встал на пути разгоняющейся машины, поднял руку и приказал: «Стоять!» Машина остановилась. Моему папе вообще трудно отказать.) Меня приняли на работу, посадили в комнату, где хранились все руководства к компьютерам IBM, и заставили записывать в них описания обновлений за несколько лет. Именно тогда я впервые увидел фразу: «Страница намеренно оставлена пустой».

Через пару дней обновления руководств мой начальник предложил мне написать простую программу на Easycoder¹. Его просьба вызвала у меня бурный энтузиазм, ведь до этого я еще не написал ни одной программы для настоящего компьютера. Впрочем, я бегло просмотрел несколько книг по Autocoder и примерно представлял, с чего следует начать.

Моя программа должна была прочесть записи с магнитной ленты и изменить идентификаторы этих записей. Значения новых идентификаторов начинались с 1 и увеличивались на 1 для каждой последующей записи. Записи с обновленными идентификаторами должны были записываться на новую ленту.

Начальник показал мне полку, на которой лежало множество стопок красных и синих перфокарт. Представьте, что вы купили 50 колод игральных карт — 25 красных и 25 синих, а потом положили эти колоды друг на друга. Так выглядели эти стопки. В них чередовались карты красного и синего цвета; каждая «колода», состоявшая примерно из 200 карт, содержала исходный код библиотеки подпрограмм. Программисты просто снимали верхнюю «колоду» со стопки (убедившись, что

¹ Ассемблер для компьютера Honeywell H200, аналог Autocoder для компьютера IBM 1401.

они взяли только красные или только синие карты) и клали ее в конец своей стопки перфокарт.

Моя программа была написана на программных формулярах — больших прямоугольных листах бумаги, разделенных на 25 строк и 80 столбцов. Каждая строка соответствовала одной карте. Программа записывалась на формуляре прописными буквами. В последних 6 столбцах каждой строки записывался ее номер. Номера обычно увеличивались с приращением 10, чтобы позднее в стопку можно было вставить новые карты.

Формуляры передавались операторам подготовки данных. В компании работало несколько десятков женщин, которые брали формуляры из большого ящика и «набивали» их на клавишных перфораторах. Эти машины были очень похожи на пишущие машинки, но они не печатали вводимые знаки на бумаге, а кодировали их, пробивая отверстия в перфокартах.

На следующий день операторы вернули мою программу по внутренней почте. Маленькая стопка перфокарт была завернута в формуляры и перетянута резинкой. Я искал на перфокартах ошибки набора. Вроде все нормально. Я положил библиотечную колоду в конец своей стопки программ и отнес ее наверх операторам.

Компьютеры были установлены в машинном зале за закрытыми дверями, в зале с регулируемым микроклиматом и фальшполом (для прокладки кабелей). Я постучал в дверь, суровый оператор забрал у меня колоду и положил ее в другой ящик. Моя программа будет запущена, когда до нее дойдет очередь.

На следующий день я получил свою колоду обратно. Она была завернута в листинг и перетянута резинкой. (Да, в те дни мы использовали *очень много* резинок!)

Я открыл листинг и увидел, что программа не прошла компиляцию. Сообщения об ошибках в листинге оказались слишком сложными для моего понимания, поэтому я отнес их своему начальнику. Он просмотрел листинг, что-то пробормотал про себя, сделал несколько пометок, взял колоду и приказал следовать за ним. Он прошел в операторскую, сел за свободный перфоратор, исправил все карты с ошибками и добавил еще пару карт. Он на скорую руку объяснил суть происходящего, но все промелькнуло в одно мгновение.

Он отнес новую колоду в машинный зал, постучал в дверь, сказал какие-то «волшебные слова» оператору, а затем прошел в компьютерный зал. Оператор установил магнитные ленты на накопители

и загрузил колоду, пока мы наблюдали. Завертелись ленты, затарахтел принтер — и на этом все кончилось. Программа заработала.

На следующий день начальник поблагодарил меня за помощь, и моя работа на этом завершилась. Очевидно, фирма ASC посчитала, что ей некогда нянчиться с 17-летними новичками.

Впрочем, моя связь с ASC на этом не завершилась. Через несколько месяцев я получил постоянную работу в вечернюю смену в ASC на обслуживании принтеров. Эти принтеры печатали всякую ерунду с образов, хранившихся на ленте. Я должен был своевременно заправлять принтеры бумагой, ставить ленты с образами, извлекать замятую бумагу и вообще следить за тем, чтобы машины нормально работали.

Все это происходило в 1970 году. Я не мог себе позволить учебу в колледже, да она меня, признаться, не особенно привлекала. Война во Вьетнаме еще не закончилась, и в студенческих городках было неспокойно. Я продолжал штудировать книги по COBOL, Fortran, PL/1, PDP-8 и ассемблеру для IBM 360. Я намеревался обойтись без учебы и как можно быстрее заняться реальным программированием.

Через год я достиг этой цели — меня повысили до штатного программиста в ASC. Я с двумя друзьями Ричардом и Тимом, которым тоже было по 19 лет, трудились вместе с тремя другими программистами над бухгалтерской системой реального времени для фирмы, занимающейся грузовыми перевозками. Мы работали на Varian 620i — простых мини-компьютерах, по архитектуре сходных с PDP-8, не считая того, что у них были 16-разрядные слова и два регистра. Программирование велось на ассемблере.

Мы написали каждую строку кода в этой системе. Да, без преувеличения каждую. Мы написали операционную систему, обработчики прерываний, драйверы ввода/вывода, файловую систему для дисков, систему подгрузки оверлеев и даже компоновщик с динамической переадресацией — не говоря уже о коде приложения. Мы написали все это за 8 месяцев, работая по 70–80 часов в неделю для соблюдения немислимо жестких сроков. Тогда я получал \$7200 в год.

Система была закончена в срок. А потом мы уволились.

Все произошло внезапно, и расставание не было дружеским. Дело в том, что после всей работы и успешной сдачи системы компания дала нам прибавку всего в 2%. Мы почувствовали себя обманутыми. Некоторые из нас нашли работу в другом месте и попросту подали заявление.

К сожалению, я избрал другой, далеко не лучший путь. Мы с приятелем вломились в кабинет директора и уволились вместе с изрядным

скандалом. Это доставило нам эмоциональное удовлетворение — примерно на день.

На следующий день я осознал, что у меня нет работы. Мне было 19 лет, я был безработным без диплома. Я прошел собеседования на нескольких вакансиях из области программирования, но они прошли неудачно. Следующие четыре месяца я проработал в мастерской по ремонту газонокосилок, принадлежащей моему сводному брату. К сожалению, ремонтника из меня не вышло, и в конце концов мне пришлось уйти. Я впал в депрессию.

Я засиживался до трех часов ночи, поедая пиццу и смотря старые фильмы ужасов по черно-белому телевизору моих родителей. Постепенно кошмары стали просачиваться с экрана в мою жизнь. Я валялся в постели до часа дня, потому что не хотел видеть очередной унылый день. Я поступил на курсы математического анализа в региональном колледже и провалил экзамен. Моя жизнь летела под откос.

Моя мать поговорила со мной и объяснила, что так жить нельзя и что я был идиотом, когда уволился, не найдя себе новую работу — да еще со скандалом и на пару с приятелем. Она сказала, что увольняться, не имея новой работы, вообще нельзя, и делать это следует спокойно, трезво и в одиночку. Она сказала, что мне следует позвонить старому начальнику и попроситься на старое место — по ее выражению, «проглотить обиду».

Девятнадцатилетние парни не склонны признавать свои ошибки, и я не был исключением. Тем не менее обстоятельства взяли верх над гордостью. В конечном итоге я позвонил своему начальнику. И ведь сработало! Он охотно принял меня на \$6800 в год, а я охотно принял его предложение.

Следующие полтора года я работал на старом месте, обращая внимание на каждую мелочь и стараясь стать как можно более ценным работником. Моей наградой стали повышения и прибавки. Все шло хорошо. Когда я ушел из этой компании, мы остались в хороших отношениях, а мне уже предложили лучшую работу.

Наверное, вы подумали, что я усвоил полученный урок и стал профессионалом? Ничего подобного. Это был лишь первый из многих уроков, которые мне еще предстояло усвоить. В дальнейшем меня уволили с одной работы за сорванный по безопасности график и чуть не уволили с другой за случайное разглашение конфиденциальной информации. Я брался за рискованные проекты и заваливал их, не обращаясь за помощью, которая, как я знал, была мне необходима. Я рьяно защищал

свои технические решения, даже если они противоречили потребностям заказчиков. Я принял на работу совершенно неквалифицированного человека, который стал тяжким бременем для моего нанимателя. И что хуже всего, из-за моих организационных ошибок уволили двух других людей.

Так что относитесь к этой книге как к каталогу моих заблуждений, исповеди в моих прегрешениях и сборнику советов, которые помогут вам избежать моих ошибок.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства <http://www.piter.com>.

1

Профессионализм



Смейся, Кертин, старина. Над нами сыграли отличную шутку — Господь Бог, природа или судьба, как тебе больше нравится. Но кто бы это ни был, у него наверняка есть чувство юмора! Ха!

Ховард, «Сокровище Сьерра-Мадре»

Итак, вы хотите стать профессиональным разработчиком? Ходить с гордо поднятой головой и объявить всему миру: «Я *профессионал!*» Хотите, чтобы люди смотрели на вас с уважением, а матери указывали на вас и говорили своим детям, что они должны вырасти такими же. Вы хотите всего этого, верно?

Оборотная сторона профессионализма

Термин «профессионализм» имеет много смысловых оттенков. Конечно, профессионализм — это своего рода почетный знак и повод для гордости, но также он является признаком ответственности. Понятно, что эти стороны профессионализма неразрывно связаны между собой: нельзя гордиться тем, за что вы не несете никакой ответственности.

Быть непрофессионалом намного проще. Непрофессионалы не несут ответственности за выполняемую работу — они оставляют ответственность своим работодателям. Если непрофессионал совершает ошибку, то мусор за ним прибирает работодатель. Но если ошибка совершается профессионалом, то устранять последствия приходится *ему самому*.

А если в ваш модуль закрадется ошибка, которая обойдется вашей компании в \$10 000? Непрофессионал пожмет плечами, скажет: «Всякое бывает», и продолжит писать следующий модуль. Профессионал должен выписать своей компании чек на \$10 000¹!

Да, когда речь идет о ваших личных деньгах, все выглядит немного иначе, верно? Но это ощущение присутствует у профессионалов постоянно. Более того, в нем заключается сущность профессионализма. Потому что профессионализм — это ответственное отношение к делу.

Ответственность

Вы прочитали введение, правда? Если не прочитали — вернитесь и прочитайте сейчас; оно задает контекст для всего остального материала.

Чтобы понять, почему так важно брать на себя ответственность, я на собственном опыте пережил последствия отказа от нее.

¹ Если, конечно, он правильно понимает профессиональную ответственность.

В 1979 году я работал на компанию Teradyne. Я был «ответственным инженером» за разработку программы, управляющей мини- и микрокомпьютерной системой для измерения качества телефонных линий. Центральный мини-компьютер подключался по выделенным или коммутируемым линиям на скорости 300 бод к десяткам периферийных микрокомпьютеров, управлявших измерительным оборудованием. Код был написан на ассемблере.

Нашими пользователями были администраторы по обслуживанию, работавшие в крупных телефонных компаниях. Каждый из них отвечал за 100 000 и более телефонных линий. Моя система помогала администраторам находить и исправлять неполадки и проблемы в телефонных линиях еще до того, как они будут замечены клиентами. Таким образом сокращалась частота жалоб клиентов — показатель, который измерялся комиссиями по предприятиям коммунального обслуживания и использовался для регулирования тарифов. Короче говоря, эти системы были невероятно важными.

Каждую ночь эти системы проводили «ночную проверку»: центральный мини-компьютер приказывал каждому из периферийных микрокомпьютеров протестировать все телефонные линии, находящиеся под его контролем. Каждое утро центральный компьютер получал список сбойных линий с характеристиками дефектов. По данным отчета администраторы по обслуживанию строили графики работы ремонтников, чтобы сбои исправлялись до поступления жалоб от клиентов.

Время от времени я рассылал нескольким десяткам заказчиков новую версию своей системы. «Рассылал» — самое правильное слово: я записывал программу на ленты и отправлял эти ленты своим клиентам. Клиенты загружали ленты, а затем перезапускали свои системы.

Очередная версия исправляла ряд незначительных дефектов и содержала новую функцию, которую требовали наши клиенты. Мы пообещали реализовать эту новую функцию к определенной дате. Я едва успел записать ленты в ночную смену, чтобы клиенты получили их к обещанной дате.

Через два дня мне позволил Том, наш менеджер эксплуатационного отдела. По его словам, несколько клиентов пожаловались на то, что «ночная проверка» не завершилась, и они не получили отчетов. У меня душа ушла в пятки — ведь чтобы вовремя выдать готовую версию программы, я не стал тестировать новый код. Я протестировал основную функциональность системы, но на тестирование проверки линий

потребовались бы много часов, а я должен был выдать программы. Ни одна из исправленных ошибок не содержалась в коде проверки, поэтому я чувствовал себя в безопасности.

Потеря ночного отчета была серьезным делом. Она означала, что у ремонтников было меньше работы, а позднее им придется отрабатывать упущенное. Также некоторые клиенты могли заметить дефект и пожаловаться. Потери данных за целую ночь было достаточно, чтобы менеджер по обслуживанию позвонил Тому и устроил ему разнос.

Я включил тестовую систему, загрузил новую программу и запустил проверку. Программа проработала несколько часов, а затем аварийно завершилась. Код не работал. Если бы я протестировал его до поставки, то данные не были бы потеряны, а менеджеры по обслуживанию не терзали бы Тома.

Я позвонил Тому и сообщил, что мне удалось воспроизвести проблему. Оказалось, что многие другие клиенты уже обращались к нему с той же проблемой. Затем он спросил, когда я смогу исправить ошибку. Я ответил, что пока не знаю, но работаю над ней, а пока клиенты могут вернуться к старой версии программы. Том рассердился — возврат стал бы двойным ударом для клиентов: они теряют данные за целую ночь и не могут использовать обещанную функцию.

Ошибку было трудно найти, а тестирование занимало несколько часов. Первое исправление не сработало. Второе — тоже. Мне понадобилось несколько попыток (а следовательно, дней), чтобы разобраться в происходящем. Все это время Том звонил мне через каждые несколько часов и спрашивал, когда будет исправлена ошибка. Он также передавал мне все, что ему говорили клиенты и как неудобно было предлагать им поставить старые ленты.

В конце концов я нашел дефект, отправил новые ленты, и все вошло в норму. Том (который не был моим начальником) остыл, и весь эпизод остался в прошлом. Но когда все было кончено, мой начальник пришел ко мне и сказал: «Это не должно повториться». Я согласился.

Поразмыслив, я понял, что отправка программы без тестирования кода проверки была безответственным поступком. Я пренебрег тестированием для того, чтобы сказать, что программа была отправлена вовремя. При этом я думал только о своей репутации, а не о клиенте и не о работодателе. А нужно было поступить ответственно: сообщить Тому, что тестирование не завершено и что я не готов сдать программу в назначенный срок. Было бы неприятно, Том расстроился бы, но клиенты не потеряли бы свои данные, и обошлось бы без звонков рассерженных клиентов.

Первое правило: не навреди

Итак, какие же принципы присущи ответственному поведению? Руководствоваться клятвой Гиппократата немного нескромно, но разве можно найти лучший источник? И в конце концов, это только логично — одаренный профессионал должен в первую очередь думать о том, чтобы его способности применялись только в добрых целях?

Какой вред может причинить разработчик? С чисто программной точки зрения он может навредить функциональности и структуре продукта. Давайте разберемся, как избежать именно такого ущерба.

Не навреди функциональности

Естественно, мы хотим, чтобы наши программы работали. Многие из нас стали программистами после того, как им удалось заставить работать свою первую программу, и это чувство хочется испытать снова. Но в работоспособности наших программ заинтересованы не только мы. Наши клиенты и работодатели хотят того же. Не забывайте — они платят нам за создание программ, которые делают именно то, что им нужно.

Функциональность программ страдает от ошибок. Следовательно, одним из признаков профессионализма должно быть написание программ с минимальным количеством ошибок.

«Но постойте! Ведь это нереально. Программный код слишком сложен, чтобы его можно было написать без ошибок».

Конечно, вы правы. Программный код слишком сложен, и ошибки будут всегда. К сожалению, это не избавляет вас от ответственности. Человеческое тело слишком сложно, чтобы изучить его от начала и до конца, но врачи все равно клянутся не причинять вреда. И если *ужони* не пытаются уйти от ответственности, то с какой стати это может быть позволено нам?

«Хотите сказать, что мы должны писать совершенный код?»

Нет, я хочу сказать, что вы должны отвечать за свое несовершенство. Тот факт, что в вашем коде заведомо будут присутствовать ошибки, не означает, что вы не несете за них ответственность. Написать идеальную программу практически невозможно, но за все недочеты несете ответственность именно вы, и никто другой.

Это верный признак настоящего профессионала — умение отвечать за свои ошибки, появление которых практически неизбежно. Итак, мой начинающий профессионал, прежде всего научитесь извиняться. Извинения необходимы, но недостаточны. Нельзя просто совершать одни и те же ошибки снова и снова. По мере вашего профессионального становления частота ошибок в вашем коде должна асимптотически стремиться к нулю. Она никогда не достигнет нуля, но вы ответственны за то, чтобы она была как можно ближе к нулю.

Контроль качества не должен ничего обнаружить

Когда вы передаете окончательную версию продукта в службу контроля качества, вы должны рассчитывать на то, что контроль не выявит никаких проблем. Было бы в высшей степени непрофессионально передавать на контроль качества заведомо дефектный код. А какой код является заведомо дефектным? Любой, в качестве которого вы не *уверены!*

Некоторые «специалисты» используют службу контроля качества для выявления ошибок. Они рассчитывают на то, что контроль качества обнаружит ошибки и вернет их список разработчикам. Некоторые компании даже выплачивают премии службе контроля качества за выявленные ошибки. Чем больше ошибок — тем больше премия.

Дело даже не в том, что это в высшей степени дорогостоящая практика, которая наносит ущерб компании и продукту. И не в том, что такое поведение срывает сроки и подрывает доверие к организации дела в группе разработки. И даже не в том, что это простое проявление лени и безответственности. Передавать на контроль качества код, работоспособность которого вы не можете гарантировать, непрофессионально. Такое поведение нарушает правило «не навреди».

Найдет ли служба контроля качества ошибки? Возможно, так что приготовьтесь извиняться, — а потом подумайте, почему эти ошибки ускользнули от вашего внимания, и сделайте что-нибудь для того, чтобы это не повторилось.

Когда служба контроля качества (или еще хуже — *пользователь*) обнаруживает ошибку, это должно вас удивить, огорчить и настроить на то, чтобы предотвратить повторение подобных событий в будущем.

Вы должны быть уверены в том, что ваш код работает

Как узнать, работает ли ваш код? Легко. Протестируйте его. Потом протестируйте еще раз. Протестируйте слева направо, потом справа налево. А теперь еще и сверху вниз!

Возможно, вас беспокоит, что столь тщательное тестирование кода отнимает слишком много времени. В конце концов, у вас есть графики и сроки, которые нужно соблюдать. Если тратить все время на тестирование, то когда писать код? Все верно! Поэтому тестирование следует автоматизировать. Напишите модульные тесты, которые можно выполнить в любой момент, и запускайте их как можно чаще.

Какая часть кода должна тестироваться этими автоматизированными модульными тестами? Мне действительно нужно отвечать на этот вопрос? Весь код! Весь. Без исключения.

Скажите, я предлагаю 100% тестовое покрытие кода? Ничего подобного. Я не *предлагаю*, а *требую*. Каждая написанная вами строка кода должна быть протестирована. Точка.

Может, это нереалистично? Почему? Вы пишете код, потому что ожидаете, что он будет выполняться. Если вы ожидаете, что код будет выполняться, то вы должны знать, что он работает. *А знать* это можно только в одном случае — по результатам тестирования.

Я являюсь основным автором и исполнителем проекта с открытым кодом FitNesse. На момент написания книги размер FitNesse достиг 60К строк, 26 из которых содержатся в 2000+ модульных тестах. По данным Еммы, покрытие этих 2000 тестов составляет около 90% кода. Почему не выше? Потому что Емма видит не все выполняемые строки! По моей оценке, степень покрытия намного выше. Составляет ли она 100%? Нет, 100% — асимптотический предел.

Но ведь некоторые части кода трудно тестировать? Да, но только потому, что этот код был так *спроектирован*. Значит, код нужно проектировать с расчетом на *простоту* тестирования. И для этого лучше всего написать тесты сначала — до того кода, который должен их пройти.

Этот принцип используется в методологии разработки через тестирование (TDD, Test Driven Development), которая будет более подробно описана в одной из следующих глав.

Автоматизированный контроль качества

Вся процедура контроля качества FitNesse заключается в выполнении модульных и приемных тестов. Если тесты проходят успешно, я выдаю продукт. При этом процедура контроля качества занимает около трех минут, и я могу выполнить ее в любой момент.

Конечно, из-за ошибки в FitNesse никто не умрет и никто не потеряет миллионы долларов. С другой стороны, у FitNesse много тысяч пользователей, а список дефектов очень невелик.

Безусловно, некоторые системы настолько критичны, что короткого автоматизированного теста недостаточно для определения их готовности к развертыванию. С другой стороны, вам как разработчику необходим относительно быстрый и надежный механизм проверки того, что написанный код работает и не мешает работе остальных частей системы. Итак, автоматизированные тесты по меньшей мере должны сообщить вам, что система с большой вероятностью пройдет контроль качества.

Не навреди структуре

Настоящий профессионал знает, что добавление функциональности в ущерб структуре — последнее дело. Структура кода обеспечивает его гибкость. Нарушая структуру, вы разрушаете будущее кода.

Все программные проекты базируются на фундаментальном предположении о простоте изменений. Если вы нарушаете это предположение, создавая негибкие структуры, то вы тем самым подрываете экономическую модель, заложенную в основу всей отрасли.

Внесение изменений не должно приводить к непомерным затратам.

К сожалению, слишком многие проекты вязнут в болоте плохой структуры. Задачи, которые когда-то решались за считанные дни, начинают занимать недели, а потом и месяцы. Руководство в отчаянных попытках наверстать потерянный темп нанимает дополнительных разработчиков для ускорения работы. Но эти разработчики только ухудшают ситуацию, углубляя структурные повреждения и создавая новые препятствия.

О принципах и паттернах проектирования, способствующих созданию гибких, удобных в сопровождении структур, написано много книг ¹.

¹ Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.

Профессиональные разработчики держат эти правила в памяти и стараются строить свои программные архитектуры по ним. Однако существует один нюанс, о котором часто забывают: *если вы хотите, чтобы ваш код был гибким, его необходимо проверять на гибкость!*

Как убедиться в том, что в ваш продукт легко вносятся изменения? Только одним способом — попытаться внести в него изменения! И если сделать это оказывается сложнее, чем предполагалось, то вы перерабатываете структуру кода, чтобы следующие изменения вносились проще.

Когда следует вносить такие изменения? *Всегда!* Каждый раз при работе с модулем следует понемногу совершенствовать его структуру. Каждое чтение кода должно приводить к доработке структуры.

Эта идеология иногда называется *безжалостным рефакторингом*. Я называю этот принцип «правилом бойскаута»: *всегда оставляйте модуль чище, чем до вашего прихода*. Всегда совершайте добрые дела в коде, когда вам представится такая возможность.

Такой подход полностью противоречит отношению некоторых людей к программному коду. Они считают, что серии частых изменений в рабочем коде опасны. Нет! Опасно оставлять код в статическом, неизменном состоянии. Если не проверять код на гибкость, то когда потребуется внести изменения, он может оказаться излишне жестким.

Почему многие разработчики боятся вносить частые изменения в свой код? Да потому что они боятся его «сломать»! А почему они этого боятся? Потому что у них нет тестов.

Мы снова возвращаемся к тестам. Если у вас имеется автоматизированный тестовый пакет, покрывающий почти 100% кода, и если этот пакет можно быстро выполнить в любой момент времени, то вы попросту не будете бояться изменять код. А как доказать, что вы не боитесь изменять код? Изменяйте его почаще.

Профессиональные разработчики настолько уверены в своем коде и тестах, что они с легкостью вносят случайные, спонтанные изменения. Они могут ни с того ни с сего переименовать класс. Заметив слишком длинный метод во время чтения модуля, они по ходу дела разбивают его на несколько меньших методов. Они преобразуют команду `switch` в полиморфную конструкцию или сворачивают иерархию наследования в линейную цепочку. Короче говоря, они относятся к коду так же, как скульптор относится к глине — они постоянно разминают его и придают новую форму.

Трудовая этика

За свою карьеру отвечаете вы сами. Ваш работодатель не обязан заботиться о вашей востребованности на рынке труда. Он не обязан обучать вас, отправлять вас на конференции или покупать книги. Всем этим должны заниматься *вы сами*. Горе тому разработчику, который доверит свою карьеру своему работодателю!

Некоторые работодатели согласны покупать вам книги, отправлять вас на семинары и конференции. Прекрасно, они оказывают вам услугу. Но никогда не думайте, что они обязаны это делать! Если ваш работодатель не делает этого за вас, подумайте, как сделать это своими силами.

Ваш работодатель также не обязан выделять вам время для учебы. Некоторые выделяют время на повышение квалификации — или даже требуют, чтобы вы это делали. Но и в этом случае они оказывают вам услугу, и вы должны быть им благодарны. Не рассчитывайте на это как на нечто само собой разумеющееся.

Вы обязаны своему работодателю некоторым количеством времени и усилий. Для примера возьмем стандартную для США 40-часовую рабочую неделю. Эти 40 часов должны быть проведены за решением проблем *вашего работодателя*, а не *ваших* личных проблем.

Запланируйте 60 рабочих часов в неделю. Первые 40 вы работаете на своего работодателя, а остальные 20 на себя. В эти 20 часов вы читаете книги, практикуетесь, учитесь и иным образом развиваете свою карьеру.

Наверняка вы подумали: «А как же моя семья? Моя личная жизнь? Я должен пожертвовать всем ради своего работодателя?»

Я не говорю обо всем вашем личном времени. Я говорю о 20 дополнительных часах в неделю. Если вы будете использовать обеденный перерыв для чтения и прослушивания подкастов и еще 90 минут в день на изучение нового языка — это решит все проблемы.

Давайте немного посчитаем. В неделе 168 часов. 40 достается вашему работодателю, еще 20 — вашей карьере. Остается 108. 56 тратится на сон, на все остальное остается 52. Возможно, вы не хотите брать на себя подобные обязательства. И это вполне нормально, но тогда не считайте себя профессионалом. Профессионалы не жалеют времени на совершенствование в своей профессии.

Возможно, вы считаете, что работа должна оставаться на рабочем месте и ее не следует брать домой. Согласен! В эти 20 часов вы должны работать не на своего работодателя, а на свою карьеру.

Иногда эти два направления совпадают. Иногда работа, выполняемая для работодателя, оказывается исключительно полезной для вашей карьеры. В таком случае потратить на нее некоторые из этих 20 часов будет вполне разумно. Но помните: эти 20 часов предназначены для вас. Они используются для того, чтобы повисить вашу профессиональную ценность.

Может показаться, что мой путь ведет к «перегоранию» на работе. Напротив, он помогает избежать этой печальной участи. Вероятно, вы стали разработчиком из-за своего энтузиазма к программированию, а ваше желание стать профессионалом обусловлено этим энтузиазмом. За эти 20 часов вы будете заниматься тем, что подкрепит ваш энтузиазм. Эти 20 часов должны быть *интересными*!

Знай свою область

Вы знаете, что такое диаграмма Насси–Шнейдермана? Если не знаете — почему? А чем отличаются конечные автоматы Мили и Мура? Должны знать. Сможете написать процедуру быстрой сортировки, не обращаясь к описанию алгоритма? Выполнить функциональную декомпозицию диаграммы информационного потока? Что означает термин «бесхозные данные»? Для чего нужны «таблицы Парнаса»?

За последние 50 лет в нашей области появилось множество новых идей, дисциплин, методов, инструментов и терминов. Сколько из них вы знаете? Каждый, кто хочет стать профессионалом, обязан знать заметную часть и постоянно увеличивать размер этой части.

Почему необходимо знать все это? Разве наша область не прогрессирует так быстро, что старые идеи теряют актуальность? Первая часть вопроса вполне очевидна: безусловно, в нашей области происходит стремительный прогресс. Но интересно заметить, что этот прогресс во многих отношениях имеет периферийную природу. Действительно, нам уже не приходится по 24 часа дожидаться завершения компиляции. И действительно, мы пишем системы, размер которых измеряется гигабайтами. Правда и то, что мы работаем в глобальной сети, предоставляющей мгновенный доступ к информации. Но с другой стороны, мы пишем те же команды `if` и `while`, что и 50 лет назад. Многое изменилось. Многое осталось неизменным.

Вторая часть вопроса так же очевидно неверна. Лишь очень немногие идеи последних 50 лет потеряли актуальность. Некоторые ушли на второй план, это правда. Концепция каскадной разработки, скажем, явно перестала пользоваться популярностью. Однако это не означает, что мы не должны знать, что это за концепция, каковы ее сильные и слабые стороны.

В целом подавляющее большинство с трудом завоеванных идей последних 50 лет ничуть не утратило своей ценности. А может, эти идеи стали еще более ценными. Вспомните проклятие Сантаяны: «Не помнящие прошлого обречены на его повторение».

Далее приводится *минимальный* список тем, в которых должен разбираться каждый разработчик.

- Паттерны проектирования. Вы должны быть способны описать все 24 паттерна из книги «Банды Четырех» и иметь практическое представление о многих паттернах из книг «Pattern-Oriented Software Architecture».
- Принципы проектирования. Вы должны знать принципы SOLID и хорошо разбираться в принципах компонентного проектирования.
- Методы. Вы должны понимать суть методологий XP, Scrum, экономной¹ разработки (Lean), Kanban, каскадной разработки, структурного анализа и структурного проектирования.
- Дисциплины. Практикуйтесь в практическом применении разработки через тестирование (TDD), объектно-ориентированного проектирования, структурного программирования, непрерывной интеграции и парного программирования.
- Артефакты. Вы должны уметь работать с UML, DFD, структурными диаграммами, сетями Петри, диаграммами переходов, блок-схемами и таблицами решений.

Непрерывное обучение

Неистовый темп изменений в нашей отрасли означает, что разработчики должны постоянно изучать большой объем материала только для того, чтобы оставаться в курсе дела. Горе проектировщикам, которые перестают программировать — они быстро оказываются не у дел. Горе

¹ Также называемой «бережливой». — *Примеч. перев.*

программистам, которые перестают изучать новые языки — им придется смотреть, как отрасль проходит мимо них. Горе разработчикам, которые не изучают новые дисциплины и методологии — их ожидает упадок на фоне процветания коллег.

Пойдете ли вы к врачу, который не знает, что сейчас происходит в медицине и не читает медицинские журналы? Обратитесь ли вы к консультанту по налогам, который не следит за налоговым законодательством и прецедентами? Так зачем работодателю нанимать разработчика, который не стремится быть в курсе дел?

Читайте книги, статьи, блоги, твиты. Посещайте конференции и собрания пользовательских групп. Участвуйте в работе исследовательских групп. Изучайте то, что лежит за пределами вашей привычной зоны. Если вы программист .NET — изучайте Java. Если вы программируете на Java — изучайте Ruby. Если вы программируете на C — изучайте Lisp. А если вам захочется серьезно поработать мозгами, изучайте Prolog и Forth!

Тренировка

Профессионалы тренируются. Настоящие профессионалы прилежно работают над тем, чтобы их навыки были постоянно отточены и готовы к применению. Недостаточно выполнять свою повседневную работу и называть ее тренировкой. Повседневная работа — это исполнение обязанностей, а не тренировка. Тренировка начинается тогда, когда вы целенаправленно применяете свои навыки за пределами своих рабочих обязанностей с единственной целью совершенствования этих навыков.

Что может означать тренировка для разработчика? На первый взгляд сама концепция выглядит абсурдно. Но давайте ненадолго задержимся и подумаем. Как музыканты совершенствуют свое мастерство? Не на концертах, а во время занятий. Как они это делают? Среди прочего, у них имеются специальные упражнения, гаммы и этюды. Музыканты повторяют их снова и снова, чтобы тренировать свои пальцы и ум и чтобы поддерживать свое мастерство на должном уровне.

Как тренируются разработчики? В этой книге целая глава посвящена разным методам тренировки, поэтому я не стану углубляться в подробности сейчас. Например, я часто применяю метод повторения простых упражнений вроде «игры в боулинг» или «разложения на простые

множители». Я называю эти упражнения *ката*. Существует много разных ката, из которых можно выбрать то, что лучше подойдет вам.

Ката обычно имеет вид простой задачи по программированию — например, написать функцию, которая раскладывает целое число на простые множители. Целью выполнения ката является не поиск решения; вы уже знаете, как решается задача. Ката тренируют ваши пальцы и ваш мозг.

Я ежедневно выполняю одну-две ката, часто в процессе погружения в работу. Я пишу их на Java, Ruby, Clojure или на каком-нибудь другом языке, который я хочу поддерживать в рабочем состоянии. Я использую ката для тренировки конкретных навыков (например, приучая пальцы к использованию клавиш ускоренного доступа) или приемов рефакторинга.

Относитесь к ката как к 10-минутной разминке по утрам и 10-минутной релаксации по вечерам.

Совместная работа

Второй лучший способ чему-то научиться — совместная работа с другими людьми. Профессиональные разработчики намеренно стараются вместе программировать, вместе тренироваться, вместе проектировать и планировать. При этом они много узнают друг от друга, выполняют свою работу быстрее и с меньшим количеством ошибок.

Это не означает, что вы должны проводить 100% своего рабочего времени за совместной работой. Одиночная работа тоже очень важна. Как бы я ни любил программировать в паре, мне абсолютно необходимо время от времени поработать одному.

Наставничество

Самый лучший способ чему-то научиться — учить других. Факты запоминаются быстрее всего тогда, когда вы должны их сообщить другим людям, за которых вы отвечаете. Таким образом, основную пользу от преподавания получает прежде всего преподаватель.

Аналогичным образом лучший способ ввести новых людей в организацию — посидеть с ними и объяснить, что и как работает. Профессионалы берут на себя персональную ответственность за обучение новичков. Они не бросают новичков, предоставляя им самим решать свои проблемы.

Знание предметной области

Каждый профессионал обязан понимать предметную область программируемых им решений. Если вы пишете бухгалтерскую систему, вы должны разбираться в бухгалтерии. Если вы пишете приложение для туристической фирмы, вы должны разбираться в туризме. Быть экспертом не обязательно, но к изучению темы необходимо относиться ответственно. Начиная проект в новой для себя области, прочитайте одну-две книги по теме. Проведите собеседование с клиентом и пользователями об основах предметной области. Поговорите с экспертами, постарайтесь понять их принципы и ценности.

Худшая разновидность непрофессионализма — просто программировать по спецификации, не понимая того, почему эта спецификация подходит для решения своей задачи. Вы должны обладать достаточными познаниями в предметной области для того, чтобы распознать и исправить возможные ошибки в спецификации.

Понимание интересов работодателя/заказчика

Проблемы вашего работодателя — это *ваши* проблемы. Вы должны понимать их и постараться найти лучшие решения. В ходе разработки системы представьте себя на месте своего работодателя и убедитесь в том, что разрабатываемые вами возможности действительно соответствуют его потребностям.

Разработчику легко представить себя на месте другого разработчика. Но когда речь идет об отношении к работодателю, многие попадают в ловушку представлений «мы и они». Профессионалы всеми силами стремятся избежать этого.

Скромность

Программирование — творческая деятельность. Во время написания кода мы творим нечто из ничего. Мы решительно наводим порядок в хаосе. Мы уверенно определяем поведение машины, которая в случае ошибки могла бы причинить невообразимый ущерб. Таким образом, программирование является актом исключительно амбициозным.

Профессионалам не свойственно ложное смирение. Профессионал знает свою работу и гордится ей. Профессионал уверен в своих

способностях и сознательно идет на риск, основываясь на этой уверенности. Профессионал не может быть робким.

Однако профессионал также знает, что в отдельных случаях он потерпит неудачу, его оценки риска окажутся неверными, а его способности — недостаточными; он посмотрит в зеркало и увидит, что оттуда ему улыбается самонадеянный болван.

Итак, оказавшись мишенью для насмешки, профессионал смеется первым. Он сам никогда не высмеивает других, но принимает заслуженные насмешки и легко отмахивается от незаслуженных. Он не издевается над коллегами, допустившими ошибку, потому что знает — следующим может быть он сам.

Профессионал понимает свою гордыню, а также то, что судьба рано или поздно заметит и найдет его слабые места. И когда ее выстрел попадет в цель, остается только следовать совету: смейтесь.

2

Как сказать «нет»



Делай. Или не делай. Не надо пытаться.

Йода

В начале 1970-х годов мы с двумя 19-летними друзьями работали над бухгалтерской системой реального времени для профсоюза грузоперевозчиков в Чикаго для компании ASC. Если у вас в памяти всплывают такие имена, как Джимми Хоффа¹, — так и должно быть. В 1971 году профсоюз грузоперевозчиков был серьезной организацией.

Наша система должна была пойти в эксплуатацию к определенной дате. На эту дату были поставлены большие деньги. Наша группа работала по 60, 70 и 80 часов в неделю, чтобы уложиться в срок.

За неделю до даты запуска система наконец-то была собрана. В ней было множество ошибок и нерешенных проблем, и мы лихорадочно разбирались с ними по списку. Нам не хватало времени на то, чтобы

¹ Американский профсоюзный лидер, исчезнувший при загадочных обстоятельствах. — *Примеч. пер.*

есть и спать, не говоря уже о том, чтобы думать. Нашим начальником в ASC был Фрэнк, отставной полковник ВВС. Это был один из тех громогласных напористых руководителей, которые предлагают выбрать: либо ты *немедленно* прыгаешь с парашютом с высоты 3000 м либо делаешь то же самое, но уже без парашюта. Мы, 19-летние парни, его терпеть не могли.

Фрэнк сказал, что все должно быть сделано к установленной дате. И говорить больше не о чем. Когда придет срок, наша система должна быть готова. Точка. И никаких «но».

Мой босс Билл был симпатичным парнем. Он проработал с Фрэнком немало лет и хорошо понимал, что возможно, а что нет. Он сказал нам, что система должна заработать к установленной дате, что бы ни произошло.

И система заработала к установленной дате. И это кончилось оглушительным провалом.

Штаб-квартира профсоюза грузоперевозчиков связывалась с нашим компьютером, находящимся за 30 миль к северу, через дюжину полудуплексных терминалов со скоростью 300 бод. Каждый терминал зависал примерно через каждые полчаса. Мы сталкивались с этой проблемой и прежде, но у нас не было времени на моделирование трафика, который операторы ввода данных создадут для нашей системы.

Ситуация усугублялась тем, что отрывные листки печатались на теле-тайпах ASR35, которые тоже подключались к нашей системе по 110-бодным телефонным линиям — и тоже зависали на середине печати.

Проблема решалась перезагрузкой. Итак, все операторы, терминалы которых еще работали, должны были завершить свою текущую работу. Когда вся работа прекращалась, они звонили нам, и мы перезагружали компьютер. Операторам зависших терминалов приходилось делать все заново. И это происходило чаще одного раза в час.

Через полдня руководитель отделения профсоюза приказал нам отключить систему и не включать ее, пока она не заработает. В итоге они потеряли половину рабочего дня, а все данные пришлось вводить заново в старой системе.

Вопли и рев Фрэнка разносились по всему зданию. Так продолжалось долго, очень долго. Затем Билл и наш системный аналитик Джалиль зашли к нам и спросили, сколько времени понадобится на обеспечение стабильной работы системы. Я сказал: «Четыре недели».

На их лицах отразился ужас, затем решимость. «Нет, — сказали они, — система должна заработать к пятнице».

Я сказал: «Послушайте, система едва заработала на прошлой неделе. Нам нужно разобраться с множеством проблем. На это понадобится четыре недели».

Но Билл и Джалиль были непреклонны: «Нет, это должно быть в пятницу. Можно хотя бы попробовать?»

Пятница была выбрана удачно — нагрузка в конце недели была намного ниже. Нам удалось найти много дефектов и исправить их до наступления понедельника. Но даже после этого вся система едва стояла, словно картонный домик. Проблемы с зависаниями по-прежнему происходили один-два раза в день. Также были и другие проблемы. Через несколько недель система была доведена до состояния, в котором жалобы прекратились, и вроде бы стала возможна нормальная жизнь.

И тогда, как я рассказывал во введении, мы все уволились. И фирма столкнулась с настоящим кризисом: ей пришлось нанимать новых программистов, чтобы разобраться с потоком жалоб от клиентов.

Кто виноват в этом фиаско? Разумеется, проблема отчасти обусловлена стилем руководства Фрэнка. Его тактика запугивания мешала ему услышать правду. Понятно, что Билл и Джалиль должны были противостоять давлению Фрэнка намного активнее. И само собой, наш руководитель группы не должен был соглашаться на требование выдать продукт к пятнице. Да и мне следовало продолжать говорить «нет» вместо того, чтобы повторять за руководителем группы.

Профессионалы говорят правду облеченным властью. У них достаточно смелости, чтобы сказать «нет» своим начальникам.

Как сказать «нет» начальнику? Ведь это ваш *начальник*! Разве вы не обязаны делать то, что говорит начальник?

Нет! Говорите «нет», если вы профессионал.

Рабам запрещается говорить «нет». Наемные работники неохотно говорят «нет». Но профессионалу *положено* говорить «нет». Более того, хорошим руководителям очень нужны люди, у которых хватает смелости сказать «нет». Только так можно действительно чего-то добиться.

Антагонистические роли

Одному из рецензентов книги эта глава очень не понравилась. Он сказал, что из-за нее он едва не отложил книгу. Ему доводилось создавать

группы, в которых не было антагонистических отношений; группы работали вместе в гармонии и без конфронтации. Я рад за этого рецензента, но не уверен в том, что его группы действительно были избавлены от конфронтации настолько, насколько ему кажется. А если так — не уверен, что они были в полной мере эффективными. По собственному опыту скажу, что трудные решения лучше принимать на основании конфронтации антагонистических ролей.

Руководители — люди, которые должны исполнять свои обязанности, и большинство руководителей знает, как выполнять свою работу на должном уровне. Часть этой работы заключается в том, чтобы как можно более жестко преследовать и защищать свои цели.

Программисты — тоже люди, которые должны исполнять свои обязанности, и большинство из них знает, как выполнять свою работу на должном уровне. И если они относятся к числу настоящих профессионалов, они будут как можно более жестко преследовать и защищать *свои* цели.

Когда ваш руководитель говорит вам, что страница входа в систему должна быть готова к завтрашнему дню, он преследует и защищает одну из своих целей. Он выполняет свою работу. Если вы хорошо знаете, что сделать страницу к завтрашнему дню невозможно, то отвечая: «Хорошо, я попытаюсь», вы не выполняете свою работу. Выполнить ее в этот момент можно только одним способом: сказать: «Нет, это невозможно».

Но разве вы не должны выполнять распоряжения начальства? Нет, ваш начальник рассчитывает на то, что вы будете защищать свои цели так же жестко, как он защищает свои. Таким образом вы вдвоем приходите к *оптимальному результату*.

Оптимальным результатом является цель, общая для вас и вашего руководителя. Фокус в том, чтобы найти эту цель, а для этого обычно необходимы переговоры.

Переговоры могут быть приятными.

Майк: «Пола, страница входа в систему мне нужна к завтрашнему дню».

Пола: «Ого! Уже завтра? Хорошо я попробую».

Майк: «Отлично, спасибо!»

Приятный разговор, никакой конфронтации. Обе стороны расстались с улыбками. Очень мило.

Но при этом обе стороны вели себя непрофессионально. Пола отлично знает, что работа над страницей займет больше одного дня, поэтому она просто врёт. Возможно, она не считает свои слова враньем. Возможно, она действительно хочет попробовать и надеется, что ей каким-нибудь чудом удастся исполнить свое обещание. Но в конечном итоге она все равно врёт.

С другой стороны, Майк принял ее «Я попробую» за «Да». И это просто глупо: он должен знать, что Пола попытается избежать конфронтации, поэтому он должен был проявить настойчивость и спросить: «Мне кажется или ты сомневаешься? Уверена, что ты сможешь сделать страницу к завтрашнему дню?»

Или еще одна приятная беседа.

Майк: «Пола, страница входа в систему мне нужна к завтрашнему дню».

Пола: «Прости, Майк, но мне понадобится больше времени».

Майк: «И как ты думаешь, когда она будет готова?»

Пола: «Как насчет двух недель — нормально?»

Майк: (записывает что-то в ежедневнике) «Хорошо, спасибо».

Приятный, но ужасно неэффективный и совершенно непрофессиональный разговор. Обе стороны потерпели неудачу в своем поиске оптимального результата. Вместо того чтобы спрашивать, устроят Майка две недели или нет, Пола должна была высказаться утвердительно: «У меня это займет две недели, Майк».

С другой стороны, Майк принял предложенный срок без вопросов, словно его личные цели не имеют никакого значения. Интересно, не собирается ли он просто сообщить своему начальнику, что демонстрацию программы заказчику придется отложить из-за Полы? Такое пассивно-агрессивное поведение морально предосудительно.

В обоих случаях ни одна из сторон не преследовала общей цели. Ни одна из сторон не пыталась найти оптимальный результат. Давайте посмотрим, как это делается.

Майк: «Пола, страница входа в систему мне нужна к завтрашнему дню».

Пола: «Нет, Майк, здесь работы на две недели».

Майк: «Две недели? По оценкам проектировщиков, работа должна была занять три дня, а прошло уже пять!»

Пола: «Проектировщики ошибались, Майк. Они выдали свою оценку до того, как служба маркетинга сформулировала окончательные требования. У меня осталось работы еще на 10 дней. Ты не видел мои обновленные оценки в вики?»

Майк: (с суровым видом и недовольным голосом) «Это недопустимо, Пола. Завтра я буду представлять клиентам демо-версию, и я должен им показать, что страница входа работает».

Пола: «Какая часть страницы входа должна работать к завтрашнему дню?»

Майк: «Мне нужна страница входа! Я должен иметь возможность войти в систему».

Пола: «Майк, я могу сделать макет страницы входа, который позволит войти в систему. Сейчас простейший вариант уже работает. Макет не проверяет имя пользователя и пароль и не отправляет забытый пароль по электронной почте. У верхнего края нет баннера с фирменным логотипом, не работает кнопка справки и всплывающая подсказка. Страница не сохраняет cookie, чтобы запомнить данные для следующего входа, и не устанавливает ограничений доступа. Но войти в систему вы сможете. Подойдет?»

Майк: «Значит, вход будет работать?»

Пола: «Да, вход будет работать».

Майк: «Отлично, Пола, ты меня спасла!» (отходит с довольным видом)

Стороны пришли к оптимальному результату. Для этого они сказали «нет», а потом выработали взаимоприемлемое решение. Они действовали как профессионалы. В разговоре присутствовал элемент конфронтации и в нем было несколько неудобных моментов, но это неизбежно, когда два человека настойчиво преследуют несовпадающие цели.

Как насчет «почему»?

Возможно, вы думаете, что Поле следовало более подробно объяснить, *почему* работа над страницей заняла намного больше времени. По собственному опыту могу сказать, что *причины* намного менее важны, чем *факт*. А факт заключается в том, что на страницу понадобится две недели. Почему две недели — это уже второстепенно.

Впрочем, объяснение может помочь Майку понять (а следовательно, и принять) этот факт. И если у Майка имеется техническая

квалификация и темперамент для понимания, такие объяснения могут быть полезными. С другой стороны, Майк может не согласиться с выводами. Возможно, он решит, что Пола делает все неправильно. Он может сказать, что ей не нужен такой объем тестирования или рецензирования или что этап 12 можно исключить из описания. Изобилие подробностей может обернуться мелочным регламентированием.

Высокие ставки

Говорить «нет» важнее всего тогда, когда ставки высоки. Чем выше ставки, тем больше ценность сказанного «нет».

Казалось бы, утверждение очевидное. Если риск настолько велик, что от успеха зависит выживание компании, вы должны без малейших колебаний предоставить руководству самую точную информацию. А это часто означает «нет».

Дон (начальник по управлению разработкой): «Итак, по нашим текущим прогнозам проект «Золотой Гусь» будет завершен через 12 недель от сегодняшнего дня, с погрешностью плюс/минус 5 недель».

Чарльз (исполнительный директор): (четверть минуты сидит молча, постепенно багровея) «То есть ты хочешь сказать, что мы опаздываем на 17 недель?»

Дон: «Да, это возможно».

Чарльз: (встает, Дон встает на секунду позже) «Черт возьми, Дон! Все должно было быть готово три недели назад! Заказчик из „Галитрона“ звонит каждый день и спрашивает, где его чертова система. И я должен сказать, что им придется подождать еще четыре месяца? Предложи что-нибудь получше».

Дон: «Чак, я тебе говорил три месяца назад после реорганизации, что нам понадобится еще четыре месяца. Я хочу сказать, ты сократил мой штат на 20%! Ты тогда сообщил „Галитрону“, что мы задержим сдачу продукта?»

Чарльз: «Ты отлично знаешь, что не сообщил. Мы не можем себе позволить потерять этот заказ, Дон. (Чарльз делает паузу, бледнеет.) Без „Галитрона“ нам крышка. Ты это знаешь, верно? А теперь после этой задержки я боюсь... Что я скажу совету директоров? (Снова садится в кресло, пытаюсь сохранить самообладание.) Дон, ты должен что-то придумать».

Дон: «Я ничего не могу сделать, Чак. Мы это уже обсуждали. „Галитрон“ не собирается сокращать требования и не соглашается на промежуточные версии. Они хотят, чтобы установка проводилась только один раз и на этом все заканчивалось. Я просто не смогу сделать это быстрее. Ничего не выйдет».

Чарльз: «Черт побери. Даже если я скажу, что от этого зависит твоя работа?»

Дон: «Если меня уволить, оценка от этого не изменится, Чарльз».

Чарльз: «Все, разговор закончен. Возвращайся к своей группе и следи за тем, чтобы проект двигался. А мне нужно сделать несколько очень неприятных звонков».

Конечно, Чарльз должен был все сказать «Галитрону» еще три месяца назад, когда он впервые услышал новый прогноз. По крайней мере сейчас он поступает правильно, сообщая информацию заказчику (и совету директоров). Но если бы Дон не настоял на своем, эти звонки могли бы быть отложены на еще более поздний срок.

Умение работать в коллективе

Все мы слышали, как важно «уметь работать в коллективе». Это означает, что вы выполняете свои функции настолько хорошо, насколько возможно, и помогаете своим коллегам, если они окажутся в беде. Человек, умеющий работать в коллективе, часто общается с другими, обращает внимание на своих коллег и добросовестно исполняет свои обязанности.

Умение работать в коллективе вовсе не означает, что вы должны со всеми соглашаться. Рассмотрим следующую ситуацию.

Пола: «Майк, у меня свежие прогнозы. Группа согласна с тем, что демо-версия будет готова через восемь недель плюс/минус одна неделя».

Майк: «Пола, мы уже запланировали сдачу демо-версии, это будет через шесть недель».

Пола: «Даже не спросив нашего мнения? Майк, ты не можешь взвалить это на нас».

Майк: «Это уже сделано».

Пола: (вздыхает) «Ладно, я вернусь в группу и посмотрю, что мы сможем выдать через шесть недель, но это будет не вся система. Некоторые функции будут отсутствовать, а загрузка данных будет неполной».

Майк: «Пола, заказчик хочет увидеть полную демо-версию».

Пола: «Этого не будет, Майк».

Майк: «Черт. Ладно, напиши описание того, что вы сможете сделать, и передай мне завтра утром».

Пола: «Хорошо, сделаю».

Майк: «Нельзя ли как-нибудь ускорить работу? Может, работать более эффективно, более творчески?»

Пола: «Куда уж эффективнее, Майк. Мы хорошо знаем задачу, и на ее реализацию нам понадобится восемь или девять недель, но не шесть».

Майк: «Можно работать сверхурочно».

Пола: «От этого все только замедлится. Помнишь, что вышло в последний раз, когда мы выгоняли народ на сверхурочные?»

Майк: «Да, но на этот раз этого не случится».

Пола: «Все будет точно так же, Майк. Поверь мне. Нам нужно восемь или девять недель, а не шесть».

Майк: «Хорошо, напиши план, но постоянно думай о том, как справиться за шесть недель. Наверняка что-нибудь можно придумать».

Пола: «Нет, Майк, нельзя. Я могу написать план на шесть недель, но в нем не будет многих важных функций и данных. Только так, и никак иначе».

Майк: «Договорились, но я уверен, что вы сможете сотворить чудо, если попытаетесь».

(Пола уходит, качая головой.)

Позднее, на собрании у директора...

Дон: «Итак, Майк, заказчик рассчитывает получить демо-версию через шесть недель. И надеется, что все будет работать».

Майк: «Да, все будет готово. Моя группа просиживает за компьютером дни и ночи, и мы справимся. Возможно, придется поработать сверхурочно и проявить творческий подход, но мы сделаем!»

Дон: «Как хорошо, что вы понимаете интересы коллектива!»

Кто же *на самом деле* «понимает интересы коллектива» в этой ситуации? Пола действует в общих интересах, потому что она по мере возможностей сообщает, что сделать можно, а что нельзя. Она жестко

защищает свою позицию, несмотря на все уговоры и нытье Майка. Майк действует в интересах коллектива из одного человека — самого Майка. Очевидно, он не думает об интересах Пола, потому что он только что пообещал за нее сделать то, что она (как она сама сказала открытым текстом) сделать не сможет. Он явно и не думал и об интересах Дона (хотя он бы с этим не согласился), потому что солгал ему.

Почему же Майк так поступил? Он хотел, чтобы Дон видел в нем человека, понимающего интересы коллектива, а также верил в свою способность манипулировать Полой для достижения шестинедельного срока. Не нужно считать Майка злым и испорченным; он просто слишком уверен в том, что ему удастся заставить других людей делать то, что он хочет.

Не пытайтесь

Худшее, что может сделать Пола в ответ на манипуляции Майка, — сказать: «Хорошо, я попытаюсь». Не хочу приплетать сюда Йоду, но в данном случае он прав. *Не надо пытаться.*

Вам не нравится эта мысль? Может, вы думаете, что пытаться что-то сделать полезно? Ведь Колумб не открыл бы Америку, если бы не пытался?

Слово «попытаться» имеет много определений. В данном случае я имею в виду значение «приложить дополнительные усилия». Какие дополнительные усилия может приложить Пола, чтобы демо-версия была готова к сроку? Если это возможно, получается, что ее группа ранее работала не в полную силу.

Обещая «попытаться», вы признаетесь в том, что ранее вы сдерживались; что у вас остался дополнительный резерв, которым вы можете воспользоваться. Вы признаетесь в том, что цель может быть достигнута посредством приложения дополнительных усилий; более того, вы фактически обязуетесь применить эти дополнительные усилия для достижения цели. Следовательно, обещая попытаться, вы обязуетесь добиться успеха. Тем самым вы взваливаете на себя тяжелое бремя. Если «попытка» не приведет к желаемому результату, это рассматривается как провал.

У вас есть дополнительный источник энергии, который вы еще не пустили в ход? И если вы задействуете его, сможете ли вы достичь поставленной цели? Или вы просто создаете условия для своего будущего провала?

Обещая попытаться, вы обещаете изменить свои планы. Прежних планов оказалось недостаточно. Обещая попытаться, вы говорите, что у вас есть новый план. Что это за план? Какие изменения вы внесете в свое поведение? Что вы собираетесь сделать иначе сейчас, когда вы «пытаетесь»?

Если у вас нет другого плана, если вы не измените свое поведение, если все будет идти точно так же, как до вашего обещания, то что тогда означает ваше «попытаюсь»?

Если вы не придерживаете часть энергии в резерве, если у вас нет нового плана, если вы не намерены изменять свое поведение и если вы достаточно уверены в исходной оценке, то ваше обещание «попытаться» в корне непорядочно. Вы *врите*. И по всей видимости, вы делаете это для того, чтобы сохранить лицо и избежать конфронтации.

Подход Пола был куда более правильным. Она продолжала напоминать Майку, что исходная оценка была неопределенной. Она всегда говорила «восемь или девять недель». Она подчеркнула существующую неопределенность и ни разу не отступила. Она ни разу не утверждала, что какие-то дополнительные усилия, или новый план, или изменение поведения смогут уменьшить эту неопределенность.

Три недели спустя...

Майк: «Пола, через три недели сдаем демо-версию. Заказчики хотят посмотреть, как работает отправка файлов».

Пола: «Майк, это не входит в согласованный список функций».

Майк: «Я знаю, но они требуют».

Пола: «Хорошо, тогда из демо-версии придется выкинуть единый вход или резервное копирование».

Майк: «Ни в коем случае! Они хотят видеть и эти функции!»

Пола: «Значит, они хотят полностью реализованную функциональность. Ты это хочешь сказать? Я же говорила, что это невозможно».

Майк: «Прости, Пола, но заказчик не уступает. Они хотят увидеть все сразу».

Пола: «Этого не будет, Майк. Просто не будет».

Майк: «Да ладно, Пола, можно хотя бы *попытаться*?»

Пола: «Майк, я могу попытаться летать по воздуху. Могу попытаться превратить свинец в золото. Могу попытаться переплыть Атлантический океан. Как ты думаешь, у меня получится?»

Майк: «Послушай, я же не требую невозможного».

Пола: «Нет, Майк, именно *требуешь*».

(Майк ухмыляется, кивает и отворачивается, собираясь отойти.)

Майк: «Я верю в тебя, Пола; знаю, что ты меня не подведешь».

Пола: (в спину Майку) «Майк, ты меня не слушаешь? Это плохо кончится».

(Майк просто машет рукой, не поворачиваясь.)

Пассивная агрессивность

Пола должна принять интересное решение. Она подозревает, что Майк не рассказывает Дону о ее оценках. Она может просто позволить Майку идти своим путем и в конечном итоге свалиться в пропасть. Ей нужно лишь позаботиться о том, чтобы в переписке были зарегистрированы все копии соответствующих служебных записок. Когда разразится катастрофа, Пола покажет всем, *что* и *когда* она говорила Майку. Это пассивно-агрессивная тактика — просто позволить Майку самому залезть в петлю.

Также Пола может попытаться предотвратить катастрофу, обратившись к Дону напрямую. Безусловно, это рискованно, но в этом и заключается суть понимания интересов коллектива. Когда прямо на вас мчится грузовой поезд и никто, кроме вас, его не видит, у вас есть выбор: либо молча отойти в сторону и посмотреть, как он задавит всех остальных, либо закричать: «Поезд! Немедленно уходите!»

Проходит два дня...

Пола: «Майк, ты сообщил Дону о моих оценках? Он сказал заказчику, что в демо-версии не будет работать функция отправки файлов?»

Майк: «Пола, ты сказала, что сделаешь это для меня».

Пола: «Нет, Майк, я этого не говорила. Я тебе сказала, что это невозможно. Вот копия служебной записки, которую я отправила тебе после нашего разговора».

Майк: «Да, но ты же сказала, что *попытаешься*, верно?»

Пола: «Мы это уже обсуждали, Майк. Помнишь, свинец и золото?»

Майк: (вздыхает) «Послушай, Пола, это очень нужно. Просто нужно. Пожалуйста, сделай все необходимое, но ты просто должна сделать это к сроку».

Пола: «Майк, ты ошибаешься. Я *должна* сделать совсем другое — сообщить Дону, если этого не сделаешь ты».

Майк: «Это нарушение субординации, так нельзя».

Пола: «А я и не хочу, Майк, но ты меня вынуждаешь».

Майк: «Ох, Пола...»

Пола: «Послушай, Майк, мы не успеем реализовать всю функциональность демо-версии вовремя. Усвой это наконец. Перестань уговаривать меня больше работать. Перестать обманывать себя, что я каким-то чудом вытащу кролика из шляпы. Пойми, что ты должен сообщить это Дону, и притом сообщить прямо сегодня».

Майк: (с широко раскрытыми глазами) «Сегодня?»

Пола: «Да, Майк, сегодня. Потому что ты, я и Дон проведем встречу, на которой обсудим функциональность, включаемую в демо-версию. Если эта встреча не состоится, я буду вынуждена сама обратиться к Дону. Вот копия служебной записки, в которой это объясняется».

Майк: «Ты просто прикрываешься!»

Пола: «Майк, я пытаюсь прикрыть нас обоих. Ты представляешь, что произойдет, если заказчик придет сюда, ожидая увидеть полную демо-версию, а мы ее не сможем предъявить?»

Чем кончится история Пола и Майка? Проработайте возможные варианты сами. Суть в том, что Пола вела себя очень профессионально. Она говорила «нет» в правильно выбранные моменты и говорила правильно. Она сказала «нет», когда на нее давили, чтобы она изменила свою оценку. Она сказала «нет» на все попытки манипуляций, лести и мольбы.

И что самое важное — она сказала «нет» самообману и бездействию Майка. Пола действовала в интересах коллектива. Майку была необходима помощь, и она использовала все, что было в ее силах, чтобы ему помочь.

Цена согласия

Чаще мы предпочитаем говорить «да». Действительно, в здоровом коллективе люди стараются найти путь к согласию. Руководители

и разработчики в хорошо управляемых группах ведут переговоры до тех пор, пока не найдут взаимоприемлемый план действий.

Но как мы уже видели, иногда, для того чтобы прийти к «да», нужно не бояться сказать «нет».

Для примера возьмем следующую историю, опубликованную Джоном Бланко в своем блоге¹. Она воспроизводится здесь с его разрешения. Во время чтения спросите себя, когда и как ему следовало сказать «нет».

ХОРОШИЙ КОД СТАЛ НЕВОЗМОЖНЫМ?

В юношеском возрасте вы решаете стать программистом. В старших классах вы учитесь писать программы по объектно-ориентированным принципам. В колледже вы применяете усвоенные принципы в таких областях, как искусственный интеллект и 3D-графика.

А после вступления в круг профессионалов начинается ваша бесконечная работа по написанию качественного на коммерческом уровне, простого в сопровождении, «идеального» кода, который выдержит испытание временем.

Качество коммерческого уровня? Ха. Это довольно забавно.

Я считаю, что мне везет. Я люблю паттерны проектирования. Мне нравится изучать теорию идеального программирования. Я запросто могу затеять часовую дискуссию по поводу неудачного выбора иерархии наследования моим партнером по XP — что отношения типа «содержит» во многих ситуациях предпочтительнее отношений «является частным случаем». Но в последнее время мне не дает покоя один вопрос...

...Почему в современном программировании стал невозможным хороший код?

ТИПИЧНОЕ ПРЕДЛОЖЕНИЕ

Работая по контракту, я провожу свои дни (и ночи) за разработкой мобильных приложений для своих клиентов. И за те многие годы, когда я этим занимался, я понял, что требования работы на заказчика не позволяют мне писать по-настоящему качественные приложения, которые мне хотелось бы создавать.

Прежде чем я начну, позвольте сказать, что меня нельзя упрекнуть в недостатке старания. Мне нравится тема чистого кода. Я не знаю никого, кто бы стремился к идеальной архитектуре программного продукта так же сильно. Проблема кроется в исполнении, и не по той причине, о которой вы подумали.

Позвольте рассказать вам историю.

В конце прошлого года одна хорошо известная компания провела конкурс на разработку приложения. Фирма — очень крупный оператор розничной торговли; для сохранения конфиденциальности назовем ее «Горилла Маркет». Представители заказчика указали, что им нужно организовать свое

¹ <http://raptureinvenice.com/?p=63>

присутствие в области приложений iPhone, а приложение должно быть готово к «черной пятнице»¹. В чем проблема? Сегодня уже 1 ноября. На создание приложения остается всего 4 недели. Да, и еще в это время Apple обычно требуется до двух недель на утверждение приложений (старые добрые времена). Выходит, приложение должно быть написано... ЗА ДВЕ НЕДЕЛИ?!?

Да. У нас две недели на создание приложения. И к сожалению, наша заявка победила. (В бизнесе фигура заказчика играет важную роль.) Никуда не денешься.

«Ничего страшного, — говорит Руководитель № 1 из «Горилла Маркета», — приложение простое. Все, что нужно, — показать пользователю несколько продуктов из нашего каталога и дать ему возможность найти адреса магазинов. На нашем сайте это уже сделано. Мы дадим готовую графику. Вероятно, вы сможете использовать — как это называется? — да, жесткое кодирование!»

В разговор вступает Руководитель № 2: «И еще нам понадобятся купоны на скидку, которые пользователь сможет предъявить на кассе. Откровенно говоря, приложение пишется „на выброс“. Давайте сделаем его, а потом для фазы II „с нуля“ будет написано другое, больше и лучше».

Так оно и происходит. Несмотря на годы постоянных напоминаний о том, что каждая затребованная заказчиком функция всегда оказывается сложнее, чем кажется из его объяснений, вы соглашаетесь. Вы действительно верите, что на этот раз все будет сделано за две недели. Да! Мы справимся! На этот раз все будет иначе! Несколько графических изображений и обращение к службе для получения адреса магазина. XML! Запросто. Мы справимся. Поехали!

Всего одного дня оказывается достаточно, чтобы я снова вернулся к реальности.

Я: Итак, дайте мне информацию, необходимую для вызова веб-службы с адресами магазинов.

ЗАКАЗЧИК: А что такое «веб-служба»?

Я:

Именно так все и происходило. Данные об адресах магазинов, выводимые в правом верхнем углу их веб-сайта, предоставлялись вовсе не веб-службой — они генерировались кодом Java. И вдобавок хостинг обеспечивался стратегическим партнером «Горилла Маркета».

В моей ситуации мне удалось добиться от «Горилла Маркета» только текущего списка магазинов в виде файла Excel. Код поиска пришлось писать «с нуля».

Позднее в этот же день последовал второй удар: заказчик хотел, чтобы данные продуктов и купонов могли еженедельно меняться. О жестком кодировании данных можно забыть! Выходит, за две недели придется на-

¹ День массовых распродаж в США; приходится на период с 22 по 29 ноября. — Примеч. перев.

писать не только приложение для iPhone, но и исполнительную часть на PHP, и интегрировать ее с... Что? Контролем качества тоже придется заниматься мне?

Чтобы компенсировать возросший объем работы, нам придется программировать немного быстрее. Забудьте про паттерн «Абстрактная фабрика». Заменяем паттерн «Компоновщик» большим и уродливым циклом `for` — некогда!

Хороший код стал невозможным.

ДВЕ НЕДЕЛИ ДО СДАЧИ ПРОЕКТА

Уверяю вас, эти две недели были довольно паршивыми. Первые два дня пропали из-за многочасовых собраний по моему следующему проекту. (Это только подчеркивает, насколько мало времени осталось для работы.) В конечном итоге на работу у меня осталось 8 дней. В первую неделю я проработал 74 часа, а в следующую... Боже... Я даже не помню, это стерлось из моих синапсов. Наверное, к лучшему.

Я провел эти восемь дней за яростным программированием. Я пустил в ход все возможные средства, чтобы справиться со своей работой: копирование/вставку (АКА повторное использование кода), «волшебные числа» (чтобы избежать дублирующихся определений констант с их последующим — о ужас! — повторным вводом) — и НИКАКИХ модульных тестов! (Кому нужны проблемы в такое время, они только отобьют охоту работать!)

Код получился довольно скверным, и у меня не было времени на рефакторинг. Впрочем, при таких сроках он был весьма неплох — ведь код все равно писался «на выброс», верно? Что-то из этого кажется вам знакомым? Подождите, дальше будет еще интереснее.

Накладывая завершающие штрихи (прежде чем переходить к написанию серверного кода), я начал поглядывать на кодовую базу и думать, что все, возможно, не так уж плохо. Ведь приложение работает, в конце концов. Я выжил!

«Боб у нас работает совсем недавно, он был очень занят и не мог позвонить раньше. А теперь он говорит, что пользователи должны вводить адреса своей электронной почты для получения купонов. Он еще не видел приложения, но думает, что это отличная идея! Кроме того, нам понадобится система построения отчетов для получения введенных адресов с сервера. И если уж речь зашла о купонах, они должны иметь ограниченный срок действия, а срок действия мы должны задавать сами. Да, и еще...»

А теперь вернемся на шаг назад. Что мы знаем о хорошем коде? Хороший код должен быть расширяемым. Простым в сопровождении. Он должен легко модифицироваться. Он должен читаться, как проза. Так вот, мой код не был хорошим.

И еще одно. Если вы хотите повысить свою квалификацию как разработчика, всегда помните: заказчик постоянно увеличивает объем работы. Он всегда хочет добавить в приложение новые возможности. Он всегда хочет вносить изменения — НА ПОЗДНЕЙ СТАДИИ.

Вот простая формула успеха:

$$\begin{aligned} & (\text{количество руководителей})^2 \\ & + 2 * \text{количество новых руководителей} \\ & + \text{количество детей у Боба} \\ & = \text{ДНЕЙ, ДОБАВЛЯЕМЫХ В ПОСЛЕДнюю МИНУТУ} \end{aligned}$$

Руководители — такие же люди, как мы. Они должны обеспечивать свои семьи (если Сатана разрешил им завести семью). Они хотят, чтобы приложение было успешным (время повышения!). Проблема в том, что все они хотят претендовать на свою долю успеха в проекте. После того как все будет сказано и сделано, они хотят указать на некоторую функцию или архитектурное решение, которое они бы могли назвать своей личной заслугой.

Но вернемся к нашему проекту. Мы добавили еще пару дней и реализовали ввод адресов электронной почты. А потом я упал в обморок от усталости.

ЗАКАЗЧИК МЕНЬШЕ БЕСПОКОИТСЯ О ПРОЕКТЕ, ЧЕМ ВЫ САМИ —————

Клиенты, несмотря на все их заявления, несмотря на очевидную срочность, никогда не беспокоятся о нарушении графика сильнее, чем вы. В день завершения работы над приложением я разослал сообщение с финальной сборкой всем ключевым участникам. Руководителям (grrr!), менеджерам и т. д. «ГОТОВО! ВОТ ВАМ ВЕРСИЯ 1.0! СЛАВА БОГУ!» Я нажал кнопку «Отправить», откинулся в кресле и с довольной ухмылкой начал представлять себе, как заказчики несут меня на руках, а на 42-й улице проходит парад, где меня венчают лаврами «Величайшего Разработчика Всех Времен». По крайней мере, мое лицо должно быть на их рекламе, верно?

Как ни странно, фирма-заказчик не торопилась меня хвалить. Я вообще не знал, что они думают по этому поводу. Я не получил никакой реакции. Ни единого сообщения. Похоже, руководство «Горилла Маркет» решило, что этот этап уже пройден, и перешло к следующему проекту.

Думаете, я вру? Убедитесь сами. Я подал заявку в Apple с незаполненным описанием приложения. Я запросил описание у «Горилла Маркета», но мне никто не ответил, а ждать было некогда (см. предыдущий абзац). Я написал снова. И снова. Подключил к этому наше руководство. Дважды мне звонили, и дважды я слышал: «Напомните, что вам было нужно?» МНЕ БЫЛО НУЖНО ОПИСАНИЕ ПРИЛОЖЕНИЯ!

Через неделю началось тестирование приложения в Apple. Обычно это время радости, но для меня это было время смертельного ужаса. Как и ожидалось, через день приложение было отклонено по самой жалкой и неубедительной причине, которую я только могу себе представить: «У приложения отсутствует описание». С функциональностью все в порядке; нет описания. И по этой причине приложение «Горилла Маркет» не было готово к «черной пятнице». Меня это порядком раздражало.

Я пожертвовал своим общением с семьей ради двухнедельного рабочего марафона, а в «Горилла Маркете» за целую неделю никто не побеспокоился создать описание приложения! Мы получили описание через час после отказа Apple.

И если до этого я испытывал раздражение, то через полторы недели я пришел в полную ярость. Оказалось, что заказчик не предоставил нам реальные данные. Продукты и купоны на сервере были фиктивными. Условными, если хотите. Код купона был равен 1234567890 — просто взят с потолка.

А потом наступило судьбоносное утро, когда я зашел на Портал — И ПРИЛОЖЕНИЕ БЫЛО ДОСТУПНО! С фиктивными данными и всем прочим! Я в ужасе названивал всем, кому было можно, и вопил: «МНЕ НУЖНЫ ДАННЫЕ!» Женский голос спросил, с кем меня соединить — с пожарными или с полицией, и я повесил трубку. Но потом я все-таки дозвонился в «Горилла Маркет» со своим «МНЕ НУЖНЫ ДАННЫЕ!» И я никогда не забуду ответ:

«Здравствуйте, это Джон. У нас сменился вице-президент, и мы решили отказаться от выпуска. Отзовите его из App Store, хорошо?»

В итоге оказалось, что не менее 11 людей зарегистрировали свои адреса в базе данных. Это означало, что теоретически 11 людей могут заявиться в «Горилла Маркет» с фальшивым купоном. Весело, правда?

В общем, во всех утверждениях заказчика правдой было только одно: код действительно писался «на выброс». Единственная проблема заключалась в том, что он вообще не был использован.

РЕЗУЛЬТАТ? СПЕШКА С ЗАВЕРШЕНИЕМ, МЕДЛЕННЫЙ ВЫХОД НА РЫНОК

Мораль этой истории: ключевые участники проекта (будь то внешний заказчик или внутренний руководитель) придумали схему, которая заставит разработчиков быстро писать код. Эффективно? Нет. Быстро? Да. Вот как работает эта схема.

- Сказать разработчику, что приложение очень простое. Это создает у группы разработки искаженное представление о масштабах работы. Кроме того, разработчики быстро берутся за работу, а тем временем...
- Функциональность проекта расширяется, причем рабочая группа оказывается виноватой в том, что не распознала эту необходимость заранее. В нашем случае жесткое кодирование контента должно было привести к усложнению обновлений. Как я мог этого не понять сразу? Я понял, но до этого я получил лживые обещания от заказчика. Или другой вариант: заказчик нанимает «нового человека», который находит какое-нибудь явное упущение. А завтра заказчик скажет, что они приняли на работу Стива Джобса, и в приложение нужно добавить алхимические трансформации? Далее...
- Проект постоянно подгоняется, чтобы работа была завершена к исходному сроку. Разработчики трудятся на максимальной скорости (и с максимальным риском ошибок, но кто станет обращать на это внимание?). До срока остается пара дней? Зачем говорить, что срок сдачи можно перенести, если работа идет так продуктивно? Нужно использовать это в своих интересах! Потом срок наступает, добавляется еще несколько дней, потом неделя — и это после того, как вы

отработаете 20-часовую смену, чтобы все было сделано вовремя. Все как на знаменитой картинке с ослим и морковкой — не считая того, что с ослим обращаются намного лучше, чем с вами.

Схема отлично придумана. Можно ли обвинять ее создателей, уверенных в том, что она работает? Просто они не видят кошмарного кода. И так происходит снова и снова, несмотря на результаты.

В условиях глобализированной экономики, когда корпорации держатся за всемогущий доллар, а повышение котировки акций связано с сокращением штата, сверхурочной работой и оффшорной разработкой, описанная стратегия экономии на разработчиках делает хороший код невозможным. Если мы, разработчики, не проявим должной осторожности, то нас просьбами/приказами/угрозами заставят писать вдвое больший объем кода за половину времени.

О невозможности хорошего кода

Когда в этой истории Джон спрашивает: «Хороший код стал невозможным?», в действительности он спрашивает: «Профессионализм стал невозможным?» В конце концов, в этой печальной истории пострадал не только код. Пострадала его семья, его работодатель, заказчики и пользователи. В проигрыше оказались все¹. И проигрыш объясняется непрофессионализмом.

Кто здесь действовал непрофессионально? Джон недвусмысленно намекает, что это были руководители «Горилла Маркета». Схема, описанная им в конце, ясно указывает на их непорядочное поведение. Но было ли это поведение *плохим*? Я так не думаю.

Они хотели иметь приложение для iPhone к «черной пятнице». Они были готовы заплатить за него. Они нашли кого-то, кто возьмется за эту работу. Так в чем их винить?

Да, в процессе общения явно возникали проблемы. И очевидно, руководители фирмы-заказчика не знали, что такое веб-служба; это обычное дело — одно подразделение крупной корпорации не знает, чем занимается другое. Но все это следовало предвидеть. Джон даже признает это, когда говорит: «Несмотря на годы постоянных напоминаний о том, что каждая затребованная заказчиком функция всегда оказывается сложнее, чем кажется из его объяснений...»

Итак, если виновником был не «Горилла Маркет», то кто?

¹ Возможно, за исключением непосредственного работодателя Джона, хотя я готов поспорить, что он тоже оказался в проигрыше.

Возможно, непосредственный работодатель. Джон явно не говорит об этом, но намекает в своей снисходительной фразе: «В бизнесе фигура заказчика играет важную роль». Может, работодатель Джона раздавал неразумные обещания «Горилла Маркету»? Он оказывал давление на Джона (прямое или косвенное), чтобы эти обещания оправдались? Джон не говорит об этом, так что мы можем только догадываться.

Но даже если так, за что в этой истории отвечает сам Джон? Я возлагаю всю ответственность исключительно на него. Это Джон согласился на исходный двухнедельный срок, отлично зная, что проекты обычно оказываются более сложными, чем кажется на первый взгляд. Это Джон согласился написать серверную часть на PHP. Это Джон согласился на требование о регистрации по электронной почте и ограничении срока действия купона. Это Джон работал по 20 часов в сутки и по 90 часов в неделю. Это Джон отказался от своей семьи и нормальной жизни, чтобы не сорвать срок сдачи.

Почему Джон так поступил? Он об этом говорит вполне определенно: «Я нажал кнопку „Отправить“, откинулся в кресле и с довольной ухмылкой начал представлять себе, как заказчики несут меня на руках, а на 42-й улице походит парад, где меня венчают лаврами „Величайшего Разработчика Всех Времен“». Короче говоря, Джону захотелось быть героем. Он увидел шанс добиться славы и ухватился за него.

Профессионалы часто совершают героические дела, но не потому, что хотят быть героями. Профессионалы становятся героями, когда они хорошо выполняют свою работу, без нарушения сроков и бюджета. Стремясь стать «героем дня», Джон действовал непрофессионально.

Джон должен был сказать «нет» на исходный двухнедельный срок. А если не сказал — то должен был сделать это, когда обнаружил, что никакой веб-службы для получения данных не существует. Он должен был сказать «нет» в ответ на требование регистрации по электронной почте и ограничения срока действия купонов. Он должен был сказать «нет» на все, что приводит к немыслимым жертвам и перерасходу времени.

Но самое главное, Джон должен был сказать «нет» своему внутреннему решению о том, что выполнить работу в установленный срок можно только одним способом — устроив неразбериху в коде. Обратите внимание, что говорит Джон о хорошем коде и модульных тестах: «Чтобы компенсировать возросший объем работы, нам придется прогнать немного быстрее. Забудьте про паттерн „Абстрактная фабрика“. Заменяем паттерн „Компоновщик“ большим и уродливым циклом `for` — некогда!»

И еще раз:

«Я провел эти восемь дней за яростным программированием. Я пустил в ход все возможные средства, чтобы справиться со своей работой: копирование/вставку (АКА повторное использование кода), „волшебные числа“ (чтобы избежать дублирующихся определений констант с их последующим — о ужас! — повторным вводом) — и НИКАКИХ модульных тестов! (Кому нужны проблемы в такое время, они только отобьют охоту работать!)»

Все эти решения и стали истинной причиной катастрофы. Джон принял тот факт, что прийти к успеху можно только через непрофессиональное поведение, и получил то, чего заслужил.

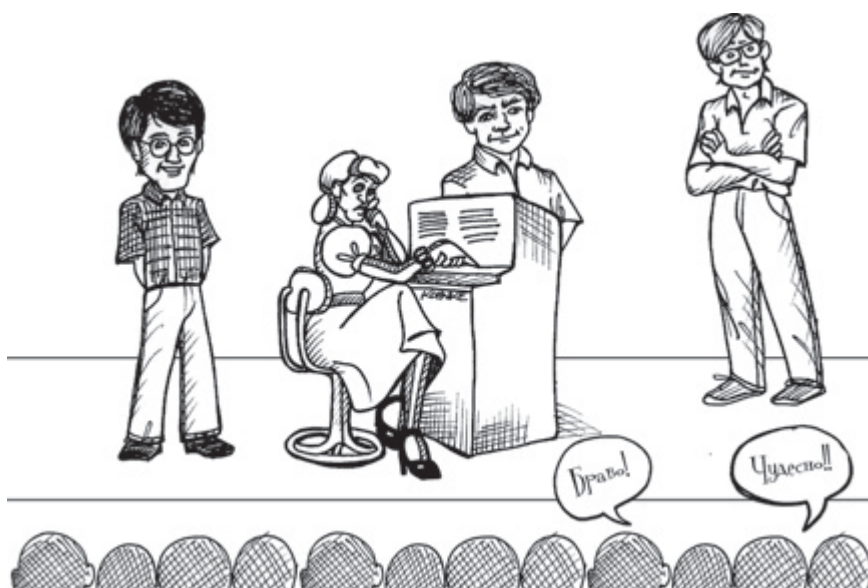
Возможно, это звучит излишне сурово. Я этого не хотел. В предыдущих главах я описал, как неоднократно совершал ту же ошибку в своей карьере. Искушение «быть героем» и «решать проблемы» велико. Однако все мы должны понять, что отказ от профессиональных принципов не решает проблемы, а создает их. Учитывая сказанное, я наконец-то могу ответить на исходный вопрос Джона:

«Хороший код стал невозможным? Профессионализм стал невозможным?»

Я говорю *«нет»!*

3

Как сказать «да»



А вы знаете, что я изобрел голосовую почту? Честное слово. Вообще-то нас, владельцев патента на голосовую почту, было трое: Кен Файндер, Джерри Фитцпатрик и я. Это было в начале 80-х годов, когда мы работали на компанию Teradyne. Наш исполнительный директор поручил нам создать продукт нового типа, и мы изобрели «электронного секретаря» (сокращенно ЭС).

Все вы отлично знаете, что такое «электронный секретарь». Это одна из тех кошмарных машин, которые отвечают на звонки в компаниях и задают всевозможные идиотские вопросы, на которые нужно отвечать нажатием кнопок («Чтобы переключиться на английский язык, нажмите 1»).

Наш ЭС отвечал на звонок и просил ввести имя нужного человека, после чего вызывал его по внутренней связи. ЭС сообщал о вызове и спрашивал, соединить ли со звонившим. Если ответ был положительным, он устанавливал связь со звонившим и отключался.

Вы могли сообщить ЭС свое текущее местонахождение. Также ему можно было задать несколько телефонных номеров. Если вы были в другом офисе, ЭС находил вас. Если вы были дома, ЭС находил вас. Если вы находились в другом городе, ЭС находил вас. А если у него это все же не получилось, он оставлял сообщение. Именно здесь была впервые применена голосовая почта.

Как ни странно, фирма Teradyne не смогла придумать, как бы ей продать «электронного секретаря». Проект вышел за рамки бюджета и был преобразован в систему CDS (Craft Dispatch System) для передачи специалистам по ремонту телефонов информации о следующем задании. Также фирма Teradyne отказалась от патента, не сообщив об этом нам (!). Текущий владелец патента подал заявку на три месяца позже нас (!!)¹.

После того как ЭС превратился в CDS (но задолго до отказа от патента), я ждал исполнительного директора компании... на дереве. Перед фасадом здания рос большой дуб, я забрался на него и ждал, пока подъедет его «Ягуар». Я перехватил его у двери и попросил уделить мне несколько минут; он согласился.

Я сказал, что проект ЭС нужно запускать заново и что мы наверняка на нем сможем заработать. Но мой собеседник удивил меня, сказав: «Хорошо, Боб, подготовь план. Покажи, как мы на этом сможем заработать. Если сделаешь, а я твоему плану поверю, мы снова запустим ЭС».

Этого я не ожидал. Предполагалось, что он скажет: «Ты прав, Боб. Давай запустим проект, а я придумаю, как на этом заработать». Но нет, он переложил бремя на меня. Не могу сказать, что меня это обрадовало — я все-таки программист, а не финансист. Мне хотелось работать над проектом ЭС, а не отвечать за прибыли и убытки. Но я не хотел показать свое разочарование, поэтому поблагодарил его и вышел из офиса со словами: «Спасибо, Расс, я непременно сделаю... когда будет время».

А теперь я передаю слово: Рой Ошеров сейчас расскажет, каким жалким было это заявление.

¹ Правда, никаких денег я на этом не потерял. Я продал свой патент Teradyne за 1 доллар согласно условиям контракта (хотя и этот доллар я не получил).

Язык обещаний

Рой Ошеров

Сказать. Ответственно отнестись. Сделать.

Обещание состоит из трех частей.

1. Вы *говорите*, что вы это сделаете.
2. Вы *ответственно относитесь* к своим словам.
3. Вы *выполняете* обещанное.

Но часто ли нам встречаются люди (конечно, это не мы сами!), которые выполняют все три части?

- Вы спрашиваете парня из технического отдела, почему сеть работает так медленно, он говорит: «Да, надо бы закупить новые маршрутизаторы». Понятно, что сделано ничего не будет.
- Вы просите участника группы провести ручное тестирование перед сдачей исходного кода, он отвечает: «Конечно. Постараюсь сделать к концу дня». И почему-то вам кажется, что завтра нужно будет спросить, провел он тестирование или нет.
- Начальник входит в комнату и бормочет: «Нам надо работать побыстрее». Вы понимаете, что на самом деле это ВАМ нужно работать побыстрее. Он ничего делать не собирается.

Лишь очень немногие люди, обещая что-то, ответственно относятся к своим словам и делают то, что обещали. Некоторые говорят и даже искренне собираются выполнить обещание, но ничего не делают. Гораздо больше людей, которые обещают, совершенно не собираясь что-то делать. Слышали, как ваши знакомые говорят: «Надо бы заняться спортом, сбросить пяток килограммов», хотя вы прекрасно знаете, что ничего делать они не будут? Такое происходит постоянно.

Откуда берется это странное ощущение, что люди в большинстве случаев слишком легкомысленно относятся к своим обещаниям?

Что еще хуже, нас порой подводит интуиция. Иногда нам *хочется* верить, что собеседник ответственно относится к своим словам, хотя в действительности это не так. Мы *хотим* верить тому, что говорит загнанный в угол разработчик — например, что двухнедельный проект будет завершен за одну неделю... хотя на самом деле это не так.

Вместо того чтобы доверять интуиции, можно по некоторым языковым признакам определить, насколько ответственно люди относятся к своим словам. Правильно выбирая слова, можно решить проблемы с частями 1 и 2 со своей стороны.

Признаки пустых обещаний

Тщательно выбирайте формулировки, которые вы используете в своих обещаниях, потому что по словам часто можно судить о дальнейшем ходе событий. Если вам не удастся подобрать нужные слова, скорее всего, вы недостаточно ответственно относитесь к сказанному или не верите в его выполнимость. Несколько примеров слов и выражений, которые являются характерными признаками пустых обещаний.

- «Нужно/должен»: «Нам нужно это сделать поскорее», «Мне бы нужно сбросить лишние килограммы», «Кто-то должен об этом позаботиться».
- «Надеюсь/хорошо бы»: «Надеюсь, это будет сделано к завтрашнему дню», «Надеюсь, мы еще встретимся и поговорим на эту тему», «Хорошо бы выкроить время для этого», «Хорошо бы, чтобы компьютер работал побыстрее».
- «Давайте»: «Давайте потом встретимся», «Давайте доделаем эту штуку».

Если вы начнете обращать внимание на эти слова, то увидите, как часто они звучат вокруг вас — и даже в том, что вы говорите другим. Вы увидите, как мы стараемся избежать ответственности за сказанное. И это недопустимо, если вы или кто-то другой полагается на эти обещания в своей работе. Впрочем, первый шаг уже сделан — вы начинаете узнавать признаки безответственного отношения к обещаниям в окружающих и в себе.

Итак, мы знаем, как выглядят пустые обещания. Как узнать настоящее, серьезное обещание?

Признаки серьезных обещаний

У всех выражений из предыдущего раздела есть нечто общее: они предполагают, что «вы» никак не влияете на происходящее и не несете за него ответственности. Во всех случаях люди ведут себя так, словно они являются *жертвами* ситуации, а не контролируют ее.

Но на самом деле *лично вы* ВСЕГДА можете хоть как-то повлиять на ситуацию, поэтому вы всегда можете ответственно пообещать *что-то* сделать.

Верными признаками серьезных обещаний являются выражения вида «Я сделаю то-то... к такому-то времени...» (например, «Я закончу работу над этим модулем ко вторнику»).

Чем так важна эта фраза? *Вы утверждаете факт того, что ВЫ что-то сделаете, с указанием четкого момента завершения*. Вы говорите не о ком-то другом, а только о себе. Вы говорите о том, что сделаете лично вы. Вы не «*надеетесь*», что это будет сделано, и не уточняете «*если будет время*»; вы просто выполните обещанное.

Давая такое устное обязательство, вы не сможете отказаться от него без нарушения обещания. Вы сказали, что сделаете, и теперь возможен только один из двух вариантов: вы либо делаете, либо не делаете. Если не делаете, то окружающие могут справедливо спросить, чего же стоят ваши обещания. Вам будет *стыдно* сказать другим, что вы не сделали обещанное (если они слышали ваше обещание). Неприятная перспектива, не так ли?

Вы принимаете на себя полную ответственность за что-либо перед аудиторией, состоящей минимум из одного человека. Вы стоите не перед зеркалом и не перед экраном компьютера. Вы разговариваете с другим человеком и обещаете что-то сделать. С этого начинается ответственное обещание. Поставьте себя в ситуацию, которая заставит вас что-то сделать.

Переход на язык обязательств поможет вам пройти следующие две фазы: ответственного отношения к словам и выполнения обещанного.

Есть несколько причин, из-за которых говорящий может не относиться ответственно к своим обещаниям (или препятствующих их выполнению).

Выполнение обещания зависит от другого человека X

Обещайте только то, что находится под вашим полным контролем. Например, если вашей целью является завершение модуля, в работе над которым также участвует другая группа, вы не можете обещать

завершить модуль с полной интеграцией. Однако вы можете обещать выполнить конкретные действия, которые приблизят поставленную цель. Вы можете:

- пообщаться часок-другой с Гарри из инфраструктурной группы, чтобы понять зависимости;
- создать интерфейс, абстрагирующий зависимости модуля от инфраструктуры другой группы;
- встречаться не менее трех раз в неделю с ответственным за сборку, чтобы обеспечить работоспособность ваших изменений в системе сборки, используемой в компании;
- создать собственную процедуру сборки, в ходе которой выполняются интеграционные тесты.

Видите разницу?

Если конечная цель зависит от кого-то другого, обещайте только конкретные действия, которые способствуют достижению конечной цели.

Вы не уверены в том, что обещание можно выполнить

Даже если конечная цель невозможна, вы можете взять на себя обязательства по выполнению действий, приближающих ее достижение. Более того, проверка достижимости цели может быть одним из таких действий!

Вместо того чтобы обещать исправить все 25 оставшихся ошибок до выхода финальной версии (что может быть невозможно), вы обещаете выполнить конкретные действия, приближающие вас к этой цели.

- Перебрать все 25 ошибок и попытаться воспроизвести их.
- Пообщаться с автором каждого сообщения об ошибке и увидеть ее воспроизведение.
- Потратить все оставшееся время на исправление ошибок.

Вы не справились

Что ж, бывает. Могло произойти что-то непредвиденное — жизнь есть жизнь. Однако вы не должны подводить тех, кто от вас чего-то ожидает. В такой ситуации лучше как можно скорее изменить ожидания.

Если вы не можете выполнить свое обещание, очень важно как можно быстрее сообщить об этом тому, кому вы обещали.

Чем быстрее вы оповестите о неудаче все заинтересованные стороны, тем больше вероятность того, что у группы будет время остановиться, переоценить текущую обстановку и решить, можно ли что-то сделать или изменить (например, приоритеты). Возможно, это позволит выполнить исходные обязательства или изменить их.

Пара примеров.

- Вы назначили встречу в кафе с коллегой, но застряли в транспортной пробке. Вы сомневаетесь в том, что вам удастся сдержать свое обещание вовремя быть на месте. Как только вы поймете, что можете опоздать, позвоните коллеге и сообщите ему об этом. Возможно, вы найдете другое место или отложите встречу.
- Вы пообещали исправить ошибку, которая на первый взгляд казалась вполне рядовой. В какой-то момент вы понимаете, что ошибка оказалась намного коварнее ваших представлений о ней, и поднимаете белый флаг. Далее группа может выбрать программу действий для выполнения этого обязательства (парная работа, проработка потенциальных решений, мозговая атака) или же изменить приоритеты и перевести вас на более простую ошибку.

Если вы никому не сообщите о потенциальной проблеме как можно быстрее, то никто не сможет вам помочь в выполнении ваших обязательств.

Резюме

Сама идея особого «языка обещаний» выглядит немного пугающе, но она поможет решить многие проблемы общения, с которыми программисты сталкиваются в своей повседневной работе: прогнозы, сроки, недоразумения при личном общении. Вас будут считать серьезным разработчиком, который держит свое слово, — а это, возможно, одна из самых высоких репутаций в нашей отрасли.

Учимся говорить «да»

Я попросил Роя разрешение на публикацию этой статьи, потому что она задела меня за живое. Я долго объяснял, как научиться говорить «нет». Однако не менее важно научиться говорить «да».

Обратная сторона «попытки»

Представьте, что Питеру поручено внести изменения в систему оценок. По его собственной оценке, работа займет пять-шесть дней. Он также полагает, что подготовка документации по изменениям займет несколько часов. В понедельник утром Мардж, его начальник, интересуется у него текущим состоянием дел.

Мардж: «Питер, изменения в системе оценок будут готовы к пятнице?»

Питер: «Я думаю, это возможно».

Мардж: «Вместе с документацией?»

Питер: «Попытаюсь сделать и ее».

Возможно, Мардж не слышит нерешительности в заявлениях Питера — на самом деле он ничего не обещает. Мардж задает вопросы, на которые нужно ответить «да» или «нет», а Питер дает неопределенные ответы.

Обратите внимание на слово «попытаюсь». В предыдущей главе мы определили его как «приложить дополнительные усилия», но здесь Питер использует определение «может, получится, а может, нет». Ему следовало бы отвечать иначе.

Мардж: «Питер, изменения в системе оценок будут готовы к пятнице?»

Питер: «Не исключено, но это может быть и понедельник».

Мардж: «Вместе с документацией?»

Питер: «На документацию уйдет еще несколько часов. Теоретически могу успеть к понедельнику, но возможно, документация будет готова только во вторник».

В данном случае Питер говорит более честно — он описывает существующую неопределенность. Возможно, Мардж удастся что-то сделать с этой неопределенностью. А может, и не удастся.

Дисциплинированное принятие обязательств

Мардж: «Питер, мне нужно четкое „да“ или „нет“. Система оценок вместе с документацией будет готова к пятнице?»

Мардж задает абсолютно правильный вопрос. Ей нужно обеспечить соблюдение графика, и ей нужен однозначный ответ насчет пятницы. Как должен ответить Питер?

Питер: «В таком случае я должен сказать „нет“. Самая ранняя дата, когда изменения и документация будут точно готовы, — это вторник».

Мардж: «Ты обещаешь сделать ко вторнику?»

Питер: «Да, ко вторнику все будет готово».

А если для Мардж очень важно, чтобы изменения и документация были готовы именно к пятнице?

Мардж: «Питер, со вторником у нас будут большие проблемы. Вилли, наш штатный технический писатель, освободится в понедельник. У него будет всего пять дней на подготовку руководства пользователя. Если документация по системе оценок не будет готова в понедельник утром, то он не уложится в срок. Ты не можешь сначала написать документацию?»

Питер: «Нет, сначала нужно внести изменения, потому что документация строится по выходным данным тестовых запусков».

Мардж: «Может, изменения с документацией как-то возможно завершить до утра понедельника?»

Теперь Питер должен принять решение. Вполне возможно, что модификация системы оценок будет завершена в пятницу; может быть, даже документация будет готова еще до того, как он отправится домой на выходные. А если работа займет больше времени, чем он рассчитывает, можно выделить несколько часов в субботу. Что он должен сказать Мардж?

Питер: «Знаешь, Мардж, если я поработаю несколько часов в субботу, то с большой вероятностью все будет готово в понедельник утром».

Решает ли это проблему Мардж? Нет, высказывание Питера просто изменяет вероятность успеха. Питер должен говорить иначе.

МАРДЖ: «Так я могу рассчитывать на утро понедельника?»

ПИТЕР: «Возможно, но гарантировать не могу».

Не исключено, что такая формулировка Мардж не устроит.

МАРДЖ: «Послушай, Питер, мне нужна определенность. Ты можешь *обещать*, что работа будет закончена к утру понедельника?»

В этот момент у Питера возникает искушение схалтурить. Возможно, работа будет завершена быстрее, если обойтись без написания тестов. Или без рефакторинга. Или если отказаться от полного регрессионного тестирования.

Именно в такие моменты проявляется профессионализм. Во-первых, такие предположения неверны — работа не будет завершена быстрее, если Питер не напишет тесты. Она не будет завершена быстрее, если он не проведет рефакторинг или пропустит полное регрессионное тестирование. Многолетний практический опыт учит нас тому, что нарушение правил только замедляет работу.

Во-вторых, профессионал обязан поддерживать определенные стандарты. Его код должен пройти тестирование, поэтому без тестов не обойтись. Его код должен быть чистым и элегантным. И он должен убедиться в том, что изменения не нарушают работу других компонентов системы.

Питер как профессионал уже дал обязательство соблюдать эти стандарты. Все остальные обязательства отходят на второй план, поэтому о всех недостойных измышлениях необходимо забыть.

ПИТЕР: «Нет, Мардж, самый ранний срок, который я действительно могу гарантировать, — это вторник. Извини, если это нарушает твои планы, но нам приходится смириться с реальностью».

МАРДЖ: «Черт, я очень рассчитывала, что ты справишься быстрее. Уверен?»

ПИТЕР: «Да, я уверен, что работа может задержаться до вторника».

МАРДЖ: «Ладно, придется поговорить с Вилли. Возможно, ему удастся изменить свой график».

В данном случае Мардж согласилась с ответом Питера и перешла к поиску других возможностей. А если все возможности исчерпаны? Если Питер был последней надеждой?

Мардж: «Послушай, Питер, я понимаю, что для тебя это будет не-легко, но мне очень нужно, чтобы все было готово к понедельнику утром. Это очень важно. Можно хоть что-нибудь сделать для этого?»

Теперь Питеру приходится думать о сверхурочных, которые, вероятно, займут большую часть выходных. Он должен абсолютно объективно оценить свою трудоспособность и резервы. Легко сказать «поработаю на выходных»; намного труднее найти в себе достаточно сил для качественного выполнения работы.

Профессионалы знают границы своих возможностей. Они знают, какой объем работы они могут выполнить сверхурочно, и знают, чем за это придется расплачиваться. В данном случае Питер вполне уверен, что нескольких дополнительных часов на неделе и работы в выходные будет достаточно.

Питер: «Хорошо, Мардж, вот что я скажу. Сейчас я позвоню домой и обговорю возможность работы на выходных со своей семьей. Если они не против, то все будет сделано к утру понедельника. Я даже приду в понедельник утром и прослежу за тем, чтобы у Вилли не было вопросов. Но потом я отправлюсь домой и буду отдыхать до среды. Договорились?»

Питер говорит абсолютно честно. Он уверен в том, что при сверхурочной работе он справится с изменениями и написанием документации. Он также знает, что пару дней после этого от него не будет проку на работе.

Итоги

Профессионал не обязан отвечать «да» на все, что от него требуют. Тем не менее он должен приложить максимум усилий к тому, чтобы это «да» стало возможным. Когда профессионалы говорят «да», они используют такие формулировки, чтобы у собеседника не возникало сомнений в надежности их обещаний.

4

Написание кода



В предыдущей книге¹ я подробно описал структуру и природу Чистого Кода. В этой главе будет рассмотрен сам *акт* написания кода, а также контекст, в котором он происходит.

Когда мне было 18 лет, я набирал текст достаточно быстро, но мне приходилось смотреть на клавиши. Я не умел печатать «вслепую». Однажды вечером я провел несколько часов за перфоратором IBM 029, стараясь не смотреть на клавиши во время набора программы, записанной на нескольких формулярах. После набора я проверил все перфокарты и выбросил те, которые содержали ошибки.

¹ Мартин Р. Чистый код. Создание, анализ и рефакторинг. СПб.: Питер, 2010.

Сначала я ошибался довольно часто, но к концу вечера набор шел почти идеально. За этот долгий вечер я понял, что качество «слепой» печати в основном зависит от *уверенности*. Мои пальцы уже знали, где находятся клавиши; мне оставалось только набраться уверенности в том, что я не ошибаюсь. Среди прочего, мне помогало то, что я сразу ощущал, когда совершаю ошибку. К концу вечера я почти мгновенно распознавал ошибки ввода и просто выбрасывал испорченную перфокарту, не глядя на нее.

Умение интуитивно ощущать свои ошибки очень важно — не только при наборе текста, но и во всем остальном. «Чувство ошибки» означает, что вы очень быстро замыкаете цикл обратной связи и все быстрее учитесь на своих ошибках. С того дня, проведенного за перфоратором, я изучал — и успешно освоил — немало других дисциплин. И во всех случаях ключом к мастерству была уверенность в себе и «чувство ошибки».

В этой главе описан мой личный набор правил и принципов написания кода. Эти правила и принципы относятся не к самому коду, а к моему поведению, настроению и отношению. Они описывают мой умственный, моральный и эмоциональный контекст написания кода. В них кроются корни моей уверенности и «чувства ошибки».

Возможно, вы не согласитесь с чем-то из сказанного. В конечном счете все это сугубо личное мнение. Более того, какие-то принципы могут вызвать у вас яростный протест. И это нормально — они не являются абсолютными истинами для кого-то, кроме меня. В них отражены воззрения всего лишь одного человека на профессиональное программирование.

Возможно, изучая и обдумывая мои личные взгляды на написание кода, вы найдете в них что-то полезное для себя.

Готовность

Написание кода — интеллектуально сложное и утомительное занятие. Мало найдется других дисциплин, требующих такого уровня концентрации и внимания. Причина заключается в том, что при написании кода вам приходится жонглировать сразу несколькими взаимно противоречивыми факторами.

1. Прежде всего ваш код должен работать. Вы должны разобраться в сути решаемой проблемы и понять, как ее решать. Вы должны

убедиться в том, что написанный код адекватно отражает решение. Вы должны проследить за каждой подробностью решения, не нарушая правил языка, платформы, текущей архитектуры и всех специфических особенностей текущей системы.

2. Ваш код должен решать задачу, поставленную заказчиком. Часто требования заказчика не отражают всего, что необходимо для решения его задачи. Вы должны понять это и обсудить с заказчиком, чтобы написанный код соответствовал истинным потребностям заказчика.
3. Ваш код должен хорошо вписываться в существующую систему. Он не должен приводить к повышению ее жесткости, непрочности или непрозрачности. Вы должны хорошо организовать управление зависимостями. Короче говоря, ваш код должен соответствовать принципам качественного проектирования¹.
4. Ваш код должен нормально читаться другими программистами. Проблема не сводится к написанию понятных комментариев. Скорее, вы должны строить код таким образом, чтобы в его структуре проявлялись ваши намерения. Сделать это непросто. Вполне возможно, что это самый сложный навык, которым должен овладеть программист.

Жонглировать всеми этими аспектами нелегко. Поддерживать необходимую концентрацию и внимание на протяжении длительного времени трудно хотя бы на физиологическом уровне. Добавьте к этому проблемы и раздражители, связанные с работой в группе и организации, а также заботы повседневной жизни. Как видите, отвлекающих факторов более чем достаточно.

Если вы не можете в достаточной степени сосредоточиться на своей работе, у вас получится плохой код. В нем будет много ошибок. Он будет иметь неверную структуру. Он получится запутанным и непрозрачным. Он не будет соответствовать истинным потребностям заказчика. Короче говоря, код придется перерабатывать или писать заново. Работа без сосредоточенности — напрасная трата времени.

Если вы устали или не можете сосредоточиться — *не пишите код*. Все равно написанное придется переделывать. Лучше подумайте, как устранить отвлекающие факторы и обрести душевное равновесие.

¹ Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

Ночное программирование

Худший код, созданный мной, был написан в 3 часа ночи. Это было в 1988 году, когда я работал в телекоммуникационной начинающей фирме Clear Communications. Мы проводили долгие часы за «трудовыми подвигами» — конечно, мечтая со временем разбогатеть.

Однажды очень поздним вечером (а вернее, очень ранним утром) для решения проблемы синхронизации я организовал отправку моим кодом сообщения самому себе через систему диспетчеризации событий (мы называли это «отправить почту»). Решение было неверным, но в 3 часа ночи оно казалось чертовски привлекательным. В самом деле, после 18 часов непрерывного программирования (не говоря уже о 60–70-часовой рабочей неделе) мне ничего другого в голову уже не приходило.

Помню, я очень гордился своей многочасовой работой. Помню, я ощущал свою *целеустремленность*. Помню, мне казалось, что работать в 3 часа ночи — удел настоящего профессионала. Как же я ошибался!

Этот код, словно бумеранг, возвращался к нам снова и снова. Он породил ошибочную архитектуру, которую использовали все и недостатки которой постоянно приходилось маскировать обходными решениями. Код приводил к появлению странных ошибок синхронизации и необъяснимых циклов обратной связи. Одно сообщение приводило к отправке другого, потом третьего — и так до бесконечности. У нас вечно не было времени на то, чтобы переписать эту халтуру (нам так казалось), но у нас всегда хватало времени для наложения очередной «заплатки». Мусор накапливался, а мой ночной код обрастал все большими последствиями и побочными эффектами. Через много лет эта история превратилась в популярную шутку нашей группы. Когда я уставал или впадал в отчаяние, коллеги говорили: «Смотрите! Боб собирается отправить себе почту!»

Мораль: не пишите код, когда вы устали. Преданность делу и профессионализм проявляются в дисциплине, а не в продолжительности работы. Обязательно следите за сном, здоровьем и образом жизни, чтобы вы могли ежедневно посвящать работе восемь *хороших* часов.

Программирование в расстроенных чувствах

Когда-нибудь пытались писать код после серьезной ссоры с женой или другом? Замечали, что у вас в мозгу запускается фоновый процесс,

который пытается разрешить конфликт или по крайней мере припомнить его? Иногда давление этого фонового процесса ощущается в груди или где-то в животе. Вы ощущаете беспокойство, как если бы выпили слишком много кофе или диетической колы. И это отвлекает от работы.

Когда я беспокоюсь по поводу спора с женой, разногласий с заказчиком или болезни ребенка, я не могу сконцентрироваться на своей работе. Я ловлю себя на том, что глазею на экран, держа руки на клавиатуре, — и ничего не делаю. Кататония. Паралич. Я нахожусь за миллион миль от рабочего места, размышляя над своей фоновой проблемой, — вместо того чтобы заниматься своим прямым делом.

Иногда мне удается *заставить* себя думать о коде. Я принуждаю себя написать одну-две строки. Усилием воли добиваюсь прохождения одного-двух тестов. Но я не могу продолжить в том же духе. Неизбежно я снова прихожу в состояние оцепенелой бесчувственности, ничего не видя открытыми глазами, переживая свои фоновые неприятности.

Я узнал, что писать код в такие моменты бесполезно. Код, который я напишу, можно сразу отправлять в мусор. Вместо того чтобы работать над кодом, я должен сначала разобраться со своими проблемами.

Конечно, многие проблемы попросту не решаются за час или два. К тому же наши работодатели вряд ли смирятся с нашей неспособностью работать в то время, пока мы переживаем свои личные неурядицы. Значит, вы должны научиться завершать фоновый процесс или по крайней мере понижать его приоритет, чтобы он не отвлекал вас постоянно.

Я решаю эту проблему посредством распределения времени. Вместо того чтобы заставлять себя программировать, пока личные проблемы беспокоят меня, я выделяю специальное время (например, час) на причину. Если мой ребенок заболел, я звоню домой и справляюсь о его состоянии. Если я поссорился с женой, я звоню ей и обсуждаю наши разногласия. Если у меня проблемы с деньгами, я обдумываю возможное решение. Я знаю, что проблема вряд ли будет решена за этот час, но по крайней мере мне с большой вероятностью удастся снизить свое беспокойство и приглушить фоновый процесс.

В идеале на борьбу с переживаниями должно тратиться личное время. Стыдно тратить рабочее время подобным образом. Профессиональные разработчики тратят личное время для того, чтобы время на работе проводилось как можно продуктивнее. Следовательно, вы должны специально выделить время дома на решение своих проблем, чтобы

не тащить их на работу. С другой стороны, если вы уже находитесь на работе, а тревожные мысли подрывают вашу производительность, лучше потратить час на их подавление, чем заставлять себя писать код, который позднее все равно будет выброшен (или еще хуже, с которым вам придется жить).

Зона потока

О сверхпроизводительном состоянии, называемом «поток» (flow), написано много литературы. Некоторые программисты называют его «зоной». Как бы оно ни называлось, вероятно, вам знакомо это ощущение предельной концентрации сознания, в которое может войти программист при написании кода. В этом состоянии программисты ощущают себя сверхэффективными и непогрешимыми, поэтому они стремятся войти в это состояние, а их самооценка часто определяется тем, какая часть времени в нем проводится.

А теперь небольшой совет от того, кто неоднократно бывал в Зоне и возвращался из нее: *Избегайте Зоны*. На самом деле это состояние не настолько уж эффективно и, безусловно, не непогрешимо. Это всего лишь умеренно-медитативное состояние, в котором мыслительные способности снижаются ради ощущения скорости.

Позвольте мне высказаться более определенно. В Зоне можно написать больше кода. Если вы практикуете разработку через тестирование (TDD), то вы скорее преодолеете цикл «красный/зеленый/рефакторинг». И у вас появится легкая эйфория или чувство достижения цели. Проблема заключается в том, что во время нахождения в Зоне теряется часть «общей картины», поэтому вы скорее примете решения, которые вам позднее придется исправлять. Код быстрее пишется в Зоне, но вам придется чаще возвращаться к нему.

Теперь когда я чувствую, что мое сознание постепенно входит в Зону, я на несколько минут отхожу от края. Я стараюсь прояснить свои мысли: отвечаю на электронную почту, просматриваю твиты. Если время идет к полудню — я делаю обеденный перерыв. Если я работаю в группе, то нахожу кого-нибудь для парного программирования.

У парного программирования есть одно важное преимущество: паре практически невозможно войти в Зону. Состояние Зоны некоммуникативно, тогда как парная работа требует интенсивного, постоянного

общения. В самом деле, одна из претензий к парному программированию как раз и заключается в том, что оно блокирует входение в Зону. Отлично! Зона — это не то место, где вам стоит находиться.

Вообще-то это не совсем так. В некоторых ситуациях Зона — именно то место, где вам стоит находиться. Когда вы *тренируетесь*! Но мы поговорим об этом в другой главе.

Музыка

Когда я работал в Teradyne в конце 1970-х, у меня была отдельная комната. Я был системным администратором нашей PDP 11/60, поэтому я был одним из нескольких программистов, имевших собственный терминал. Это был терминал VT100 со скоростью передачи данных 9600 бод, подключенный к PDP11 25 метрами кабеля RS232, который я лично прокладывал над подвесным потолком из офиса в машинный зал.

В моей комнате стояла стереосистема: старый проигрыватель, усилитель и динамики. У меня была довольно серьезная коллекция винила, включая Led Zeppelin, Pink Floyd, ... В общем, вы поняли.

Обычно я включал свою стереосистему перед тем, как писать код. Мне казалось, что она способствует моей концентрации. Тем не менее я ошибался.

Однажды я просматривал модуль, который редактировал во время прослушивания начала «Стены». Комментарии в коде содержали слова из песни и упоминания о звуковых вставках: пикирующие бомбардировщики, плачущие дети и т. д. И тогда я понял: читатель такого кода намного больше узнает о музыкальной коллекции автора (меня самого), чем о задаче, для решения которой был написан этот код.

Мне стало ясно, что я просто не могу хорошо писать код во время прослушивания музыки. Музыка не помогает мне сосредоточиться. Более того, на само прослушивание расходуются ресурсы, необходимые моему мозгу для написания чистого, хорошо спланированного кода.

Возможно, у вас дело обстоит иначе. Возможно, вам музыка *помогает* писать код. Я знаю много людей, которые программируют в наушниках. Я допускаю, что музыка помогает им, но также подозреваю, что в действительности она лишь помогает им войти в Зону.

Помехи

Представьте, что вы программируете за рабочей станцией. Как вы отреагируете на заданный вопрос? Огрызнетесь? Смерите холодным взглядом? Ваш «язык тела» намекнет, что вы заняты? Короче, вы среагируете невежливо? Или вы прервете свою текущую работу и любезно поможете? Проявите ли вы такое же отношение, какое хотели бы видеть от других, если бы сами обратились к ним за помощью?

Невежливые ответы часто исходят из Зоны. Возможно, вас раздражает, что вас вытаскивают из Зоны или кто-то мешает вашим попыткам войти в Зону. Как бы то ни было, грубость часто объясняется вашей связью с Зоной.

В некоторых случаях виновником оказывается не Зона, а лично вы. Просто вы пытаетесь разобраться в сложном вопросе, требующем полной концентрации. У этой задачи есть несколько решений.

Парная работа отлично помогает справиться с помехами. Ваш напарник погружается в контекст задачи, а вы разбираетесь с телефонными звонками и вопросами коллег. Когда вы возвращаетесь к напарнику, он быстро помогает восстановить интеллектуальный контекст на момент до возникновения помехи.

Также большую помощь оказывает разработка через тестирование. Если какой-то тест не проходит, то он определяет текущий контекст вашей работы. Разобравшись с помехой, вы возвращаетесь к нему и продолжаете работать над устранением проблемы.

Разумеется, вам не удастся полностью избежать помех, которые *неизбежно будут отвлекать вас и* приводить к потерям времени. Помните, что в следующий раз помощь может понадобиться вам. Таким образом, профессионализм проявляется в вежливой готовности прийти на помощь.

Творческий кризис

Иногда работа попросту не идет. Такое бывало со мной, и я видел, как такое случается с другими. Вы сидите за компьютером... и ничего не происходит.

Часто вы находите другие дела. Вы читаете электронную почту или Твиттер. Вы просматриваете книги, графики и документы. Вы устраи-

ваете собрания. Вы затеваете дискуссии. Короче, вы делаете *что угодно*, чтобы вам не пришлось снова оказаться за компьютером.

Из-за чего возникают подобные «творческие кризисы»? Мы уже обсудили многие причины. Лично для меня главным фактором является нехватка сна. Если я слишком мало сплю, то я просто не могу программировать. Также важную роль играют беспокойство, страх и депрессия.

Как ни странно, у этой проблемы существует очень простое решение. Оно срабатывает почти всегда. Оно легко реализуется, и оно может обеспечить необходимый импульс для написания большого объема кода.

Найдите напарника для парного программирования.

Просто невероятно, насколько эффективно оно работает. Как только вы садитесь с кем-то рядом, все проблемы, мешавшие работе, немедленно исчезают. Работа в паре приводит к физиологическим изменениям. Я не знаю, что именно происходит, но определенно чувствую. То ли в мозгу, то ли в теле происходит химическое изменение, которое позволяет мне преодолеть застой и снова взяться за работу.

Учтите, что решение не идеально. Иногда изменение длится час-два, а за ним следует истощение настолько серьезное, что мне приходится отрываться от напарника и искать место для отдыха. Иногда, даже когда я работаю в паре, у меня хватает сил только соглашаться с тем, что делает напарник. И все же для меня типичной реакцией на работу в паре является восстановление творческого потенциала.

Творческий ввод

Существуют и другие меры для предотвращения застоя. Я уже давно узнал, что результаты творческой работы зависят от выбора источников.

Я читаю много книг на разные темы. Я читаю материалы по программированию, политике, биологии, астрономии, физике, химии, математике и многим другим темам. Однако я обнаружил, что мои механизмы творческой работы лучше всего активизирует научная фантастика.

Для вас это может быть что-то другое — хороший детектив, поэзия или даже любовный роман. Наверное, дело в том, что творчество порождает творчество. Также стоит учитывать элемент эскапизма. Часы, проведенные вдали от повседневных забот под активным воздействием интересных, творческих идей, вызывают почти непреодолимое желание создать что-нибудь самому.

Не все формы творческой деятельности работают одинаково эффективно. Просмотр телевизора обычно не влияет на мой творческий процесс. Поход в кино работает лучше, но ненадолго. Музыка не помогает создавать код, но способствует созданию презентаций и подготовке видеоматериалов. И все же из всех форм творческих источников ничто не воздействует на меня лучше, чем старая добрая «космическая опера».

Отладка

Один из худших сеансов отладки за всю мою карьеру случился в 1972 году. Терминалы, подключенные к бухгалтерской системе профсоюза грузоперевозчиков, зависали один-два раза в день. Сознательно воспроизвести ошибку было невозможно. Ошибка не отдавала предпочтений какому-то конкретному терминалу или приложению. Не важно, чем занимался пользователь перед зависанием: сейчас терминал работает нормально, а в следующую минуту безнадежно зависает.

На диагностику требовались недели. Тем временем грузоперевозчики испытывали все большее раздражение. Каждый раз при зависании пользователю приходилось прекращать работу и ждать, пока все остальные пользователи тоже выполнят свои текущие операции. После этого они звонили нам, и мы перезагружали компьютер. Короче, настоящий кошмар.

Первая пара недель была проведена за простым опросом пользователей, работавших за зависающими терминалами. Мы спрашивали, чем они занимались в тот момент и что делалось до этого. Мы спрашивали других пользователей, не заметили ли они чего-нибудь необычного на своих терминалах в момент зависания. Собеседования приходилось проводить по телефону, потому что терминалы были установлены в пригородах Чикаго, а мы работали на 30 миль к северу.

У нас не было журналов, счетчиков или отладчиков. Все взаимодействие с внутренним состоянием системы осуществлялось через индикаторы и тумблеры передней панели. Мы могли остановить компьютер и просмотреть содержимое памяти по словам. Однако заниматься этим более 5 минут было невозможно, потому что грузоперевозчикам была нужна их система.

Мы потратили несколько дней на написание простого инспектора, работавшего в режиме реального времени. Им можно было управлять с телетайпа ASR-33, который служил нам консолью. Инспектор позволял читать и изменять содержимое памяти во время работы системы.

Мы добавили журнальные сообщения, которые выводились на теле-тайп в критических ситуациях. Мы создали в памяти счетчики событий, которые запоминали информацию состояний для ее просмотра инспектором. И конечно, весь код создавался «с нуля» на ассемблере и тестировался по вечерам, когда система не использовалась.

Работа терминалов управлялась прерываниями. Символы, передаваемые терминалам, хранились в циклических буферах. Каждый раз при передаче символа последовательным портом срабатывало прерывание и к отправке готовился следующий символ циклического буфера.

Со временем выяснилось, что терминал зависал из-за рассинхронизации трех переменных, управлявших циклическим буфером. Мы понятия не имели, почему это происходило, но это было хоть что-то. Где-то в 5К строк кода супервизора содержалась ошибка, которая некорректно работала с одним из этих указателей.

Новая информация также позволила нам снимать блокировку с терминалов вручную! Мы могли при помощи инспектора присвоить этим трем переменным значения по умолчанию, и терминалы, как по волшебству, начинали работать снова. Вскоре мы написали маленький фрагмент кода, который проверял синхронизацию счетчиков и восстанавливал их в случае необходимости. Сначала код запускался специальным тумблером пользовательского прерывания на передней панели, когда заказчики по телефону сообщали о зависании. Позднее мы просто выполняли код восстановления каждую секунду.

Месяц спустя проблема с зависанием исчезла — по крайней мере с точки зрения профсоюза грузоперевозчиков. Время от времени один из их терминалов приостанавливался на полсекунды, но с базовой скоростью передачи 30 символов в секунду никто этого не замечал.

Но почему происходила десинхронизация счетчиков? Мне было 19, и я был полон решимости разобраться.

Автором кода супервизора был Ричард, уехавший на учебу в колледж. Никто из нас толком не разбирался в супервизоре, потому что Ричард относился к своему созданию очень ревниво. Код принадлежал ему, и нам было не положено разбираться в нем. Но теперь Ричарда не было, поэтому я нашел листинг толщиной в несколько дюймов и начал просматривать его страницу за страницей.

Циклические буферы в этой системе были обычными структурами данных FIFO, то есть очередями. Прикладные программы заносили символы с одного конца очереди, пока она не заполнялась. Обработчики прерываний извлекали символы с другого конца очереди, когда принтер

был готов принять их. Если в очереди не оставалось символов, принтер останавливался. Из-за ошибки приложения считали, что очередь заполнена, а обработчики прерываний — что она пуста. Обработчики прерываний выполнялись в другом «программном потоке», отдельно от остального кода. Таким образом, счетчики и переменные, доступные для обоих обработчиков и остального кода, должны быть защищены от параллельного обновления. В нашем случае это означало, что перед выполнением любого кода, работавшего с этими тремя переменными, необходимо было запретить прерывания. К тому моменту, когда я сел за код, мне уже стало ясно: нужно искать участок кода, который работает с переменными без предварительного запрета прерываний.

Конечно, сейчас появилось множество мощных инструментов для поиска всех мест изменения переменных в программе. За считанные секунды вы найдете все строки кода, которые работают с переменными. За минуты можно будет определить, где именно автор забыл запретить прерывания. Однако наша история происходила в 1972 году, и у меня таких инструментов не было. Были только мои глаза.

Я просмотрел каждую страницу кода в поисках переменных. К сожалению, переменные использовались везде — почти на каждой странице программа обращалась к ним тем или иным образом. При многих обращениях прерывания не запрещались — они ограничивались чтением, а следовательно, были безвредными. Вдобавок в этом конкретном ассемблере было невозможно проверить, доступна ли переменная только для чтения, без анализа логики кода. Каждое чтение переменной могло сопровождаться ее обновлением. И если при этом прерывания не были запрещены, содержимое переменных могло быть легко испорчено.

Мне потребовалось несколько дней интенсивного изучения кода, но в конце концов я нашел ошибку. В середине кода отыскилось одно место, в котором одна из трех переменных обновлялась без предварительного запрета прерываний.

Я занялся вычислениями. Уязвимость существовала на протяжении двух микросекунд. В системе дюжина терминалов передавала данные на скорости 30 символов в секунду, так что прерывания происходили каждые 3 микросекунды или около того. С учетом размера супервизора и тактовой частоты процессора зависания от этой уязвимости должны были происходить с частотой примерно 1–2 раза в день. Есть!

Конечно, я исправил ошибку, но у меня не хватило смелости отключить автоматический запуск проверки и исправления счетчиков. До сих пор не уверен в том, что в системе не было другой «дыры».

Время отладки

По какой-то неведомой причине разработчики не считают отладку естественной частью процесса разработки. Им кажется, что отладка сродни физиологической потребности: ей занимаются просто потому, что это неизбежно. Однако время отладки обходится фирме ровно в такую же сумму, что и время написания кода, поэтому любые меры по его сокращению будут полезны.

Сейчас я провожу за отладкой намного меньше времени, чем десять лет назад. Я не проводил точных измерений, но, по моим оценкам, продолжительность отладки сократилась раз в 10. Я добился этого воистину радикального сокращения переходом на методологию разработки через тестирование (TDD, Test Driven Development), которая будет рассматриваться в следующей главе.

Независимо от того, используете ли вы TDD или другую методологию аналогичной эффективности¹, вы как профессионал обязаны стремиться по возможности приблизить время отладки к нулю. Конечно, нуль — цель асимптотическая, но от этого она не перестает быть целью.

Врачи не любят заново делать операции пациентам, чтобы исправить свои прошлые ошибки. Адвокаты не любят повторно браться за «заваленные» дела. Врач или адвокат, который слишком часто допускает ошибки, не будет считаться профессионалом. Аналогичным образом разработчик, создающий слишком много ошибок, действует непрофессионально.

Выбор темпа

Программирование — марафон, а не спринт. Невозможно выиграть забег, набрав максимальную скорость на старте. Побеждает тот, кто бережет силы и разумно выбирает темп. Марафонец должен заботиться

¹ Я не знаю ни одной методологии, которая бы сравнилась по эффективности с TDD, — но вдруг она известна вам?

о своем организме как до, так и во время соревнований. Точно так же и профессиональные программисты берегут силы и творческий потенциал.

Умейте остановиться

Не можете уйти домой, пока не решили свою задачу? Уйти не только можно, но и нужно! Творческое мышление и интеллектуальная деятельность — недолговечные состояния нашего разума. Когда мы устаем, они исчезают. Если вы будете силой заставлять свой уставший мозг решить задачу в поздний час, скорее всего, это кончится лишь дополнительной усталостью и снижением вероятности того, что задачу удастся решить в душе (или в машине).

Если вы зашли в тупик, если вы устали — отвлекитесь на время. Дайте своему творческому подсознанию отдохнуть от задачи. Внимательно относясь к своим ресурсам, вы сделаете больше за меньшее время и с меньшими усилиями. Сами определяйте темп работы для себя и своей группы. Изучите свои закономерности проявления творческих способностей и озарений и используйте их вместо того, чтобы подчинять насильно.

По дороге домой

Многие задачи были успешно решены мной в машине, когда я возвращался с работы домой. Вождение требует больших затрат нетворческих интеллектуальных ресурсов. Ваши глаза, руки и часть ума заняты управлением машиной, и вы отвлекаетесь от своей задачи на работе. И в этом отвлечении есть что-то такое, что помогает вашему разуму находить другие, более творческие пути поиска решений.

Душ

Неожиданно много задач было решено мной в душе. Возможно, утренние водные процедуры помогают мне проснуться и припомнить все решения, которые возникли в моем мозгу, пока я спал.

Работая над задачей, вы иногда подбираетесь к ней так близко, что не видите всех возможных вариантов. Вы упускаете элегантные решения, потому что творческая часть вашего разума подавляется излишней сосредоточенностью. Иногда лучшее решение задачи — пойти домой,

поужинать, посмотреть телевизор и лечь спать, а на следующее утро проснуться и принять душ.

Отставание от графика

Вы *будете* отставать от графика. Это происходит с самыми лучшими из нас. Это происходит с самыми прилежными. Оценки срываются, а результат запаздывает.

Чтобы свести к минимуму проблемы, связанные с отставанием, помните о двух важнейших аспектах: раннем обнаружении и прозрачности. Хуже всего, когда вы до последнего момента уверяете окружающих, что работа будет завершена вовремя, а потом подводите всех. Не делайте этого. Регулярно проверяйте ход проекта по отношению к конечной цели и представьте три¹ обоснованные конечные даты: лучшую, номинальную и худшую. Будьте по возможности честны в своих оценках. Оптимизму в них не место! Представьте все три даты своей группе и ключевым участникам проекта. Ежедневно обновляйте их.

Надежда

Что делать, если из этих чисел следует, что вы *можете* не успеть к критической дате? Например, через десять дней начинается торговая промышленная выставка, на которой должен быть представлен продукт. С другой стороны, тройственная оценка времени готовности подсистемы, над которой вы работаете, равна 8/12/20.

Не надейтесь, что вам удастся сделать все за 10 дней! Надежда убивает проекты. Надежда срывает графики и рушит репутации. Надежда навлечет на вас большие неприятности. Если выставка начнется через 10 дней, а номинальная оценка составляет 12 дней, вы не успеете к сроку. Убедитесь в том, что группа и ключевые участники проекта понимают ситуацию, и не успокаивайтесь до тех пор, пока не будет сформулирован альтернативный план. И не позволяйте надеяться другим.

Спешка

А если начальник приглашает вас на беседу и настоятельно просит успеть к сроку? Если он настаивает, что вы должны сделать «все

¹ Эта тема гораздо подробнее рассматривается в главе 10.

возможное»? Не отступайте от своих оценок! Исходные оценки всегда точнее любых изменений, вносимых под давлением. Скажите начальнику, что вы уже рассмотрели все варианты (потому что вы их действительно рассмотрели) и что ускорить работу можно только одним способом — усечением части функциональности. Не поддавайтесь искушению ускорить темп.

Горе несчастному разработчику, который уступит давлению и попробует успеть к сроку. Он начинает искать обходные пути и работать сверхурочно в тщетной надежде совершить чудо. Это верный путь к катастрофе, потому что он внушает вам, вашей группе и ключевым участникам проекта неоправданные надежды. Все это оборачивается только нежеланием взглянуть в лицо реальности и откладыванием неприятных, но необходимых решений.

Спешка бессмысленна. Вы не заставите себя программировать быстрее. Вы не заставите себя быстрее решать задачи. А если попытаетесь — вы только замедлите работу и устроите хаос, который замедлит работу других.

Итак, вы должны ответить начальнику, группе и ключевым участникам проекта так, чтобы у них не осталось иллюзий.

Сверхурочные

Начальник говорит: «А если увеличить рабочий день на пару часов? Если работать по воскресеньям? Наверняка как-нибудь можно выжать лишние часы, чтобы успеть ко времени».

Сверхурочная работа возможна, а иногда просто необходима. Иногда можно завершить работу к вроде бы невозможной дате, работая по 10 часов в сутки, с одним-двумя воскресеньями. Однако это очень рискованно. Вряд ли 20% увеличение продолжительности рабочего дня позволит вам выполнить на 20% больше работы. Что еще важнее, сверхурочная работа на протяжении более чем двух-трех недель наверняка приведет к провалу.

Следовательно, на сверхурочную работу можно соглашаться *только* при выполнении некоторых условий: 1 — лично вы можете ее себе позволить; 2 — аврал будет продолжаться недолго, не более двух недель, и 3 — у *руководства* имеется *резервный план* на случай, если ваши усилия завершатся неудачей.

Последний критерий имеет решающее значение. Если ваш начальник не может объяснить, что он собирается делать в случае неудачного исхода, не соглашайтесь на сверхурочную работу.

Ложная готовность

Вероятно, худший из всех видов непрофессионализма со стороны программиста — это попытка выдать недоделку за готовый продукт. Иногда это просто открытая ложь, что достаточно плохо. Но гораздо опаснее другая ситуация — попытки подвести рациональную основу под новое определение «готовности». Мы убеждаем себя в том, что сделано достаточно, и переходим к следующей задаче. Мы говорим, что оставшуюся работу можно выполнить позднее, когда у нас будет больше времени.

Эта болезнь заразна. Если одному программисту такое поведение сходит с рук, другие видят и следуют его примеру. Один из них расширяет определение «готовности» еще дальше, и все остальные следуют его примеру. Я видел, как подобные ухищрения доходили до ужасающих крайностей. Один из моих клиентов под «готовностью» понимал регистрацию изменений в базе данных. При этом код мог даже не компилироваться!

Когда группа попадает в эту ловушку, начальство слышит, что все идет нормально. Все отчеты о ходе работ показывают, что работа будет завершена к сроку. Ситуация напоминает пикник слепых на железнодорожных рельсах: никто не видит приближающийся грузовой состав незавершенной работы, пока не будет слишком поздно.

Определение «готовности»

Проблема ложной готовности решается созданием независимого определения «готовности». Для этого следует поручить бизнес-аналитикам и специалистам по тестированию создать автоматизированные приемочные тесты¹, без прохождения которых продукт не может считаться готовым. Тесты пишутся на тестовых языках — таких как FitNesse, Selenium, RobotFX, Cucumber и т. д. Тесты должны быть понятны для бизнесменов и ключевых участников проекта, не связанных с технической стороной, и они должны выполняться по возможности часто.

¹ См. главу 7.

Помощь

Программирование — трудная работа. Чем вы моложе, тем слабее в это верится. В конце концов, программный код — всего лишь последовательность команд `if` и `while`. Но по мере накопления опыта вы понимаете, что важнейшую роль играет способ объединения этих команд `if` и `while`. Нельзя просто свалить их в одну кучу и надеяться на лучшее. Вместо этого систему необходимо тщательно разбить на небольшие, понятные блоки, которые должны быть как можно меньше связаны друг с другом, — а это действительно сложно.

Более того, программирование настолько сложно, что одному человеку с этой работой не справиться. Даже самому квалифицированному специалисту пригодятся мысли и идеи других программистов.

Как помогать другим

Ответственные программисты должны быть готовы помогать друг другу. Программист, который изолируется в своем офисе или кабинке и отказывается отвечать на вопросы других, нарушает профессиональную этику. Ваша работа не настолько важна, чтобы вы не могли выделить немного времени на помощь другим. Честь профессионала обязывает предложить ближним помощь тогда, когда это потребуется.

Это не означает, что вы должны отказаться от «личного времени». Конечно, оно необходимо, но к его выбору следует подойти вежливо и честно. Например, вы можете сообщить, что между 10:00 и полуднем вас нельзя беспокоить, а с 13:00 до 15:00 ваша дверь открыта для других.

Также будьте внимательны к состоянию работы ваших коллег. Если кто-то испытывает затруднения, предложите помощь. Просто удивительно, насколько эффективной порой оказывается такая помощь. Дело не в том, что вы намного умнее своего коллеги; просто свежая точка зрения нередко становится мощным катализатором для решения проблем.

Когда вы кому-то помогаете, сядьте и напишите код вместе. Запланируйте на помощь не менее часа. Возможно, вам понадобится меньше времени, но торопиться не стоит. Сосредоточьтесь на задаче и приложите максимум усилий. Скорее всего, вы в конечном итоге от такого сотрудничества получите больше, чем отдадите.

Как принимать помощь

Когда кто-то предлагает вам помощь, будьте признательны. Примите помощь с благодарностью и отнеситесь к ней со всем вниманием. Не отказывайтесь от помощи, потому что вам не хватает времени. Выделите на разговор около минут 30. Если особой пользы от предложенной помощи не видно, вежливо извинитесь и завершите беседу с благодарностью. Помните: профессионализм обязывает вас не только предлагать, но и принимать предложенную помощь.

Научитесь просить о помощи. Когда вы зашли в тупик или не можете разобраться в задаче, попросите других помочь вам. Если вы находитесь в общей комнате, просто скажите: «Мне нужна помощь». В остальных случаях можно воспользоваться Твиттером, электронной почтой или телефоном. Обратитесь за помощью — в конце концов, это также является делом профессиональной этики. Непрофессионально оставаться в тупике, когда помощь так доступна.

Думаете, я сейчас нарисую радужную картину — хор поет проникновенную песню, пушистые зайчики прыгают на спины единорогам и все дружно отправляются к радуге надежды и изменений? Нет, не совсем так. Видите ли, программисты обычно бывают самонадеянными, эгоцентричными интровертами. Мы идем в эту область не потому, что мы любим людей. Большинство из нас приходит в программирование, потому что нам нравится концентрироваться на мелочах, жонглировать множеством абстрактных концепций и иным образом доказывать себе, что мы являемся обладателями выдающегося интеллекта, — а вовсе не для того, чтобы разбираться с досадными сложностями других людей.

Да, это стереотип. Да, это обобщение со множеством исключений. Однако реальность такова, что программисты не склонны к сотрудничеству¹. Тем не менее сотрудничество играет исключительно важную роль в эффективном программировании. А раз для многих из нас оно не является инстинктом, значит, нам нужны *методологии*, которые заставляют нас сотрудничать.

¹ К мужчинам это относится в гораздо большей степени, чем к женщинам. У меня была замечательная беседа с @desi (Дези Макадам, основательница DevChix) о мотивации женщин-программистов. Я сказал ей, что для меня заставить программу работать — примерно то же самое, что на охоте убить огромного зверя. Она сказала, что для нее и для других женщин, с которыми она общалась, написание кода является созидательным актом.

Обучение

Позднее этой теме будет посвящена целая глава. А пока позвольте мне просто сказать, что ответственность за обучение менее квалифицированных программистов возлагается на их опытных коллег. Учебных курсов недостаточно. Книг недостаточно. Ничто не приведет молодого разработчика к высокой производительности быстрее, чем его собственное желание в сочетании с эффективным обучением со стороны старших товарищей. Итак, еще одна сторона профессиональной этики проявляется в том, что опытные программисты берут под опеку молодых программистов и обучают их. Аналогичным образом профессиональная обязанность неопытных программистов заключается в том, чтобы найти наставника и перенять его опыт.

5

Разработка через тестирование



Методология разработки через тестирование, или TDD (Test Driven Development), появилась в нашей отрасли уже более 10 лет. Изначально она применялась на волне экстремального программирования (XP, eXtreme Programming), но с тех пор была принята на вооружение Scrum и практически всеми остальными гибкими (Agile) методологиями. Даже группы, не использующие гибкие методологии, применяют TDD.

Когда в 1998 году я впервые услышал о «упреждающем тестировании», я отнесся к нему скептически. Да и кто бы поступил иначе? Кто *начинает* работу с написания модульных тестов? Кто будет делать подобные глупости?

Но к тому времени у меня был уже 30-летний опыт профессионального программирования; я видел, как в отрасли появляются и исчезают новые идеи. Я прекрасно понимал, что ничего не стоит отвергать заранее, особенно если рекомендует такой человек, как Кент Бек.

Так в 1999 году я отправился в Медфорд, штат Орегон, чтобы встретиться с Кентом и научиться у него новой методологии. Результат был просто поразительным!

Мы с Кентом сели у него в офисе и начали программировать простую задачу на Java. Я хотел просто написать свой примитивный код, но Кент воспротивился и провел меня по всему процессу шаг за шагом. Сначала он написал крошечную часть модульного теста, которую и кодом-то нельзя было назвать. Затем он написал код, достаточный для того, чтобы тест компилировался. Затем он написал еще один тест и еще немного кода.

Такой рабочий цикл полностью противоречил всему моему опыту. Я привык писать код не менее часа, прежде чем пытаться откомпилировать или запустить его. Но Кент буквально выполнял свой код каждые 30 секунд или около того. Это было невероятно! Но самое интересное, что этот рабочий цикл был мне знаком! Я сталкивался с ним много лет назад, когда еще ребенком¹ программировал игры на интерпретируемых языках вроде Basic или Logo. В этих языках не было сборки как таковой: вы просто добавляли строку кода и запускали программу. Рабочий цикл проходил очень быстро. И по этой причине программирование на этих языках бывало *очень* производительным.

Но в *настоящем* программировании такой рабочий цикл казался абсурдным. В настоящем программировании вы тратили много времени на написание кода, а потом еще больше времени на то, чтобы заставить его компилироваться. И еще больше времени на отладку. Я ведь *был программистом C++, черт побери!* А в C++ процессы сборки и компоновки могли длиться минутами, а то и часами. Тридцатисекундные рабочие циклы казались немыслимыми. Тем не менее передо мной сидел Кент, который писал свою программу на Java с 30-секундными циклами — и без малейшего намека на то, что работа замедлится. И тогда до меня дошло, что эта простая методология позволяет программировать на настоящих языках с продолжительностью рабочего цикла, типичной для Logo! И я капитально «подсел» на TDD!

¹ С высоты своего возраста я считаю ребенком всех, кому меньше 35 лет. Когда мне было за 20, я тратил довольно много времени на написание глупых игр на интерпретируемых языках. Я программировал космические «стрелялки», приключенческие игры, имитаторы скачек, «змейки», азартные игры... и т. п.

Вердикт вынесен

С того времени я узнал, что TDD — нечто много большее, чем простой трюк для сокращения рабочего цикла. Методология обладает множеством преимуществ, которые будут описаны ниже.

Но сначала я должен сказать следующее.

- Вердикт вынесен!
- Дебаты завершены.
- Команда GOTO вредна.
- И TDD работает.

Да, за прошедшие годы о TDD было написано много противоречивых статей и блогов. На первых порах встречалась серьезная критика и сомнения, но в наши дни все дискуссии завершены. Кто бы что ни говорил, TDD работает.

Я знаю, что это утверждение кажется слишком жестким и односторонним, но в конце концов, хирургам уже не нужно доказывать полезность мытья рук. И я не думаю, что программистам нужно защищать TDD.

Как можно называть себя профессионалом, если вы не *знаете*, что весь ваш код работает? А как можно знать, что весь ваш код работает, если вы не тестируете его при каждом внесении изменений? А как тестировать код при каждом внесении изменений, не имея автоматизированных модульных тестов с очень высоким покрытием? Но можно ли создать автоматизированные модульные тесты с очень высоким покрытием без применения TDD?

Впрочем, последнее предложение стоит рассмотреть более подробно. Что же это, собственно, такое — TDD?

Три закона TDD

1. Новый рабочий код пишется только после того, как будет написан модульный тест, который не проходит.
2. Вы пишете ровно такой объем кода модульного теста, какой необходим для того, чтобы этот тест не проходил (если код теста не компилируется, считается, что он не проходит).

3. Вы пишете ровно такой объем рабочего кода, какой необходим для прохождения модульного теста, который в данный момент не проходит.

Эти три закона заставляют вас использовать рабочий цикл продолжительностью около 30 секунд. Сначала вы пишете маленькую часть модульного теста. За эти считанные секунды вы упоминаете в коде имя класса или функции, которые еще не были написаны; естественно, модульный тест не компилируется. Следовательно, далее вы должны написать рабочий код, с которым тест откомпилируется. Но писать больше кода нельзя, поэтому вы переходите к написанию дополнительного кода модульного теста.

Цикл прокручивается снова и снова. Добавляем небольшой фрагмент в тестовый код. Добавляем небольшой фрагмент в рабочий код. Два кодовых потока растут одновременно, превращаясь во взаимодополняющие компоненты. Соответствие между тестами и рабочим кодом напоминает соответствие между антителом и антигеном.

Длинный перечень преимуществ

Уверенность

Разработчик, принявший TDD как профессиональную методологию, пишет десятки тестов каждый день, сотни тестов каждую неделю, тысячи тестов каждый год. И все эти тесты постоянно находятся «под рукой» и запускаются при каждом внесении в код каких-либо изменений.

Я являюсь основным автором и ответственным за сопровождение FitNesse¹ — системы приемочного тестирования на базе Java. На момент написания книги код FitNesse состоял из 64 000 строк, из которых 28 000 содержались в 2200 отдельных модульных тестах. Эти тесты обеспечивают покрытие по меньшей мере 90% рабочего кода², а их выполнение занимает около 90 секунд.

Каждый раз, когда я изменяю какую-либо часть FitNesse, я запускаю модульные тесты. Если они проходят, то я практически полностью уверен, что изменения ничего не нарушили. Насколько «практически полностью»? Достаточно, чтобы опубликовать обновленную версию!

¹ <http://fitnesse.org>

² 90% — минимальная оценка. На самом деле значение намного выше. Точную величину покрытия трудно рассчитать, потому что программные инструменты «не видят» код, выполняемый во внешних процессах или блоках catch.

Весь процесс контроля качества FitNesse сводится к команде `ant release`. Эта команда собирает FitNesse «с нуля», а затем запускает все модульные и приемочные тесты. Если все тесты проходят успешно, я публикую результат.

Снижение плотности дефектов

FitNesse не является критически важным приложением. Если в FitNesse закрадется ошибка, никто не умрет и никто не потеряет миллионы долларов. Исходя из этого, я могу себе позволить опубликовать новую версию на основании только прохождения тестов. С другой стороны, у FitNesse тысячи пользователей, и при том, что за последний код кодовая база расширилась на 20 000 строк, мой список дефектов состоит только из 17 позиций (многие из которых имеют чисто косметическую природу). Таким образом, я знаю, что плотность дефектов в FitNesse чрезвычайно низка.

И этот эффект не уникален. Существенное снижение количества дефектов при использовании TDD описано в ряде отчетов¹ и исследований². От IBM до Microsoft, от Sabre до Symantec — компании и группы сообщают о снижении количества дефектов в 2, 5 и даже 10 раз. Настоящий профессионал не может игнорировать такие показатели.

Смелость

Почему мы не исправляем плохой код сразу же, как только увидим его? Наша первая реакция на неаккуратно написанную, запутанную функцию: «Ну и мешанина, надо бы исправить». Вторая реакция: «Пусть это сделает кто-нибудь другой!» Почему? Потому что вы знаете:

¹ http://www.objectmentor.com/omSolutions/agile_customers.html

² E. Michael Maximilien, Laurie Williams, “Assessing Test-Driven Development at IBM,” http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF

B. George, and L. Williams, “An Initial Investigation of Test-Driven Development in Industry,” <http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperV8.pdf>

D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,” *IEEE Computer*, Volume 38, Issue 9, pp. 43–50.

Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams, “Realizing quality improvement through test driven development: results and experiences of four industrial teams,” Springer Science + Business Media, LLC 2008: http://research.microsoft.com/en-us/projects/esm/nagappan_tdd.pdf

притронувшись к коду, вы рискуете его «сломать»; а если код будет сломан, то он автоматически переходит под вашу ответственность.

А если вы твердо уверены, что чистка кода ничего не нарушит? Если вы просто нажимаете кнопку и через 90 секунд узнаете, что изменения ничего не нарушили, а принесли только пользу?

Это одно из величайших преимуществ TDD. Если у вас имеется пакет тестов, которому можно доверять, вы перестаете бояться вносить изменения. Видя плохой код, вы просто чистите его «на месте». Код становится глиной, из которой лепятся простые, эстетичные структуры.

Когда программист перестает бояться чистить код, он чистит его! Чистый код проще понять, проще изменять и проще расширять. С упрощением кода вероятность дефектов становится еще ниже. Происходит стабильное улучшение кодовой базы — вместо «загнивания кода», столь привычного для нашей отрасли. Разве профессиональный программист может допустить, чтобы загнивание продолжалось?

Документация

Вы когда-нибудь использовали сторонние библиотеки? Фирма-разработчик обычно присылает красивое руководство с десятками глянцевых иллюстраций с кружочками и стрелочками; на обратной стороне каждой иллюстрации приводится абзац текста с описанием настройки, развертывания и других операций с этой библиотекой. А в самом конце, где-нибудь в приложении, прячется маленький невзрачный раздел с примерами кода.

Куда вы первым делом заглянете в таком руководстве? Если вы программист, то вы обратитесь к примерам кода. Вы сделаете это, потому что знаете: код расскажет всю правду. Цветные глянцевые иллюстрации с кружочками и стрелочками выглядят очень мило, но если вы хотите узнать, как использовать код, необходимо читать код.

Каждый модульный тест, написанный с соблюдением трех законов, представляет собой пример использования системы, сформулированный в виде программного кода. Если вы выполняли три закона, то в вашем коде будет модульный тест, описывающий создание каждого объекта в системе, для каждого способа создания таких объектов. В нем будет модульный тест, описывающий вызов каждой функции в системе, для каждого осмысленного способа вызова. Для каждой операции, по поводу которой у вас могут возникнуть вопросы, будет модульный тест, подробно описывающий ее выполнение.

Модульные тесты представляют собой документы, описывающие самый нижний архитектурный уровень системы. Они однозначны, точны, написаны на языке, понятном для аудитории, и достаточно точны и формальны для выполнения. Это самая лучшая низкоуровневая документация, которая только возможна. Какой профессионал не захочет создать такую документацию?

Архитектура

Если вы соблюдаете три закона и пишете тесты раньше рабочего кода, вы сталкиваетесь с дилеммой. Часто вы точно знаете, какой код нужно написать, но три закона приказывают сначала написать модульный тест, который не пройдет, потому что код еще не существует! Таким образом, вам приходится думать о тестировании еще не написанного кода.

Проблема с тестированием кода заключается в необходимости изоляции этого кода. Часто бывает трудно тестировать функцию, вызывающую другие функции. Чтобы написать такой тест, необходимо каким-то образом отделить функцию от всех остальных. Иначе говоря, необходимость тестирования заставляет вас продумать хорошую архитектуру приложения.

Если вы не начинаете с написания тестов, то ничто не мешает вам свалить все функции в одну кучу, не поддающуюся тестированию. Если тесты пишутся позднее, возможно, вам удастся протестировать входное и выходное поведение этой кучи, но, скорее всего, с тестированием отдельных функций возникнут большие проблемы.

Таким образом, соблюдение трех законов и опережающее написание тестов приводит к более качественной архитектуре с меньшим количеством привязок. Какой профессионал откажется от инструментов, применение которых приводит к совершенствованию архитектуры?

«Но я могу написать тесты позднее», скажете вы. Нет, не можете. Конечно, *некоторые* тесты можно написать позднее. Можно даже обеспечить высокое покрытие, если вы проследите за его измерением. Однако тесты, написанные позднее, лишь *защищают* от ошибок — тогда как тесты, написанные с опережением, их активно *атакуют*. Тесты, написанные позднее, пишутся разработчиком, который уже сформировал код и знает, как решалась задача. Такие тесты никак не сравнятся по полноте и актуальности с тестами, написанными заранее.

Выбор профессионалов

Из всего сказанного следует, что TDD — выбор профессионалов. Эта методология повышает уверенность, придает смелости разработчикам, снижает количество дефектов, формирует документацию и улучшает архитектуру. При таком количестве доводов в пользу TDD отказ от использования этой методологии можно считать проявлением непрофессионализма.

Чем TDD не является

При всех своих достоинствах TDD — не религия и не панацея. Выполнение трех законов не гарантирует ни одного из перечисленных преимуществ. Плохой код можно написать даже при предварительном написании тестов. Да и сами тесты тоже могут быть написаны плохо.

В некоторых ситуациях три закона оказываются просто непрактичными или неподходящими. Такие ситуации встречаются редко, но они все же возможны. Ни один профессиональный разработчик не станет применять методологию, которая в конкретной ситуации приносит больше вреда, чем пользы.

6

Тренировка



Все профессионалы оттачивают свое мастерство на специальных упражнениях. Музыканты играют гаммы. Доктора тренируются в наложении швов и выполнении других хирургических приемов. Адвокаты репетируют речи. Солдаты участвуют в учениях. Профессионалы тренируются везде, где важна эффективность выполнения работы. Эта глава полностью посвящена возможности тренировки навыков программирования.

Азы тренировки

Концепция тренировки в программировании появилась довольно давно, но собственно тренировкой она была признана лишь в начале нового тысячелетия. Вероятно, первый формальный пример тренировочной программы был напечатан на странице 6 учебника Кернигана–Ричи ¹.

```
main()  
{  
    printf("hello, world\n");  
}
```

Кто из нас не писал эту программу в той или иной форме? Мы используем ее для того, чтобы опробовать новую среду программирования или новый язык. Когда мы пишем и выполняем эту программу, это доказывает, что мы точно так же можем написать и откомпилировать любую другую программу.

Когда я был намного моложе, освоение нового компьютера обычно начиналось для меня с написания программы SQUINT, вычисляющей квадраты целых чисел. Я писал ее на ассемблере, BASIC, FORTRAN, COBOL и множестве других языков. Все эти многочисленные версии одной программы также доказывали, что я могу заставить компьютер сделать то, что мне нужно.

Первые персональные компьютеры появились в магазинах в начале 1980-х годов. Каждый раз, когда мне представлялась возможность поработать за одним из них (VIC-20, Commodore-64 или TRS-80), я писал небольшую программу для вывода бесконечного потока символов '\ ' и '/ '. Рисунки, которые строились такими программами, радовали глаз и выглядели намного сложнее маленькой программы, которая их строила.

И хотя эти программы были чисто учебными, программисты в целом не тренировались. Откровенно говоря, нам это просто не приходило в голову. Мы были слишком заняты написанием кода, чтобы думать о совершенствовании мастерства. Да и зачем? В те годы программирование не требовало хорошей реакции или проворных пальцев. Первые экранные редакторы появились только в конце 1970-х годов. Большая часть нашего рабочего времени проходила за ожиданием компиляции или отладкой длинных, безобразных потоков кода. Короткие циклы TDD еще не были изобретены, поэтому те нетривиальные возможности, которые открываются благодаря тренировке, были попросту не нужны.

¹ Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Upper Saddle River, NJ: Prentice Hall, 1975.

Двадцать два нуля

Но с начала эпохи программирования прошло много времени. Некоторые обстоятельства сильно изменились, другие не изменились вовсе.

Одной из первых машин, для которых я программировал, была PDP-8/I. Компьютер имел тактовую частоту 1,5 мс и 4096 12-разрядных слов основной памяти, был размером с холодильник и потреблял значительную мощность. Его дисковый накопитель позволял хранить 32К 12-разрядных слов, а взаимодействие с оператором осуществлялось через телетайп со скоростью передачи 10 символов в секунду. Нам казалось, что это очень мощная машина, на которой можно творить чудеса.

Недавно я купил новый ноутбук Macbook Pro. Он оснащен двухъядерным процессором 2,8 ГГц, 8 Гбайт памяти, 512-гигабайтным SSD-накопителем и 17-дюймовым ЖК-экраном с разрешением 1920i 1200. Я ношу его в рюкзаке. Он легко умещается у меня на коленях и потребляет менее 85 ватт.

По сравнению с PDP-8/I мой ноутбук работает в восемь тысяч раз быстрее, имеет в два миллиона раз больше памяти, в шестнадцать миллионов раз большую емкость запоминающего устройства, потребляет 1% мощности, занимает 1% места и стоит в 25 раз меньше. Посчитаем:

$$8000 \cdot 2000\,000 \cdot 16\,000\,000 \cdot 100 \cdot 100 \cdot 25 = 6,4 \cdot 10^{22}.$$

Это *очень большое* число. Разница составляет 22 *порядка*! Это расстояние в ангстремах от нас до альфы Центавра; количество электронов в серебряном долларе; масса Земли в Майклах Мурах — словом, большое, очень большое число. И теперь такой компьютер стоит у меня на коленях — и вероятно, на ваших тоже!

И что я делаю с этой мощностью, возросшей на 10 в 22 степени? Примерно то же, что я делал на PDP-8/I. Я пишу команды `if`, циклы `while` и команды присваивания.

О, инструменты для написания этих команд заметно улучшились, и языки стали более мощными. Но сама природа команд за это время не изменилась. Код 2010 года будет понятен программисту из 1960-х годов. Глина, из которой мы лепим свои программы, не изменилась за четыре десятилетия.

Длительность рабочего цикла

Но сам процесс работы серьезно изменился. В 1960-е годы я мог ждать день или два для получения результатов компиляции. В конце 1970-х годов программа из 50 000 строк компилировалась около 45 минут. Даже в 1990-е годы долгая сборка казалась нормой.

В наши дни программисты не ждут результатов компиляции ¹. В их распоряжении появилась такая немыслимая вычислительная мощь, что цикл «красный-зеленый-рефакторинг» может прокручиваться буквально за секунды.

Например, я веду проект FitNesse, написанный на Java и состоящий из 64 000 строк кода. Полная сборка со всеми модульными и интеграционными тестами занимает менее 4 минут. Если тесты проходят, то я публикую новую версию продукта. Таким образом, *весь процесс контроля качества, от исходного кода до развертывания, занимает менее 4 минут*. Время компиляции ничтожно мало. Тесты выполняются за считанные секунды. Выходит, цикл компиляции/тестирования может прокручиваться *по 10 раз в минуту!*

Конечно, не всегда стоит работать с такой скоростью. Часто бывает лучше замедлиться и *подумать*². Но в некоторых ситуациях прокрутка цикла с максимальной скоростью оказывается в *высшей степени* производительной.

Для быстрого выполнения чего угодно необходима тренировка. Быстрая прокрутка цикла «код/тест» требует очень быстрого принятия решений. А для быстрого принятия решений необходимо успешно распознавать огромное количество ситуаций и проблем, а также просто *знать* решения.

Представьте двух мастеров боевых искусств во время поединка. Каждый должен понять, что пытается сделать другой, и правильно среагировать за считанные миллисекунды. В бою вам не удастся остановить время, проанализировать ситуацию и выбрать подходящий ответ. В бою нужно просто *реагировать*. Ваше тело реагирует, пока разум работает над стратегией более высокого уровня.

Пока вы прокручиваете цикл «код/тест» по несколько раз в минуту, ваше тело знает, какие клавиши нужно нажимать. Основная часть

¹ А если некоторые программисты и ждут, то это трагическая случайность, которая свидетельствует об их неаккуратности. В современном мире время сборки должно измеряться секундами, а не минутами — и уж конечно, не часами.

² Рик Хики называет этот метод «разработкой через лежание в гамаке».

мозга распознает ситуацию и реагирует за миллисекунды, выдавая подходящее решение, тогда как мозг концентрируется на проблеме более высокого уровня.

И в боевых искусствах, и в программировании скорость зависит от тренированности. И в обоих случаях тренировка проходит примерно одинаково: мы выбираем набор пар «проблема/решение» и повторяем их снова и снова, пока не будем знать наизусть.

Представьте гитариста — скажем, Карлоса Сантану. Музыка в его голове просто переходит в пальцы. Он не задумывается над положением пальцев или приемами игры. Его ум свободен для планирования высокоуровневых мелодий и гармоничных сочетаний, тогда как его тело преобразует эти планы в низкоуровневые движения пальцев.

Но для достижения такой простоты игры необходима практика. Музыкант тренируется в исполнении этюдов, гамм и риффов снова и снова, пока не будет знать их наизусть.

Додзё программирования

С 2001 года я провожу демонстрацию TDD, которую я называю «игрой в кегли»¹. Это маленькое упражнение занимает около 30 минут. Оно выявляет конфликт в архитектуре, развивается до кульминационной точки и преподносит сюрприз напоследок. Я написал целую главу об этом примере².

За годы я проводил эту демонстрацию сотни, а то и тысячи раз. Я достиг в ней настоящего мастерства! Я мог бы повторить ее во сне. Я свел к минимуму количество нажатий клавиш, подобрал самые удобные имена переменных и оптимизировал структуру алгоритма, пока она не стала идеальной. Хотя я тогда и не знал этого, это была моя первая ката.

В 2005 году я посетил конференцию XP2005 в Шеффилде (Великобритания). Там я участвовал в презентации под названием «Додзё программирования» (Coding Dojo), которую проводили Лорен Босса-вит и Эммануэль Галло. Все присутствующие открыли свои ноутбуки

¹ Эта ката стала очень популярной; Google выдает информацию о многих ее разновидностях. Описание оригинала находится по адресу <http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>

² Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

и программировали вместе с докладчиками, которые применяли методологию TDD для реализации игры Конвея «Жизнь». Докладчики назвали это упражнение «ката» и сообщили, что исходная идея ¹ принадлежала «Прагматику»² Дэйву Томасу.

С тех пор многие программисты стали использовать метафору боевых искусств для своих тренировочных сеансов. Название «Додзё программирования» тоже прижилось³. Иногда несколько программистов встречаются и тренируются вместе, как мастера боевых искусств. Иногда тренировки проходят в одиночку — тоже по аналогии с боевыми искусствами. Около года назад я обучал группу разработчиков в Омахе. За обедом они пригласили меня на свой сеанс «Додзё программирования». Я наблюдал за тем, как 20 разработчиков открыли свои ноутбуки и клавиша за клавишей повторяли действия своего преподавателя, который выполнял кату «игры в кегли».

В додзё используются разные виды упражнений. Некоторые из них представлены ниже.

Ката

В боевых искусствах термином *ката* называется строго определенный набор отретенированных движений, имитирующих действия одной стороны в поединке. Их целью (в реальности недостижимой) является достижение совершенства. Мастер стремится к тому, чтобы научить свое тело идеально выполнять каждое движение и объединить отдельные приемы в плавную серию. Хорошо исполняемые ката очень красивы.

Но несмотря на красоту, ката изучаются не для сценического исполнения. Мастера изучают их для того, чтобы обучить ум и тело реагировать на конкретные боевые ситуации. Отработанные движения должны выполняться инстинктивно, чтобы они срабатывали именно в тот момент, когда они нужны.

Программные ката представляют собой строго определенный набор отретенированных нажатий клавиш и перемещений мыши, имитирующих решение некоторой программной задачи. Вы не решаете задачу, потому что уже знаете решение. Вместо этого вы тренируетесь в выполнении действий и принятии решений, необходимых для решения задачи.

¹ <http://codekata.pragprog.com>

² Мы используем префикс «Прагматик», чтобы отличить его от «Большого» Дэйва Томаса из OTI.

³ <http://codingdojo.org/>

И снова целью является асимптотическое приближение к совершенству. Упражнение повторяется снова и снова, чтобы научить ваш мозг реагировать, а пальцы двигаться. В ходе тренировки могут обнаружиться определенные улучшения и повышение эффективности ваших действий или самого решения.

Отработка нескольких ката помогает запомнить «горячие клавиши» и идиомы навигации. Она также хорошо работает при изучении таких дисциплин, как разработка через тестирование (TDD) и непрерывная интеграция (CI, Continuous Integration). Но самое важное — ката помогают закрепить в подсознании пары «задача/решение»; столкнувшись с этими задачами в реальном программировании, вы попросту будете знать, как они решаются.

Программист, как и мастер боевых искусств, должен знать разные ката и регулярно тренировать их, чтобы они не стерлись из памяти. Описания многих ката находятся по адресам <http://katas.softwarecraftsmanship.org> и <http://codekata.pragprog.com>. Некоторые из моих любимых ката:

- Игра в кегли: <http://butunclebob.com/ArticleS.UncleBob.TheBowling-GameKata>
- Простые числа: <http://butunclebob.com/ArticleS.UncleBob.ThePrime-Factors-Kata>
- Перенос текста: <http://thecleancoder.blogspot.com/2010/10/craftsman-62-dark-path.html>

Если вам захочется непростых испытаний, попробуйте выучить ката настолько хорошо, чтобы выполнять их под музыку. Сделать это очень трудно¹.

Вадза

Когда я занимался джиу-джитсу, большая часть времени в додзё проводилась за парной отработкой *вадза* (техники). Вадза напоминает ката с участием двух человек. Все действия точно запоминаются и воспроизводятся напарниками. Один играет роль нападающего, другой обороняется. Движения повторяются снова и снова, а спортсмены меняются ролями.

¹ <http://katas.softwarecraftsmanship.org/?p=71>

Программисты могут тренироваться аналогичным образом в игре, которая называется *пинг-понг*¹. Два напарника выбирают ката или простую задачу. Один пишет модульный тест, а другой должен заставить этот тест проходить. Затем они меняются ролями.

Если напарники выбирают стандартную ката, то результат заранее известен, а программисты отрабатывают работу с клавиатурой и мышью, а также четкость запоминания ката. С другой стороны, если напарники выбирают новую задачу, игра становится более интересной. Программист, пишущий тест, оказывает заметное влияние на то, как будет решаться задача. Также в его власти установка ограничений. Например, если программисты решают реализовать алгоритм сортировки, автор теста может легко установить ограничения по скорости и затратам памяти, которые усложнят задачу его партнера. В таком виде игра становится соревновательной... и интересной.

Рандори

Рандори — свободный спарринг. Во время тренировок по джиу-джитсу мы определяли боевые сценарии, а затем отрабатывали их. Иногда один человек должен был защищаться, тогда как остальные последовательно атаковали его. Иногда два и более нападающих выступали против одного защищающегося (обычно им был наш сэнсэй, который почти всегда выигрывал). Иногда одна пара вступала в единоборство с другой и т. д.

Имитация поединка не имеет нормального аналога в программировании; тем не менее во многих программных додзё играют в игру, которая тоже называется рандори. Она очень похожа на вадза для двоих напарников, решающих задачу. Однако в рандори играет много участников, а правила слегка изменены. На экране, проецируемом на стену, один участник пишет тест. Другой участник обеспечивает прохождение теста, а затем пишет следующий тест. Ход передается по кругу или участники просто выстраиваются в очередь. В любом случае такие упражнения бывают *очень* забавными.

Удивительно, насколько много можно узнать из таких упражнений. Вы получаете глубокое представление о том, как другие люди подходят к решению задач. Полученная информация помогает расширить ваш кругозор и повысить квалификацию.

¹ <http://c2.com/cgi/wiki?PairProgrammingPingPongPattern>

Расширение кругозора

Профессиональные программисты часто страдают от однообразия решаемых задач. Работодатели часто ограничиваются одним языком, платформой и предметной областью, в которой должен работать программист. Если вы не позаботитесь о расширении собственного кругозора, это может привести к нежелательному сужению резюме и менталитета. Такие программисты нередко оказываются неподготовленными к изменениям, периодически сотрясающим нашу отрасль.

Проекты с открытым кодом

Один из способов «держаться на переднем крае» позаимствован из практики адвокатов и врачей: выполняйте общественно-полезную работу, участвуя в проекте с открытым кодом. Таких проектов очень много; пожалуй, нет лучшего способа пополнить ваш творческий арсенал, чем поработать над проектом, который принесет пользу другим.

Если вы программируете на Java — поучаствуйте в проекте Rails. Если вы пишете большой объем кода C++ для своего работодателя, найдите проект Python и присоединитесь к нему.

Этика тренировки

Профессиональные программисты тренируются в личное время. Ваш работодатель не обязан заботиться о поддержании вашей квалификации или расширении вашего резюме. Пациенты не платят врачам, тренирующимся в наложении швов. Футбольные болельщики (обычно) не платят, чтобы посмотреть, как игроки бегают на тренировках. Зрители не платят за то, чтобы послушать, как музыканты играют гаммы. И работодатели программистов не обязаны оплачивать им время тренировок.

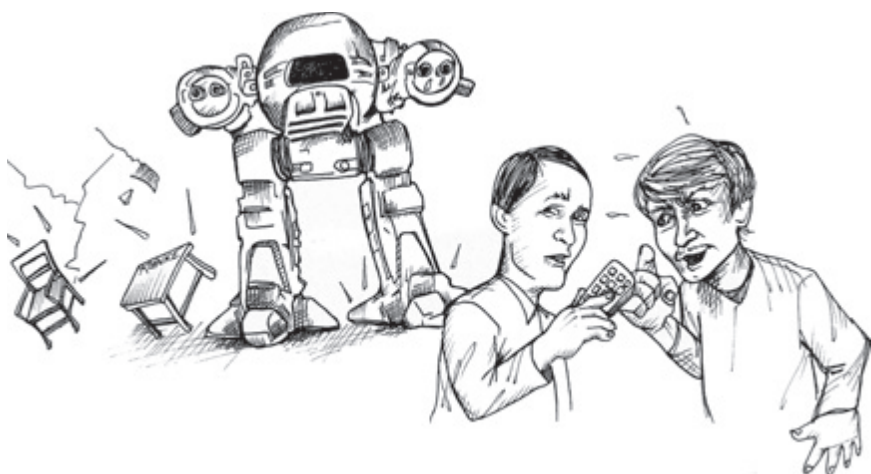
Так как тренировки проводятся в ваше личное время, вы не обязаны использовать языки и платформы своей основной работы. Выберите любой язык и проявите свои способности полиглота. Если вы работаете на платформе .NET, потренируйтесь в использовании Java или Ruby — дома или в обеденный перерыв.

Заключение

*Все профессионалы тренируются тем или иным образом. Они делают это, потому что хотят как можно лучше справиться с порученным делом. Более того, они делают это в свое личное время, поскольку знают, что ответственность за повышение их квалификации лежит на них самих, а не на их работодателях. За тренировки вам никто *не платит*. Вы занимаетесь ими, чтобы вам *платили* за основную работу — и платили *хорошо*.*

7

Приемочное тестирование



Роль профессионального разработчика подразумевает не только программирование, но и общение. Помните, что принцип «мусор на входе/мусор на выходе» применим и к программистам, поэтому профессиональные программисты следят за тем, чтобы их общение с другими участниками группы и бизнесменами было точным и плодотворным.

Передача требований

Одним из самых распространенных аспектов общения между программистами и бизнесом является разработка требований. Бизнесмены описывают то, что по их мнению им нужно, а программисты создают то, что по их мнению им описали.

По крайней мере так должно быть. На практике информационный обмен проходит чрезвычайно сложно, с высоким риском ошибок.

В 1979 году во время работы в Teradyne меня посетил Том, руководитель отдела установки и обслуживания. Он попросил меня научить его работать в текстовом редакторе ED-402, чтобы он мог организовать простую систему обработки заявок на устранение неисправностей.

ED-402 был коммерческим редактором, написанным для компьютера M365 — клона PDP-8. Это был очень мощный текстовый редактор со встроенным языком сценариев, который мы использовали для разнообразных операций с текстом.

Том не был программистом, но приложение, которое он себе представлял, было довольно простым. Том полагал, что я быстро научу его, а он сам напишет приложение. Я по наивности думал то же самое. В конце концов, язык сценариев представлял собой обычный макроязык команд редактирования, с простейшими условными и циклическими конструкциями.

Итак, мы сели за компьютер, и я спросил, что должно делать его приложение. Том начал с экрана ввода данных. Я показал, как создать текстовый файл со сценарными командами и как ввести в сценарий символьное представление команд редактирования. Но когда я взглянул ему в глаза, ответный взгляд был совершенно пустым. Он не понял ни единого слова из моего объяснения.

Тогда я впервые встретился с этим явлением. Для меня символьное представление команд редактирования было простым и естественным: например, для представления команды Control+B (команда перемещения курсора в начало текущей строки) в файл сценария просто включались символы ^B. Но Тому это казалось бессмысленным: он не мог перейти от редактирования файла к редактированию файла, который редактировал файл.

Том не был глуп. Наверное, он просто понял, что задача намного более сложная, чем ему казалось изначально, и не захотел тратить свое время и умственные силы на изучение такой хитроумной концепции, как использование редактора для управления редактором.

Мало-помалу я начал сам реализовывать приложение Тома, пока он сидел и наблюдал. Через 20 минут стало ясно, что он уже думает не о том, как сделать все самому, а о том, как объяснить *мне*, что именно *ему* нужно.

На это ушел целый день. Том описывал некоторую возможность, а я реализовывал ее, пока он наблюдал. Рабочий цикл составлял не более

5 минут, поэтому ему не нужно было отходить и заниматься чем-то другим. Он просил меня сделать X, и через 5 минут реализация X работала.

Часто Том рисовал то, что он хотел, на листке бумаги. Некоторые из его требований было трудно реализовать средствами ED-402; тогда я предлагал что-то другое. В конечном итоге мы договаривались до чего-то, что могло работать, и я реализовывал согласованный вариант. Но потом мы опробовали его, и Том менял свое решение. Он говорил что-нибудь вроде: «Да, но это не совсем то, что мне нужно. Давай попробуем иначе». Час за часом мы пробовали, экспериментировали и придавали приложению нужную форму. Мы опробовали одно, потом другое, потом третье. Вскоре мне стало абсолютно ясно, что Том — скульптор, а я — инструмент в его руках.

В конечном итоге Том получил желаемое приложение, но не имел ни малейшего понятия о том, как построить следующее приложение самостоятельно. С другой стороны, я получил важный урок относительно того, как заказчик определяет свои потребности. Оказалось, что его представление о функциональности приложения часто не переживает контакта с компьютером.

Преждевременная точность

И бизнесмены, и программисты часто попадают в ловушку преждевременной точности. Бизнесмены хотят точно знать размер прибыли, прежде чем давать согласие на проект. Разработчики хотят точно знать, что им предстоит сделать, прежде чем давать прогнозы по проекту. Обе стороны желают иметь точную информацию, которой быть заведомо не может, причем часто они готовы потратить целое состояние на попытки ее получения.

Принцип неопределенности

К сожалению, на бумаге все выглядит совсем не так, как в рабочей системе. Когда бизнесмены видят, что получается из реализации их требований в системе, они понимают, что хотели совсем не этого. А иногда уже после знакомства с реализацией они начинают лучше понимать, что же им на самом деле нужно — чаще всего не то, что они видят.

В игру вступает «эффект наблюдателя», или принцип неопределенности. Когда вы демонстрируете новую возможность бизнес-участникам,

у них появляется больше информации, чем было прежде, а эта новая информация влияет на их восприятие системы в целом. В конечном итоге чем точнее формулируете свои требования, тем быстрее они теряют актуальность в ходе реализации системы.

Стремление к точности оценки

Разработчики тоже попадают в ловушку точности. Они знают, что должны выдать оценку системы, и часто думают, что их оценка должна быть точной. Ничего подобного!

Во-первых, даже с идеальной информацией в ваших оценках будет наблюдаться огромный разброс. Во-вторых, принцип неопределенности разносит раннюю точность в пух и прах. Требования будут изменяться, а это приведет к тому, что точность потеряет смысл.

Профессиональные разработчики понимают, что оценки могут (и должны) базироваться на требованиях с низкой точностью. Они понимают, что оценка — это именно оценка. Они всегда включают в свои оценки диапазоны погрешности, чтобы бизнес-участники понимали наличие неопределенности (см. главу 10).

Поздняя неоднозначность

Как решить проблему преждевременной точности? Откладывая точные определения на как можно более поздний срок. Профессиональные разработчики не формулируют требования до того, как будут готовы к разработке. Однако такой подход может привести к другому несчастью: поздней неоднозначности.

Между ключевыми участниками проекта часто возникают разногласия. В таких случаях часто бывает проще «заговорить» проблему, чем решать ее. Они находят такую формулировку требования, с которой будут согласны все, без реального разрешения спора. Однажды я слышал, как Том Демарко сказал: «неоднозначность в документе с требованиями появляется из-за разногласий между участниками¹».

Конечно, неоднозначность не всегда появляется из-за споров или разногласий. Иногда ключевые участники проекта просто считают, что читатели требований знают, что они имеют в виду. Возможно, их

¹ XP Immersion 3, May, 2000. <http://c2.com/cgi/wiki?TomsTalkAtXpImmersionThree>

формулировки абсолютно ясны им самим в их контексте, но означают нечто совершенно иное для программиста, читающего документ. Подобная контекстная неоднозначность также может возникать и при личном общении заказчиков с программистами.

Сэм (ключевой участник проекта): «Так, еще нужно организовать резервное копирование журнальных файлов».

Пола: «С какой частотой?»

Сэм: «Ежедневно».

Пола: «Понятно. И где будут храниться резервные копии?»

Сэм: «В смысле?»

Пола: «Наверное, они должны сохраняться в определенном подкаталоге?»

Сэм: «Да, пожалуй».

Пола: «И как он будет называться?»

Сэм: «Может, backup?»

Пола: «Да, нормально. Значит, журнал будет ежедневно сохраняться в каталоге backup. В какое время?»

Сэм: «Ежедневно».

Пола: «Нет, в какое именно время суток?»

Сэм: «В любое».

Пола: «В полдень?»

Сэм: «Нет, только не во время торгов. Лучше в полночь».

Пола: «Хорошо, пусть будет полночь».

Сэм: «Отлично, спасибо!»

Пола: «Всегда рада помочь».

Позднее Пола рассказывает о задаче своему коллеге Питеру.

Пола: «Значит, журнал должен копироваться в подкаталог с именем backup ежедневно в полночь».

ПИТЕР: «Понятно. А как будет называться файл?»

Пола: «Я думаю, *log.backup* подойдет».

ПИТЕР: «Хорошо».

В другом офисе Сэм общается по телефону с заказчиком.

Сэм: «Да, да, журнальные файлы будут сохраняться».

Карл: «Очень важно, чтобы ни один журнал не был потерян. Нельзя исключать, что нам придется когда-нибудь вернуться к любому из этих файлов, даже через месяцы или годы — в случае сбоя питания, какой-нибудь катастрофы, судебного спора и т. д.».

Сэм: «Не беспокойтесь, я только что говорил с Полой. Журналы будут сохраняться в каталоге с именем backup ежедневно в полночь».

Карл: «Хорошо, это подойдет».

Вы заметили неоднозначность? Заказчик ждет, что сохраняться будут все журналы, а Пола думает, что достаточно сохранить журнал за последний день. Если заказчик захочет вернуться к резервной копии месячной давности, он найдет только данные последнего дня.

В данном случае и Пола, и Сэм оказались не на высоте. Профессиональные разработчики (и менеджеры) должны следить за тем, чтобы из требований была исключена всякая неоднозначность.

Это *сложная* задача, и мне известен только один способ ее решения.

Приемочные тесты

Термин «приемочные тесты» перегружен множеством значений. Одни полагают, что речь идет о тестах, выполняемых пользователями перед приемкой очередной версии продукта. Другие понимают под этим термином контроль качества. В этой главе под термином «приемочные тесты» понимаются тесты, написанные при совместном участии заинтересованных сторон и программистов для проверки выполнения требований.

Что такое «выполнено»?

Одна из самых распространенных неоднозначностей, с которыми сталкиваются профессиональные разработчики, связана с самим понятием «выполнения». Когда разработчик говорит, что он выполнил свою задачу, что он имеет в виду? Что он готов передать свой код в эксплуатацию с полной уверенностью? Что его код готов к прохождению контроля

качества? А может, что он дописал свой код и даже смог один раз выполнить его, но еще не подвергал тестированию?

Я работал с группами, имевшими разные представления о терминах «выполнено» и «готово». В одной группе использовались термины «сделано» и «совсем сделано». У профессиональных разработчиков есть только одно определение: выполнено — значит выполнено. Сделано — значит, что весь код написан, все тесты пройдены, служба контроля качества и ключевые участники приняли результат. Сделано.

Но как добиться такого уровня выполнения, не снижая темпа перехода между итерациями? Нужно создать набор автоматизированных тестов, прохождение которых означает выполнение всех перечисленных условий! Если приемочные тесты вашей подсистемы проходят успешно, значит, работа выполнена.

Профессиональные разработчики расширяют определение требований до автоматизированных приемочных тестов. Они общаются с ключевыми участниками и специалистами по контролю качества, стремясь к тому, чтобы автоматизированные тесты полностью отражали определение «выполнения».

Сэм: «Так, еще нужно организовать резервное копирование журнальных файлов».

Пола: «С какой частотой?»

Сэм: «Ежедневно».

Пола: «Понятно. И где будут храниться резервные копии?»

Сэм: «В смысле?»

Пола: «Наверное, они должны сохраняться в определенном подкаталоге?»

Сэм: «Да, пожалуй».

Пола: «И как он будет называться?»

Сэм: «Может, backup?»

Том (тестер): «Погодите, backup — слишком общее название. Что именно будет храниться в этом каталоге?»

Сэм: «Резервные копии».

Том: «Резервные копии чего?»

Сэм: «Журнальных файлов».

Пола: «Но ведь журнальный файл всего один?»

Сэм: «Нет, их много. По одному на каждый день».

Том: «То есть у нас будет один *активный* журнал и много резервных копий?»

Сэм: «Конечно».

Пола: «О! А я думала, копии будут постоянно замещаться».

Сэм: «Нет, заказчик хочет, чтобы они хранились неограниченно долго».

Пола: «Для меня это новость. Хорошо, что вовремя разобрались».

Том: «Имя подкаталога должно сообщать, что именно в нем хранится».

Сэм: «Там будут храниться старые неактивные журналы».

Том: «Тогда мы назовем его `old_inactive_logs`».

Сэм: «Отлично».

Том: «И когда будет создаваться этот каталог?»

Сэм: «В смысле?»

Пола: «Каталог будет создаваться при запуске системы, но только в том случае, если он не существует».

Том: «Понятно, это наш первый тест. Я запускаю систему и проверяю, создан ли каталог `old_inactive_logs`. Затем я добавляю файл в этот каталог, завершаю работу, снова запускаю систему и проверяю, что каталог и файл находятся на положенном месте».

Пола: «На выполнение этого теста уйдет много времени. Инициализация системы уже занимает около 20 секунд, и время только растет. К тому же я не хочу собирать систему заново при каждом запуске приемочных тестов».

Том: «Что ты предлагаешь?»

Пола: «Создадим класс `SystemStarter`. Основная программа будет загружать его с группой объектов `StartupCommand`, построенных на базе паттерна «Команда». Затем во время запуска системы `SystemStarter` просто приказывает всем объектам `StartupCommand` выполнить свои команды. Один из этих объектов `StartupCommand` создает старый каталог `old_inactive_logs`, но только в том случае, если он не существует».

Том: «Ладно, тогда мне остается только протестировать этого потомка `StartupCommand`. Я напишу для этого простой тест `FitNesse`».

(Том идет к доске.)

«Первая часть будет выглядеть примерно так»:

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does not exist
when the command is executed
then the old_inactive_logs directory should exist
and it should be empty
```

«А вторая часть будет примерно такой»:

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory exists
and that it contains a file named x
When the command is executed
Then the old_inactive_logs directory should still exist
and it should still contain a file named x
```

Пола: «Да, этого должно хватить».

Сэм: «А это все действительно необходимо?»

Пола: «Сэм, какое из этих двух утверждений недостаточно важно для проверки?»

Сэм: «Просто я хочу сказать, что для написания всех этих тестов потребуется немало лишней работы».

Том: «Да, но не больше, чем для написания плана ручного тестирования. И *намного* меньше, чем для многократного выполнения тестов вручную».

Взаимодействие сторон

Основные цели приемочных тестов — взаимодействие сторон, ясность и точность требований. В результате согласования приемочных тестов разработчики, ключевые участники и тестеры достигают понимания планируемого поведения системы. Достижение такой ясности — обязанность всех сторон. Профессиональные разработчики считают своей обязанностью работать с ключевыми участниками и тестерами и принимать меры к тому, чтобы все стороны знали, что именно они собираются построить.

Автоматизация

Приемочные тесты *всегда* должны быть автоматизированными. В других моментах жизненного цикла программных продуктов находится

место для ручного тестирования, но *такие* тесты никогда не должны выполняться вручную. Причина проста: затраты.

Взгляните на рис. 7.1. Руки, которые вы на нем видите, принадлежат менеджеру по контролю качества крупной интернет-компании. В документе, который он держит, содержится *оглавление* его плана *ручного* тестирования. Он оплачивает целую армию тестеров из других стран, которые выполняют этот план каждые шесть недель. Каждое тестирование обходится примерно в миллион долларов. Он только что вернулся с собрания, на котором руководитель сообщил, что бюджет тестирования будет урезан примерно на 50%, а теперь спрашивает меня: «Какую половину этих тестов не нужно выполнять?»

Назвать происходящее катастрофой значило бы не сказать ничего. Затраты на ручное тестирование настолько велики, что фирма решила отказаться от него — и просто жить дальше, *не зная, работает ли половина ее продукта!*

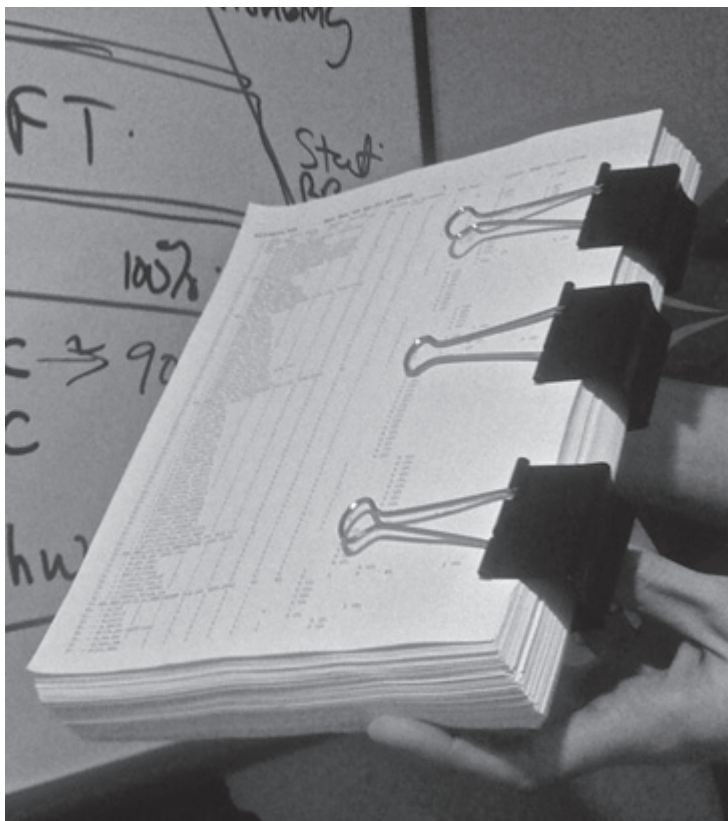


Рис. 7.1. План ручного тестирования

Профессиональные разработчики не допускают возникновения подобных ситуаций. Затраты на проведение автоматизированных тестов настолько малы по сравнению с затратами на ручное тестирование, что написание сценариев, запускаемых вручную, не имеет никакого экономического смысла. Профессиональные разработчики считают своей обязанностью проследить за тем, чтобы приемочные тесты проводились в автоматизированном режиме.

Существует множество программных инструментов (как коммерческих, так и с открытым кодом), автоматизирующих приемочные тесты. FitNesse, Cucumber, cucumber4duke, robot framework и Selenium — этот список далеко не полон. Во всех этих инструментах автоматизированные тесты определяются в такой форме, что даже не-программисты могут читать их, понимать и даже создавать.

Дополнительная работа

Замечание Сэма насчет лишней работы понятно. На первый взгляд *кажется*, что написание подобных приемочных тестов потребует значительных усилий. Но взглянув на рис. 7.1, мы видим, что эту работу неправильно называть «лишней». Написание тестов всего лишь является работой по определению спецификации системы. Только на таком уровне детализации мы, программисты, понимаем, что означает «выполненная работа». Только на таком уровне детализации ключевые участники проекта могут убедиться в том, что система, за которую они платят, делает то, что требуется. И только на таком уровне детализации возможна успешная автоматизация тестирования. Так что не стоит рассматривать эти тесты как лишнюю работу — лучше рассматривайте их как значительную экономию времени и денег. Тесты предотвратят ошибки в реализации системы и помогут узнать, когда ваша работа закончена.

Кто и когда пишет приемочные тесты?

В идеальном мире ключевые участники проекта и служба контроля качества сотрудничают в написании этих тестов, а разработчики проверяют их на логическую непротиворечивость. В реальном мире ключевые участники редко находят время или желание погружаться на нужный уровень детализации, поэтому они перепоручают эту обязанность бизнес-аналитикам, специалистам по контролю качества или даже разработчикам. Если окажется, что тесты должны писать

разработчики, по крайней мере проследите за тем, чтобы это были *не те* разработчики, которые занимаются реализацией тестируемой функциональности.

Бизнес-аналитики обычно пишут «оптимистичные» версии тестов, потому что эти тесты описывают аспекты, обладающие коммерческой ценностью. Служба контроля качества обычно пишет «пессимистичные» тесты с проверкой всевозможных граничных условий, исключений и аномальных случаев. И это понятно, потому что задача контроля качества — думать о том, что может пойти не так.

Согласно принципу «поздней точности» приемочные тесты следует писать как можно позднее, обычно за несколько дней до реализации. В проектах на базе гибких методологий тесты пишутся *после* выбора функций для следующей итерации или спринта.

Первые приемочные тесты должны быть готовы к первому дню итерации. Новые тесты должны появляться ежедневно вплоть до середины итерации, когда готовы должны быть все тесты. Если к середине итерации некоторые приемочные тесты еще не готовы, переведите нескольких разработчиков на их срочную доработку. Если это происходит часто, включите в команду дополнительных бизнес-аналитиков и/или специалистов по контролю качества.

Роль разработчика

Работа по реализации некоторого аспекта системы начинается тогда, когда для этого аспекта готовы все приемочные тесты. Разработчики выполняют приемочные тесты и убеждаются в том, что те не проходят. Затем они связывают приемочные тесты с системой и начинают работать над реализацией нужного аспекта, обеспечивая прохождение приемочных тестов.

Пола: «Питер, сможешь мне?»

ПИТЕР: «Разумеется, а что случилось?»

Пола: «Вот наш приемочный тест. Как видишь, он не проходит».

```
given the command LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does not exist
when the command is executed
```

```
then the old_inactive_logs directory should exist  
and it should be empty1
```

ПИТЕР: «Да, все результаты красные. Ни один сценарий еще не написан. Давай я напишу первый»:

```
|scenario|given the command _|cmd|  
|create command|@cmd|
```

ПОЛА: «А у нас уже есть операция createCommand»?

ПИТЕР: «Да, в пакете CommandUtilitiesFixture, который я написал на прошлой неделе».

ПОЛА: «Хорошо, давай запустим тест».

ПИТЕР: (запускает тест) «Первая строка стала зеленой, переходим к следующей».

Не обращайте внимания на все эти сценарии (scenarios) и оснастки (fixtures) — это всего лишь служебный код, который необходимо написать для связывания тестов с тестируемой системой. Достаточно сказать, что все перечисленные инструменты предоставляют те или иные средства поиска по шаблону для распознавания и разбора инструкций теста и последующего вызова функций, передающих данные тестируемой системе. Все это делается достаточно просто, а полученные сценарии и оснастки могут использоваться в разных тестах.

Суть в том, что разработчик должен связать приемочные тесты с системой, а затем обеспечить их прохождение.

Обсуждение тестов и пассивно-агрессивная позиция

Авторы тестов — люди, и они тоже допускают ошибки. Иногда при переходе к реализации становится очевидно, что тест выглядит бессмысленно. Тесты бывают слишком запутанными или громоздкими.

¹ Исходное условие: команда LogFileDirectoryStartupCommand

Исходное условие: до выполнения команды каталог old_inactive_logs существует

И содержит файл с именем х

После выполнения команды каталог old_inactive_logs должен существовать и должен содержать файл с именем х. — *Примеч. пер.*

Они могут базироваться на нелепых предпосылках. А иногда они попросту неверны. Все это может быть весьма неприятно, если вы — разработчик, который должен обеспечить прохождение теста.

Ваша задача как профессионального разработчика — обсудить ситуацию с автором теста для его улучшения. *Никогда* не выбирайте пассивно-агрессивную позицию, когда вы говорите себе: «Как написано в тесте, так я и сделаю».

Помните: профессионал обязан помочь своей группе создать программу настолько качественную, насколько это возможно. А это означает, что все должны уделять внимание возможным ошибкам и промахам коллег и совместно работать над их исправлением.

Пола: «Том, с этим тестом что-то не то».

ensure that the post operation finishes in 2 seconds.

Том: «По-моему, все нормально. Наше требование гласит, что пользователи не должны ждать больше двух секунд. В чем проблема?»

Пола: «Проблема в том, что мы можем гарантировать выполнение требования только в статистическом смысле».

Том: «По-моему, это только слова. В требованиях ясно сказано: две секунды».

Пола: «Верно, и мы можем обеспечить этот результат в 99,5% случаев».

Том: «Пола, в требовании этого нет».

Пола: «Мы живем в реальном мире. Я не могу предоставить других гарантий».

Том: «Сэм будет в бешенстве».

Пола: «Вообще-то я с ним уже пару раз обсуждала эту тему. Он согласен, если типичная продолжительность операции будет не более двух секунд».

Том: «Ну и как мне писать этот тест? Я же не могу сказать: „операция обычно заканчивается за две секунды“».

Пола: «Сформулируй на статистическом уровне».

Том: «Предлагаешь выполнить операцию тысячу раз и проверить, что она заняла более двух секунд в пяти и менее случаях? Абсурд».

Пола: «Нет, это займет слишком много времени. А как насчет этого?»

execute 15 post transactions and accumulate times.
ensure that the Z score for 2 seconds is at least 2.57¹

Том: «А что еще за z-показатель?»

Пола: «Так, статистика. А как тебе такая формулировка?»

execute 15 post transactions and accumulate times.
ensure odds are 99.5% that time will be less than 2 seconds²

Том: «Да, это уже понятнее, но можно ли доверять математике?»

Пола: «Я обязательно включу все промежуточные вычисления в отчет по тестированию, чтобы ты мог проверить вычисления, если у тебя останутся сомнения».

Том: «Хорошо, меня это устроит».

Приемочные тесты и модульные тесты

Не путайте приемочные тесты с *модульными* (unit tests). Модульные тесты пишутся *программистами для программистов*. Они представляют собой формальные архитектурные документы с описанием нижнего уровня структуры и поведения кода. Их читателями являются не бизнесмены, а программисты.

Приемочные тесты создаются *бизнесменами для бизнесменов* (даже если в конечном итоге их пишете вы, разработчик). Они представляют собой формальные описания требований, определяющие поведение системы с точки зрения бизнеса. Их читателями являются бизнесмены и программисты.

Возможно, у кого-то возникнет соблазн избавиться от лишней работы, предположив, что тесты двух видов избыточны. Но хотя модульные и приемочные тесты часто тестируют одно и то же, никакой избыточности в них нет.

¹ Выполнить 15 операций отправки данных и просуммировать время.

Убедиться в том, что z-показатель для двух секунд не менее 2,57. — *Примеч. пер.*

² Выполнить 15 операций отправки данных и просуммировать время.

Убедиться, что вероятность того, что операция займет не более двух секунд, составляет не менее 99,5%. — *Примеч. пер.*

Во-первых, даже если они тестируют одно и то же, при этом используются разные пути и механизмы. Модульные тесты углубляются во внутреннюю реализацию системы и вызывают методы конкретных классов. Приемочные тесты обращаются к системе на значительно более высоком уровне — уровне API или даже уровне пользовательского интерфейса. Таким образом, пути выполнения этих тестов сильно различаются.

Но настоящая причина, по которой эти тесты нельзя назвать избыточными, заключается в том, что тестирование не является их главной функцией. Тот факт, что они что-то проверяют, вторичен. Модульные и приемочные тесты прежде всего являются документами и лишь потом — тестами. Их главная цель — формальное документирование архитектуры, структуры и поведения системы. Автоматическая проверка архитектуры, структуры и поведения чрезвычайно полезна, но истинной целью является именно документирование.

Графические интерфейсы и другие сложности

Графический интерфейс трудно определить заранее. Теоретически это возможно, но редко удастся сделать хорошо. Дело в том, что эстетика — предмет субъективный, а следовательно, переменчивый. Разработчики любят *повозиться* со своими графическими интерфейсами, доводить их до ума и шлифовать. Они хотят использовать разные шрифты, цвета, макеты и схемы выполнения операций. Графические интерфейсы находятся в постоянном развитии.

Это обстоятельство усложняет написание приемочных тестов для графических интерфейсов. Задача решается таким проектированием системы, при котором графический интерфейс можно было бы рассматривать как API, а не как набор кнопок, ползунков, таблиц и меню. На первый взгляд кажется странно, но в действительности речь идет просто о качественном проектировании.

В области проектирования программных систем существует принцип, называемый принципом единственной обязанности (SRP, Single Responsibility Principle). Он гласит, что при проектировании следует разделять аспекты системы, которые могут изменяться по разным причинам, и группировать вместе те аспекты, которые изменяются по одним и тем же причинам. Графические интерфейсы не являются исключением.

Макет, формат и схема выполнения операций в графическом интерфейсе могут меняться по эстетическим причинам и по соображениям эффективности, но базовая функциональность графического интерфейса остается неизменной. Таким образом, при написании приемочных тестов для графического интерфейса следует пользоваться базовыми абстракциями, которые изменяются относительно редко.

Например, на странице могут находиться несколько кнопок. Вместо создания тестов, имитирующих нажатия кнопок по их положению на странице, следует предусмотреть возможность имитации нажатий с идентификацией по именам. Еще лучше, если у каждой кнопки будет уникальный идентификатор. При написании теста намного приятнее выбирать кнопку с идентификатором `ok_button`, чем кнопку в столбце 3 строки 4 таблицы элементов.

Выбор интерфейса для тестирования

И все же лучше писать тесты, которые активизируют функции тестируемой системы через API, а не через графический интерфейс. При этом должен использоваться тот же API, который используется графическим интерфейсом. В этом нет ничего нового: специалисты в области проектирования десятилетиями говорили нам, что графический интерфейс нужно отделять от бизнес-логики.

Тестирование через графический интерфейс всегда создает проблемы (если вы не ограничиваетесь тестированием одного лишь графического интерфейса). Дело в том, что графический интерфейс с большой вероятностью изменится, а это делает тесты весьма непрочными. Когда каждое изменение интерфейса нарушает работу тысячи тестов, вы либо начинаете отказываться от тестов, либо перестаете изменять графический интерфейс. Ни один из этих вариантов хорошим не назовешь. Итак, пишите тесты бизнес-логики так, чтобы они проходили через API, находящийся за графическим интерфейсом.

Некоторые приемочные тесты определяют поведение самого графического интерфейса. Естественно, такие тесты должны проходить через графический интерфейс. Однако они не тестируют бизнес-логику, а следовательно, не требуют ее связывания с графическим интерфейсом. По этой причине на время тестирования самого графического интерфейса лучше изолировать его от бизнес-логики и заменить последнюю заглушками.

Ограничьтесь минимальным объемом тестов графического интерфейса. Они слишком непрочны из-за изменчивости графического интерфейса. Чем больше у вас тестов графического интерфейса, тем меньше вероятность того, что они останутся неизменными.

Непрерывная интеграция

Проследите за тем, чтобы все модульные и приемочные тесты запускались несколько раз в день в механизме непрерывной интеграции. Этот механизм должен инициироваться вашей системой управления исходным кодом. Каждый раз, когда кто-то вносит в базу новый модуль, механизм непрерывной интеграции должен инициировать сборку с последующим запуском всех тестов в системе. Результаты запуска должны рассылаться всем участникам группы.

Стоп-сигнал

Очень важно, чтобы тесты непрерывной интеграции все время проходили успешно. Они никогда не должны завершаться отказом. В случае отказа вся группа прекращает заниматься текущими делами и направляет все усилия на то, чтобы обеспечить успешное прохождение всех тестов. Сборка в системе с непрерывной интеграцией должна рассматриваться как экстренное событие, своего рода «стоп-сигнал».

Я общался с группами, которые недостаточно серьезно относились к отказам в тестах. Такие группы обычно были «слишком заняты», чтобы решать проблему немедленно, поэтому тесты откладывались в сторону до лучших времен. В одном случае нерабочие тесты были попросту исключены из сборки, потому что программистов раздражали сообщения об отказах. Позднее, уже после сдачи продукта заказчику, они вдруг вспомнили, что забыли вернуть тесты в сборку. Это выяснилось уже после того, как разгневанный заказчик забросал их сообщениями об ошибках.

Заключение

Передать информацию о подробностях сложно. Это в полной мере относится к программистам и ключевым участникам проекта, обсуждающим подробности приложения. Слишком легко каждой из сторон

махнуть рукой и считать, что другая сторона ее понимает. Слишком часто стороны соглашаются с тем, что они поняли друг друга, и расходятся с совершенно разными убеждениями.

Мне известен только один способ эффективного исключения коммуникационных ошибок в общении программистов с ключевыми участниками проектов — написание автоматизированных приемочных тестов. Эти тесты формализованы, полностью однозначны и всегда остаются синхронизированными с приложением. Они являются идеальным документом, определяющим требования к проекту.

8

Стратегии тестирования



Профессиональные разработчики тестируют свой код. Однако тестирование не сводится к написанию нескольких модульных или приемочных тестов. Написание этих тестов — дело полезное, но отнюдь не достаточное. Любой группе профессиональных разработчиков нужна хорошая *стратегия тестирования*.

В 1989 году я работал над первой версией Rational Rose. Каждый месяц или около того начальник службы контроля качества объявлял день «охоты за ошибками». Все участники проекта, от программистов до начальников, от секретарей до администраторов баз данных, садились за Rose и пытались вызвать сбой в программе. За разные типы ошибок присуждались призы. Ошибка, приводящая к аварийному завершению

приложения, могла быть награждена обедом для двоих. Тот, кто обнаруживал больше всего ошибок, мог выиграть поездку на выходные в Монтеррей.

Контроль качества не должен находить дефекты

Я уже говорил это прежде и скажу снова. Несмотря на то что в вашей компании может существовать отдельная группа контроля качества, занимающаяся тестированием программных продуктов, группа разработки должна стремиться к тому, чтобы контроль качества не выявлял никаких дефектов.

Конечно, вряд ли эта цель будет достигаться постоянно. В конце концов, когда группа умных людей решительно и непреклонно стремится выявить все дефекты и недостатки продукта, она наверняка чего-нибудь найдет. И все же каждый раз, когда служба контроля качества что-нибудь обнаруживает, это должно стать тревожным сигналом для разработчиков. Они должны спросить себя, как это произошло, и принять меры для предотвращения подобных инцидентов в будущем.

Служба контроля качества — часть команды

Из предыдущего раздела может сложиться впечатление, что служба контроля качества и группа разработки противодействуют друг другу, что их отношения имеют антагонистический характер. Этого быть не должно. Служба контроля качества и группа разработки совместно работают для повышения качества системы, а оптимальная роль для специалистов по контролю качества должна заключаться в создании спецификаций и описании характеристик системы.

Создание спецификаций

Служба контроля качества должна работать совместно с бизнес-стороной для создания автоматизированных приемочных тестов, которые представляют собой истинную спецификацию и документированные требования к системе. Последовательно, от итерации к итерации, они получают информацию о требованиях со стороны бизнеса и преобразуют их в тесты, которые описывают желаемое поведение системы для разработчиков (см. главу 7, «Приемочное тестирование»). Как

правило, бизнес-сторона создает «оптимистичные» тесты, а служба контроля качества — «пессимистичные» тесты с проверкой всевозможных граничных условий, исключений и аномальных случаев.

Описание характеристик системы

Еще одна роль службы контроля качества — применение дисциплины исследовательского тестирования¹ для описания характеристик истинного поведения работающей системы и передачи информации о нем группе разработки и бизнес-стороне. В этой роли служба контроля качества *не занимается* интерпретацией требований — она идентифицирует фактическое поведение системы.

Пирамида автоматизации тестирования

Профессиональные разработчики для создания модульных тестов обычно применяют методологию разработки через тестирование (TDD, Test Driven Development). Группы профессиональных разработчиков используют приемочные тесты для составления спецификации своей системы и механизм непрерывной интеграции (см. главу 7, с. 122) для предотвращения регрессии. Однако эти тесты составляют лишь часть картины. Какими бы полезными ни были модульные и приемочные тесты, нам также понадобятся тесты более высокого уровня, которые будут следить за тем, чтобы контроль качества не обнаруживал никаких дефектов. На рис. 8.1 изображена пирамида автоматизации тестирования² — графическое представление всевозможных тестов, необходимых при профессиональной организации разработки.

Модульные тесты

У основания пирамиды располагаются модульные тесты. Они пишутся программистами для программистов на языке программирования

¹ http://www.satisfice.com/articles/what_is_et.shtml

² Mike Cohn, *Succeeding with Agile*, Boston, MA: Addison-Wesley, 2009.



Рис. 8.1. Пирамида автоматизации тестирования

системы. Целью этих тестов является определение спецификации системы на самом нижнем уровне. Выполнение тестов в контексте непрерывной интеграции помогает проследить за тем, что исходные намерения программистов были успешно реализованы.

Модульные тесты должны обеспечивать покрытие кода, настолько близкое к 100%, насколько это возможно. В общем случае покрытие должно превышать 90%, причем оно должно быть *реальным* — в отличие от фальшивых тестов, которые выполняют код без проверки утверждений относительно его поведения.

Компонентные тесты

В эту категорию входит часть приемочных тестов, упоминавшихся в прошлой главе. Обычно эти тесты пишутся для отдельных компонентов системы. В компонентах системы инкапсулируются бизнес-правила, поэтому тесты компонентов становятся приемочными тестами для бизнес-правил.

Как показано на рис. 8.2, компонентный тест изолирует отдельный компонент системы. Он передает компоненту входные данные, получает от него результаты и проверяет их на соответствие входным данным.

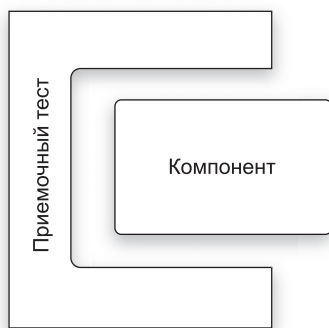


Рис. 8.2. Компонентный приемочный тест

Для изоляции остальных компонентов системы применяются макеты и тест-дублиеры.

Компонентные тесты пишутся специалистами по контролю качества и бизнес-стороной при помощи разработчиков. Они формируются в средах компонентного тестирования — таких как FitNesse, JBehave или Cucumber. (Компоненты графического интерфейса тестируются в специализированных средах вроде Selenium или Watir.) Тесты должны быть написаны так, чтобы бизнес-сторона могла читать и интерпретировать их (или даже создавать своими силами).

Компонентные тесты обеспечивают покрытие примерно половины системы. Они в основном ориентированы на «оптимистичные» ситуации и наиболее очевидные граничные условия, исключения и аномальные ситуации. Подавляющее большинство «пессимистичных» ситуаций покрывается модульными тестами, и на уровне компонентных тестов они не имеют смысла.

Интеграционные тесты

Тесты этой категории имеют смысл только в больших системах с множеством компонентов. Как видно из рис. 8.3, эти тесты группируют компоненты и тестируют взаимодействия между ними. Другие компоненты системы, как обычно, изолируются при помощи подходящих макетов и тест-дублеров.

Интеграционные тесты являются «хореографическими»: они не тестируют бизнес-правила, а лишь проверяют, насколько хорошо компоненты

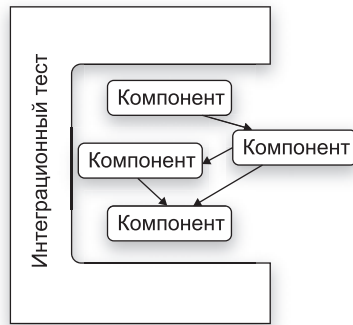


Рис. 8.3. Интеграционный тест

группы «танцуют» друг с другом. Они проверяют правильность связывания компонентов и обмена информацией между ними.

Интеграционные тесты обычно пишутся системными архитекторами (или ведущими проектировщиками) системы. Они проверяют основательность архитектурной структуры системы. Именно на этом уровне обычно встречаются тесты производительности и пропускной способности.

Интеграционные тесты обычно пишутся на том же языке и в той же среде, что и компонентные тесты. Они редко выполняются в контексте непрерывной интеграции, потому что обычно занимают слишком много времени. Вместо этого тесты выполняются периодически (еженежно, еженедельно и т. д.) с интервалом, определяемым их авторами.

Системные тесты

Автоматизированные тесты, проверяющие работу всей интегрированной системы. По сути, они представляют собой предельный случай интеграционных тестов. Системные тесты не проверяют бизнес-правила напрямую. Вместо этого они проверяют, что компоненты системы правильно связаны друг с другом, а взаимодействие между ними проходит по исходному плану. Тесты производительности и пропускной способности обычно относятся к этой категории.

Эти тесты пишутся системными архитекторами и ведущими специалистами с технической стороны. Как правило, они пишутся на том же языке и в той же среде, что и интеграционные тесты пользовательского

интерфейса. Системные тесты выполняются относительно редко (в зависимости от продолжительности их выполнения), но чем чаще — тем лучше.

Системные тесты покрывают 10% системы. Это объясняется тем, что они предназначены для проверки правильности не поведения системы, а ее конструкции. Правильность поведения нижележащего кода и компонентов уже была проверена на нижних уровнях пирамиды.

Исследовательские тесты

В этой категории разработчикам приходится поработать за клавиатурой и монитором. Исследовательские тесты не автоматизируются и *не оформляются в сценарии*. Они предназначены для исследования системы на предмет выявления неожиданного поведения и подтверждения поведения ожидаемого. И для исследования системы в этом направлении необходим человеческий мозг с человеческим же творческим мышлением. Построение письменного плана для тестирования такого рода противоречит его цели.

В некоторых группах имеются специалисты для выполнения этой работы. В других группах попросту объявляется «день охоты за ошибками», когда максимальное количество людей, включая начальников, секретарей, программистов, тестеров и авторов технической документации, «издеваются» над системой в попытках «сломать» ее. Покрытие кода не входит в число целей исследовательского тестирования. Мы вовсе не собираемся проверять все бизнес-правила и все пути выполнения. Целью является проверка нормального поведения системы в условиях использования человеком, а также творческое выявление как можно большего количества «странностей».

Заключение

TDD — мощная методология, а приемочные тесты — ценное средство выражения и проверки соблюдения требований. Но они являются лишь частью общей стратегии тестирования. Для достижения цели «Контроль качества не должен находить дефекты» группа разработки должна работать в тесном взаимодействии со службой контроля качества

для создания иерархии модульных, компонентных, интеграционных, системных и исследовательских тестов. Тесты следует выполнять как можно чаще, чтобы обеспечить максимальную обратную связь и гарантировать постоянную «чистоту» системы.

9

Планирование



Восемь часов — на удивление короткий промежуток времени. Это всего лишь 480 минут, или 28 800 секунд. Вы как профессионал должны использовать эти драгоценные секунды как можно более эффективно. Какую стратегию вы выберете, чтобы избежать напрасных затрат своего времени? Как организовать эффективное управление временем?

В 1986 году я жил в Литл-Сэндхерсте (Великобритания). Я руководил отделом разработки TeraType, в котором работало 15 человек. Мои дни были сплошным хаосом из телефонных звонков, импровизированных встреч, обсуждения проблем обслуживания «на местах» и непредвиденных событий, прерывавших мою работу. Чтобы моя работа выполнялась, мне пришлось установить жесткую дисциплину управления временем.

- Я просыпался в 5 утра и ехал на велосипеде в свой офис к 6 часам. Это давало мне 2,5 спокойных часа до того, как начинался ежедневный хаос.
- Приехав на место, я писал на доске расписание дня. Я делил время на 15-минутные интервалы и записывал, чем буду заниматься в каждый из интервалов.
- Первые 3 часа моего расписания были полностью распланированы. Начиная с 9 утра я оставлял один свободный 15-минутный интервал на каждый час; это позволяло мне легко перенести большинство прерываний в один из открытых интервалов и продолжить работу.
- Время после обеда оставалось нераспланированным, потому что я знал: к этому моменту сумятица достигает апогея, и в оставшуюся часть дня мне придется разгрести навалившиеся дела. В те редкие дневные периоды, когда хаос не вмешивался в мою работу, я просто трудился над самой важной задачей, пока не происходило что-то непредвиденное.

Моя схема не всегда работала успешно. Просыпаться в 5 утра не всегда получалось, иногда непредвиденные обстоятельства нарушали мою тщательно продуманную стратегию и поглощали весь день. И все же большей частью мне удавалось держаться на поверхности.

Встречи

Встречи обходятся примерно в \$200 в час на каждого участника. Здесь учитываются зарплаты, премии, затраты на использование помещения и т. д. Когда вы в следующий раз будете проводить рабочую встречу, посчитайте затраты на нее. Результат вас сильно удивит.

Две истины о встречах:

- 1) встречи необходимы;
- 2) встречи часто оказываются бесплодной тратой времени.

Часто эти две истины относятся к одной и той же встрече. Одним участникам встреча кажется бесценной, другим — лишней или бесполезной.

Профессионалы знают, что встречи обходятся дорого. Они также знают, что их собственное время драгоценно; им нужно писать код и выдерживать график. По этой причине они активно сопротивляются посещению встреч, которые не приносят немедленной и значительной пользы.

Отказ от участия

Вы не обязаны посещать каждую встречу, на которую вас приглашают. Более того, посещать слишком много встреч непрофессионально. Время нужно расходовать разумно. Будьте очень осмотрительны с выбором встреч, которые нужно посетить или от которых нужно вежливо отказаться. Человек, приглашающий вас на встречу, не отвечает за управление вашим временем — за него отвечаете только *вы*. Таким образом, получив приглашение, не принимайте его, если только ваше участие не объясняется немедленной и значительной необходимостью для текущей работы.

Иногда встреча посвящена теме, которая представляет для вас интерес, но не является строго необходимой. Решайте сами, сможете ли вы выделить для нее время. Будьте осторожны — такие встречи способны поглотить все ваше рабочее время.

Иногда вы знаете, что ваше участие может принести пользу, но встреча не является непосредственно значимой для того, чем вы сейчас занимаетесь. Решайте, компенсируются ли потери для вашего проекта пользой для других проектов. Возможно, это прозвучит цинично, но вы в первую очередь отвечаете за *свои* проекты. Тем не менее взаимопомощь групп часто приносит пользу, так что вы можете обсудить возможность своего участия со своей группой и начальством.

Иногда с просьбой о вашем присутствии на встрече к вам обращается какое-нибудь авторитетное лицо — например, технический специалист очень высокого уровня или руководитель другого проекта. Вам придется решать, перевешивает ли этот авторитет ваш рабочий график. Как и в предыдущем случае, в принятии решения может помочь группа или начальство.

Одна из самых важных обязанностей вашего руководителя — удерживать вас от участия в лишних встречах. Хороший руководитель более чем охотно защитит ваш отказ от участия, потому что ваше время представляет для него такую же ценность, как и для вас.

Уход со встречи

Встречи не всегда идут так, как планировалось изначально. Иногда вы осознаете, что отказались бы от участия в текущей встрече, если бы знали больше. На встрече могут подниматься новые темы или в обсуждении

на первый план выходит чья-то излюбленная тема. За годы я выработал простое правило: если на встрече стало скучно — уходите.

Еще раз напомним, что вы обязаны эффективно управлять своим временем. Если вы застряли на встрече, которая лишь понапрасну расходует ваше время, найдите способ вежливо покинуть эту встречу.

Разумеется, не нужно в гневе выбегать из помещения с восклицанием: «Какая скука!» Не нужно проявлять неуважение. Просто спросите в подходящий момент, насколько необходимо ваше присутствие. Объясните, что вы не можете выделить больше времени, и спросите, нельзя ли ускорить обсуждение или изменить повестку дня. Важно понять, что оставаться на встрече, расходующей ваше время, в ход которой вы не можете внести сколько-нибудь значительного вклада, было бы непрофессионально. Вы обязаны разумно тратить время и деньги своего работодателя, поэтому в выборе подходящего момента для обсуждения вашего ухода нет ничего зазорного.

Повестка дня и цель

Почему мы смиряемся с затратами на проведение встреч? Потому что иногда бывает нужно собрать всех участников в одной комнате для достижения конкретной цели. Чтобы время участников расходовалось эффективно, встреча должна иметь четкую повестку дня с указанием цели и времени обсуждения по каждой теме.

Если вас просят прийти на встречу, убедитесь в том, что вы знаете, какие темы будут обсуждаться, сколько времени на них выделено и какая цель должна быть достигнута. Если вам не удастся получить четкие ответы на эти вопросы, вежливо откажитесь от участия.

Если, придя на встречу, вы обнаруживаете, что повестка дня сорвана или изменена, предложите отложить новую тему и вернуться к исходной повестке. Если этого не произойдет, вежливо покиньте встречу при первой возможности.

Пятиминутка

«Пятиминутки» являются частью канона гибких методологий. Название происходит от того факта, что участники обычно стоят во время проведения встречи. Все участники поочередно отвечают на три вопроса:

1. Что я сделал вчера?
2. Что я буду делать сегодня?
3. Какие сложности мне предстоит решить?

Вот и все. Ответ на каждый вопрос должен занимать не более 20 секунд, так что на каждого участника уйдет не более минуты. Даже в группе из 10 человек встреча будет закончена за 10 минут.

Встречи планирования итераций

Самая сложная разновидность встреч в каноне гибких методологий. При плохом исполнении они занимают слишком много времени. Для хорошего проведения таких встреч необходимы соответствующие навыки, которые стоит освоить.

На встречах планирования итераций из списка реализации должны выбираться те элементы, которые должны быть выполнены в ходе следующей итерации. Оценки и прогнозы коммерческой ценности выполняются заранее. При действительно хорошей организации работ приемочные/компонентные тесты должны быть уже написаны или хотя бы определены в черновом варианте.

Встреча должна проходить достаточно быстро: краткое обсуждение по каждому элементу списка реализации, после чего элемент либо выбирается, либо отклоняется. Ни на один элемент не должно уходить более 5 или 10 минут. Если потребуется более подробное обсуждение, запланируйте его на другое время с частью группы.

У меня есть простое эмпирическое правило: встреча не должна занимать более 5% общего времени итерации. Таким образом, для недельной итерации (40 часов) встреча должна завершаться в пределах двух часов.

Ретроспективные встречи по итерациям и демонстрации

Эти встречи проводятся в конце каждой итерации. Участники группы обсуждают, что прошло хорошо, а что плохо. Ключевые участники видят демонстрацию новых возможностей. При неправильной организации эти встречи могут занимать очень много времени; планируйте их за 45 минут до завершения последнего дня итерации. Выделяйте не более

20 минут на ретроспективу и 25 минут на демонстрацию. Не забудьте, что прошли всего одна-две недели, так что материала для обсуждения не так уж много.

Споры и разногласия

Кент Бек однажды сказал мне очень важную вещь: «Любой спор, который не удастся завершить за 5 минут, не может быть решен обсуждением». Если спор занимает слишком много времени, значит, не существует четких доказательств в пользу одной из сторон. В таких ситуациях спор обычно имеет религиозную подоплеку, а не базируется на фактах.

Технические разногласия порой заходят за край. У каждой стороны имеются всевозможные обоснования своей позиции, которые редко подкрепляются данными. Без данных любой спор, который не приходит к согласию за несколько минут (от 5 до 30), попросту не способен прийти к согласию. Единственное, что можно сделать в такой ситуации, — это раздобыть данные.

Некоторые люди пытаются выиграть в споре за счет демонстрации характера. Они кричат, пытаются давить или изображают снисходительность. Это не важно; сила воли не может разрешить спор на продолжительное время. Данные — могут.

Другие занимают пассивно-агрессивную позицию. Они соглашаются просто для того, чтобы закончить спор, а затем саботируют результат, отказываясь участвовать в решении. Они говорят себе: «Вы так хотели, вот теперь сами и разбирайтесь». Вероятно, это худший из вариантов непрофессионального поведения. Никогда, никогда не поступайте подобным образом. Если вы соглашаетесь, то вы *обязаны* участвовать.

Как получить данные, необходимые для урегулирования разногласий? Иногда можно провести эксперимент, устроить имитацию или моделирование. А иногда лучше всего попросту бросить монетку, чтобы выбрать один из двух путей.

Если все получилось, значит, этот путь был работоспособным. Если возникли проблемы — вернитесь и опробуйте другой путь. Заранее согласуйте время и набор критериев, по которым будет приниматься решение об отказе от выбранного пути.

Остерегайтесь встреч, которые проводятся только для подавления разногласий и привлечения поддержки одной из сторон. Также избегайте встреч, на которых присутствует только одна из спорящих сторон.

Если спор необходимо урегулировать, попросите каждую сторону изложить свои аргументы группе не более чем за 5 минут, после чего проведите голосование. Вся встреча займет не более 15 минут.

Мана концентрации

Простите, если этот раздел отдает то ли метафизикой «эпохи Водолея», то ли ролевой системой «Dungeons & Dragons». Просто я отношусь к этому вопросу именно так.

Программирование — интеллектуальная деятельность, для которой необходимы продолжительные периоды сосредоточения и концентрации. Концентрация — ценный ресурс, как и мана ¹. После того как вы израсходуете свою ману концентрации, вам придется восстанавливать ее, в течение часа и более занимаясь более простой работой.

Я не знаю, что такое «мана концентрации», но мне кажется, что это некая физическая субстанция (а может быть, ее отсутствие), влияющая на внимательность и восприимчивость. Чем бы она ни была, вы *чувствуете*, когда она есть, и чувствуете, когда ее нет. Профессиональные разработчики учатся управлять своим временем так, чтобы в полной мере использовать свою ману концентрации. Мы пишем код, когда резерв маны высок, а когда ее остается мало — занимаемся другими, менее творческими делами.

Кроме того, мана концентрации — недолговечный ресурс. Если не использовать ее, пока она есть, то, скорее всего, вы ее потеряете. Это одна из причин, по которым встречи часто приводят к таким значительным затратам. Если вы потратите всю ману концентрации на встрече, то у вас останется меньше ресурсов для программирования.

Беспокойство и раздражение также приводят к потере маны концентрации. Вчерашняя ссора с супругой, царапина на крыле машины, счет, который вы забыли оплатить на прошлой неделе, — все это приводит к быстрому поглощению маны.

¹ Мана — стандартный ресурс в фэнтезийных и ролевых играх типа «Dungeons & Dragons». Каждый игрок обладает определенным количеством маны — магической субстанции, которая расходуется на применение магических заклинаний. Чем мощнее заклинание, тем больше на него расходуется маны. Восстановление маны происходит медленно, с фиксированным ежедневным приращением. Неопытные игроки способны легко израсходовать всю ману за несколько применений.

Сон

Важность сна невозможно переоценить. Хороший ночной сон восстанавливает большую часть маны концентрации. Семичасовой сон часто возвращает мне полную восьмичасовую дозу маны. Профессиональные разработчики управляют своим графиком сна так, чтобы резерв маны концентрации достигал максимума к тому моменту, когда они утром приходят на работу.

Кофеин

Несомненно, поглощение умеренных доз кофеина повышает эффективность использования маны концентрации. Но будьте осторожны! Кофеин также вводит в вашу концентрацию странную неустойчивость. При избытке кофеина ваше внимание способно смещаться в непредсказуемых направлениях. Слишком сильная кофеиновая стимуляция нередко приводит к тому, что вы тратите весь день, концентрируясь на малозначительных вещах.

Использование и переносимость кофеина — личное дело. Лично я обхожусь чашкой крепкого кофе утром и диетической колой в обед. Иногда эта доза удваивается, но превышаете крайне редко.

Перезарядка

Мана концентрации частично перезаряжается отвлекающими занятиями. Хорошая долгая прогулка, беседа с друзьями, просто взгляд из окна — все это поможет восстановить резерв маны.

Одни люди медитируют. Другие выбирают восстановительный сон. Третьи слушают подкасты или листают журналы.

Мой опыт показывает, что после того, как мана уйдет, сохранять концентрацию силой воли невозможно. Вы можете писать код, но вам почти наверняка придется переписывать его на следующий день или жить с этой разлагающейся массой неделями или месяцами. Уж лучше потратить полчаса или даже час на расслабление.

Физические упражнения

В физических упражнениях — таких как боевые искусства, тай-чи или йога — есть нечто особенное. Хотя все они требуют значительной концентрации, это концентрация сильно отличается от необходимой для

программирования. Она имеет не интеллектуальную, а физическую природу. И иногда физическая концентрация помогает восстановить умственную, причем это нечто большее, чем простая перезарядка. Я обнаружил, что регулярные физические упражнения повышают мой потенциал умственной концентрации.

Моей любимой формой физической концентрации является велоспорт. Я езжу на велосипеде час или два, иногда на 20–30 километров. Мой маршрут пролегает по тропе, параллельной реке Дес-Плейнс, так что мне не приходится иметь дело с машинами.

Во время поездок я слушаю подкасты об астрономии и политике. Иногда я просто ставлю свою любимую музыку, а иногда снимаю наушники и слушаю природу.

Некоторые люди предпочитают поработать руками. Одни увлекаются столярным делом, другие — сборкой моделей или садоводством. Впрочем, в любой физической работе независимо от ее вида есть что-то такое, что помогает нам работать головой.

Ввод и вывод

Также мне кажется очень важным, чтобы мои результаты подпитывались соответствующим «вводом». Написание программного кода — творческая работа. Обычно мои творческие способности в наибольшей степени проявляются тогда, когда я сталкиваюсь с творческим трудом других людей. Поэтому я читаю много научной фантастики — творческое воображение авторов каким-то образом стимулирует мои собственные творческие способности в области программирования.

Помидоры и распределение времени

Для управления своим временем и концентрацией я также использую чрезвычайно эффективный *помидорный метод планирования*¹. Основная идея очень проста: вы ставите стандартный кухонный таймер (традиционно такие таймеры оформляются в виде помидора) на 25 минут. Во время работы таймера ничто не должно мешать вашей работе. Если звонит телефон, вы отвечаете и вежливо просите перезвонить через

¹ <http://www.pomodorotechnique.com/>

25 минут. Если кто-то заходит к вам, чтобы задать вопрос, вы вежливо предлагаете зайти через 25 минут. Независимо от вида прерывания оно откладывается до момента срабатывания таймера. Вряд ли найдется так уж много неотложных дел, которые не могут подождать 25 минут!

По сигналу таймера-«помидора» вы немедленно прекращаете свою текущую работу. Пришло время разобраться со всеми проблемами, возникавшими во время работы таймера. Затем вы делаете перерыв примерно на пять минут, снова ставите таймер на 25 минут и начинаете следующий «помидорный» период. Каждый четвертый период отдыха делается более продолжительным, 30 минут или около того.

Этот метод управления временем описан достаточно подробно; я рекомендую познакомиться с ним подробнее. Тем не менее даже это краткое описание дает начальное представление об этой методике.

При использовании этого метода ваше время делится на продуктивное и непродуктивное. «Помидорное» время продуктивно. Именно в эти периоды выполняется настоящая работа. Остальное время тратится на отвлекающие факторы, встречи, перерывы или другую деятельность, не связанную напрямую с выполнением ваших задач.

Сколько «помидоров» можно сделать за день? В хорошие дни — 12 и даже 14. В плохие может выйти 2 или 3. Если подсчитать «помидоры» и построить график, вы быстро получите представление о том, какая часть рабочего времени расходуется продуктивно, а какая тратится на «всякую всячину».

Некоторые профессионалы настолько привыкают к этой методике, что оценивают затраты времени на свои задачи в «помидорах» и вычисляют свою еженедельную «помидорную скорость». Однако это всего лишь дополнительные «плюсы» метода. Главное преимущество помидорного метода заключается в том, что 25-минутное окно продуктивного времени агрессивно защищается от любых вмешательств.

Уклонение от работы

Иногда у вас попросту «сердце не лежит» к работе. Например, порученная задача вас пугает, кажется неудобной или скучной. А может, вы думаете, что она приведет к конфликтам или неминуемо загонит в сложную ситуацию? Или вам просто не хочется ее выполнять?

Инверсия приоритетов

Независимо от причины мы всегда находим способы избежать выполнения работы. Мы убеждаем себя, что у вас есть более срочная задача, и занимаемся ей. Это называется *инверсией приоритетов*: вы искусственно повышаете приоритет задачи, чтобы отложить другую задачу, обладающую настоящим приоритетом. Инверсия приоритетов — это ложь, которую мы рассказываем самим себе. Нам не хватает смелости обратиться к тому, что нужно сделать, и мы убеждаем себя, что другая задача более важна. Мы знаем, что это не так, и все же обманываем себя.

Хотя правильнее сказать, что мы обманываем не себя. В действительности мы готовим ложь для тех, кто спросит нас, чем мы занимаемся и почему занимаемся именно этим. Мы заранее готовим защиту от мнения других.

Конечно, такое поведение непрофессионально. Профессионал оценивает приоритет каждой задачи независимо от своих личных страхов и предпочтений и решает эти задачи в порядке приоритетов.

Тупики

Тупик — явление, хорошо знакомое всем разработчикам программных продуктов. Вы принимаете решение и идете по техническому пути, который приводит вас в никуда. Чем больше упорства вы проявляете в своем решении, чем дальше зайдете на этом пути. А если на кон поставлена ваша профессиональная репутация, то вы будете блуждать вечно.

Опыт и благоразумие помогут избежать некоторых тупиков, но обойти их все не удастся. Так что в действительности вы должны быстро понять, что ваш путь завел в тупик, и иметь смелость для отступления. Иногда это называется *«правилом ямы»*: если вы оказались в яме, прежде всего перестаньте копать.

Профессионал не увлекается идеей настолько, чтобы у него не хватило сил отказаться от нее и вернуться к исходной точке. Он непредвзято относится к другим идеям, чтобы у него оставались другие варианты на случай, если он все же окажется в тупике.

Грязь, болота и трясины

Грязь еще хуже, чем тупик. Она вас замедляет, но не останавливает полностью. Грязь препятствует вашему продвижению, но вы все равно можете двигаться вперед, действуя методом «грубой силы». Грязь опаснее тупиков, потому что вы всегда видите путь впереди, и он всегда кажется короче, чем путь назад (хотя на самом деле это не так).

Я видел продукты и целые компании, уничтоженные грязью в программном коде. Я видел, как производительность работы групп падала едва ли не до нуля всего за несколько месяцев. Ничто не оказывает более основательного и долгосрочного отрицательного эффекта на производительность группы программистов, чем грязь в программном коде. Ничто.

Проблема в том, что возникновение грязи, как и тупики, неизбежно. Опыт и благоразумие помогут вам избегать его, но рано или поздно вы примете решение, которое заведет вас в грязь.

Возникновение грязи весьма коварно. Вы создаете решение простой задачи, всеми силами стремясь к тому, чтобы код оставался простым и чистым. С ростом масштаба и сложности задачи вы расширяете ее кодовую базу, по возможности стараясь сохранять ее чистоту. В какой-то момент вы понимаете, что изначально приняли неверное архитектурное решение и ваш код плохо масштабируется в направлении смещения требований.

Здесь и находится критическая точка! Вы все еще можете вернуться и исправить архитектуру. Но вы также можете продолжить движение вперед. Кажется, что возврат обойдется слишком дорого, потому что вам придется перерабатывать существующий код, однако в будущем он обойдется еще дороже. Если вы будете двигаться вперед, то система сползет в грязь, и вполне возможно, что она из нее уже не выберется.

Профессионалы опасаются грязи намного сильнее, чем тупиков. Они всегда обращают внимание на грязь, которая начинает неограниченно разрастаться, и прикладывают все необходимые усилия к ее устранению — по возможности раннему и быстрому.

Движение вперед по грязи (когда вы знаете, что это грязь) является худшей из разновидностей инверсии приоритетов. Двигаясь вперед, вы обманываете себя, обманываете вашу группу, обманываете свою компанию и заказчиков. Вы говорите им, что все будет хорошо, хотя на самом деле вы ведете их к общей катастрофе.

Заключение

Профессиональные разработчики серьезно относятся к управлению своим временем и концентрацией. Они понимают соблазн инверсии приоритетов и борются с ним, поскольку это является делом чести. Они непредвзято относятся к альтернативным решениям. Они никогда не увлекаются решением настолько, что не могут отказаться от него. А еще они всегда следят за возможным возникновением грязи и вычищают ее при первых признаках. Нет более печального зрелища, чем группа разработчиков, бесцельно плетущаяся по постоянно углубляющейся трясине.

10

Оценки



Оценка — одно из самых простых, но при этом самых рискованных задач, с которыми сталкиваются профессиональные разработчики. От оценки напрямую зависит коммерческая ценность проекта. От нее зависят наши репутации. Неверные оценки становятся причиной наших страхов и провалов. Оценка — это первый клин, вбиваемый между бизнесменами и разработчиками. Это источник почти всего недоверия, связанного с этими отношениями.

В 1978 году я был ведущим программистом для 32-килобайтной встроенной программы Z-80, написанной на языке ассемблера. Программа «прошивалась» на 32 перепрограммируемых микросхемах, которые вставлялись в три платы (до 12 микросхем на каждой).

У нас в эксплуатации были сотни устройств, установленных на центральных телефонных станциях по всем Соединенным Штатам. Каждый раз, когда мы исправляли ошибку или добавляли новую функцию, нам приходилось отправлять техников к каждому устройству для замены всех 32 микросхем!

Это был настоящий кошмар. Микросхемы и платы были непрочными, контакты на микросхемах гнулись и ломались. Нечаянные изгибы плат могли повредить места пайки. Риск ошибок и поломок был огромен, а затраты для компании — слишком высоки.

Мой начальник Кен Файндер пришел ко мне и попросил что-нибудь сделать. Нам был нужен способ замены микросхем, который бы не требовал замены всех остальных микросхем. Если вы читали мои книги или слушали мои лекции, то вы знаете, что я много говорю о возможности независимого развертывания. Именно тогда я впервые усвоил этот урок.

Проблема заключалась в том, что программа представляла собой единый скомпонованный исполняемый файл. Добавление новой строки кода приводило к изменению адресов всех последующих строк. Так как в каждой микросхеме попросту хранился один килобайт адресного пространства, то изменялось содержимое практически всех микросхем.

Решение было довольно простым. Микросхемы нужно было изолировать друг от друга. Содержимое каждой микросхемы преобразовывалось в независимую единицу компиляции, которая могла записываться независимо от всех остальных.

Я определил размеры всех функций в приложении и написал простую программу, которая «укладывала» их, словно фрагменты головоломки, на микросхемах, оставляя 100 байт свободного пространства для расширения. В начале каждой микросхемы размещалась таблица указателей на все функции данной микросхемы. Во время загрузки эти указатели перемещались в память. Весь код системы был изменен таким образом, чтобы функции никогда не вызывались напрямую — только через векторы, хранящиеся в памяти.

Да, вы правильно поняли: микросхемы превратились в аналоги объектов с v-таблицами, а функции вызывались полиморфно. Именно так я узнал некоторые принципы объектно-ориентированного программирования задолго до того, как познакомился с понятием «объект».

Выгода от такого решения была огромной. Мы получили возможность не только ограничиться установкой отдельных микросхем, но и вносить

исправления «на месте», перенаправляя векторы в оперативной памяти. Это значительно упростило отладку и «горячие» исправления.

Но я отклоняюсь от темы. Когда Кен пришел ко мне с предложением решить проблему, он предложил подумать об указателях на функции. Я потратил день-два на формальное изложение идеи, а затем представил подробный план. Он спросил, сколько времени займет работа; я сказал, что около месяца.

Мне понадобилось *три* месяца.

Я напивался только дважды в жизни и только один раз напился основательно. Это произошло на праздновании Рождества в Teradyne в 1978 году. Мне тогда было 26 лет.

Праздник проводился в офисе Teradyne, который в основном состоял из открытого лабораторного пространства. Все пришли рано, а сильная снежная буря не позволила оркестру и фирме выездного обслуживания добраться до нас. К счастью, выпивки было достаточно. Я не очень хорошо помню этот вечер, а то, что *помню*, предпочел бы забыть. И все же я не могу не поделиться одним важным моментом.

Я сидел по-турецки на полу с Кеном (мой начальник — ему тогда было 29 лет, и он был трезв), сетуя на то, сколько времени у меня заняла векторизация. Алкоголь освободил мои скрытые страхи и неуверенность по поводу исходной оценки. Надеюсь, я не клал голову ему на плечо — впрочем, такие подробности не очень четко отложились у меня в памяти.

Помню, я спросил, не сердится ли он на меня и не считает ли, что работа заняла слишком много времени. И как бы смутно я ни помнил тот вечер, его ответ четко отложился у меня в памяти на последующие десятилетия. Он сказал: «Да, я думаю, что прошло много времени, но я вижу, что ты прилежно трудишься, а работа не стоит на месте. И нам это действительно нужно. Так что я не сержусь».

Что такое «оценка»?

Проблема в том, что оценки можно рассматривать по-разному. Бизнес любит рассматривать их как обязательства. Разработчики предпочитают рассматривать оценки как предположения. Между этими точками зрения существуют принципиальные различия.

Обязательства

Обязательство — нечто такое, что вы обязаны сделать. Если вы обязуетесь сделать что-то к определенной дате, то к этой дате «что-то» должно быть готово. Если для этого вам придется работать по 12 часов в сутки и по выходным, пропускать семейные праздники — так тому и быть. Вы приняли обязательство и его необходимо соблюдать.

Профессионалы не принимают на себя обязательств, если только они не уверены в возможности их выполнения. Да, все просто. Если вам предлагают принять на себя обязательство, а вы не уверены, что сможете справиться с задачей, — ваша честь требует отказаться. Если вам предлагают принять обязательство, которое, на ваш взгляд, выполнимо, но потребует сверхурочных, работы по выходным и пропущенного семейного отдыха — решайте сами; но если уж пообещали — выполняйте.

Обязательствам присуща *определенность*. Другие люди принимают ваши обязательства и используют их как базу для построения собственных планов. Нарушение обязательств обернется огромным ущербом для вашей репутации; это проявление непорядочности, лишь немногим лучшее открытой лжи.

Оценка

Оценка, прежде всего, является предположением. Она не подразумевает никаких обязательств. Вы ничего не обещаете. Нарушение оценки ни в коей мере не повредит вашей репутации. Мы выдаем оценки, прежде всего, потому, что мы *не знаем*, сколько времени займет работа.

К сожалению, многие разработчики очень слабы в оценках. Дело не в том, что оценка требует какого-то тайного мастерства — вовсе нет. Дело в том, что мы часто не понимаем истинной природы оценки.

Оценка — это не число, а распределение. Возьмем следующий диалог:

Майк: «Сколько, по твоим оценкам, понадобится для завершения работы?»

Питер: «Три дня».

Питер действительно управится с работой за три дня? Возможно, но насколько вероятно? Правильный ответ: понятия не имеем. Что имел в виду Питер и что узнал Майк? Если Майк вернется через три дня и работа не будет выполнена, должен ли он удивляться? А почему,

собственно? Питер не давал никаких обязательств. Питер не сказал ему, насколько три дня вероятнее четырех или пяти.

А если бы Майк поинтересовался у Питера, насколько высока вероятность его оценки?

Майк: «Какова вероятность того, что ты справишься за три дня?»

Питер: «Пожалуй, справлюсь».

Майк: «Можешь назвать число?»

Питер: «Пятьдесят или шестьдесят процентов».

Майк: «Значит, есть довольно высокая вероятность, что тебе понадобится четыре дня».

Питер: «Да. Может понадобиться даже пять или шесть, хотя я в этом сомневаюсь».

Майк: «До какой степени сомневаешься?»

Питер: «О, я не знаю... Я на девяносто пять процентов уверен, что работа будет сделана менее чем за шесть дней».

Майк: «То есть может быть и семь?»

Питер: «Ну, если все пойдет наперекосяк... Черт, если *все* пойдет наперекосяк, может быть десять и даже одиннадцать дней. Но ведь вероятность такого совпадения очень мала, верно?»

Мы постепенно начинаем видеть истину. Оценка Питера представляет собой вероятностное распределение. Своим мысленным взором он видит вероятность завершения задачи так, как показано на рис. 10.1.

Мы видим, почему Питер выдал исходную оценку в 3 дня — это самый высокий столбец гистограммы. Соответственно, Питеру она представляется наиболее вероятной продолжительностью выполнения задачи.

Но Майк смотрит на происходящее иначе. Он обращает внимание на правый край распределения и беспокоится о том, что Питеру может понадобиться более 11 дней.

Должен ли Майк беспокоиться об этом? Конечно! Закона Мерфи¹ еще никто не отменял, поэтому могут возникнуть непредвиденные осложнения.

¹ Закон Мерфи гласит, что если какая-нибудь неприятность может случиться, то она обязательно произойдет.

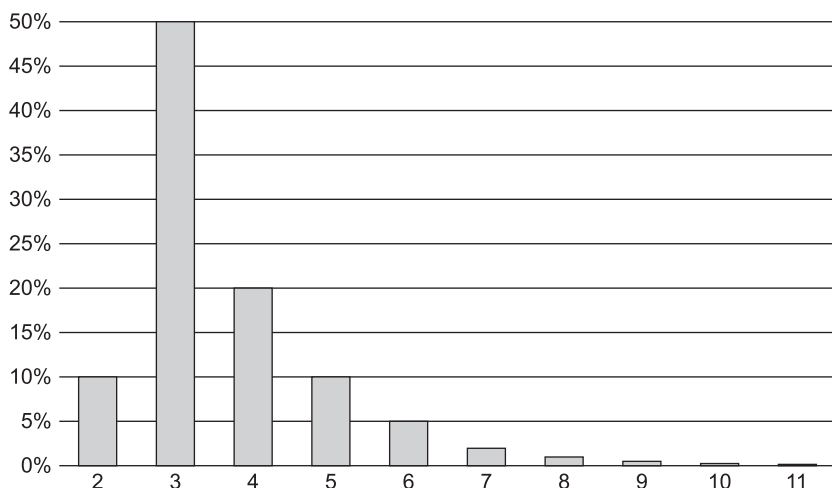


Рис. 10.1. Вероятностное распределение

Подразумеваемые обязательства

Майк сталкивается с проблемой. Он не уверен в том, сколько времени потребуется Питеру для выполнения работы. Чтобы свести к минимуму неопределенность, он может попробовать добиться от Питера обязательства. Питер не в состоянии что-либо обещать с полной уверенностью.

ПИТЕР: «Нет, Майк. Как я уже сказал, работа будет выполнена за три, а может, за четыре дня».

МАЙК: «Тогда пишем четыре?»

ПИТЕР: «Нет, теоретически может быть пять или шесть».

Пока все идет честно. Майк попросил принять обязательство, а Питер аккуратно отказался его давать. Майк пытается применить другую тактику.

МАЙК: «Хорошо, Питер, но ты можешь хотя бы попытаться уложиться в шесть дней?»

Просяба Майка звучит достаточно невинно, и безусловно, Майк руководствуется лучшими намерениями. Но чего именно Майк хочет от Питера? Что значит «попытаться»?

Мы уже говорили об этом в главе 2. В это слово вкладывается разный смысл. Если Питер согласится, то он фактически возьмет на себя обязательство уложиться в шесть дней.

Какие еще возможны интерпретации? Что именно Питер собирается сделать, чтобы «попытаться»? Он собирается работать более 8 часов? Очевидно, это подразумевается в его согласии. Он собирается работать по выходным? Да, это тоже подразумевается. Пропускать семейные праздники? Да, и это тоже. Все это входит в понятие «попытаться». И если Питер чего-то не сделает, Майк сможет обвинить его в том, что он приложил недостаточно стараний.

Профессионалы проводят четкое различие между оценками и обязательствами. Они не берут на себя обязательств, пока не будут твердо уверены в успехе. Также они следят за тем, чтобы избегать неявных обязательств. Они по возможности четко оговаривают вероятностное распределение своих оценок, чтобы руководители могли строить соответствующие планы.

PERT

Программа PERT (Program Evaluation and Review Technique) была создана в 1957 году ВМС США для проектирования подводных лодок Polaris. Одним из элементов PERT является способ вычисления оценок. Схема PERT предоставляет очень простой, но исключительно эффективный способ преобразования оценок в вероятностные распределения, подходящие для начальства. При оценке задачи предоставляются три числа (так называемый анализ по трем переменным):

- *O*: оптимистическая оценка. Это значение выбирается *предельно* оптимистично. Задача может быть выполнена за это время только в том случае, если все без исключения пройдет гладко. Более того, чтобы математическая теория сработала, вероятность такого исхода должна быть менее 1%¹. Как видно из рис. 10.1, в ситуации Питера это один день;
- *N*: номинальная оценка (наиболее вероятная). На гистограмме она будет представлена самым высоким столбцом. На рис. 10.1 номинальная оценка составляет 3 дня;

¹ Точное значение для нормального распределения равно 1:769, или 0,13. Вероятно, можно безопасно говорить о вероятности 1:1000.

- P : пессимистическая оценка. Эта оценка также должна быть крайне предельно пессимистической. В ней следует учесть все возможные неприятности, кроме ураганов, ядерной войны, блуждающих «черных дыр» и других катастроф. Математическая база также работает только в том случае, если вероятность этого исхода много меньше 1%. В ситуации Питера пессимистическая оценка представлена крайним правым столбцом (12 дней).

По этим трем оценкам вероятностное распределение описывается следующей формулой:

$$\mu = \frac{O + 4N + P}{6},$$

где μ — ожидаемая продолжительность задачи. В случае Питера она составит $(1+12+12)/6$, или около 4,2 дня. Для большинства задач оценка получается слегка завышенной, потому что правая часть распределения длиннее левой¹.

$$\sigma = \frac{P - O}{6},$$

где σ — среднее квадратическое отклонение распределения времени выполнения задачи². Фактически это мера неопределенности задачи: если это число велико, то и неопределенность тоже велика. В нашем примере оно равно $(12 - 1)/6$, или около 1,8 дня.

По оценке Питера 4,2/1,8 Майк понимает, что задача, скорее всего, будет завершена за пять дней, но также может занять 6 и даже 9 дней.

Но Майк управляет не одной задачей — он ведет проект с множеством задач. Питеру поручены три задачи, над которыми он должен работать последовательно. Оценки продолжительности выполнения этих задач, представленные Питером, приведены в табл. 10.1.

¹ Предполагается, что модель PERT используется для аппроксимации бета-распределения. Это разумное предположение, потому что минимальная длительность выполнения задачи обычно определяется намного точнее, чем максимальная [Макконелл С. Сколько стоит программный проект. СПб.: Питер; М.: Русская Редакция, 2007. Рис. 1.3].

² Если вы не знаете, что это такое, — найдите хороший учебник по теории вероятностей и математической статистике. Понять эту концепцию нетрудно, но она вам очень пригодится.

Таблица 10.1. Задачи Питера

Задача	Оптимистическая оценка	Номинальная оценка	Пессимистическая оценка	μ	σ
Альфа	1	3	12	4,2	1,8
Бета	1	1,5	14	3,5	2,2
Гамма	3	6,25	11	6,5	1,3

Что происходит с задачей «бета»? Похоже, Питер достаточно уверен в ней, но непредвиденные факторы могут серьезно затормозить его работу. Как Майку интерпретировать эти результаты? Сколько времени следует планировать на завершение всех трех задач?

Оказывается, путем простых вычислений Майк может объединить все задачи Питера и создать вероятностную оценку для всего набора задач. Вычисления весьма тривиальные:

$$\mu_{\text{sequence}} = \sum \mu_{\text{task}}.$$

Для любой последовательности задач предполагаемая продолжительность выполнения вычисляется простым суммированием продолжительностей всех задач последовательности. Таким образом, если Питер должен выполнить три задачи с оценками 4,2/1,8, 3,5/2,2 и 6,5/1,3, то вероятнее всего, на их выполнение Питеру понадобится около 14 дней: $4,2 + 3,5 + 6,5$.

$$\sigma_{\text{sequence}} = \sqrt{\sum \sigma_{\text{task}}^2}.$$

Среднеквадратическое отклонение последовательности равно квадратному корню из суммы квадратов среднеквадратичных отклонений задач. Таким образом, стандартное отклонение всех трех задач Питера равно примерно 3.

$$\begin{aligned} & (1,8^2 + 2,2^2 + 1,3^2)^{1/2} = \\ & = (3,24 + 2,48 + 1,69)^{1/2} = \\ & = 9,77^{1/2} = \sim 3,13. \end{aligned}$$

Из этого результата Майк узнает, что Питеру на решение его задач, вероятно, потребуется 14 дней, но с достаточно большой вероятностью может потребоваться 17 (1s) и даже 20 дней (2s). Решение задач может затянуться и на более долгий срок, но это маловероятно.

Вернемся к таблице оценок. Разве вам не хочется предположить, что все три задачи будут выполнены за 5 дней? В конце концов,

оптимистические оценки равны 1, 1 и 3. Даже номинальные оценки в сумме дают всего 10 дней. Откуда взялись 14 дней с возможными 17 и даже 20? Дело в том, что суммирование неопределенности в серии задач добавляет реализма в исходный план.

Любому программисту со сколько-нибудь значительным опытом работы знакомы проекты, которые изначально оценивались оптимистически, а затем занимали в 3–5 раз больше времени. Простая схема PERT — один из разумных способов предотвращения подобных излишне оптимистических ожиданий. Профессионалы очень тщательно относятся к выбору разумных сроков, несмотря на давление и уговоры.

Оценка времени выполнения

Майк и Питер совершили ужасную ошибку. Майк спрашивает Питера, сколько времени ему потребуется на выполнение работы. Питер дал честный ответ с тремя переменными, но как насчет мнения его коллег? Может, у них есть свое мнение по этому поводу?

Самый важный ресурс оценки — это люди, которые вас окружают. Они могут видеть то, что не видите вы. Они помогут вам оценить ваши задачи точнее, чем если бы вы делали это самостоятельно.

Широкополосный дельфийский метод

В 1970-е годы Барри Бем представил метод экспертной оценки, названный «широкополосным дельфийским методом»¹. За прошедшие годы появилось много разновидностей этого метода — как формальных, так и неформальных. Но у всех них есть нечто общее: принцип консенсуса.

Стратегия проста. Группа экспертов собирается, обсуждает задачу и оценивает ее сложность. Обсуждение и оценка повторяются до тех пор, пока не будет достигнуто согласие.

Исходный метод, описанный Бемом, требует проведения нескольких собраний и составления документов; на мой вкус все это оборачивается лишними церемониями и непроизводительными затратами. Я предпочитаю более экономичные методы — такие как описанный ниже.

¹ Barry W. Boehm, *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

Метод быстрого голосования

Все участники сидят за столом. Задачи рассматриваются последовательно. Для каждой задачи проводится краткое обсуждение, описываются возможные усложняющие факторы и возможная реализация. Затем участники опускают руку под стол и поднимают от 0 до 5 пальцев в зависимости от того, сколько, по их мнению, займет решение задачи. Модератор считает до трех, после чего все участники одновременно показывают руки.

Если оценки согласуются, участники переходят к следующей задаче. В противном случае они продолжают обсуждение, чтобы определить причины расхождений. Цикл повторяется до тех пор, пока их оценки не будут согласованы. Согласие не обязано быть абсолютным — если оценки достаточно близки друг к другу, это тоже хорошо. Например, смешанный набор из 3 и 4 считается согласием. Но если все подняли 4 пальца кроме одного человека, поднявшего один палец, у участников появляется тема для обсуждения.

Масштаб оценки определяется в начале встречи. Трудоемкость задачи может определяться как непосредственно количеством пальцев, так более сложными метриками типа «количество пальцев, умноженное на 3» или «количество пальцев в квадрате».

Одновременность предъявления пальцев очень важна. Участники не должны изменять свои оценки на основании оценок, которые они видят у других.

Покер планирования

В 2002 году Джеймс Греннинг написал отличную статью¹ с описанием «покера планирования». Эта разновидность широкополосного дельфийского метода стала настолько популярной, что несколько разных компаний использовали идею для создания маркетинговых сувениров в виде колод для покера планирования². Даже существует специальный сайт *planningpoker.com*, который может использоваться распределенными группами для проведения сеансов покера планирования в Интернете.

¹ James Grenning, «Planning Poker or How to Avoid Analysis Paralysis while Release Planning», April 2002, <http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html>

² <http://store.mountaingoatsoftware.com/products/planning-poker-cards>

Идея очень проста. Всем участникам экспертной группы раздаются карты с разными числами. Числа от 0 до 5 работают достаточно хорошо; такая система логически эквивалентна системе с показанными пальцами.

Выберите задачу и обсудите ее. В какой-то момент модератор просит всех выбрать карту. Участники группы берут карту, которая соответствует их внутренней оценке, и держат ее «рубашкой» вверх, чтобы остальные не видели номинал карты. Затем модератор дает сигнал показать карты.

Остальное — как в методе быстрого голосования. Если оценки участников согласуются, оценка принимается. В противном случае карты возвращаются в руку, а участники продолжают обсуждение задачи.

По поводу правильного выбора номиналов карт существуют целые научные теории. Некоторые специалисты дошли до использования карт, номиналы которых определяются числами Фибоначчи. Другие включают в колоду карты со знаком бесконечности и вопросительным знаком. На мой взгляд, пяти карт 0, 1, 3, 5, 10 вполне достаточно.

Аффинная оценка

Несколько лет назад Лоуэлл Линдстром показал мне необычную разновидность широкополосного дельфийского метода. Я довольно успешно применял его с разными заказчиками и группами.

Все задачи записываются на картах без каких-либо оценок. Экспертная группа стоит возле стола или стены, на которой карты распределены случайным образом. Участники группы не говорят между собой — они просто сортируют карты. Карты задач, которые занимают больше времени, перемещаются направо, а карты меньших задач — налево.

Любой участник группы может в любой момент переместить любую карту, даже если она была перемещена другим участником. Карты, перемещенные более h раз, откладываются в сторону для обсуждения.

Со временем безмолвная сортировка завершается и начинается обсуждение. Участники разбираются во всех разногласиях по поводу порядка карт. Возможно, для достижения согласия придется провести краткое обсуждение или нарисовать от руки несколько условных схем.

На следующем этапе между картами рисуются линии, представляющие трудоемкость задачи в днях, неделях, условных пунктах и т. д. Традиционно используются пять значений, образующих последовательность Фибоначчи (1, 2, 3, 5, 8).

Анализ по трем переменным

Широкополосный дельфийский метод хорошо подходит для выбора одной номинальной оценки. Но как говорилось ранее, в большинстве случаев желательно иметь три оценки для создания вероятностного распределения. Оптимистическую и пессимистическую оценку для каждой задачи можно очень быстро получить при помощи любой из разновидностей широкополосного дельфийского метода. Например, если вы используете покер планирования, попросите группу поднять карты для пессимистической оценки и выберите наибольшую. Потом сделайте то же самое для оптимистической оценки и возьмите наименьшее значение.

Закон больших чисел

В оценке заложена ошибка. Собственно, поэтому они и называются оценками. Один из способов контроля ошибок основан на *законе больших чисел*¹. В частности, из этого закона следует, что при разбиении большой задачи на несколько меньших и независимой их оценке сумма оценок меньших задач будет более точной, чем одна оценка большей задачи. Возрастание точности объясняется тем, что погрешности оценки меньших задач взаимно компенсируются.

Честно говоря, это утверждение оптимистическое. Погрешности в оценке обычно связаны с недооценкой, а не с переоценкой, так что компенсация вряд ли идеальна. Тем не менее разбиение больших задач с независимой оценкой меньших все равно полезно. *Некоторые* ошибки взаимно компенсируются, а разбиение задачи поможет лучше понять ее суть и выявить возможные неожиданности.

Закключение

Профессиональные разработчики знают, как предоставить бизнес-стороне практическую оценку, пригодную для планирования. Они не дают обещаний, которые не могут сдержать, и не принимают обязательств, в соблюдении которых они не уверены.

Когда профессионал принимает обязательство, он указывает конкретные числа, а затем ориентируется на них. Однако чаще профессионалы

¹ http://en.wikipedia.org/wiki/Law_of_large_numbers

все же не дают таких обязательств. Вместо этого они дают вероятностные оценки, определяющие предполагаемое время завершения и дисперсию.

Профессиональные разработчики общаются с другими участниками группы для согласования оценок, передаваемых руководству.

В этой главе описаны лишь отдельные примеры методов, используемых профессиональными разработчиками для создания практических оценок. Список не полон, а представленные методы не обязательно являются лучшими. Это всего лишь некоторые методы, которые хорошо сработали в моем случае.

11

Под давлением



В книгах нередко пишут о феномене «внетелесных переживаний»: представьте, что вы наблюдаете за операционным столом, на котором хирург делает вам операцию на сердце. Хирург пытается спасти вам жизнь, но время ограничено — операция должна быть сделана за короткое время.

Какого поведения вы ждете от врача? Вы предпочитаете, чтобы он был спокойным и собранным? Чтобы он отдавал ясные, четкие приказы ассистентам? Чтобы он помнил все, чему его учили, и придерживался действующих норм?

А если врач потеет и ругается? Швыряет инструменты? Обвиняет руководство в нереалистичных требованиях и постоянно жалуется на нехватку времени? Как он должен вести себя — как профессионал или как типичный разработчик?

Профессионал спокоен и решителен, даже когда он находится под давлением. Даже с ростом давления он не забывает полученные знания и методы, зная, что это лучший путь к выполнению установленных сроков и обязательств.

В 1988 году я работал в Clear Communications. Это была начинающая фирма, которая так никогда и не «поднялась». Мы прошли через первый цикл финансирования, потом второй и третий.

Исходная идея выглядела неплохо, но проектирование так и осталось на бумаге. Сначала продукт состоял из программных и аппаратных компонентов. Затем он стал чисто программным. Программная платформа изменилась — с PC мы перешли на Sparc. Заказчики сменили требования: с высокопроизводительных рабочих станций они переключились на низкопроизводительные.

В конечном итоге изменилась даже исходная концепция продукта — компания пыталась найти возможности получения прибыли. Не думаю, что за четыре года, которые я провел в ней, она получила хотя бы цент.

Не стоит и говорить, что мы, разработчики, находились под значительным давлением. В офисе за терминалами было проведено немало долгих ночей и еще более долгих выходных. На C писались функции по 3000 строк. Шли жаркие споры с криками и переходами на личности. Было все: интриги и саботаж, удары кулаком о стену, гневное швыряние ручек в стену, карикатуры на неприятных коллег и бесконечный запас гнева и стресса.

Сроки определялись событиями. Новые функции спешно готовились к торгово-промышленной выставке или выпуску демо-версии для заказчика. Какую бы глупость ни потребовал заказчик, она должна быть готова к следующей демо-версии. Времени вечно не хватало. Работа всегда отставала от графика. Планы всегда превышали наши возможности.

Работая по 80 часов в неделю, можно было стать героем. Слепив на скорую руку демо-версию для заказчика, можно было стать героем. Если вы прилагали достаточно усилий, вас могли повысить. А если нет — вас увольняли. Это была начинающая фирма и здесь полагалось

выкладываться «на полную». И в 1988 году, с моим 20-летним опытом, я «купился» на это.

Я был начальником отдела разработки, и это я говорил работавшим на меня программистам, что работать нужно еще больше и быстрее. Это я был одним из тех парней с 80-часовой неделей, я писал функции C по 3000 строк в 2 часа ночи, пока мои дети спали дома. Это я бросал ручку в стену и кричал в гнев. Это я увольнял людей, если они не справлялись. Это было ужасно.

Однажды моя жена заставила меня хорошенько присмотреться к отражению в зеркале. Мне не понравилось то, что я увидел. Она сказала, что быть со мной рядом стало не очень приятно. Мне пришлось согласиться. Но все это мне не понравилось, поэтому я в гнев выбежал из дома и пошел неизвестно куда. Я шел минут тридцать, охваченный волнением; а потом пошел дождь.

И у меня в голове что-то щелкнуло. Я начал смеяться. Я смеялся над своими переживаниями. Я смеялся над стрессом. Я смеялся над человеком в зеркале; над несчастным болваном, который отравляет жизнь себе и другим во имя — чего?

В этот день все изменилось. Я прекратил безумные сверхурочные, прекратил работу в условиях постоянного стресса. Я перестал бросать ручки в стену и писать функции C по 3000 строк. Я решил, что я буду получать удовольствие от своей работы, выполняя ее хорошо, а не занимаясь глупостями.

Я расстался со своей должностью настолько профессионально, насколько это было возможно, и стал консультантом. С того дня я уже никогда не называл другого человека «начальником».

Как избежать давления

Лучший способ сохранять спокойствие под давлением — избегать ситуаций, *создающих* давление. Возможно, это не решит проблему полностью, но по крайней мере сведет к минимуму и сократит продолжительность напряженных периодов.

Обязательства

Как мы узнали в главе 10, важно избегать принятия обязательств на сроки, в соблюдении которых вы не уверены. Бизнес всегда будет

стараться добиться от вас таких обязательств, потому что он хочет полностью исключить риск. А мы должны позаботиться о том, чтобы риски объективно оценивались и представлялись бизнесу таким образом, чтобы тот мог правильно управлять ими. Принятие нереалистичных обязательств противоречит этой цели и оказывает плохую услугу как бизнесу, так и нам самим.

Иногда обязательства принимаются за нас. Иногда мы вдруг узнаем, что начальство что-то пообещало заказчику, не проконсультировавшись с нами. В таких случаях профессиональная честь требует помочь бизнесу найти возможность выполнения этих обязательств. Тем не менее та же профессиональная честь *не требует* от нас *принятия* этих обязательств.

Различие достаточно принципиальное. Профессионал всегда помогает бизнесу найти способ достижения его целей, но он не всегда принимает на себя обязательства, принятые за него. В конечном итоге, если нам не удастся выполнить обязательства, принятые за нас, то ответственность за это должен нести тот, кто эти обязательства принял.

Легко сказать... Когда ваш бизнес разваливается, а зарплата откладывается из-за нарушенных обязательств, трудно устоять перед давлением. Но если вы вели себя профессионально, то по крайней мере сможете искать новую работу с достоинством и чистой совестью.

Как сохранить чистоту

Чтобы двигаться быстро и не нарушать сроков, в коде необходимо сохранять чистоту. Профессионал не поддается искушению устроить грязь в коде, чтобы быстро двигаться вперед. Грязно — всегда значит медленно!

Сохранение чистоты в системе, коде и архитектуре помогает избежать давления. Это вовсе не означает, что вы должны до бесконечности «полировать» код. Речь о другом: о непримиримом отношении к грязи. Мы знаем, что грязь замедлит нашу работу, приведет к срыву сроков и нарушению обязательств. Соответственно мы всеми силами пытаемся сохранить результаты нашей работы в настолько чистом состоянии, насколько это возможно.

Дисциплина в кризисных ситуациях

Чтобы понять, во что вы по-настоящему верите, понаблюдайте за собой в кризисной ситуации. Если во время кризиса вы не отклоняетесь от своих рабочих методов, значит вы действительно убеждены в их эффективности.

Если вы следуете методологии разработки через тестирование (TDD) в обычное время, но отказываетесь от нее во время кризиса, значит вы не верите в полезность TDD. Если ваш код остается чистым в обычное время, а в кризис вы разводите в нем грязь, значит вы не верите, что грязь замедляет вашу работу. Если вы используете парное программирование во время кризиса, но не в обычной ситуации, значит вы полагаете, что парное программирование эффективнее индивидуального.

Выберите те методы, с которыми вы комфортно ощущаете себя в кризисной ситуации. *А потом используйте их постоянно.* Использование этих методов — лучший способ избежать кризиса.

Не изменяйте свое поведение в напряженной ситуации. Если ваши методы действительно оптимальны, то они должны соблюдаться даже в самые тяжелые времена.

Как вести себя в тяжелой ситуации

Предотвращение и устранение кризисов — дело, конечно, хорошее, но иногда вы оказываетесь под давлением независимо от своего желания. Иногда проект просто занимает больше времени, чем планировалось изначально. Иногда исходная архитектура оказывается неверной и ее приходится перерабатывать. Иногда вы лишаетесь ценного работника или заказчика. Иногда вы принимаете обязательство, которое не удастся выполнить. Что делать в таких случаях?

Без паники

Возьмите свой стресс под контроль. Бессонные ночи не помогут выполнить работу быстрее. Ругань тоже не поможет. Но самое худшее, что

вы можете сделать, — это спешка! Боритесь с этим искушением любой ценой. Спешка только затянет вас еще глубже на дно.

Наоборот, притормозите и продумайте задачу. Проложите курс к лучшему из возможных решений, а затем двигайтесь по этому курсу в разумном, стабильном темпе.

Взаимодействие

Уведомите свою группу и начальство о неприятностях. Изложите свой план по выходу из кризиса. Обратитесь к ним за информацией и советом. Избегайте сюрпризов. Ничто не сердит людей и не делает их менее рациональными так, как сюрпризы. Сюрпризы повышают уровень стресса десятикратно.

Доверяйте своим методам

Если вы попали в трудную ситуацию, *доверяйте своим методам*. Они для того и *нужны*, чтобы помочь вам выбраться из тяжелой ситуации. В такое время следует особенно тщательно следить за соблюдением правил, а не ставить их под вопрос или отказываться от них.

Вместо того чтобы в панике искать хоть что-нибудь, что поможет вам ускорить работу, вы должны сознательно и целенаправленно соблюдать требования выбранных методов. Если вы следуете методологии разработки через тестирование, пишите еще больше тестов, чем обычно. Если вы выбрали бескомпромиссный рефакторинг, действуйте еще бескомпромисснее. Если вы ограничиваете размеры функций, делайте их еще меньше. Чтобы выйти из тяжелой ситуации, необходимо довериться тому, в эффективности чего вы уже уверены — вашим методам.

Помощь

Парное программирование! Когда становится жарко, найдите напарника, который захочет программировать в паре с вами. Так вы выполните свою работу быстрее и с меньшим количеством дефектов. Партнер поможет вам держаться в рамках методологий и не впадать в панику. Партнер увидит то, что вы упустили, подаст полезную идею и примет эстафету, если вы устанете.

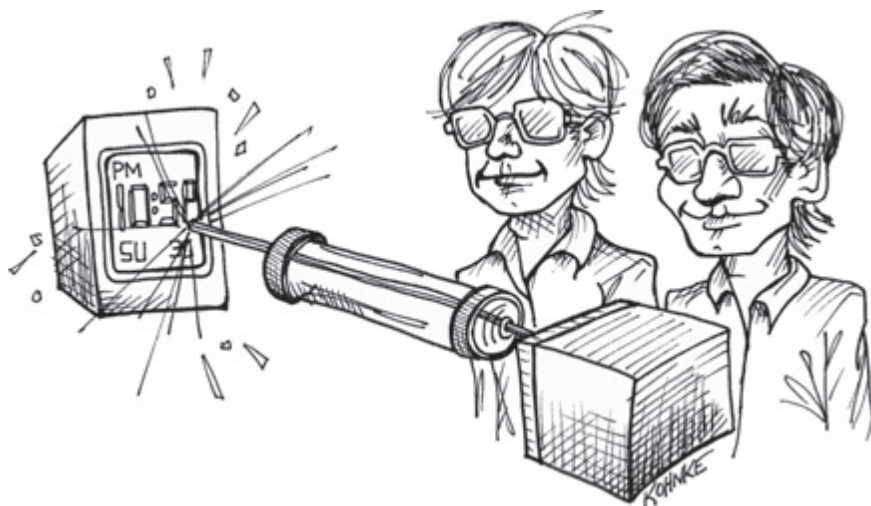
Соответственно, когда вы видите, что кто-то из ваших коллег попал в напряженную ситуацию, предложите ему поработать в паре. Помогите ему выбраться из ямы, в которой он оказался.

Заклучение

Держите напряженные ситуации под контролем. Прежде всего постарайтесь не попадать в них, а если не получилось — держитесь. Чтобы избежать проблемной ситуации, контролируйте свои обязательства, соблюдайте правила методологий и поддерживайте чистоту в коде. Чтобы выдержать давление, сохраняйте спокойствие, общайтесь с коллегами, соблюдайте правила и обращайтесь за помощью.

12

Сотрудничество



Программы обычно создаются группами разработчиков. Одним из условий эффективной работы группы является профессиональное взаимодействие участников. Быть одиночкой или отшельником в группе непрофессионально.

В 1974 году мне было 22 года. Прошло полгода с момента брака с моей замечательной женой Энн-Мэри. До рождения нашего первого ребенка, Анджелы, оставался еще год. Тогда я работал в одном из подразделений Teradyne, называвшемся Chicago Laser Systems.

Рядом со мной трудился мой приятель по средней школе Тим Конрад. В свое время мы с ним занимались всякими интересными вещами:

собирали компьютеры в его подвале, мастерили «лестницы Якова»¹ в моем, учили друг друга программировать для PDP-8 и собирать настоящие калькуляторы из микросхем и транзисторов.

На работе мы программировали систему, которая использовала лазеры для высокоточной обрезки электронных компонентов (резисторов, конденсаторов и т. д.) В частности, мы нарезали кристаллы для первых цифровых часов, Motorola Pulsar.

Программирование велось на компьютере M365, клоне Teradyne PDP-8. Система писалась на ассемблере, а исходные файлы хранились на магнитных лентах. Хотя мы могли править код в экранном редакторе, процесс был достаточно сложным, поэтому для чтения кода и предварительной правки использовались в основном печатные листинги.

У нас не было никаких средств поиска по кодовой базе. Не было возможности найти все места вызова заданной функции или использования заданной константы. Как нетрудно представить, это основательно тормозило нашу работу.

И вот однажды мы с Тимом решили написать генератор перекрестных ссылок. Программа должна была читать ленты с исходным кодом и выводить на печать списки всех символических имен с указанием файла и номера строки, в которой это имя использовалось.

Исходная программа была достаточно простой. Она читала ленту, разбирала ассемблерный код, строила таблицу символических имен и добавляла в нее ссылки. Программа прекрасно работала, но была жутко медленной. Обработка главной операционной программы (MOP) занимала около часа.

Такая скорость объяснялась тем, что растущая таблица символических имен хранилась в одном буфере. Каждый раз, когда программа находила новую ссылку, она вставлялась в буфер, а остаток буфера сдвигался на несколько байт, чтобы освободить место.

Мы с Тимом не были экспертами по структурам данных и алгоритмам. Мы никогда не слыхали о хеш-таблицах и бинарном поиске. Мы понятия не имели, как ускорить алгоритм. Мы просто знали, что наша программа работает слишком медленно.

Тогда мы стали пробовать одно решение за другим. Мы объединяли ссылки в связанный список. Мы оставляли пропуски в массиве и увеличивали буфер только после их заполнения. Мы пытались создавать

¹ Деревянная игрушка; см. http://en.wikipedia.org/wiki/Jacob%27s_ladder_%28toy%29. — *Примеч. перев.*

связанные списки пропусков. Мы опробовали множество безумных идей. Мы стояли у доски в офисе, рисовали диаграммы структур данных и занимались вычислениями для прогнозирования быстродействия. Мы ежедневно приходили на работу с новыми идеями. Постоянно шел интенсивный творческий обмен.

В конечном итоге мы сократили время работы программы до 15 минут, что было довольно близко к времени простого чтения ленты. Тогда мы успокоились.

Программисты и люди

Как правило, программисты не любят работать с людьми. Они находят, что межличностные отношения слишком хлопотны и непредсказуемы. Программисты предпочитают четкое, предсказуемое поведение компьютеров. Они счастливы, часами просиживая в комнате, глубоко сосредоточившись на какой-нибудь интересной задаче.

Ладно, это слишком серьезное обобщение, а из правил существует множество исключений. Многие программисты успешно работают с людьми и им нравится это занятие. Однако в среднем по группам тенденция именно такова, как я изложил. Нас, программистов, радует умеренная сенсорная недостаточность и глубокое погружение в работу — своего рода «кокон».

Программисты и работодатели

В 1970-е и 1980-е годы, во время работы в Teradyne, я хорошо освоил процесс отладки. Мне нравилось это занятие, я бросался на сложные задачи с пылом и энтузиазмом. Ни одна ошибка не могла долго прятаться от меня!

Устраняя очередную ошибку, я чувствовал себя победителем! Я отправлялся к своему боссу Кену Файндеру и с энтузиазмом рассказывал, какая интересная мне попала ошибка. Но однажды Кен в отчаянии воскликнул: «Ошибки не интересны. Их просто нужно исправлять!»

В этот день я усвоил важный урок. Хорошо с энтузиазмом относиться к тому, чем вы занимаетесь. Но при этом также стоит помнить о целях людей, которые вам платят.

Первая обязанность профессионального программиста — заботиться об интересах своих работодателей. Это означает, что вы должны общаться

со своими начальниками, бизнес-аналитиками, тестерами и другими участниками группы, чтобы *глубоко понимать* коммерческие цели проекта. Никто не заставляет вас становиться экспертом в области бизнеса. Просто *нужно* понимать, зачем вы пишете свой код и какую пользу он приносит вашей фирме.

Худшее, что может сделать профессиональный программист, — в блаженном неведении укрыться в технологическом убежище, когда бизнес горит и рушится вокруг него. Ваша *работа* — удерживать бизнес на плаву!

Итак, профессиональный программист должен выделить время на изучение коммерческой стороны дела. Он говорит с пользователями о программном продукте, с которым они работают. Он общается с людьми из отдела продаж и маркетинга по поводу возникающих проблем. Он говорит с начальством, чтобы понять как краткосрочные, так и долгосрочные цели группы.

Короче говоря, профессиональный программист обращает внимание на корабль, на котором он плывет.

За всю мою карьеру меня уволили с должности программиста всего один раз. Это произошло в 1976 году, когда я работал на Outboard Marine Corp. Я помогал писать систему автоматизации производства, которая использовала компьютеры IBM System/7 для контроля за десятками автоматов алюминиевого литья в главном цехе предприятия.

С технической точки зрения это была интересная и сложная работа. Архитектура System/7 приводила меня в восторг, а система автоматизации производства тоже была весьма захватывающей.

У нас была хорошая группа. Руководитель группы Джон был компетентным и целеустремленным специалистом. Два моих коллеги-программиста были приятными людьми, готовыми прийти на помощь. Под наш проект была выделена специальная лаборатория, в которой мы работали. Наш бизнес-партнер участвовал в работе и находился с нами в лаборатории. Наш начальник Ральф был компетентным и ответственным.

Все было замечательно. Проблема была во мне. Я с энтузиазмом относился к проекту и технологии, но в возрасте 24 лет я не мог заставить себя думать о бизнесе или внутренней политической структуре.

Первую ошибку я совершил в первый же день. Я пришел на работу без галстука. Я надел галстук на собеседование, и я видел, что все носят галстуки, но не сделал выводов. Итак, в первый день Ральф подошел ко мне и просто сказал: «Мы здесь носим галстуки».

Не могу описать, насколько меня это бесило. Я ощущал глубочайшее раздражение. Я носил галстук каждый день и ненавидел его. Но почему? Ведь я знал, куда я поступаю на работу. Я знал принятые правила. Почему меня это раздражало? Потому что я был эгоистичным, самолюбленным типом.

Я просто не мог приходить на работу вовремя. И мне казалось, что это не важно. В конце концов, я «хорошо справлялся». И это было правдой: я действительно очень хорошо справлялся с написанием своих программ. Бесспорно, я был лучшим программистом в группе. Я мог писать код быстрее и лучше других. Я быстрее находил и устранял проблемы. Я знал, что я был ценным специалистом, так что время и даты для меня значили мало.

Решение о моем увольнении было принято тогда, когда я опоздал на одну из контрольных точек проекта. По всей видимости, Джон сказал нам, что в следующий понедельник ему нужна демо-версия рабочих функций системы. Наверняка я знал об этом, но просто не обратил внимания.

Система находилась в стадии активной разработки. До реальной эксплуатации было еще далеко. Не было причин оставлять систему в рабочем состоянии, когда в лаборатории никого не было. Я последним уходил с работы в пятницу, и, видимо, я оставил систему в неработоспособном состоянии. Тот факт, что понедельник был важной датой, просто не отложился у меня в памяти.

В понедельник я пришел на час позже и увидел, как все мрачно собрались вокруг нерабочей системы. Джон спросил: «Почему система не работает сегодня, Боб?» Я ответил: «Не знаю», и сел за отладку. Я даже тогда не вспомнил о демо-версии, запланированной на понедельник, но по поведению моих коллег было совершенно ясно, что произошло что-то нехорошее. Тогда Джон подошел ко мне, прошептал на ухо: «А если бы пришел Стенберг?» и отошел в негодовании.

Стенберг был вице-президентом по автоматизации. Сегодня его должность назвали бы «руководителем технической службы». Вопрос показался мне бессмысленным. «Ну и что? — подумал я. — Система же не в эксплуатации, так в чем проблема?»

В этот день я получил первое предупредительное письмо. В нем мне предлагалось немедленно изменить свое отношение к работе, иначе «последует немедленное увольнение». Я был в ужасе!

Мне пришлось проанализировать свое поведение и понять, что же я делал не так. Я поговорил с Джоном и Ральфом с твердым намерением изменить себя и свой стиль работы.

И я это сделал! Я перестал опаздывать, начал обращать внимание на внутреннюю политику. Я начал понимать, почему Джон беспокоился по поводу Стенберга. Я понял, как подвел его, оставив на понедельник неработоспособную систему.

Но было слишком поздно. Через месяц я получил второе предупреждение по поводу какой-то тривиальной ошибки. В этот момент мне следовало понять, что письма были простой формальностью, а решение об увольнении уже было принято. Но я твердо решил все исправить и работал еще прилежнее.

Еще через несколько недель мне сообщили об увольнении.

Мне пришлось идти домой к своей беременной 22-летней жене и рассказывать, что я потерял работу. Мне бы не хотелось, чтобы такое когда-нибудь повторилось в моей жизни.

Программисты и программисты

У программистов часто возникают трудности при работе в тесном контакте с другими программистами. Порой это создает серьезные проблемы.

Принадлежность кода

Один из худших признаков неправильно функционирующей команды — когда каждый программист возводит стену вокруг своего кода и запрещает другим программистам прикасаться к нему. Я был в местах, где программисты даже запрещали другим *смотреть* на свой код. Это верный путь к катастрофе.

Однажды я консультировал компанию, производившую высококлассные принтеры. Машины состояли из множества разных компонентов: систем подачи бумаги, печати, укладки бумаги, сшивания листов, резак и т. д. С точки зрения бизнеса эти системы обладали разной ценностью. Система подачи листов была важнее системы укладки, но ни одно устройство не могло сравниться по важности с системой печати.

Каждый программист работал над своей системой. Один писал код для системы подачи листов, другой — код для системы сшивания и т. д.

Все они бдительно охраняли свою технологию и не позволяли никому притрагиваться к своему коду. Политический вес программистов был напрямую связан с коммерческой ценностью устройства. Программист, работавший над устройством печати, обладал непререкаемым авторитетом.

Для технологии такое положение дел имело катастрофические последствия. Мне, консультанту, было хорошо видно массовое дублирование кода и разноречивость в интерфейсах между модулями. Но никакие аргументы с моей стороны не могли убедить программистов (или представителей бизнеса) изменить подход к работе. Ведь их зарплата была связана с важностью устройств, которыми они занимались.

Коллективная принадлежность кода

Разрушьте стены принадлежности кода — код должен принадлежать всей группе. Я предпочитаю работать с группами, в которых любой участник может проверить любой модуль и внести те изменения, которые сочтет нужным. Я предпочитаю, чтобы код принадлежал группе, а не конкретным людям.

Профессиональные разработчики не запрещают коллегам работать со своим кодом. Они не возводят вокруг своего кода стены принадлежности. Напротив, они стараются работать друг с другом над как можно большей частью кода. И работая над другими частями системы, они учатся друг у друга.

Парное программирование

Многие программисты не любят идею парного программирования. Мне это кажется странным, потому что в напряженных ситуациях большинство программистов *объединяется* в пары. Почему? Потому что парное программирование очевидным образом является самым эффективным способом решения задачи. Вспомните старую поговорку: «Две головы лучше, чем одна». Но если парное программирование является самым эффективным способом решения задач в напряженных ситуациях, с чего бы ему не быть самым эффективным способом решения задач в нормальное время?

Я не собираюсь цитировать научные исследования, хотя у меня в запасе найдется немало подходящих цитат. Я не буду рассказывать «случаи из жизни», хотя и их у меня тоже предостаточно. Я даже не собираюсь

указывать, какую часть времени следует проводить за парным программированием. Скажу одно — *профессионалы работают в парах*. Почему? Потому что по крайней мере для некоторых задач эта методология наиболее эффективна. Впрочем, это не единственная причина.

Профессионалы также объединяются в пары, потому что это лучший способ обмена знаниями. Профессионалы не создают «банки знаний» — вместо этого они изучают разные аспекты системы и бизнеса, объединяясь в пары. Они понимают, что хотя у каждого участника группы имеется своя должность, все участники должны быть готовы моментально переключаться в случае необходимости.

Профессионалы объединяются в пары, потому что это лучший способ рецензирования кода. Ни в одной системе не должно быть кода, который не был прорецензирован другими программистами. Существует много механизмов рецензирования кода; в большинстве своем они ужасающе неэффективны. Самый эффективный и действенный способ рецензирования кода — участие в его написании.

Как работать мозжечком

Однажды утром на самом пике бума интернет-коммерции я приехал на поезде в Чикаго. Когда я ступил на платформу, мне в глаза бросился огромный плакат над выходом. Хорошо известная фирма-разработчик приглашала на работу программистов. На плакате было написано: «Поработайте мозжечком рядом с лучшими!»

Столь выдающаяся глупость меня поразила. Жалкие бестолковые рекламщики пытались произвести впечатление на умных, эрудированных, технически грамотных программистов — на людей, которые плохо переносят глупость. Рекламщики пытались создать впечатление, будто на новой работе вы будете обмениваться знаниями с другими умными людьми. К сожалению, упомянутая в рекламе часть мозга — мозжечок — отвечает за координацию движений, а не за интеллект. И те люди, которых пытались привлечь этой рекламой, только ухмылялись над глупой ошибкой.

Но в этой рекламе меня заинтриговало другое. Я представил себе группу людей, пытающихся «поработать мозжечком». Поскольку мозжечок находится в задней части мозга, я представил себе группу программистов, рассевшихся по кабинкам затылками друг к другу, уставившихся на мониторы и оградившихся от внешних раздражителей наушниками. Это нельзя назвать группой.

Профессионалы работают вместе. Невозможно работать вместе, рассевшись по углам с наушниками. Участники группы должны сидеть за общим столом, повернувшись друг к другу *лицом*. Они должны чувствовать опасения своих коллег, слышать их раздраженное бормотание. Между ними должно идти постоянное общение — как вербальное, так и на уровне «языка тела». Они должны взаимодействовать как единое целое.

Возможно, вам кажется, что в одиночку вы будете работать эффективнее. Даже если это и так, отсюда вовсе не следует, что *группа* будет работать эффективнее. И в действительности крайне маловероятно, что в одиночку вы действительно лучше справляетесь со своей работой.

В одних ситуациях работа в одиночку — именно то, что нужно. Иногда вам просто нужно долго и напряженно думать над задачей. В других случаях задача настолько тривиальна, что привлекать к работе другого человека было бы попросту расточительно. Но в общем случае лучше выделять значительную часть времени на работу в тесном контакте с другими и парное программирование.

Заключение

Возможно, кто-то из нас пришел в программирование не для того, чтобы работать с людьми. Ему не повезло: все программирование неразрывно связано с работой с людьми. Мы должны общаться со стороной бизнеса, и мы должны общаться друг с другом.

Знаю, знаю: мы бы предпочли работать в закрытой комнате с шестью большими экранами, каналом ТЗ, параллельным массивом сверхбыстрых процессоров, неограниченным объемом памяти и дискового пространства, а также бесконечным запасом диетической колы и острых кукурузных чипсов. Увы, так не будет. Если вы хотите заниматься программированием, придется научиться общаться.

13

Группы и проекты



Представьте, что вам нужно выполнить множество мелких проектов. Как распределить их между программистами? А если проект только один, но очень большой?

Формирование группы

За прошедшие годы я консультировал многие банки и страховые компании. Единственное, что у них было общего — странный подход к распределению проектов. Банковский проект часто представляет собой относительно малую задачу, которая может быть выполнена одним или двумя программистами за несколько недель. В проект обычно включался руководитель, который одновременно вел другие проекты. К нему добавлялся бизнес-аналитик, который предоставлял требования для других проектов. Дальше добавлялись программисты, которые также работали над другими проектами. Потом один-два тестера, у которых тоже были другие проекты.

Заметили закономерность? Проект настолько мал, что никто не может заниматься только им одним. Все участники уделяют проекту 50% или даже 25% времени.

А теперь правило: половины человека не бывает.

Бессмысленно приказывать программисту посвятить половину времени проекту А, а другую половину — проекту В, особенно если в двух проектах участвуют разные руководители, бизнес-аналитики, программисты и тестеры. Разве подобную мешанину можно назвать группой?

«Притертая» группа

Группы формируются не сразу. Между участниками постепенно налаживаются отношения. Они учатся работать друг с другом, узнают странности, сильные и слабые стороны своих коллег. Со временем участники «притираются» друг к другу.

В «притертой» группе есть что-то волшебное: она способна творить чудеса. Участники понимают друг друга, поддерживают и требуют максимальной отдачи. Благодаря их взаимодействию достигаются результаты.

«Притертая» группа обычно содержит около дюжины участников. Их может быть и больше (до 20) или меньше (до 3), но оптимальный размер обычно где-то около 12. В группу должны входить программисты, тестеры и аналитики. И у нее должен быть руководитель проекта.

Соотношение численности программистов и тестеров/аналитиков изменяется в широких пределах, но пропорция 2:1 вполне разумна. Таким образом, хорошо «притертая» группа из 12 человек может состоять из

семи программистов, двух тестеров, двух аналитиков и руководителя проекта.

Аналитики разрабатывают требования и пишут для них автоматизированные приемочные тесты. Тестеры тоже пишут автоматизированные приемочные тесты, но отличаются от аналитиков направленностью. И те и другие пишут тесты, но аналитики ориентируются на коммерческую ценность, а тестеры — на правильность работы кода. Аналитики пишут «оптимистичные» тесты, а тестеры беспокоятся о том, что может пойти не так, и пишут тесты для выявления сбоев и граничных случаев.

Руководитель проекта следит за прогрессом и принимает меры к тому, чтобы группа понимала сроки и приоритеты.

Один из участников группы может совмещать выполнение своих обязанностей с ролью наставника, ответственного за соблюдение технологических процессов и методов. Он становится своего рода «коллективной совестью», когда у группы возникает соблазн нарушить правила под давлением сроков.

Созревание

Чтобы участники разобрались в своих различиях и договорились между собой, а группа действительно «притерлась», потребуется время. Процесс может занять полгода или даже год. Но когда это случается, происходит чудо. «Притертая» группа вместе планирует, вместе решает задачи, вместе разбирается с проблемами и добивается результатов.

Разбивать такие группы только из-за завершения проекта слишком расточительно. Лучше сохранить группу в прежнем составе и поручать ей новые проекты.

Что сначала — группа или проект?

Банки и страховые компании пытались формировать группы на базе проектов. Такой подход неверен в принципе: группы просто не могут «притереться». Участники работают над проектом в течение короткого времени, притом посвящая ему лишь часть своего рабочего времени, а следовательно, не имеют возможности сработаться.

Профессиональные фирмы-разработчики поручают проекты существующим «притертым» группам, а не формируют группы для конкретного проекта. «Притертая» команда может вести сразу несколько проектов

и распределять работу в соответствии со своими предпочтениями, квалификацией и способностями участников.

Но как управлять такой группой?

У каждой группы имеется определенная скорость работы¹, а проще говоря, объем работы, выполняемый за фиксированный промежуток времени. Некоторые группы измеряют свою скорость в *пунктах* в неделю (пункты — единица сложности). Функциональность каждого проекта, над которым работает группа, разбивается на блоки, сложность которых оценивается в пунктах. В дальнейшем производительность группы оценивается по количеству пунктов, реализованных за неделю.

Скорость — статистическая метрика. Группа может реализовать 38 пунктов в одну неделю, 42 пункта в другую и 25 в третью. По мере накопления данных вычисляется усредненный показатель.

Руководство может задать цели для каждого проекта. Например, если средняя скорость группы равна 50 и группа работает над тремя проектами, руководство может попросить группу распределить усилия в пропорции 15/15/20.

Помимо того что над проектами работает «притертая» команда, у этой схемы есть и другое преимущество. В напряженной ситуации бизнес-сторона может сказать: «Проект В в критическом состоянии; в следующие три недели направьте на него 100% усилий».

Столь быстрое перераспределение приоритетов практически невозможно со случайно подобранными группами, тогда как «притертая» группа, работающая над двумя или тремя проектами одновременно, легко выполняет подобные «развороты на месте».

Дилемма владельца проекта

Один из аргументов против метода, за который я выступаю, заключается в том, что он частично лишает владельцев проекта уверенности и власти. Владельцы проекта, для которого была создана специальная группа, могут рассчитывать на полную отдачу этой группы. Они знают, что формирование и роспуск группы обходится недешево, поэтому

¹ Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003, pp. 20–22; Mike Cohn, *Agile Estimating and Planning*, Upper Saddle River, NJ: Prentice Hall, 2006.

бизнес не будет отнимать у них группу по краткосрочным тактическим соображениям.

С другой стороны, если проекты даются «притертым» группам и эти группы ведут сразу несколько проектов, бизнес может менять приоритеты по своему усмотрению. Из-за этого владелец проекта менее уверен в будущем. У него могут внезапно отобрать ресурсы, на которые он рассчитывает.

Честно говоря, я предпочитаю вторую ситуацию. Руки бизнеса не должны быть связаны искусственными сложностями формирования и роспуска групп. Если бизнес решит, что один проект обладает более высоким приоритетом, он должен иметь возможность быстро перераспределить ресурсы. Владелец проекта должен сам привести веские доводы в его пользу.

Заключение

Создать хорошую группу сложнее, чем создать проект. Соответственно лучше формировать группы с постоянным составом, которые переходят от одного проекта к другому и могут заниматься несколькими проектами одновременно. При формировании группы следует дать ей достаточно времени для того, чтобы она сработалась, а затем использовать ее как инструмент для успешного выполнения многих проектов.

14

Наставники, ученики и мастерство



Уровень выпускников в области компьютерных технологий меня постоянно разочаровывает. Дело не в том, что они недостаточно умны или талантливы — просто их не учили тому, что необходимо знать настоящему программисту.

Диплом для неподготовленных

Однажды я проводил собеседование с девушкой, занятой дипломной работой в области компьютерных технологий в известном университете. Она подала документы на летнюю стажировку. Я попросил ее написать для меня решение несложной задачи, а она ответила: «Вообще-то я не пишу код».

Пожалуйста, перечитайте предыдущий абзац, а потом переходите к следующему.

Я спросил, какие курсы по программированию она посещала во время обучения. Она ответила, что не посещала никаких.

Возможно, вам стоит прочитать главу с самого начала. Вы хотя бы убедитесь в том, что вы не попали в альтернативную Вселенную и вам не приснился страшный сон.

Вы спросите, как студент с учебной программой магистра в области компьютерных технологий мог обойтись без единого курса программирования? Я тогда задал себе тот же вопрос. И до сих пор не нахожу ответа.

Конечно, это самый выдающийся случай в серии разочарований, испытанных мной во время собеседований с выпускниками. Не все выпускники в области компьютерных технологий настолько плохи — вовсе нет! Однако я заметил у них нечто общее: почти все они *изучали программирование самостоятельно* до поступления в университет и продолжали изучать его, несмотря на учебу в университете.

Не поймите меня неправильно. Я считаю, что в университете можно получить отличное образование. Просто наряду с этим можно украдкой проскользнуть через систему и получить диплом... не получив ничего более.

Другая проблема: даже самые лучшие курсы по компьютерным технологиям обычно не готовят молодого выпускника к тому, с чем он столкнется в этой отрасли. И дело даже не столько в недостатках учебных программ, как в реалиях почти всех дисциплин. То, что вы узнаете во время учебы, часто очень сильно отличается от того, с чем вы сталкиваетесь на работе.

Обучение

Как мы учимся программировать? Позвольте рассказать вам одну историю о наставниках и учениках.

Digi-comp I, мой первый компьютер

В 1964 году моя мама подарила мне на 12-летие маленький механический компьютер. Он назывался Digi-Comp I ¹ и состоял из трех пластиковых триггеров и шести пластиковых конъюнкторов. Выходы триггеров можно было соединять с входами конъюнкторов или наоборот — выходы конъюнкторов со входами триггеров. Короче говоря, устройство позволяло создать трехразрядный конечный автомат.

В комплект входила инструкция с несколькими программами. Чтобы запрограммировать машину, следовало протолкнуть пластиковую трубочку (короткий отрезок соломинки для коктейля) в штифт на триггере. В руководстве было точно указано, куда вставлять трубочки, но ни слова не говорилось о том, что эти трубочки делают. Знали бы вы, как это меня раздражало!

Я часами разглядывал машину и пытался понять, как она работает на самом нижнем уровне; но даже для спасения жизни я бы не смог заставить ее сделать то, что мне нужно. На последней странице руководства было сказано, что если я заплачу доллар, то мне пришлют руководство по программированию машины².

Я послал доллар и стал ждать с нетерпением двенадцатилетнего подростка. Когда книга пришла, я проглотил ее за день. Это было простое изложение булевой алгебры с азами булевой логики, законов ассоциативности и дистрибутивности и теоремы де Моргана. В руководстве было показано, как выразить задачу в виде последовательности логических формул и как сократить эти формулы для 6 конъюнкторов.

Я написал свою первую программу. Я написал уравнения, сократил их и отобразил на трубки и штифты машины. И *программа заработала!*

Даже сейчас от воспоминаний у меня мурашки бегут по спине. Те впечатления определили судьбу 12-летнего подростка почти полвека

¹ В Интернете есть много сайтов, моделирующих этот забавный маленький компьютер.

² У меня до сих пор сохранилось это руководство. Оно занимает почетное место на одной из моих книжных полок.

назад. Программирование захватило меня, и моя жизнь уже никогда не будет прежней.

Помните момент, когда заработала ваша первая программа? Она изменила вашу жизнь или открыла перед вами путь, с которого вы уже не смогли свернуть?

Я не мог во всем разобраться сам. У меня были наставники. Очень добрые и способные люди (которым я многим обязан) не пожалели времени на то, чтобы написать изложение булевой алгебры, доступное для 12-летнего подростка. Они связали математическую теорию с практическими навыками программирования маленького механического компьютера; благодаря им я смог заставить компьютер сделать то, что мне было нужно.

Я только что снял с полки свой экземпляр этого судьбоносного учебника. Я храню его в пластиковом пакете с застёжкой. Время берет свое: страницы пожелтели и стали хрупкими. Но сила слов сияет, как и прежде. Элегантное изложение булевой алгебры занимает всего три страницы неплотного текста. Пошаговый анализ формул каждой из исходных программ до сих пор производит впечатление. Это был настоящий шедевр; работа, которая изменила жизнь по крайней мере одного молодого человека. А я, скорее всего, даже не узнаю имен авторов.

ЕСР-18 в средней школе

В 15-летнем возрасте, когда я учился в старших классах, мне нравилось проводить время в математической лаборатории. Однажды туда привезли устройство размером с циркулярный станок. Это был учебный компьютер для средних школ, он назывался ЕСР-18. Нашей школе был предоставлен двухнедельный пробный период.

Я стоял в стороне и слушал разговоры учителей и техников. У машины были 15-разрядные слова (что такое «слово»?) и барабанный накопитель на 1024 слова. (Тогда я уже знал, что это такое, но только теоретически.)

Когда машину включили, она издала свист наподобие того, который издает реактивный самолет при взлете. Я предположил, что это раскручивался барабан. Когда устройство набрало обороты, оно работало относительно тихо.

Машина была очаровательной. Она напоминала офисный стол, над которым возвышалась потрясающая панель управления — как на капитанском мостике боевого корабля. Панель была украшена рядами

лампочек, которые также можно было нажимать, как кнопки. Сидя за таким столом, человек ощущал себя словно в кресле капитана Керка¹.

Я наблюдал за тем, как техники нажимали кнопки. Когда кнопку нажимали, лампочка загоралась, а при повторном нажатии она гасла. Также они нажимали другие кнопки с названиями типа «Загрузить» и «Выполнить».

Кнопки в каждом ряду были объединены в пять групп по три. Мой Digi-Comp тоже был трехразрядным, так что я мог читать восьмеричные цифры в двоичной форме. Было нетрудно понять, что каждая строка представляет пять восьмеричных цифр.

Я слышал, как техники, нажимавшие кнопки, что-то бормочут. Они нажимали 1, 5, 2, 0, 4 в строке «Буфер памяти», говоря при этом: «Сохранить в 204». Они нажимали 1, 0, 2, 1, 3 и бормотали: «Загрузить 213 в аккумулятор». Там был ряд кнопок с подписью «Аккумулятор!»

Через десять минут моему 15-летнему разуму было абсолютно ясно, что 15 означает «сохранить», а 10 — «загрузить», что в аккумуляторе находились сохраняемые или загружаемые данные, а остальные числа были номерами одного из 1024 слов на барабане. (Так вот что такое «слово»!)

Слово за слово (непреднамеренный каламбур), мой пытливый ум все глубже проникал в коды инструкций и концепции. К тому моменту, когда техники ушли, я уже понимал основные принципы работы машины.

Этим же днем, во время часов для самостоятельной работы, я пробрался в математическую лабораторию и начал экспериментировать с компьютером. К тому времени я уже отлично знал, что проще попросить прощения, чем добиться разрешения! Я ввел программу, которая умножала содержимое аккумулятора на 2 и прибавляла 1. Я ввел в аккумулятор 5, запустил программу — и увидел в аккумуляторе 13! Программа работала!

Я ввел еще несколько таких же простых программ, и они тоже работали так, как положено. Я был повелителем Вселенной!

Сутки спустя я понял, насколько я был глуп — и как мне повезло. Я нашел в лаборатории памятку, в которой были перечислены все инструкции и коды операций, в том числе и тех, которые я не мог узнать, наблюдая за техниками. Я с радостью узнал, что известные мне коды были интерпретированы правильно, а остальные вызвали

¹ Персонаж сериала «Звездный путь». — *Примеч. перев.*

прилив энтузиазма. Однако среди новых инструкций я заметил инструкцию остановки HLT. Так уж совпало, что инструкция остановки была словом из одних нулей. И еще совпало то, что я включал в конец каждой из своих программ слово из одних нулей, чтобы стереть содержимое аккумулятора. Концепция остановки мне просто не приходила в голову. Мне казалось, что программа сама остановится, когда все сделает!

Помню, я однажды сидел в математической лаборатории, наблюдая за тем, как один из учителей боролся со своей программой. Он пытался ввести два числа в десятичной форме с подключенного телетайпа, а потом распечатать их сумму. Каждый, кто пытался программировать подобные задачи на машинном языке мини-компьютеров, знает, что они отнюдь не тривиальны. Нужно прочесть символы, разбить их на цифры, затем преобразовать в двоичную форму, просуммировать, преобразовать их обратно в десятичную систему и закодировать в символы. Поверьте, когда программа вводится в двоичном виде с передней панели, все намного хуже!

Я наблюдал за тем, как он вставил в программу команду остановки и запустил. (О! Хорошая мысль!) Примитивная точка прерывания позволила ему проанализировать содержимое регистров и понять, что сделала программа. Помню, как он пробормотал: «Ого! Как быстро!»

Понятия не имею, какой алгоритм он использовал. Такое программирование для меня еще оставалось чем-то вроде волшебства. И он ни слова не сказал мне, пока я смотрел ему через плечо. Никто не говорил со мной об этом компьютере. Думаю, меня считали помехой, на которую не стоит обращать внимания, — чем-то вроде мошки, порхающей по лаборатории. Достаточно сказать, что ни ученик, ни учителя не обладали высокими навыками социальных коммуникаций.

В итоге его программа заработала. Выглядело это потрясающе: он медленно вводил два числа, потому что, несмотря на более ранние восклицания, компьютер работал довольно медленно (подумайте, сколько времени занимало чтение последовательных слов с вращающегося барабана в 1967 году). Когда он нажал «ввод» после второго числа, компьютер яростно помигал лампочками, а потом начал выводить результат. На каждую цифру уходило около секунды. Он вывел все цифры кроме последней, потом секунд пять мигал еще яростнее, вывел последнюю цифру и остановился.

Откуда взялась пауза перед последней цифрой? Этого я так и не узнал. Но зато я понял, что выбор решения задачи может иметь принципиальные

последствия для пользователя. Даже при том, что программа выводила правильный ответ, с ней *все равно* было что-то не так.

Это тоже было обучение. Конечно, не такое обучение, на какое я бы мог надеяться. Было бы намного лучше, если бы один из этих учителей взял меня под опеку и стал работать со мной. Но даже наблюдение за ними позволяло мне стремительно получать новые знания.

Нетрадиционное обучение

Я рассказал эти две истории, потому что они описывают два совершенно разных типа обучения, ни один из которых обычно не подразумевается под этим термином. В первом случае я учился у авторов очень хорошо написанного учебника. Во втором случае я учился, наблюдая за людьми, которые активно старались не обращать на меня внимания. В обоих случаях приобретенные знания были глубокими и основательными.

Конечно, у меня были и другие учителя. Добрый сосед, работавший в Teletype, подарил мне коробку с 30 телефонными реле. Вот что я вам скажу: дайте парню реле и трансформатор от игрушечной железной дороги — и он покорит мир!

Другой добрый сосед увлекался любительским радио и научил меня пользоваться мультиметром (который я незамедлительно сломал). Владелец магазина офисных принадлежностей разрешал мне зайти и «поиграть» с его очень дорогим программируемым калькулятором. А еще рядом был отдел продаж Digital Equipment Corporation, который разрешал мне зайти и «поиграть» с PDP-8 и PDP-10.

Также был большой Джим Карлин — BAL-программист, который спас меня от увольнения с первой работы. Он помог отладить программу на Cobol, выходящую за пределы моего понимания. Он научил меня читать дампы ядра, форматировать код при помощи пустых строк, звездочек и комментариев. Он дал мне первый толчок на пути к мастерству. Жаль, что я не мог отплатить услугой за услугу, когда гнев моего начальства обрушился на него год спустя.

Но по правде говоря, список подошел к концу. В начале 1970-х годов было не так уж много опытных программистов. Во всех остальных местах, где я работал, я был самым опытным. Не было никого, кто бы помог разобраться мне, что такое настоящее профессиональное программирование. Не было образца для подражания, который бы научил

меня, как себя вести и на что обращать внимание в первую очередь. Все это мне пришлось узнавать самому, и процесс был отнюдь не из простых.

Горький опыт

Как я уже рассказывал, меня все-таки уволили с той работы по автоматизации производства в 1976 году. Хотя с технической точки зрения я был очень компетентным, я не научился обращать внимание на бизнес и его цели. Даты и сроки ничего не значили для меня. Я забыл о важной демонстрации программы в понедельник утром, оставил систему в неработоспособном состоянии в пятницу и опоздал на работу в понедельник под осуждающими взглядами всех остальных. Мой начальник прислал мне уведомление о том, что я должен немедленно изменить свое отношение к работе или меня уволят. Для меня это был «тревожный звонок»: я пересмотрел свои взгляды на жизнь и карьеру и внес существенные изменения в свое поведение — кое-что об этом вы уже читали. Но было поздно, слишком поздно. Маховик был уже запущен, и мелочи, на которые ранее никто не обратил бы внимания, вдруг стали важны. Итак, как я ни старался, в конечном итоге меня вывели из фирмы.

Не стоит и говорить, что приносить такие вести беременной жене с двухлетней дочерью невесело. Но я собрался с духом и перенес полезные жизненные уроки на следующую работу, на которой я оставался 15 лет и которая заложила настоящий фундамент текущей карьеры.

В конечном итоге я выжил и преуспел. Но должен быть и другой, более эффективный путь. Мне было бы намного проще, если бы у меня был настоящий наставник — тот, кто покажет мне «что к чему»; человек, за которым я мог бы наблюдать, помогая с выполнением мелких задач, который будет рецензировать и направлять мою раннюю работу. Человек, который станет образцом для подражания и научит меня необходимым профессиональным ценностям и навыкам. Сэнсэй. Руководитель. Наставник.

Ученичество

А что происходит в медицине? Думаете, больницы берут на работу выпускников и с первого дня отправляют их в операционную выполнять операции на сердце? Конечно, нет.

Профессиональная медицина разработала методологию интенсивного обучения, плотно укутанную ритуалами и сглаженную традицией. Профессиональная медицина наблюдает за университетами и следит за тем, чтобы выпускники получали лучшее образование. В процессе учебы время *приблизительно поровну* распределяется между занятиями в классах и клинической работой в больницах с профессионалами.

После выпуска и до получения лицензии новоиспеченные врачи обязаны пройти годичную практику под руководством специалистов, которая называется интернатурой. Происходит интенсивное обучение на месте работы: интерна окружают образцы для подражания и наставники.

После завершения интернатуры разные медицинские специализации требуют от трех до пяти лет дальнейшей практики, называемой ординатурой. Врач-стажер постепенно набирается уверенности, выполняя еще более серьезные задачи, оставаясь в окружении (и под наблюдением) более опытных врачей.

Многие специальности требуют от одного до трех лет аспирантуры, в течение которой продолжается обучение студента по специальности и накопление практического опыта.

И только *после этого* молодой специалист допускается к экзаменам и аттестации.

Мое описание профессиональной медицины несколько идеализировано и, вероятно, крайне неточно. Но факт остается фактом: когда риск велик, никто не отправляет недавних выпускников в операционную, время от времени подбрасывая им пациентов и ожидая, что из этого выйдет что-нибудь путное. Так почему же это происходит в области программирования?

Конечно, количество смертей из-за ошибок в программах относительно невелико. С другой стороны, экономические потери весьма значительны. Из-за недостаточной подготовки своих разработчиков компании теряют огромные суммы.

По какой-то причине в отрасли разработки ПО родилась мысль, что программист есть программист и сразу же после получения диплома можно приступать к написанию кода. Некоторые фирмы нанимают парней прямо со школьной скамьи, собирают из них «группы» и поручают строить критически важные системы. Безумие!

Художники так не поступают. Сантехники так не поступают. Электрики так не поступают. Наверное, даже повара в «МакДональдсе» так не поступают! Мне кажется, что компании, нанимающие выпускников

в области компьютерных технологий, должны тратить на их обучение больше, чем «Макдональдс» тратит на подготовку своих работников.

И не стоит обманывать себя, будто это не важно. Ставки высоки. Наша цивилизация живет на программах. Именно программы занимаются перемещением и обработкой информации, наполнившей нашу повседневную жизнь. Программы управляют двигателями, передачей и тормозами наших машин. Они поддерживают баланс наших банковских вкладов, рассылают счета и получают оплату. Программы стирают нашу одежду и сообщают время. Они выводят изображение на экраны телевизоров, отправляют текстовые сообщения, делают телефонные звонки и развлекают нас, когда нам скучно. Они повсюду.

Раз мы доверяем разработчикам все аспекты наших жизней, от пустяковых до самых важных, на мой взгляд, разумный период обучения и практики под руководством специалистов будет вполне уместным.

Период ученичества

Итак, как же молодые выпускники *должны* вливаться в ряды профессиональных программистов? Какой путь они должны пройти? С какими препятствиями столкнуться? Каких целей они должны достичь? Давайте рассмотрим профессиональные уровни программистов по убыванию квалификации.

Мастер

К этой категории относятся программисты, возглавлявшие более одного серьезного программного проекта. Как правило, они имеют более чем 10-летний стаж работы с разными системами, языками и операционными системами. Они умеют руководить и координировать работу нескольких команд, являются квалифицированными проектировщиками и разработчиками архитектур и могут запросто запрограммировать что угодно. Им предлагались руководящие должности, но они либо отклонили предложение, либо вернулись обратно после согласия, либо интегрировали их в свою основную техническую роль. Для поддержания своей квалификации в этой роли они читают техническую литературу, учатся, тренируются, работают и учат. Именно мастера несут ответственность за реализацию проекта с технической стороны.

Ремесленник

Рядовые программисты — обученные, компетентные и энергичные. В этот период своей карьеры они учатся работать в группах и выполнять функции руководителя. Они хорошо разбираются в современной технологии, но обычно им не хватает опыта работы с разнообразными системами. Обычно ремесленник знает один язык, одну систему, одну платформу; но он старается узнать больше. Стаж работы в этой категории сильно различается; среднее значение составляет около 5 лет. На ближнем конце оси находятся недавние ученики, а на дальнем — зарождающиеся мастера.

Наставниками ремесленников являются мастера или более опытные ремесленники. Молодым ремесленникам редко предоставляется самостоятельность. За их работой плотно наблюдают, а их код проверяется. По мере накопления опыта самостоятельность растет, контроль становится менее прямолинейным и в конечном итоге преобразуется в равноправное рецензирование кода.

Ученики/интерны

Карьера выпускника начинается с позиции ученика. У учеников нет никакой самостоятельности — их очень плотно контролируют ремесленники. Сначала ученики вообще не выполняют никаких задач, просто помогая ремесленникам. Это должно быть время чрезвычайно интенсивного парного программирования. Именно в это время изучаются и закрепляются методы и приемы. Именно в это время закладывается фундамент системы профессиональных ценностей.

Ремесленники становятся учителями. Они следят за тем, чтобы ученики знали принципы и паттерны проектирования, методы и ритуалы. Ремесленники обучают их TDD, рефакторингу, искусству оценки и т. д. Они назначают ученикам книги, раздают упражнения и учебные задачи; они следят за их прогрессом.

Ученичество должно длиться не менее года. За это время, если ремесленники пожелают принять новичка в свои ряды, они обращаются к мастерам с рекомендацией. Мастера изучают новичка — как в личном собеседовании, так и посредством анализа достижений. Если мастера соглашаются, то ученик переходит в ремесленники.

Реальность

И снова скажу, что это описание идеализированное и гипотетическое. Но если немного изменить терминологию, вы поймете, что оно не так уж сильно отличается от сегодняшней ситуации. Выпускников опекают молодые руководители групп, которых опекают руководители проектов и т. д. Проблема в том, что в большинстве случаев опека имеет нетехническую природу! В большинстве компаний технического контроля нет вообще. Программисты получают прибавки и повышения, потому что... потому что так положено.

Различия между сегодняшним положением дел и моей идеализированной программой заключается в ориентированности последней на техническое обучение, опеку и контроль. Оно проявляется в самом представлении о том, что профессиональные ценности и техническую сметку нужно объяснять, культивировать и взращивать. В нашем современном стерильном подходе отсутствует ответственность старших за обучение молодежи.

Профессионализм

Итак, теперь мы имеем возможность определить само понятие *профессионализма*. Что оно собой представляет? Чтобы понять это, начнем с понятия «*профессионал*». Оно ассоциируется у нас с искусностью и качеством, создает впечатление опытности и компетентности. Профессионалом мы называем того, кто работает быстро, но без спешки, кто разумно оценивает ситуацию и выполняет свои обязательства. Профессионал знает, когда нужно говорить «нет», но честно пытается сказать «да».

Профессионализм — *склад ума*, присущий профессионалам. Профессионализм — стиль жизни с определенными представлениями о ценностях, методологиях, приемах, подходе и ответах на вопросы.

Но как профессионалы переходят на такой стиль жизни? Как они приобретают этот склад ума?

Профессионализм передается от одного человека к другому. Старшие обучают ему младших. Коллеги обмениваются им между собой. Старшие, наблюдая за младшими, видят его со стороны и учатся заново. Профессионализм распространяется словно своего рода интеллектуальный вирус. Вы «заражаетесь» профессионализмом, наблюдая за другими и позволяя укорениться ему в своем сознании.

Как убедить людей

Нельзя убедить людей быть профессионалами. Нельзя убедить их принять профессиональное отношение к делу. Аргументы неэффективны. Данные непоследовательны. Ситуационные исследования не значат ничего. Принятие профессионального отношения является не столько рациональным, сколько эмоциональным решением. Это очень человеческое решение.

Как же подвести людей к переходу на профессиональное отношение? Да, оно заразительно, но только в том случае, если вы можете наблюдать за его проявлением. Следовательно, вы должны демонстрировать его. Станьте образцом для подражания. Вы становитесь профессионалом сами и показываете свой профессионализм другим. А затем пусть концепция сама сделает всю работу.

Заключение

Учебное заведение может научить теории программирования. Но оно не учит и не может научить практике, методам и профессионализму. Все это приобретается годами личной практики — как в роли обучаемого, так и в роли наставника. Пришло время всем нам, работникам отрасли программирования, признать, что задачу воспитания следующего поколения разработчиков придется решать нам, а не университетам. Пришло время принять программу наставничества, интернатуры и долгосрочной опеки.

Приложение

Инструментарий



В 1978 году я работал в Teradyne над телефонной тестовой системой, о которой упоминал ранее. Система состояла примерно из 80 тысяч строк кода ассемблера M365. Исходный код хранился на магнитных лентах.

Ленты напоминали 8-дорожечные стереокассеты, которые были так популярны в 1970-е годы. Лента была склеена, а накопитель мог перематывать только в одном направлении. Ленты в кассетах имели длину 10, 25, 50 и 100 футов. Чем длиннее была лента, тем больше времени занимала «перемотка», так как накопителю приходилось просто перематывать ее вперед до «точки загрузки». Перемотка 100-футовой ленты до точки загрузки занимала около 5 минут, поэтому мы осмотрительно подходили к выбору длины лент¹.

¹ Ленты можно было перематывать только в одном направлении. Если происходила ошибка чтения, ленту нельзя было перемотать назад и прочесть сбойный участок заново. Приходилось прекращать то, что вы делаете, перематывать ленту к точке загрузки и начинать все с начала. Это происходило два-три раза в день. Ошибки записи тоже происходили достаточно часто, и накопитель не мог их обнаружить. По этой причине мы всегда записывали ленты парами, а затем проверяли пары после завершения. Если одна из лент была записана с ошибкой, мы немедленно делали копию. Если испорчены были обе ленты (что бывало очень редко), то вся операция повторялась. Такая была жизнь в 1970-е годы.

На логическом уровне ленты делились на файлы. На одну ленту можно было записать столько файлов, сколько на ней помещалось. Чтобы найти нужный файл, приходилось загружать ленту, а затем последовательно читать файлы, пока не будет найден нужный. На стене висела распечатка каталога с исходным кодом; по ней мы определяли, сколько файлов нужно пропустить, чтобы найти нужный.

На полке в лаборатории лежала эталонная 100-футовая копия ленты с исходным кодом. Чтобы отредактировать файл, мы ставили в один накопитель эталонный экземпляр, а в другой — 10-футовую пустую (рабочую) ленту. Эталонная лента проматывалась до нужного файла, после чего файл копировался на рабочую ленту. Тогда мы «перематывали» обе ленты в начало, и эталонная лента ставилась обратно на полку.

На доске в лаборатории была прикреплена специальная распечатка эталонной ленты. Когда мы создавали копии файлов, которые требовалось отредактировать, мы прикрепляли цветную булавку рядом с именем этого файла. Так производился контроль изменений!

Редактирование производилось в экранном режиме. Мы пользовались очень хорошим текстовым редактором ED-402 — близким аналогом *vi*. Страница читалась с ленты, мы редактировали ее содержимое, записывали ее обратно и читали следующую. Страница обычно состояла примерно из 50 строк кода. Мы не могли «заглянуть вперед» на ленту и увидеть предстоящие страницы и не могли вернуться назад к уже отредактированным страницам. Поэтому мы использовали листинги.

В листингах отмечались все предстоящие изменения, *после чего* файлы редактировались в соответствии с пометкой. *Никто* не писал и не изменял код спонтанно! Это было бы самоубийством.

После внесения изменений во все файлы, которые требовалось отредактировать, мы объединяли их с содержимым эталонного экземпляра. Полученная лента использовалась для проведения компиляции и тестирования.

Когда мы завершали тестирование и убеждались в том, что изменения работают, мы смотрели на распечатку. Если на ней не было новых кнопок, то рабочая лента просто переименовывалась в эталонную, а наши кнопки снимались с доски. Если на доске появлялись новые кнопки, то мы снимали свои и передавали рабочую ленту их владельцу для слияния результатов.

В группе было трое разработчиков. У каждого были кнопки своего цвета, поэтому мы могли легко определить, кто взял на редактирование те или иные файлы. А поскольку мы все работали в одной лаборатории

и постоянно говорили друг с другом, текущее состояние кодовой базы постоянно хранилось в наших головах. Так что распечатка часто оказывалась излишней, и часто мы обходились без нее.

Инструменты

Современным разработчикам доступны самые разнообразные инструменты программирования. От некоторых стоит держаться подальше, но есть и такие, которыми должен уверенно владеть любой разработчик. В этой главе описан мой личный текущий инструментарий. Я не привожу полного описания всех представленных инструментов, так что обзор не является исчерпывающим. Я описываю лишь то, что использую сам.

Управление исходным кодом

В том, что касается управления исходным кодом, обычно стоит использовать программы с открытым кодом. Почему? Потому что они пишутся разработчиками и для разработчиков. Иначе говоря, разработчики пишут программы с открытым кодом для самих себя, когда им понадобится работоспособное решение.

Существует немало дорогих, коммерческих «корпоративных» систем контроля версий. По моему мнению, они продаются не столько разработчикам, сколько руководителям, менеджерам и «инструментальным группам». Список функций таких программ выглядит впечатляюще. К сожалению, в нем часто не оказывается того, что действительно нужно разработчикам. И главной проблемой обычно оказывается *скорость*.

«Корпоративные» системы управления исходным кодом

Вполне возможно, что ваша фирма уже вложила целое состояние в «корпоративную» систему управления исходным кодом. В таком случае примите мои соболезнования. Вероятно, по политическим соображениям вы не сможете просто сказать: «Дядюшка Боб говорит, что ей не надо пользоваться». Тем не менее существует простой выход.

Вы можете регистрировать исходный код в «корпоративной» системе в конце каждой итерации (каждые две недели или около того), а в середине итераций использовать систему с открытым кодом. Все будут довольны, вы не нарушите корпоративных правил, а ваша производительность останется высокой.

Пессимистическая и оптимистическая блокировка

В 1980-е годы пессимистическая блокировка казалась хорошей идеей. В конце концов, простейший путь к решению проблем параллельного обновления — их распараллеливание. Если я редактирую файл, то вам лучше к нему не прикасаться. Система цветных кнопок, использованная нами в конце 1970-х, была своего рода механизмом пессимистической блокировки. Если файл был помечен кнопкой, то другие не должны были его редактировать.

Конечно, у пессимистической блокировки есть свои недостатки. Если заблокировать файл и уйти в отпуск, то все остальные пользователи, которые захотят работать с файлом, окажутся в тупике. Даже если файл останется заблокированным на день-два, это задержит работу других.

Современные инструменты значительно лучше справляются со слиянием параллельно редактируемых исходных файлов. Ведь это нетривиальная задача: программа анализирует два разных файла и предка этих двух файлов, а затем применяет различные стратегии для определения способа интеграции параллельных изменений. И надо сказать, хорошо справляется с этой работой.

Так что время пессимистической блокировки прошло. Теперь нам не нужно устанавливать блокировку файлов при редактировании. Более того, вообще не нужно беспокоиться о блокировке отдельных файлов — мы запрашиваем сразу всю систему и редактируем нужные файлы.

Когда все будет готово к регистрации изменений, выполняется операция обновления. Она сообщает нам, не было ли более ранней регистрации изменений другими пользователями, выполняет автоматическое слияние большинства изменений, находит конфликты и помогает выполнить остальные слияния. После этого объединенный код регистрируется в базе.

Позднее в этой главе я достаточно подробно освещу роль автоматизированных тестов и непрерывной интеграции в этом процессе. А пока

достаточно сказать, что код, не прошедший всех тестов, ни при каких условиях не должен регистрироваться в базе. *Никогда.*

CVS/SVN

Одна из традиционных систем управления исходным кодом — CVS — была хороша для своего времени, но в современных проектах она уже начинает отставать. Хотя CVS отлично подходит для работы с отдельными файлами и каталогами, она не слишком хорошо справляется с переименованием файлов и удалением каталогов. А уж подкаталоги Attic... Чем меньше говорить об этом, тем лучше.

С другой стороны, система Subversion работает очень хорошо. Она позволяет получить доступ ко всей системе за одну операцию. В ней легко выполняются операции обновления, слияния и закрепления изменений. При отсутствии разветвляющихся изменений системы SVN достаточно просты в управлении.

Разветвление

До 2008 года я избегал любых форм ветвления, кроме простейших. Если разработчик создает ветвь, то эта ветвь должна быть возвращена в основную линию до конца итерации. Но я так сурово относился к ветвлению, что оно крайне редко применялось в тех проектах, в которых я участвовал.

Если вы используете SVN, то я по-прежнему считаю, что это хорошая политика. Однако в последнее время появились новые инструменты, которые полностью изменяют ситуацию. Я говорю о *распределенных* системах управления исходным кодом. В этой категории моим любимым инструментом является *git*. Сейчас я расскажу об этой системе более подробно.

git

Я начал использовать *git* в конце 2008 года. Эта система полностью изменила мой подход к управлению исходным кодом. Объяснения того, почему эта программа так сильно изменила правила игры, выходят за рамки книги. Тем не менее сравнение рис. П.1 с рис. П.2 красноречивее многих слов, которые я здесь приводить не буду.

- More bug fixes
- Docs now say that Java 1.5 is required.
- Bug fix
- Many usability and behavioral improvements.
- Clean up
- Added PAGE_NAME and PAGE_PATH to pre-defined variables.
- Added ** to !path widget.
- link to the fixture gallery
- fixture gallery release 2.0 (2008-06-09) copied into the trunk wiki at
- Firefox compatability for invisible collapsible sections; removed .ce
- Updated documentation suite for all changes since last release.
- Enhancement to handle nulls in saved and recalled symbols. Adde
- Added a "Prune" Properties attribute to exclude a page and its chil
- Fixed type-o
- Added check for existing child page on rename.
- Added "Rename" link to Symbolic Links property section; renamed
- Adjusted page properties on recently added pages such that they c
- Enhanced Symbolic Links to allow all relative and absolute path for
- Cleaned up renamPageReponder a bit more.
- Cleaned Up PathParser names a bit. Pop -> RemoveNameFromE
- Cleaned up RenamePageResponder a bit. Fixed TestContentsHel
- updated usage message
- Fixed a bug wherein variables defined in a parent's preformatted bl
- Added explicit responder "getPage" to render a page in case query
- Tweaks to TOC help text.
- New property: Help text; TOCWidget has rollover balloon with new
- Redundant to the JUnit tests and elemental acceptance tests.
- Removed the last of the [acd] tags.
- !contents -f option enhancement to show suite filters in TOC list; fix
- TOC enhancements for properties (-p and PROPERTY_TOC and F
- 1) Render the tags on non-WikiWord links;
- Added http:// prefix to google.com for firewall transparency.
- Isolate query action from additional query arguments. For example
- Accommodate query strings like "?suite&suiteFilter=X"; prior logic v
- Cleaned up AliasLinkWidget a bit.

Рис. П.1. Проект FitNesse под управлением Subversion

На рис. П.1 показан ход разработки проекта FitNesse за несколько недель под управлением SVN. Вы видите эффект моей жесткой политики отказа от ветвления. Мы попросту не использовали его, зато в главной ветке часто выполнялись операции обновления, слияния и закрепления.

На рис. П.2 показана структура нескольких недель разработки того же проекта с использованием *git*. Как видно из рисунка, операции

ветвления и слияния происходят постоянно. Дело не в том, что я ослабил свою политику ветвления; просто такой рабочий процесс стал самым очевидным и эффективным. Отдельные разработчики создают ветви с очень коротким сроком жизни, а затем объединяют их с результатами своих коллег тогда, когда считают нужным.

Также обратите внимание на то, что на рисунке не видна «настоящая» главная ветвь. Дело в том, что ее попросту нет. Разработчики хранят

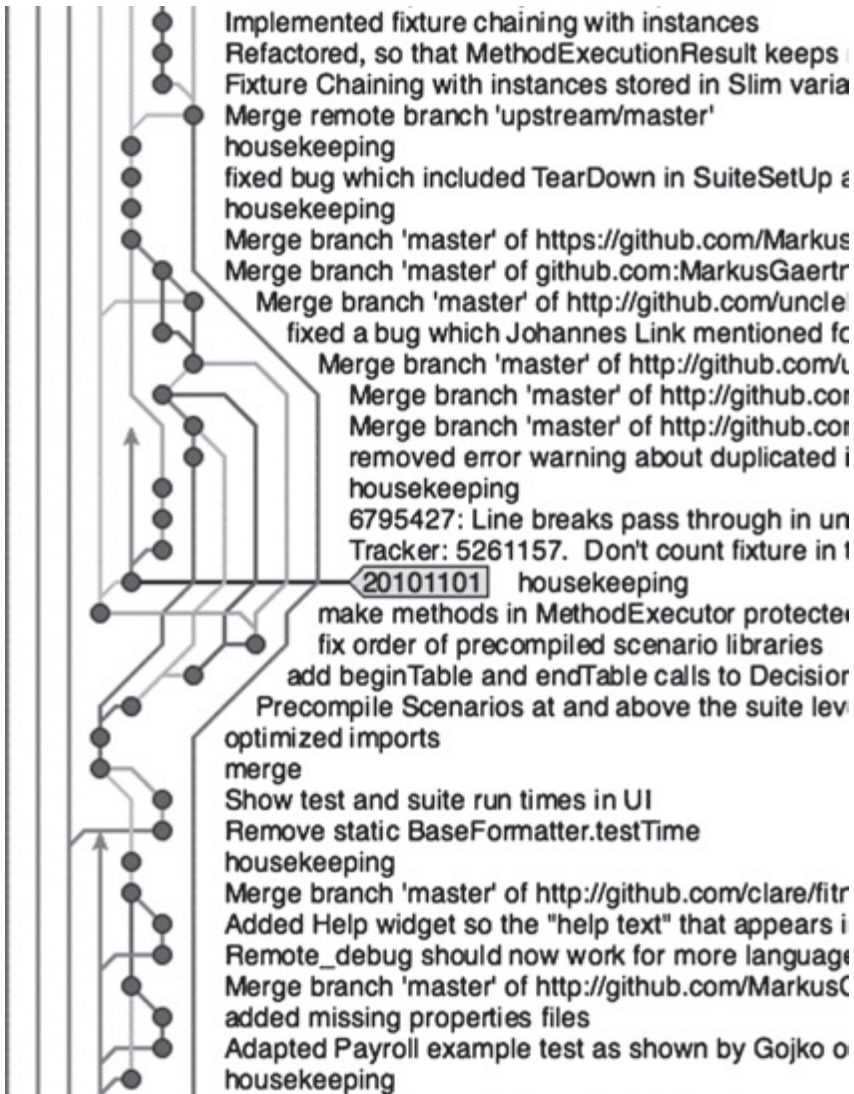


Рис. П.2. Проект FitNesse под управлением git

на своих локальных компьютерах копию всей истории проекта. Они вносят изменения и регистрируют их в своей локальной копии, а затем синхронизируют с копиями коллег по мере надобности.

Правда, я поддерживаю специальный эталонный репозиторий, в котором хранятся все опубликованные версии и внутренние сборки. Но называть его «главной ветвью» значило бы совершенно не понимать суть дела. В действительности это всего лишь удобный «снимок» всей истории, которая сохраняется локально всеми разработчиками.

Если вы не поняли что-то из сказанного, это нормально. На первых порах *git* вызывает немало затруднений. К тому, как работает эта система, нужно привыкнуть. Но будьте уверены: *git* и другие подобные системы — это будущее управления исходным кодом.

IDE/редактор

Мы, разработчики, проводим большую часть времени за чтением и редактированием кода. Инструменты, используемые нами для этих целей, значительно изменились за прошедшие годы. Некоторые из них обладают неимоверной мощностью, а некоторые почти не изменились с 1970-х годов.

vi

Казалось бы, эпоха использования *vi* как основного редактора для разработки давно прошла. В наши дни появились инструменты, значительно превосходящие *vi* по своим возможностям, и другие простые текстовые редакторы того же типа. Однако в последнее время наблюдается заметный всплеск популярности *vi*, который объясняется его простотой, удобством использования, скоростью и гибкостью. Хотя *vi* и уступает *Emacs* или Eclipse по широте возможностей, он остается быстрым и мощным редактором.

При этом я уже не являюсь опытным пользователем *vi*. Когда-то меня называли «богом» *vi*, но это время давно прошло. Я использую *vi* время от времени, когда мне нужно быстро отредактировать текстовый файл. Я недавно использовал его для быстрого изменения исходного файла Java в удаленном режиме. Но объем полноценного кода, написанного мной в *vi* за последние 10 лет, ничтожно мал.

Emacs

Emacs до сих пор остается одним из самых мощных редакторов и, вероятно, останется им еще несколько ближайших десятилетий. Его мощь обеспечивается внутренней моделью на базе *lisp*. В категории инструментов редактирования общего назначения ни один другой редактор даже отдаленно не может с ним конкурировать. С другой стороны, я думаю, что *Emacs* не может полноценно конкурировать со специализированными интегрированными средами разработки (IDE), которые сейчас заняли лидирующее положение. Редактирование кода не относится к задачам редактирования общего назначения.

В 1990-е годы я был фанатом *Emacs*. Я просто не рассматривал другие варианты. Тогдашние редакторы с управлением мышью были смешными игрушками, к которым ни один разработчик не мог относиться серьезно. Но в начале 2000-х я познакомился с IntelliJ — моим фаворитом среди IDE, и уже не оглядывался назад.

Eclipse/IntelliJ

Я — пользователь IntelliJ. Я люблю эту систему. Я использую ее для написания кода на Java, Ruby, Clojure, Scala, Javascript и многих других языках. Она была написана программистами, которые понимают, что нужно программисту при написании кода. За прошедшие годы IntelliJ почти никогда не подводила меня, а впечатления почти всегда оставались положительными.

Среда Eclipse по своей мощи и масштабу сравнима с IntelliJ. Эти две среды качественно превосходят Emacs по возможностям редактирования Java-кода. В этой категории существуют и другие IDE, но я не упоминаю их здесь, потому что у меня нет непосредственного опыта их использования.

Эти интегрированные среды отличаются от таких инструментов, как Emacs, прежде всего чрезвычайно мощными возможностями манипулирования кодом. Например, IntelliJ позволяет извлечь суперкласс из класса всего одной командой. Вы можете переименовывать переменные, извлекать методы, преобразовывать наследование в композицию... и это далеко не полный список.

С этими инструментами редактирование кода переходит с уровня строк и символов на уровень более сложных манипуляций. Разработчик думает не о нескольких следующих символах и строках, которые он собирается ввести, а о следующих преобразованиях. Короче говоря, модель

программирования в этих новых системах значительно изменяется и становится более производительной.

Конечно, за широту возможностей приходится платить. Освоение интегрированных сред требует немалого времени и усилий, и фаза начальной подготовки проекта становится довольно заметной. Кроме того, эти инструменты весьма требовательны — для их работы необходимы значительные вычислительные мощности.

TextMate

Редактор TextMate мощен и нетребователен к ресурсам. Правда, он не умеет выполнять потрясающие манипуляции, на которые способны IntelliJ и Eclipse. У него нет мощного *lisp*-ядра и библиотеки *Emacs*. Он не обладает скоростью и гибкостью *vi*. С другой стороны, его можно быстро освоить, а выполняемые операции интуитивно понятны.

Я использую TextMate время от времени, особенно для спонтанного программирования на C++. В крупном проекте я бы использовал *Emacs*, но для небольших задач на C++ мне просто лень с ним возиться.

Отслеживание задач

Сейчас я использую Pivotal Tracker. Эта система проста и элегантна, она хорошо интегрируется с гибкими/итеративными методологиями и позволяет всем заинтересованным сторонам и разработчикам быстро общаться друг с другом. Я очень доволен ей.

В очень мелких проектах я иногда использую Lighthouse. Эта система очень проста и удобна в настройке и использовании, но по широте возможностей она и близко не подходит к Tracker.

Также в некоторых случаях я просто использую вики. Такое решение хорошо подходит для внутренних проектов. Вики позволяет применить любую организационную схему на ваше усмотрение, никто не заставляет вас использовать конкретный процесс или жесткую структуру. Этот вариант тоже чрезвычайно прост для понимания и использования.

Иногда лучшей системой отслеживания задач оказывается настенная доска с набором карточек. Доска делится на столбцы (например, «Нужно сделать», «В работе» и «Готово»). Разработчики просто переносят карточки из одного столбца в другой по мере работы над задачами. Пожалуй, это одна из самых распространенных систем отслеживания задач в современных группах, использующих гибкие методологии.

Я рекомендую своим клиентам начать с подобной «ручной» системы, прежде чем приобретать специализированные программы. Освоив ручную систему, клиент получает знания, которые позволят ему сделать разумный выбор — иногда в пользу той же доски с карточками.

Счетчики дефектов

Группе разработчиков определенно необходим список текущих задач. К их числу относятся как задания на реализацию новых возможностей и функций, так и исправления ошибок. Для группы разумного размера (от 5 до 12 разработчиков) такой список должен содержать от нескольких десятков до сотен позиций. Не тысяча! Если в вашей системе тысячи ошибок, с ней что-то не так. Если ваш список насчитывает тысячи задач и/или функций, с ней что-то не так. В общем случае список должен быть относительно небольшим, поэтому для работы с ним должно быть достаточно такого несложного инструмента, как вики, Lighthouse или Tracker.

На рынке имеются коммерческие программы, которые выглядят неплохо. Я видел, как мои клиенты пользуются ими, но пока не имел возможности поработать лично. Не имею ничего против таких инструментов — при условии, что список задач остается небольшим и легко управляемым. Когда средства отслеживания задач вынуждены работать со списками из тысяч позиций, само слово «отслеживание» теряет смысл. Список текущих задач превращается в «свалку текущих задач» (и часто пахнет соответствующим образом).

Непрерывная сборка

В последнее время для обеспечения непрерывной сборки я использую Jenkins. Система нетребовательна, проста, а работа с ней не требует длительной подготовки. Вы загружаете программу, запускаете ее, проводите несложную настройку конфигурации — а дальше все работает. Очень удобно.

Мой подход к непрерывной сборке прост: свяжите ее с системой управления исходным кодом. Каждый раз, когда в базе регистрируется измененный код, должна выполняться непрерывная сборка, отчет о результатах которой выдается группе.

Группа просто обязана следить за тем, чтобы сборка всегда проходила успешно. Если сборка не проходит, это должно стать тревожным

сигналом, а группа должна немедленно собраться для решения проблемы. Ни при каких условиях недействующая сборка не должна существовать более суток.

В проекте FitNesse я требую, чтобы каждый разработчик выполнял сценарий непрерывной сборки перед регистрацией своих изменений. Сборка занимает менее 5 минут, так что она необременительна. Если в ходе сборки обнаруживаются проблемы, разработчики устраняют их до регистрации. Таким образом, автоматическая сборка редко сталкивается с какими-либо проблемами. Чаще всего проблемы при автоматической сборке возникают из-за настроек рабочей среды, поскольку моя среда автоматической сборки значительно отличается от сред разработчиков.

Инструменты модульного тестирования

Для каждого языка существуют свои специализированные средства модульного тестирования. Лично я использую *JUnit* для Java, *rspec* для Ruby, *NUnit* для .Net, *Midje* для Clojure и *CppUTest* для C и C++.

Впрочем, какой бы инструмент модульного тестирования вы ни выбрали, все они должны обладать набором базовых свойств и функций.

1. Тесты должны запускаться легко и быстро. Не важно, как именно это делается — при помощи плагинов IDE или простых утилит командной строки; главное, чтобы разработчики могли запускать эти тесты по своему усмотрению. Команда запуска должна быть тривиально простой. Например, я запускаю свои тесты *CppUTest* в *TextMate* простым нажатием клавиш *command+M*. Я связал эту комбинацию с запуском *makefile*, который автоматически выполняет тесты и выводит короткий однострочный отчет в том случае, если все тесты прошли успешно. *IntelliJ* поддерживает *JUnit* и *rspec*, поэтому от меня требуется лишь нажать кнопку. Для выполнения *NUnit* я использую плагин *Resharper*.
2. Программа должна выдавать четкий визуальный признак прохождения/непрохождения теста. Не важно, будет ли это зеленая полоса на графическом экране или консольное сообщение «Все тесты прошли». Суть в том, чтобы вы могли быстро и однозначно определить, что все тесты прошли. Если для определения результата тестирования приходится читать многострочный отчет или, того

хуже, сравнивать выходные данные двух файлов, — условие не выполнено.

3. Программа должна выдавать четкий визуальный признак прогресса. Не важно, будет ли это графический индикатор или строка постепенно появляющихся точек — главное, чтобы пользователь видел, что процесс идет, а тестирование не остановилось и не прервалось.
4. Программа должна препятствовать взаимодействию между отдельными тестовыми сценариями. *JUnit* решает эту проблему, создавая новый экземпляр тестового класса для каждого тестового метода; таким образом предотвращается использование переменных экземпляров для взаимодействия между тестами. Другие инструменты запускают тестовые методы в случайном порядке, чтобы исключить возможную зависимость от конкретного порядка выполнения тестов. Независимо от конкретного выбора механизма, программа должна принять меры к обеспечению независимости тестов. Зависимые тесты — опасная ловушка, которую необходимо избегать.
5. Программа должна по возможности упрощать написание тестов. Например, *JUnit* предоставляет удобный API для проверки условий (assertions), а также использует рефлексии и атрибуты Java для того, чтобы отличать тестовые функции от обычных. В результате хорошие IDE могут автоматически идентифицировать тесты, а вы избавляетесь от хлопот с подготовкой тестов.

Инструменты компонентного тестирования

Инструменты этой категории предназначены для тестирования компонентов на уровне API. Их роль — определение поведения компонента на языке, понятном для бизнеса и специалистов по контролю качества. В идеале бизнес-аналитики и специалисты по контролю качества должны писать спецификации с использованием данных инструментов.

Определение

Инструменты компонентного тестирования в большей степени, чем какие-либо другие инструменты, выражают наше представление о том, что понимать под «выполненной» работой. Когда бизнес-аналитики и специалисты по контролю качества создают спецификацию,

определяющую поведение компонента, и когда эта спецификация выполняется как набор тестов, которые либо проходят, либо нет, понятие «готово» принимает совершенно однозначный смысл: «Проходят все тесты».

FitNesse

Мой любимый инструмент компонентного тестирования — FitNesse. Я написал большую часть его кода, и я являюсь его основным автором. В общем, это мое детище.

FitNesse — система на базе вики, которая позволяет бизнес-аналитикам и специалистам по контролю качества писать тесты в очень простом табличном формате. Эти таблицы имеют много общего с таблицами Парнаса по форме и предназначению. Тесты легко объединяются в пакеты, которые можно выполнить в любой момент.

Система FitNesse написана на Java, однако она может использоваться для тестирования систем на любых языках, потому что она взаимодействует с базовой системой тестирования, которая может быть написана на любом языке. В настоящее время поддерживаются языки Java, C#/ .NET, C, C++, Python, Ruby, PHP, Delphi и другие.

В основе FitNesse лежат две системы: Fit и Slim. Система Fit была написана Уордом Каннингемом, послужила источником вдохновения для FitNesse и является ее близким родственником. Slim — более простая и лучше портируемая система тестирования, которой сейчас пользуются многие сторонники FitNesse.

Другие инструменты

Я знаю еще несколько других инструментов, которые можно отнести к категории компонентного тестирования.

- Система RobotFX была разработана инженерами Nokia. Как и в FitNesse, в ней используется табличный формат, но не на базе вики. RobotFX просто работает с неструктурированными файлами, созданными в Excel или другой аналогичной программе. Программа написана на Python, но при использовании соответствующих «мостов» может тестировать системы, написанные на любом языке.
- Green Pepper — коммерческая программа, в некоторых аспектах похожая на FitNesse. Базируется на популярной вики *confluence*.

- Cucumber — текстовая программа, работающая на ядре Ruby, но подходящая для тестирования на многих разных платформах. В языке Cucumber используется популярный стиль Given/When/Then.
- JBehave — аналог и «идеологический родитель» Cucumber. Программа написана на Java.

Инструменты интеграционного тестирования

Инструменты компонентного тестирования также могут использоваться для выполнения многих интеграционных тестов, но они плохо подходят для тестов, управляемых через пользовательский интерфейс.

В общем случае количество тестов, управляемых через пользовательский интерфейс, должно быть минимальным, потому что пользовательские интерфейсы хорошо известны своей нестабильностью. Из-за этой нестабильности тесты, выполняемые через пользовательский интерфейс, становятся крайне непрочными.

Впрочем, некоторые тесты должны выполняться через пользовательский интерфейс — прежде всего это тесты самого интерфейса. Кроме того, некоторые комплексные тесты должны проходить через всю собранную систему, включая пользовательский интерфейс.

Для тестирования пользовательского интерфейса я предпочитаю использовать программы Selenium и Watir.

UML/MDA

В начале 1990-х годов я очень надеялся, что отрасль автоматизированной разработки ПО кардинально изменит работу программиста. Заглядывая в будущее из этих бурных дней, я думал, что к сегодняшнему дню все уже будут программировать с диаграммами на более высоком уровне абстракции, а текстовый код останется в прошлом.

Как же я ошибался! Мои мечты не только не сбылись, но и любые попытки движения в этом направлении потерпели полный крах. Дело даже не в том, что не существует инструментов и систем, демонстрирующих потенциал такого подхода; просто эти инструменты толком не реализуют эту мечту и ими никто не хочет пользоваться.

Я мечтал, что разработчики избавятся от подробностей текстового кода и начнут строить системы на более высоком диаграммном уровне. В самых дерзких мечтах программисты вообще становились лишними. Архитекторы могли строить целые системы по диаграммам UML. Ядро системы — огромное, бесстрастное и равнодушное к бедам обычных программистов — преобразует диаграммы в исполняемый код. Так выглядела великая мечта модельно-ориентированной архитектуры (MDA, Model Driven Architecture).

К сожалению, у великой мечты имеется один крошечный изъян. MDA предполагает, что проблемы создает код. Однако код сам по себе не создает проблем и никогда не создавал. Проблема заключается в *детализации*.

Детализация

Программисты управляют подробностями. Да, именно этим мы и занимаемся. Мы описываем поведение системы в мельчайших подробностях. Для этой цели мы используем текстовые языки, потому что они чрезвычайно удобны (для примера возьмите хотя бы английский).

Какими же подробностями мы управляем?

Вы знаете, чем различаются два символа `\n` и `\r`? Первый, `\n` — перевод строки (line feed), а второй, `\r` — возврат каретки (carriage return). Что такое «каретка»?

В 1960-х и начале 1970-х годов основным устройством вывода для компьютеров был телетайп. Модель ASR33¹ была самой распространенной.

Устройство состояло из печатающей головки, которая могла печатать до 10 символов в секунду. Печатающая головка состояла из цилиндра, на котором были нанесены рельефные символы. Цилиндр поворачивался так, чтобы к бумаге был обращен правильный символ, после чего молоточек ударял по цилиндру и прижимал его к бумаге. Между цилиндром и бумагой проходила красящая лента, краска с которой переносилась на бумагу в виде символа.

Печатающая головка перемещалась на каретке. С каждым символом каретка сдвигалась на одну позицию вправо, вместе с ней смещалась и печатающая головка. Когда каретка доходила до конца 72-символьной строки, каретку приходилось возвращать в начало строки, отправляя символ возврата каретки (`\r = 0x0D`); без этого печатающая головка

¹ http://en.wikipedia.org/wiki/ASR-33_Teletype

продолжала бы печатать символы в 72 столбце, и от многократной печати там появился бы черный прямоугольник.

Конечно, этого было недостаточно. Возврат каретки не приводил к смещению бумаги к следующей строке. Если бы после возврата каретки не передавался символ перевода строки (`\n = 0x0A`), то новая строка была бы напечатана поверх старой.

Итак, для телетайпа ASR33 строки должны были завершаться последовательностью `«\r\n»`. Однако и здесь была необходима осторожность, потому что возврат каретки мог занять более 100 миллисекунд. Если ограничиться отправкой `«\n\r»`, то следующий символ мог оказаться напечатанным во время обратного хода каретки, а в середине строки появился бы смазанный символ. Для надежности символы конца строки часто дополнялись одним или двумя символами «забой» (`0xFF`) ¹.

В 1970-е годы, когда телетайпы стали постепенно выходить из употребления, в операционных системах UNIX последовательность конца строки сократилась до `\n`. Однако другие операционные системы — например DOS — продолжали использовать обозначение `\r\n`.

Когда вам в последний раз попался текстовый файл, использующий «неправильное» обозначение? Я сталкиваюсь с этой проблемой не реже раза в год. Два идентичных файла с исходным кодом не сравниваются, а их контрольные суммы различаются, потому что в них используются разные завершители строк. Текстовые редакторы некорректно переносят слова или разделяют строки двойным интервалом, потому что они интерпретируют `\r\n` как две строки. Некоторые программы понимают `\r\n`, но не распознают `\n\r...` И так далее.

Вот что я имею в виду под детализацией. Попробуйте-ка закодировать логику обработки завершителей строк на UML!

Без изменений и надежд

Движение MDA обещало, что использование диаграмм вместо кода позволит исключить из программирования большое количество второстепенных подробностей. До настоящего момента эти обещания не

¹ Эти символы были чрезвычайно полезны для редактирования перфолент. По правилам символы «забой» игнорировались при вводе. Их код `0xFF` означал, что на ленте были пробиты отверстия во всех позициях. Таким образом, любой символ можно было преобразовать в «забой» — для этого было достаточно пробить «забой» поверх старого символа. Итак, если вы допускали ошибку во время ввода программы, можно было вернуться к предыдущему символу, нажать клавишу «забой», а потом продолжить ввод.

оправдались. Как оказалось, в коде не так уж много лишних подробностей, которые можно было бы исключить при помощи диаграмм. Более того, диаграммы тоже содержат свои второстепенные подробности. Они обладают своей грамматикой, синтаксисом, правилами и ограничениями. В конечном счете различия в уровне детализации оказываются незначительными.

Движение MDA обещало, что работа с диаграммами будет осуществляться на более высоком уровне абстракции, чем работа с кодом — подобно тому, как Java находится на более высоком уровне абстракции по сравнению с ассемблером. Но и эти обещания тоже не оправдались. Различия в уровне абстракции в лучшем случае незначительны.

В завершение представьте, что в один прекрасный день будет изобретен действительно полезный диаграммный язык. Но ведь рисовать диаграммы будут не архитекторы, а программисты. Диаграммы попросту станут новым кодом, и программистам придется «рисовать» этот код — ведь в конечном итоге все сводится к подробностям, а управлять ими приходится именно программистам.

Заключение

С тех времен, как я занялся программированием, появилось великое множество новых и чрезвычайно мощных инструментов. Мой текущий инструментарий ограничивается небольшим подмножеством этого арсенала. Я использую *git* для управления исходным кодом, Tracker для отслеживания текущих задач, Jenkins для непрерывной сборки, интегрированную среду IntelliJ, XUnit для тестирования и FitNesse для компонентного тестирования.

Я работаю на компьютере Macbook Pro с процессором 2.8ГГц Intel Core i7, с 17-дюймовым монитором, 8 Гбайт памяти, 512Гбайт SSD и двумя дополнительными экранами.

Алфавитный указатель

C

Cucumber, 207
CVS, 197

E

Eclipse, 201
Emacs, 201

F

FitNesse, 206

G

Git, 197
Green Pepper, 206

I

IDE/редактор, 200
IntelliJ, 201

J

JBehave, 207
Jenkins, 203

M

MDA, 207

P

PERT, 151

R

RobotFX, 206

S

SVN, 197

T

TDD
 дебют, 87
 документация, 92
 определение, 25
 плотность дефектов, 91
 преимущества, 90
 рабочий цикл, 90
TextMate, 202

U

UML, 207

V

Vi, 200

A

автоматизированные
 приемочные тесты, 111
автоматизированный контроль
 качества, 26

антагонистические роли, 37

аффинная оценка, 156

Б

безжалостный рефакторинг, 27

бизнес-цели, 168

блокировка, 196

В

вадза, 101

вероятность, 148

ветвление, 197

встречи

отказ от участия, 134

повестка дня, 135

споры и разногласия, 137

Г

гибкость, 27

графические интерфейсы, 120

группы, 38

притертые, 176

скорость, 178

создание под конкретный
проект, 176

сохранение, 177

управление, 178

З

закон больших чисел, 157

И

инверсия приоритетов, 142

интеграционные тесты

инструменты, 207

стратегия тестирования, 128

интеграция, непрерывная, 122

интерны, 190

К

ката, 99, 100

код

ночное программирование, 70

принадлежность, 171

управление, 195

компонентные тесты

в стратегии тестирования, 127

инструменты, 205

контроль качества

автоматизация, 26

выявление ошибок, 24

идеальный случай, 115, 125

конфликты на встречах, 137

концентрация, 138

кофеин, 139

Л

ложная готовность, 83

М

модельно-ориентированная
архитектура (MDA), 208

модульные тесты, 93

инструменты, 204

приемочные тесты, 119

музыка, 73

Н

неоднозначность требований, 108

неопределенность, 107

непрерывная интеграция, 122

непрерывная сборка, 203

непрофессионализм, 20

неявные обязательства, 151

номинальная оценка, 151

О

обещания, 57
общение
 давление, 161
 приемочные тесты, 111
 требования, 105
оптимистическая
 блокировка, 196
 оценка, 151
открытый код, 103
отладка, 76
оценка
 PERT, 151
 анализ по трем
 переменным, 157
 вероятность, 148
 закон больших чисел, 157
 обязательства, 148
 определение, 147
 оптимистическая, 151
 пессимистическая, 152

П

парное программирование, 72, 163
пассивно-агрессивная
 позиция, 39, 117
паттерны проектирования, 30
перезарядка, 139
пессимистическая
 блокировка, 196
 оценка, 152
планирование итераций, 136
плотность дефектов, 91
подробности, 208
покер планирования, 155
помехи, 74
помидорный метод, 140
поток, 72
предметная область, знание, 33

прерывания, 77
приемочные тесты
 автоматизированные, 111
 графический интерфейс, 120
 лишняя работа, 113
 модульные тесты, 119
 непрерывная интеграция, 122
 обсуждение, 118
 определение, 110
 пассивно-агрессивная
 позиция, 117
 роль разработчика, 116
принадлежность кода, 171
принципы проектирования, 30
профессионализм, 20

Р

работодатель, 33, 168
рабочий цикл, в TTD, 90
рандори, 102
ремесленники, 190
репутация, 22
ретроспективные встречи, 136
роли, антагонистические, 37

С

сверхурочная работа, 82
системные тесты, 129
совместная работа, 32
сотрудничество, 166

Т

темп, 79
тестирование
 интеграционное, 128
 компонентное, 127
 модульное, 119
 непрерывная интеграция, 122
 определение, 110

тестирование (*продолжение*)

приемочное, 110

роль разработчика, 116

системное, 129

точность, преждевременная, 107

требования

неопределенность, 107

ответственность, 20

передача, 105

поздняя неоднозначность, 108

преждевременная точность, 107

тупики, 142

У

уверенность, 90

управление временем, 140

управление исходным кодом, 195

Ц

цели, 38, 135

Ш

широкополосный дельфийский
метод, 154

Роберт Мартин
Идеальный программист.
Как стать профессионалом разработки ПО
Перевел с английского Е. Матвеев

Заведующий редакцией
Руководитель проекта
Ведущий редактор
Художественный редактор
Корректоры
Верстка

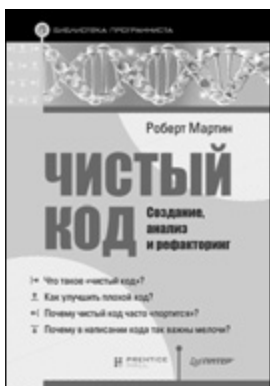
А. Кривцов
А. Юрченко
Ю. Сергиенко
А. Татарко
В. Листова, В. Нечаева
Л. Харитонов

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 25.10.11. Формат 70х100/16. Усл. п. л. 18,060. Тираж 1500. Заказ 0000.
Отпечатано по технологии СtР в ОАО «Первая Образцовая типография»,
обособленное подразделение «Печатный двор». 197110, Санкт-Петербург, Чкаловский пр., 15.

Чистый код: создание, анализ и рефакторинг

The Clean Coder: A Code of Conduct for Professional Programmers

Р. Мартин



ISBN: 978-5-459-00858-6

Объем: 464 с.

Дата выхода: март 2010

Даже плохой программный код может работать. Однако если код не является «чистым», это всегда будет мешать развитию проекта и компании-разработчика, отнимая значительные ресурсы на его поддержку и «укрощение». Эта книга посвящена хорошему программированию. Она полна реальных примеров кода. Мы будем рассматривать код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Прочитав книгу, вы узнаете много нового о коде. Более того, вы научитесь отличать хороший код от плохого. Вы узнаете, как писать хороший код и как преобразовать плохой код в хороший. Книга состоит из трех частей. В первой части излагаются принципы, паттерны и приемы написания чистого кода; приводится большой объем примеров кода. Вторая часть состоит из практических сценариев нарастающей сложности. Каждый сценарий представляет собой упражнение по чистке кода или преобразованию проблемного кода в код с меньшим количеством проблем. Третья часть книги — концентрированное выражение ее сути. Она состоит из одной главы с перечнем эвристических правил и «запахов кода», собранных во время анализа. Эта часть представляет собой базу знаний, описывающую наш путь мышления в процессе чтения, написания и чистки кода.

Управление разработкой ПО

Head First Software Development

Д. Пилон, Р. Майлз



ISBN: 978-5-459-00522-6

Объем: 464 с.

Дата выхода: март 2011

Даже опытные разработчики программного обеспечения постоянно сталкиваются с трудностями при реализации программных проектов: например, из-за смены требований заказчика ПО или непонимания конечными пользователями логики работы с новой программой. Если вы не собираетесь пасовать перед этими и другими распространенными проблемами управления IT-проектами, изучите с помощью этой уникальной книги передовые методы и практики, наработанные в области разработки программного обеспечения. Здесь вы получите всю необходимую информацию о каждом этапе жизненного цикла разработки программного обеспечения: переговоры с заказчиком и формализация клиентских требований, планирование процесса разработки ПО и подготовка технического задания, постановка задач и написание инструкций, тестирование ПО и устранение «багов». Особенностью этой книги является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию и разработке ПО.

Паттерны проектирования

Head First Design Patterns

Э. Фримен, Э. Фримен, К. Сьерра, Б. Бейтс



ISBN: 978-5-459-00435-9

Объем: 565 с.

Дата выхода: март 2011

В мире постоянно кто-то сталкивается с такими же проблемами программирования, которые возникают и у вас. Многие разработчики решают совершенно идентичные задачи и находят похожие решения. Если вы не хотите изобретать велосипед, используйте готовые шаблоны (паттерны) проектирования, работе с которыми посвящена эта книга. Паттерны появились, потому что многие разработчики искали пути повышения гибкости и степени повторного использования своих программ. Найденные решения воплощены в краткой и легко применимой на практике форме. Особенностью данного издания является уникальный способ подачи материала, выделяющий серию «Head First» издательства O'Reilly в ряду множества скучных книг, посвященных программированию. Книга будет интересна широкому кругу веб-разработчиков, от начинающих до профессионалов, желающих освоить работу с паттернами проектирования.

Изучаем JavaScript

Head First JavaScript

М. Моррисон



ISBN: 978-5-459-00322-2

Объем: 592 с.

Дата выхода: август 2011

Итак, вы готовы сделать шаг вперед в своей практике веб-разработок и перейти от верстки в HTML и CSS к созданию полноценных динамических страниц? Вы хотите перейти на новый уровень и стать веб-программистом в полном смысле этого слова? Тогда самое время познакомиться с самым «горячим» языком программирования — JavaScript! Эта книга — ваш счастливый билет в сложный и увлекательный мир современного веб-программирования, благодаря которому Вы, наконец, прекратите копипейстить чужой код и сможете писать свой собственный. С помощью этой книги вы узнаете: — все о языке JavaScript: от переменных до циклов; — почему разные браузеры по-разному реагируют на код, и как написать универсальный код, поддерживаемый всеми браузерами; — почему с кодом JavaScript вам никогда не придется беспокоиться о перегрузках страниц и ошибках передачи данных. Не пугайтесь, даже если ранее Вы не написали ни одной строчки кода — благодаря своему уникальному визуальному формату подачи материала эта книга с легкостью проведет вас по всеми пути: от написания первого простейшего java-скрипта до создания сложных веб-проектов, которые будут работать во всех современных браузерах.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



Зарегистрируйтесь на нашем сайте в качестве партнера по адресу www.piter.com/ePartners



Получите свой персональный уникальный номер партнера



Выбирайте книги на сайте www.piter.com, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт www.piter.com)

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробнее о Партнерской программе

ИД «Питер» читайте на сайте

WWW.PITER.COM



ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru


УКРАИНА


Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com


Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

 Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73. E-mail: fukanov@piter.com**

 **Издательский дом «Питер»** приглашает к сотрудничеству авторов. Обращайтесь
по телефонам: **Санкт-Петербург – (812) 703-73-72, Москва – (495) 974-34-50**

 Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

 Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

ДАЛЬНИЙ ВОСТОК

Владивосток

«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

Воронеж, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»
Московское шоссе, д. 15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

УРАЛ

Екатеринбург, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru





КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.



Нет времени ходить по магазинам?



наберите:



www.piter.com



Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD



**И наконец, вы нигде не купите
наши книги дешевле!**