

РУКОВОДСТВО ДЛЯ НАСТОЯЩИХ
САМУРАЕВ

Гибкое управление IT-проектами

Как мастера Agile
делают
выдающееся ПО



 **ПИТЕР®**



Джонатан Расмуссон

The Agile Samurai

How Agile Masters Deliver Great Software

Jonathan Rasmusson

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

РУКОВОДСТВО ДЛЯ НАСТОЯЩИХ
САМУРАЕВ

Джонатан Расмуссон

Гибкое управление IT-проектами

Как мастера Agile
делают выдающееся ПО



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

2012

ББК 65.9(2)212.14

УДК 338.24

P24

Расмуссон Дж.

P24 Гибкое управление IT-проектами. Руководство для настоящих самураев. — СПб.: Питер, 2012. — 272 с.: ил.

ISBN 978-5-459-01205-7

Прочитав эту книгу, вы получите знания и навыки, необходимые для того, чтобы запустить, проработать и успешно завершить гибкий проект, причем приведенный материал вас изрядно развеселит. Если вы — руководитель проектов, это издание станет для вас инструментом, который поможет реализовать гибкий проект от начала и до конца. Если же вы — аналитик, программист, тестировщик, разработчик пользовательских взаимодействий или проект-менеджер, книга даст вам идеи и базовые знания, необходимые для того, чтобы стать ценным членом команды разработчиков.

«Руководство для настоящих самураев» обходится без лишней теории, из-за которой другие книги совсем не отвечают духу гибкости. Она полна испытанных методов, невыдуманных историй, приятного юмора и прикладных упражнений-руководств, которые помогут вам делать правильные вещи наилучшим способом.

ББК 65.9(2)212.14

УДК 338.24

Права на издание получены по соглашению с Pragmatic Bookshelf. 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 978-1934356586 англ.
ISBN 978-5-459-01205-7

© 2010 Jonathan Rasmusson.
© Перевод на русский язык ООО Издательство «Питер», 2012
© Издание на русском языке, оформление
ООО Издательство «Питер», 2012

Оглавление

| | |
|--------------------------|----|
| Об авторе | 10 |
| Благодарности | 10 |
| Рад встрече с вами | 11 |
| От издательства | 14 |

Часть I. Введение в гибкую разработку

| | |
|---|----|
| Глава 1. Сущность гибкой разработки | 16 |
| 1.1. Как создавать что-то полезное каждую неделю | 17 |
| 1.2. Как происходит планирование при гибкой разработке | 20 |
| 1.3. Сделано — значит сделано | 22 |
| 1.4. Три простые истины | 23 |
| Глава 2. Знакомство с командой разработчиков | 27 |
| 2.1. Что особенного в проектах, связанных с гибкой разработкой .. | 28 |
| 2.2. Принципы действия гибкой команды | 30 |
| 2.3. Роли, которые встречаются в типичной команде | 36 |
| 2.4. Советы по подбору команды для гибкой разработки | 47 |

Часть II. Концептуализация проекта при гибкой разработке

| | |
|---|----|
| Глава 3. Главное — никого не забыть | 52 |
| 3.1. Из-за чего погибает большинство проектов | 53 |

| | |
|--|-----|
| 3.2. Не избегайте сложных вопросов | 54 |
| 3.3. Знакомство со стартовой колодой | 55 |
| 3.4. Как это работает. | 55 |
| 3.5. Сущность стартовой колоды | 56 |
| Глава 4. Общее представление о ситуации. | 58 |
| 4.1. Вопрос: зачем мы здесь? | 59 |
| 4.2. Создание блицрезюме | 61 |
| 4.3. Разработка оформления продукции. | 64 |
| 4.4. Создание списка функций | 67 |
| 4.5. Встреча с коллегами | 69 |
| Глава 5. Воплощение в реальность | 75 |
| 5.1. Демонстрация решения | 76 |
| 5.2. Что нас беспокоит | 77 |
| 5.3. Сколько времени займет реализация проекта | 81 |
| 5.4. Чем можно пожертвовать | 84 |
| 5.5. Определение средств, необходимых для достижения результата | 90 |
| Часть III. Планирование проекта при гибкой разработке | |
| Глава 6. Сбор пользовательских историй | 98 |
| 6.1. Проблема с документацией | 99 |
| 6.2. Знакомство с пользовательскими историями. | 102 |
| 6.3. Элементы хороших пользовательских историй | 103 |
| 6.4. Как организовать семинар по сбору пользовательских историй | 112 |
| Глава 7. Оценка: филигранное искусство угадывания | 119 |
| 7.1. Сложности, связанные с приблизительными оценками | 120 |
| 7.2. Как сделать из лимонов лимонад | 122 |

| | |
|--|-----|
| 7.3. Как это работает | 127 |
| Глава 8. Гибкое планирование: столкновение с реальностью | 135 |
| 8.1. Проблемы, связанные со статичным планированием | 136 |
| 8.2. Знакомство с гибким планированием | 138 |
| 8.3. Изменение объема работ | 142 |
| 8.4. Ваш первый план | 144 |
| 8.5. График прогресса разработки (burn-down chart) | 153 |
| 8.6. Перевод проекта в режим гибкой разработки | 156 |
| 8.7. Практическая реализация | 158 |

Часть IV. Выполнение гибкого проекта

| | |
|--|-----|
| Глава 9. Управление итерациями | 168 |
| 9.1. Как создавать что-то ценное каждую неделю | 169 |
| 9.2. Гибкая итерация | 169 |
| 9.3. Требуется помощь | 171 |
| 9.4. Этап 1. Анализ и проектирование (подготовка к работе) | 172 |
| 9.5. Этап 2. Разработка (выполнение работы) | 177 |
| 9.6. Этап 3. Тестирование (проверка работы) | 180 |
| 9.7. Канбан | 181 |
| Глава 10. Создание гибкого плана коммуникации | 187 |
| 10.1. Четыре вещи, которые необходимо сделать в ходе любой итерации | 188 |
| 10.2. Собрание по планированию историй | 188 |
| 10.3. Презентация | 190 |
| 10.4. Планирование следующей итерации | 190 |
| 10.5. Как выполнить мини-ретроспективу | 192 |
| 10.6. Как не нужно проводить планерку | 194 |

| | |
|--|-----|
| 10.7. Делайте все, что считаете целесообразным | 195 |
| Глава 11. Визуальное оформление рабочего пространства | 200 |
| 11.1. В бой вступает тяжелая артиллерия! | 201 |
| 11.2. Как оформить визуальное рабочее пространство | 205 |
| 11.3. Демонстрация намерений | 206 |
| 11.4. Создание и совместное использование общего рабочего языка | 207 |
| 11.5. Отслеживание ошибок (bugs) | 209 |

Часть V. Создание гибкого ПО

| | |
|---|-----|
| Глава 12. Тестирование компонентов: убеждаемся в том, что все работает | 214 |
| 12.1. Добро пожаловать в Вегас, малыш! | 215 |
| 12.2. Приступаем к тестированию компонентов | 217 |
| Глава 13. Рефакторинг: погашение технического долга | 225 |
| 13.1. Как «развернуться на пятачке». | 226 |
| 13.2. Технический долг | 227 |
| 13.3. Погашение долга с помощью рефакторинга | 229 |
| Глава 14. Разработка на основе тестирования | 237 |
| 14.1. Сначала пишем тесты | 238 |
| 14.2. Использование тестов для разрешения сложных ситуаций | 242 |
| Глава 15. Непрерывная интеграция: обеспечение готовности к работе | 248 |
| 15.1. Презентация | 249 |
| 15.2. Культура подготовки продукта к производству | 251 |
| 15.3. Что такое непрерывная интеграция | 252 |
| 15.4. Как это работает | 253 |

| | |
|---|-----|
| 15.5. Организация постановки кода на учет | 254 |
| 15.6. Организация автоматизированной сборки | 255 |
| 15.7. Работа с небольшими фрагментами | 257 |
| 15.8. Что дальше? | 259 |
| Приложения | 262 |
| Приложение А. Принципы гибкой разработки | 262 |
| Приложение Б. Интернет-ресурсы | 264 |
| Приложение В. Список литературы | 265 |

Об авторе

Джонатан Расмуссон, опытный предприниматель и бывший консультант по гибкой разработке (agile coach) в компании ThoughtWorks, занимается консалтингом на международном уровне, помогая клиентам находить наилучшие пути совместной работы и взаимодействий. Если Джонатан не едет на мотоцикле на работу по лютому канадскому морозу, то в свободное время он ведет блог <http://agilewarrior.wordpress.com>, где делится своим опытом из области гибкой разработки.

Благодарности

Эта книга никогда бы не получилась, если бы не поддержка Таннис — любви всей моей жизни — и трех наших чудесных детей, Лукаса, Роуэна и Брина, которые прошли вместе со мной весь этот путь.

Подобная книга также не могла бы появиться без участия чудесного редактора и издателя. Сюзанна Пфальцер достойна всяческих похвал.

Следует также упомянуть тех разработчиков-первопроходцев, на труд которых я опирался при написании книги. Это Кент Бек, Мартин Фаулер, Рон Джеффрис, Боб Мартин, Джошуа Кериевски, Том и Мэри Поппендик, Кэти Сьерра и чудесный коллектив компании ThoughtWorks.

И, разумеется, эта книга никогда бы не состоялась, если бы не невероятно плодотворное сотрудничество и вдумчивый подход, проявленные рецензентами и комментаторами: Ноэлем Раппином, Аланом Френсисом, Кевин Гиси, Джессикой Уотсон, Томасом Гендроном, Дэйвом Клейном, Майклом Сикорским, Дэном Нортон, Джанет Грегори, Санджай Манчиганти, Венди Линдемманн, Джеймсом Эвери, Робинот Даймондом, Томом Поппендиком, Элис Тот, Иэном Дизом, Меган Армстронг, Рамом Сваминатаном, Хэзер Карп, Чедом Фурнье, Мэттом Хьюзом, Майклом Менардом, Тони Семана, Ким Шриер и Рихейлом Кристофом. Особая благодарность

Ким Уимпсетт и Стиву Питеру за превосходное техническое редактирование и набор текста.

Спасибо вам, мама и папа, за то, что любите и вдохновляете меня.

Также спасибо Дэйву и Энди за их прекрасную компанию, в которой молодые и целеустремленные авторы могут творить и делиться своей работой со всем миром.

Самурай гибкой разработки — это неистовый программист-профессионал, умеющий справляться с самыми сложными софтверными проектами и укладываться в практически нереальные сроки, делая это легко и изящно.

Мастер-сэнсэй

Рад встрече с вами

Гибкая методология разработки программ (agile) напоминает нам, что, хотя компьютеры и исполняют код, его написанием и поддержкой занимаются все-таки люди.

Гибкая разработка — это концептуальный каркас, мировоззрение и подход к созданию программ. Она отличается экономностью, быстротой и прагматизмом. Это, конечно, не палочка-выручалочка, но все же такая парадигма радикально повышает шансы на успех, стимулируя вашу команду выдавать максимум, на который она способна.

В книге описывается реализация проекта, в котором применяется методология гибкой разработки. Хочу сказать, что такой проект действительно должен превосходить все ожидания. Ваши проекты будут заканчиваться вовремя и полностью укладываться в бюджет. Кроме того, клиенты будут полностью довольны создаваемыми для них программами и с удовольствием дадут вам новые заказы и примут посильное участие в работе.

Внутри коллектива вам предстоит освоить определенные навыки.

- ❑ Умение успешно подготавливать и запускать с пол-оборота собственный гибкий проект. Процесс должен быть настолько ясен, чтобы у вас вообще

не возникало вопросов из области «на какой стадии находится работа?» и «как это называется?».

- ❑ Умение собирать требования, оценивать и планировать работу прозрачно, открыто и честно.
- ❑ Умение работать с жаром. Вы узнаете, как превратить свой гибкий проект в отлаженный механизм, бесперебойно производящий высококачественный код, готовый к реальному использованию.

Если вы руководитель проекта, эта книга послужит инструментом для подготовки собственного гибкого проекта и ведения его от начала до конца. Если вы аналитик, программист, тестировщик, разработчик взаимодействия с пользователем, то сможете почерпнуть из книги идеи и базовые знания, необходимые для того, чтобы стать полноправным членом команды, занимающейся гибкой разработкой.

Как читать эту книгу

Можете свободно переходить к чтению любой главы, которая вас заинтересует. Но если вы хотите правильно построить процесс работы с самого начала, рекомендую прочесть книгу от начала и до конца.

В части I дается краткий обзор гибкой методологии разработки и объясняется, как работают команды, использующие данную парадигму.

В части II рассказывается об одной из наиболее многообещающих деталей предлагаемого метода — о подборе средств, которые составят арсенал вашей команды при реализации проекта. Речь пойдет о так называемой стартовой колоде (inception deck).

Часть III посвящена отзывам пользователей о проектах, выполненных в режиме гибкой разработки. Здесь же рассказано, как построить первый свой план такого проекта.

В части IV речь идет о выполнении работы. Именно отсюда вы узнаете, как из плана получается нечто реальное — готовая программа, с которой может работать клиент.

Наконец, в части V подытоживается все сказанное. Здесь описаны основные гибкие методы написания ПО, которым нужно следовать, чтобы постоянно повышать качество своих программ и снижать расходы на их текущую поддержку.

Неслучайный юмор

Не относитесь к этой книге с излишней серьезностью — желательно, чтобы вы изучали изложенный материал с чувством юмора.

Для этого текст сопровождается иллюстрациями, отступлениями и даже анекдотами, которые помогают понять, каково это — заниматься гибкой разработкой.

Боевые истории — это «фронтовые сводки» о работе над реальными гибкими проектами, а также впечатления о некоторых успехах (и неудачах), которые пришлось пережить мне и моим «братьям по оружию» в этом драматичном деле — гибкой разработке.



Упражнения под заголовком «*А теперь попробуем*» приглашают вас оторваться от чтения, немного поразмышлять и попробовать себя на практике.

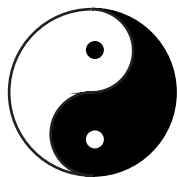


А вот и *Мастер-сэнсэй* — легендарный гуру гибкой разработки, мудрый и опытный во всех аспектах рассматриваемой парадигмы.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

Он будет вашим проводником и духовным наставником в странствии по гибкой разработке, время от времени обращая ваше внимание на важные принципы работы, например:



Принцип гибкой разработки

На создание готовой программы должно уходить минимум времени — от пары недель до пары месяцев. И предпочтительнее, чтобы это был максимально короткий срок.

Мастер-сэнсэй будет делиться с вами своими глубокими озарениями и напутствиями, как применять на практике методы гибкой разработки.

Ресурсы в Интернете

У англоязычного издания этой книги есть собственный сайт <http://pragprog.com/titles/jtrap>, где подробнее рассказано о книге. Работать с ним можно следующим образом:

- ❑ участвовать в дискуссиях на форумах, обмениваться мнениями с другими читателями, энтузиастами гибкой разработки и со мной;
- ❑ помогать совершенствовать книгу. Для этого нужно сообщать о найденных ошибках, в том числе о фактических недочетах в материале и об опечатках.

Итак, начнем.

От издательства

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

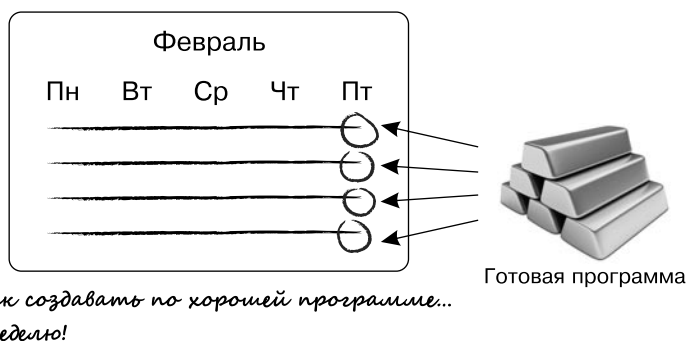
Мы будем рады узнать ваше мнение!

На сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

Часть I

Введение в гибкую разработку

Сущность гибкой разработки



Что нужно, чтобы каждую неделю создавать еще одну полезную программу?

На этот вопрос я попытаюсь ответить в данной главе. Вы узнаете, как выглядит процесс написания программы с точки зрения клиента. Также будет показано, как много ненужных вещей мы зачастую преподносим заказчику и как часто нам не удается сделать то, что действительно важно, — регулярно предоставлять клиенту хорошие функциональные программы.

К концу главы вы в общих чертах поймете, как планируется гибкая разработка, как судить об успехе проекта и как всего три простые истины помогают с честью выдерживать самые сжатые сроки и реализовывать сверхсложные проекты.

1.1. Как создавать что-то полезное каждую неделю

Давайте на секунду отвлечемся от гибкой разработки и поставим себя на место клиента. Это ваши деньги и ваш проект, для реализации которого вы наняли команду профессионалов высшего класса.

Что поможет вам убедиться в том, что приглашенная команда действительно не сидела сложа руки? Кипа документов, планов и отчетов? Или регулярное еженедельное предоставление полнофункциональных, протестированных программ, включающих только самое нужное?

Итак, если вы уже начали смотреть на программу с точки зрения клиента, значит, наши дела пойдут хорошо.

1. *Нужно разбить крупную проблему на мелкие задачи.*



Неделя — это ведь совсем немного. Казалось бы, нельзя успеть выполнить за неделю какую-либо работу. Чтобы дело пошло, нужно разбить большую и неподъемную с виду задачу на маленькие, более простые и управляемые.

2. *Нужно сосредоточиться только на том, что действительно важно, и забыть обо всем остальном.*

Большая часть того, чем мы занимаемся при разработке программ, не представляет никакой или почти никакой пользы для нашего клиента.

Разумеется, нам понадобится документация. Конечно, не обойтись и без планов. Но они нужны только как вспомогательные средства на пути к созданию готовой программы.

Создавая что-то стоящее каждую неделю, вы просто вынуждены собраться и отбросить все, что не приносит пользы. В результате вы как будто отправляетесь в дорогу налегке, захватив с собой только самое необходимое.

3. *Нужно убедиться, что создаваемый вами продукт работает.*

Создание чего-то полезного каждую неделю подразумевает, что плоды вашего труда должны быть качественными. Для этого необходимо тестирование — как можно больше и как можно раньше. Прошли те времена, когда все лишнее из проекта убирали только в его финале. Теперь ежедневное тестирование уже становится образом жизни. Вся ответственность лежит на вас.

4. *Работа требует участия клиента.*

Чтобы достичь цели, нужно регулярно останавливаться и сверять курс с клиентом. Его участие можно сравнить со светом фар, который рассекает густой туман, когда вы несетесь по трассе со скоростью 100 км/ч. Без таких консультаций клиент не может отслеживать вашу работу — в результате вы оба можете попасть в кювет.

5. *При необходимости нужно изменять курс.*

Первоначальный план



Текущий план



При работе случается всякое. Положение изменяется. То, что неделю назад казалось важным, сегодня может быть отбраковано. Если выстроить план и слепо ему следовать, вы не сможете справиться с непредвиденными обстоятельствами в случае их возникновения. Поэтому, если реальность нарушает ваши планы, меняйте планы, а не реальность.

6. *Вы берете на себя ответственность.*

Когда вы ставите перед собой цель каждую неделю создавать что-то стоящее и отчитываться перед клиентом, на что тратите его деньги, вы берете на себя определенную ответственность.

- ◆ Вы отвечаете за качество.
- ◆ Вы придерживаетесь расписания.
- ◆ Вы устанавливаете определенную планку.
- ◆ Вы тратите средства так, как если бы они были вашими.

Внимание!



Так работать
понравится не каждому!

Считаю ли я, что настанет день, когда все будут работать именно так? Ни в коем случае — ведь я же не удивляюсь тому, что большинство людей питается неправильно и не утруждает себя физкультурой.

Создание чего-то ценного каждую неделю — дело не для слабонервных. Выбирая такой подход к работе, вы словно попадаете в луч прожектора. В нем никуда не спрятаться. Ваши творения либо полезны, либо нет.

Но если вам нравится быть на виду, вы сторонник качественной работы и вас распирает желание действовать, то именно для вас работа в команде специалистов, применяющих гибкую методологию, может быть не только очень плодотворной, но и чертовски интересной.

А если вас все же пугает перспектива работать в темпе «на все про все недели» — не отчаивайтесь. Большинство команд, работающих в гибком режиме, начинает с двухнедельных проектов (а если команда очень большая — то с трехнедельных).

Это просто метафора, суть которой сводится к тому, что вы должны регулярно предоставлять клиенту готовые программы, и для этого требуется определенная отдача с его стороны, а при необходимости — изменение курса. Вот и все.



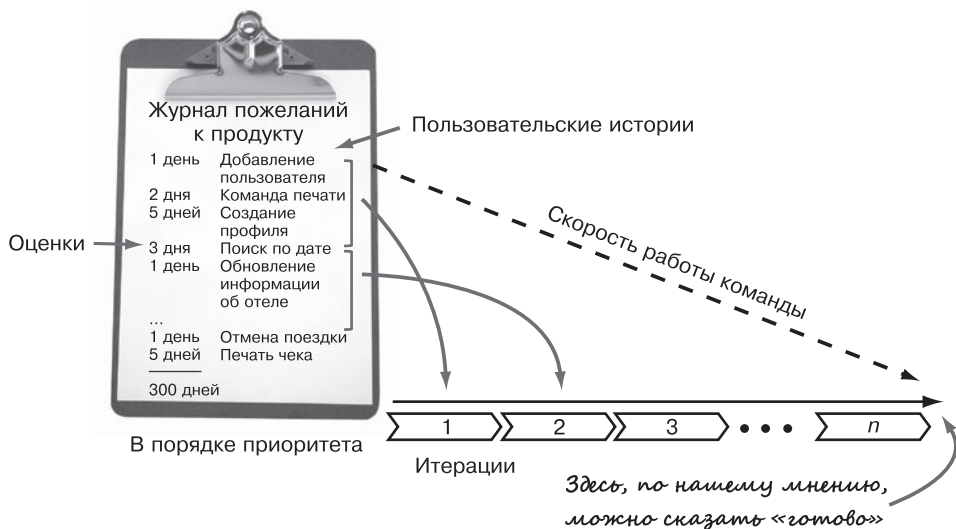
*Принцип
гибкой разработки*

Наш наивысший приоритет — заслужить доверие клиента, регулярно и последовательно предоставляя ему хорошие программы.

Теперь давайте рассмотрим гибкое планирование.

1.2. Как происходит планирование при гибкой разработке

Планирование проекта в гибкой манере схоже с подготовкой к долгим насыщенным выходным. Не буду писать длинных списков необходимых дел и задач, лучше давайте поговорим о таких нетривиальных вещах, как *журнал пожеланий к продукту* и *пользовательские истории*.



В гибкой разработке под журналом пожеланий (master story list)¹ понимается список задач, которые предстоит решить программисту. В нем упоминаются все важнейшие функции (пользовательские истории, user stories) — пожелания, предъявляемые клиентом к программе. Приоритетность тех или иных функций определяет сам пользователь, ваша команда разработчиков оценивает эти приоритеты и закладывает базовый план проекта.

Механизм выполнения всех задач в гибком проекте — это так называемая *итерация*. Под итерацией понимается короткий одно- или двухнедельный период, в ходе которого команда берет важнейшие пользовательские истории и преобразует их в действующие, протестированные программные функции.

Члены вашей команды могут прикинуть, сколько работы способен взять на себя каждый из них, измеряя *скорость работы команды* (сколько дел мож-

¹ В английских источниках также употребляются термины backlog и product backlog, в русском языке встречается понятие «бэклог». — Примеч. пер.

но выполнить за одну итерацию). Отслеживая скорость работы команды и используя эти данные для того, чтобы спрогнозировать, сколько удастся сделать в будущем, вы сможете ставить реальные сроки и соблюдать их, а ваша команда не будет брать на себя чрезмерных обязательств.

Когда оказывается, что вам и вашему клиенту предстоит сделать слишком много, ваш единственный выход — сделать меньше. *Гибко подходя к объему задач*, вы сможете сохранить сбалансированные планы, а ваши обещания останутся реалистичными.

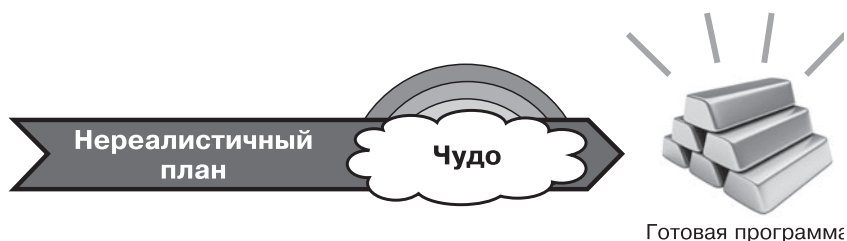
И если реальность идет вразрез с планом, нужно менять план. *Адаптивное планирование* — это краеугольный камень гибкой разработки.

Пока это все, что требуется сказать о гибком планировании. Такой метод планирования будет подробнее рассмотрен в главе 8.



Если сроки поджимают, то, как говорится, «надо — значит надо». Просто советую убедиться, что вы жертвуете собой ради стоящего дела, а не ради совершенно нереального обещания, данного примерно год назад при обзоре показателей эффективности работы.

Действительно, нереальные обещания время от времени даются и команды разработчиков сталкиваются с необходимостью сотворить невозможное. Но это неправильно. Работать с расчетом на чудо — не лучший способ реализации проекта и еще более порочный принцип по отношению к ожиданиям клиента.



При гибкой разработке необходимость в чудесах такого рода отпадает, так как вы собираетесь работать с клиентами открыто и честно с самого начала — рассказываете, что предстоит сделать, и позволяете принимать осознанные решения о функционале программы, финансировании и сроках.

Все дело в выборе. Можете продолжать верить в миф о том, что все раз — и получится. Или можно разработать вместе с клиентом такой план, в который поверите и вы и он.

1.3. Сделано — значит сделано

Допустим, ваши дедушка с бабушкой за небольшое вознаграждение попросили соседского сына-подростка стрести граблями опавшие листья во дворе на даче, сложить в мешок и отнести в лес. Сочтут ли дедушка с бабушкой работу выполненной, если парень сделает что-то из следующего:

- ☐ составит отчет о том, как он спланировал работу с граблями;
- ☐ предложит элегантный метод работы;
- ☐ составит тщательный и полный план тестирования?

Ничего подобного! Парень не получит ни копейки, пока не уберет листья, не уложит их в мешок и не отнесет куда следует.

При гибкой разработке применяется тот же принцип. В данном случае под реализацией функции понимается решение всех задач, необходимых для получения готового к работе кода.



Анализ, проектирование, написание кода, тестирование, разработка и дизайн уровня взаимодействия с пользователем (и проверка удобства этого уровня) — здесь перечислено все. Это не означает, что уже в первой версии программы у нас будут готовы все прибабасы или что в конце каждой итерации у нас будет получаться готовый продукт. Но мы собираемся к этому стремиться.

Если с продуктом нельзя работать на практике — это означает, что он не готов. И именно поэтому разработчик, исповедующий гибкую методологию, должен серьезно к ней относиться и понимать три простые истины.



Принцип гибкой разработки

Основной критерий успеха — практическая применимость программы.

1.4. Три простые истины

Далее описаны три простые истины, действующие при руководстве проектами. Если руководствоваться ими, удастся справиться со значительной долей нервозности и сбоев, с которыми регулярно приходится сталкиваться в софтверных проектах.

Три простые истины

1. Невозможно собрать все необходимые требования еще до начала проекта.
2. Любые требования, которые удастся собрать, обязательно изменятся.
3. На реализацию проекта обязательно понадобится больше времени и денег, чем было запланировано.

Принимая первую истину, мы перестаем бояться предстоящего проекта, который еще не известен нам во всех деталях. Мы понимаем, что требования еще предстоит разузнать и что работу вполне можно начинать еще до окончания сбора всей информации.

Осознавая вторую истину, мы перестаем бояться перемен или избегать их. Мы знаем, что рано или поздно они наступят. Принимаем их такими, какие они есть. И при необходимости корректируем наши планы в соответствии с ними.

Наконец, соглашаясь с третьей истиной, мы уже не драматизируем, если не успеваем сделать все запланированное в установленные сроки и испытываем недостаток ресурсов. Это нормальное состояние для любого интересного проекта. Вы делаете единственную вещь, которую можете сделать, — устанавливаете приоритеты, выполняете самую важную работу в первую очередь, а менее важную оставляете на потом.

Усвоив три эти простые истины управления проектами, вы увидите, что львиная доля стресса и волнений, традиционно связанных с написанием программ, куда-то исчезает. Вы сможете думать и изобретать с такой степенью сосредоточенности и ясности, какой не может похвастаться практически никакая другая область промышленности.

И не забывайте...

Путей всегда много!

Кристально чистое выполнение

Скрам

Бережливая разработка

Экстремальное программирование

Канбан

Ваш собственный метод!



Как и мороженое, гибкая разработка может иметь разные оттенки вкуса.

- ❑ У вас есть скрам (Scrum) — этот метод управления охватывает гибкий проект как оболочка.
- ❑ У вас есть экстремальное программирование (Extreme Programming, XP) — требующие высокой дисциплины основные практики разработки программ, жизненно важные для любого гибкого проекта.

- ❑ У вас есть бережливая разработка (Lean) — сверхэффективный аналог производственной системы, нацеленный на постоянную оптимизацию рабочего процесса (такая философия принята в компании «Тойота»).

А потом у вас появляется собственный гибкий метод. Например, такой метод может пригодиться для поиска выхода из ситуации, когда вы проехали со своей семьей на машине пару тысяч километров и обнаружили, что парк развлечений, в который вы направлялись, закрыт на ремонт.

Эта книга, как и вся остальная литература по гибкой разработке, — обычный обмен опытом. В ней рассказано о методах, которые я и другие авторы подобных книг нашли полезными при работе с клиентами. В данном издании я буду делиться с вами находками и инновациями, относящимися ко всем направлениям гибкой разработки, а несколько подобных методов нам придется изобрести самостоятельно. Читайте, изучайте, ставьте перед собой задачи и берите от книги именно то, что вам требуется.

Но учитывайте, что ни одна книга или метод не дадут вам ответов на все вопросы — в любом случае что-то придется додумывать самостоятельно. Каждый проект особенный, и хотя определенные принципы и практики будут действовать всегда¹, именно от вашей уникальной ситуации и контекста работы будут зависеть тонкости их применения.

Несколько слов о терминологии

В большинстве гибких методологий термины уже в основном устоялись, но некоторые понятия по-разному называются в скраме и экстремальном программировании.

В книге я постараюсь соблюдать единство терминологии (вообще, мне ближе экстремальное программирование), но хочу оговориться, что следующие понятия равнозначны и являются синонимами:

- ❑ *итерация* (iteration) — то же, что и *спринт* (sprint);
- ❑ *журнал пожеланий* (master story list) — то же, что и *бэклог* (product back log);
- ❑ *клиент* (customer) — то же, что и *владелец* (product owner)².

¹ <http://agilemanifesto.org>.

² «Владелец» — термин условный. Обычно так называют заказчика или его представителя, который определяет требования к продукту. В agile-команде эту роль может исполнять менеджер проекта, бизнес-аналитик или клиент. — *Примеч. ред.*

Что дальше?

Итак, мы познакомились с основами. Теперь пришло время включить вторую передачу и поговорить о том, что такое команда.

В следующей главе, которая рассказывает о командах гибких разработчиков, мы рассмотрим, как должна быть построена такая команда, как она должна работать над проектом, а также о некоторых вещах, которые каждый член команды должен уяснить *до* начала проекта.

Глава 2

Знакомство с командой разработчиков



Команда разработчиков при гибком методе работы — это нечто совершенно особенное. В типичном гибком проекте нет жестко заданных ролей. Все могут делать что угодно. И все же при всем хаосе, кажущейся неразберихе и отсутствии формальной иерархии отлаженными командами гибких разработчиков как-то удается регулярно создавать классные программы.

В этой главе будет подробно рассмотрено, что обеспечивает работу гибкой команды. Мы изучим характеристики хороших команд, построенных по такому принципу, различия между гибкими командами, а также обсудим несколько рекомендаций, упрощающих поиск квалифицированных сотрудников.

К концу главы вы будете представлять, как выглядит типичная гибкая команда, как самому собрать такую команду и чему эти люди должны научиться, прежде чем ринуться в бой.

2.1. Что особенного в проектах, связанных с гибкой разработкой

Прежде чем перейти к описанию тонкостей работы гибкой команды, нужно прояснить некоторые общие моменты, касающиеся гибких проектов в целом.

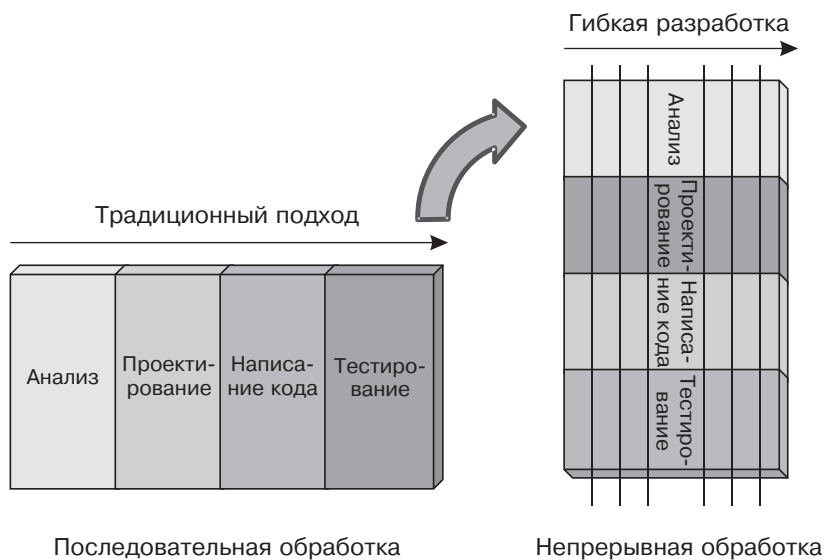
Первым делом отмечу, что в гибких проектах границы между ролями действительно размыты. Когда все идет хорошо, у человека, вливающегося в гибкую команду, возникает ощущение, что вся компания работает над маленьким стартапом. Люди принимают за все сразу и делают все, что может приблизить проект к цели, — независимо от роли или должности конкретного участника.



Разумеется, у всех сохраняются основные обязанности и люди обычно занимаются тем, в чем они особенно хороши. Но в гибком проекте такие узкоспециальные роли, как аналитик, программист и тестировщик, на самом

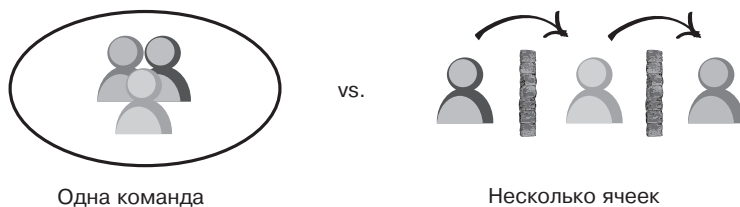
деле не существуют — как минимум не существуют в традиционном понимании этих ролей.

Вторая деталь, специфичная для гибких команд, заключается в том, что анализ, проектирование, написание кода и тестирование идут постоянно, то есть не прекращаются.



Это означает, что все этапы работы перестают изолироваться друг от друга. Люди, выполняющие работу, должны быть объединены в единое целое и вместе ежедневно заниматься проектом.

Третий аспект, который нужно прояснить заранее, — насколько важна для гибкости работы такая концепция одной команды и командной ответственности.



Качество выполнения гибкого проекта от начала до конца — задача всей команды. Отдела обеспечения качества (Quality Assurance, QA) нет, качество обеспечиваете вы сами, когда проводите анализ, пишете код или управляете проектом. Качество гарантируется на каждом шагу, поэтому

в гибком проекте вы не услышите вопроса: «И как отдел гарантии качества проморгал эту ошибку?»

Итак, размытие ролей, постоянное сосредоточение на разработке и ответственность всей команды за все этапы проекта — вот что наверняка встретится вам при работе с гибкими командами.

Теперь давайте рассмотрим некоторые типичные дела, которыми занимаются гибкие команды. Это поможет нам самим успешно набирать такие команды.

2.2. Принципы действия гибкой команды

Прежде чем вы со своей командой начнете добиваться успеха, придется побороться за некоторые вещи, а также заточить команду для этих успехов.

Совместное размещение рабочих мест

Есть одна вещь, которая помогает радикально увеличить КПД вашей команды, — все должны работать в одном месте.

Команды, члены которых работают рядом, действительно показывают более высокие результаты. Ответы на вопросы находятся быстро. Проблемы решаются сразу же после их появления. Уменьшается количество трений между различными звеньями. Люди быстрее начинают доверять друг другу. С такой маленькой командой, работающей как единый организм, очень сложно соперничать.

Итак, если команды, работающие в одном помещении, так хороши, означает ли это, что удаленная команда не сможет выполнять гибкие проекты? Конечно, сможет.

Работа в удаленной команде стала для многих специалистов образом жизни. И хотя плотно сбитая, работающая в одном офисе команда всегда будет иметь некоторую фору перед удаленной, есть секреты, помогающие минимизировать это преимущество.

Например, в самом начале реализации проекта можно выделить определенный бюджет на то, чтобы собирать разработчиков вместе. Даже если такая встреча продлится всего несколько дней (еще лучше, если удастся поработать в таком режиме пару недель), время, потраченное на знакомство друг с другом, привыкание к юмору коллег и совместные обеды, чрезвычайно помога-

ет превратить разношерстную команду в крепкий, высокопроизводительный коллектив. Итак, постарайтесь для начала собрать всех вместе.

В конце концов, в вашем распоряжении масса технических средств (Skype, видеоконференции, социальные сети), помогающих превратить удаленную команду в группу, не уступающую по производительности работникам из маленького офиса.

Привлечение клиентов

В наше время все еще существует масса программ, которые пишутся командами разработчиков совершенно без участия клиентов. Это прискорбно и просто преступно.

Как у команды может получиться выдающийся, инновационный продукт, если сами люди, для которых он пишется, не участвуют в работе?

Заинтересованный клиент не пропускает презентаций, задает вопросы, реагирует на ход работы, направляет разработку и делится идеями, помогающими команде сделать нечто неординарное. Такие клиенты становятся ключевыми членами команды и полноправными коллегами разработчиков.

Стимулируйте незапланированное сотрудничество

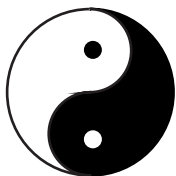


В книге о компании «Пиксар» (The Pixar Touch) Стив Джобс рассказывает, как сильно успех фильмов этой компании зависел от незапланированного сотрудничества. После выпуска фильма «История игрушек-2» (Toy Story II), который чуть не поставил всю компанию на грань банкротства, руководство осознало, что сотрудники оказались слишком разобщены, изолированы друг от друга. Все могло закончиться крахом, если бы не собрали всех участников работы над фильмом вместе.

Именно для этого студия «Пиксар» приобрела участок 20 акров в Эмеривилле, штат Калифорния, и собрала всех сотрудников под одной крышей. Результат последовал незамедлительно. Контакты наладились, сотрудничество оптимизировалось, и коллеги смогли выпускать крупный фильм каждый год.

Вот почему гибкие методы (например, экстремальное программирование и скрам) предлагают бороться за участие клиента в работе. Это выражается в выделении особой роли *«заказчик в команде»* (on-site customer) и в существовании в скраме специальной роли *«владелец продукта»* (product owner). Это очень важная работа. Подробнее все роли мы обсудим чуть позже.

Следующий принцип также проясняет, почему любой успешный гибкий проект требует участия клиента.



*Принцип гибкой
разработки*

Бизнесмен и разработчик должны трудиться
на протяжении проекта вместе, изо дня в день.

Вы можете спросить: «А что делать, если мой клиент не хочет участвовать в работе?» Возможно, клиент просто не может идти в ногу со временем или ему не особенно нужен этот проект, а быть может, он просто не считает, что вы движетесь к цели.

Какой бы ни была причина, вы должны завоевать определенное доверие клиента, это обязательно.

В следующий раз при встрече с клиентом скажите ему, что через две недели окончательно разберетесь с какой-то его проблемой.

Не просите разрешения. Не делайте из этого особенной церемонии. Просто возьмите какую-нибудь проблему или досадную помеху и покончите с ней.

Затем сделайте так. Встретьтесь с клиентом через две недели, докажите ему, что проблема полностью исчерпана, и проделайте такую же операцию снова. Найдите другую проблему и заставьте ее исчезнуть.

Возможно, вам придется поступить так два или три раза, а то и больше, чтобы клиент стал интересоваться текущими проблемами. Просто знайте, что рано или поздно такой интерес придет.

Клиент начинает смотреть на вас иначе и понимать, кто вы такой: толковый специалист, с которым нужно считаться, чтобы дело спорилось.

Понимаете, может быть тысяча причин, по которым ваш клиент не участвует в работе. Вероятно, он устал от проектов, выполняемых в IT-отделе, или эта программа для него не так важна. Не исключено, что вы недостаточно подробно рассказали ему о том, как важна именно его роль для успеха всего проекта. А быть может, клиент на самом деле очень занят.

Я пытаюсь сказать, что, если вам нужно завоевать определенное доверие, начните понемногу наращивать его, и в итоге вы добьетесь своего.

Самоорганизация

Гибкая команда предпочитает, чтобы перед ней поставили цель, а затем не мешали всей команде разработать способ ее оптимального достижения. Для этого гибкая команда должна быть самоорганизующейся системой.

Самоорганизация заключается в том, что при необходимости нужно переступить через свое эго и вместе с командой понять, как вы, будучи командой (со всеми вашими индивидуальными навыками, пристрастиями и талантами), сможете выполнить конкретный проект максимально качественно.

«Разумеется, Бобби хорошо клепает код. Но у него еще особый талант к проектированию, он поможет нам сделать некоторые макеты».

«Да, Сьюзи — одна из лучших наших тестировщиц, но ее настоящий конек — выстраивание перед клиентом перспектив работы. У нее это отлично получается, и ей нравится это делать».

Это не означает, что разработчик должен быть экспертом по визуальному дизайну, а тестировщики — брать на себя управление проектом.

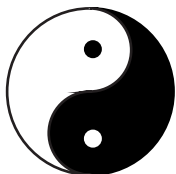
Скорее это признание того, что для создания оптимальной команды нужно исповедовать принцип «роль для человека, а не человек для роли».

Итак, как добиться самоорганизации своей команды?

- ❑ Команда допускается к планированию, оцениванию и может распоряжаться проектом.
- ❑ Вы не придаете особого значения ролям и их названиям, а сосредотачиваетесь на бесперебойном производстве функциональных, протестированных программ.

- ❑ Вы ищете людей, способных брать на себя инициативу, то есть тех, кто сам прокладывает себе путь, а не сидит и не дожидается остальных.

Короче, вы отпускаете вожжи и позволяете сотрудникам самостоятельно делать свою работу.



Принцип гибкой разработки

Самая лучшая архитектура, требования и проекты рождаются в самоорганизующихся командах.

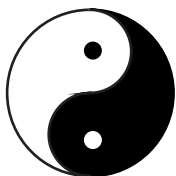
Теперь необходимо отметить, что самоорганизация как таковая, конечно же, очень хороша, но вся соль в том, к чему она приводит, — в расширении возможностей и индивидуальной ответственности.

Ответственная и полноправная

Хорошая гибкая команда всегда стремится отвечать за результаты своей работы. Эти люди знают, что клиент полагается на них и понимает, что без такой команды не обойтись. Поэтому члены гибкой команды никогда не увиливают от ответственности, которая неотделима от стремления достигать новых результатов с самого первого дня.

Разумеется, индивидуальная ответственность развивается только в командах, наделенных реальной полнотой полномочий. Давая команде право самостоятельно принимать решения и делать то, что она считает нужным, вы освобождаете пространство для инициативы и работы по собственному усмотрению. Люди начинают решать проблемы, не дожидаясь, пока им позволят это сделать.

На данном этапе никто не застрахован от ошибок. Но поверьте, игра стоит свеч.



Принцип гибкой разработки

Подбирайте для проекта мотивированных людей. Обеспечивайте им необходимую поддержку и хорошие условия труда и просто позвольте им сделать свою работу.

Конечно, создать ответственную и полноправную команду сложнее, чем просто сказать об этом. Не каждый готов к свободе действий. Зачем напрягаться, если можно просто прийти на работу, сесть и ждать, что тебе скажут.

Если вы чувствуете, что возникают проблемы с индивидуальной ответственностью, их легко решить — попросите команду продемонстрировать, как работает создаваемая программа.

Обычный прием, ставящий команду перед реальным клиентом, которому нужно показать, как выполняется поставленная задача, помогает кардинально повысить личную ответственность каждого члена.

Во-первых, команда осознает, что на нее и на результат ее работы рассчитывают реальные люди. Самые настоящие люди с насущными проблемами, для решения которых нужна заказанная программа.

Во-вторых, после первой же неудачной демонстрации для команды сразу же станет очень важно подготовить программу так, чтобы в следующий раз все работало. Люди сами будут брать на себя ответственность за решение подобных задач. Если этого не произойдет, значит, у вас большие проблемы.

Многофункциональность

Многофункциональной (cross-functional) называется команда, которая может реализовать требования клиента от начала и до конца. То есть команда должна обладать необходимым опытом и навыками и гарантировать, что сможет создать любую функцию, о которой попросит клиент.

Набирая людей в команду, ищите многостаночников, умеющих заниматься самыми разными делами. Говоря о таких программистах, я имею в виду людей, каждый из которых ориентируется во всем технологическом стеке (а не только в пользовательском интерфейсе или интерфейсе базы данных). Например, в случае тестировщиков и аналитиков это означает, что нам нужны люди, умеющие не только выполнять качественное тестирование, но и глубоко анализировать поставленные перед ними требования.

Специалисты привлекаются по мере необходимости, если команда не обладает каким-нибудь специфическим навыком (например, для настройки базы данных). Но в основном команда работает в тесном контакте на протяжении всего выполнения проекта.

Кто унес мой сыр?

«Кто унес мой сыр?» (*Who Moved My Cheese?* [Joh98]) — это бизнес-притча о мышках, которые проснулись однажды утром и обнаружили, что большой кусок сыра, вокруг которого им вольготно жилось, куда-то исчез. Его унесли. И теперь мыши в растерянности размышляли, что делать.

Кому-то переход к гибкой разработке может показаться таким отлучением от сыра.

Аналитик, переходящий к гибкому проекту, осознает, что этап анализа никогда не закончится.

Разработчик должен быть готов к тому, что ему придется писать тесты (и немало!).

То есть нужно понимать, что, предлагая людям работать в таком режиме, вы отбираете у кого-то сыр. И все, что вы можете для них сделать, — помочь им найти новый сыр (показать, как должны измениться их роли).

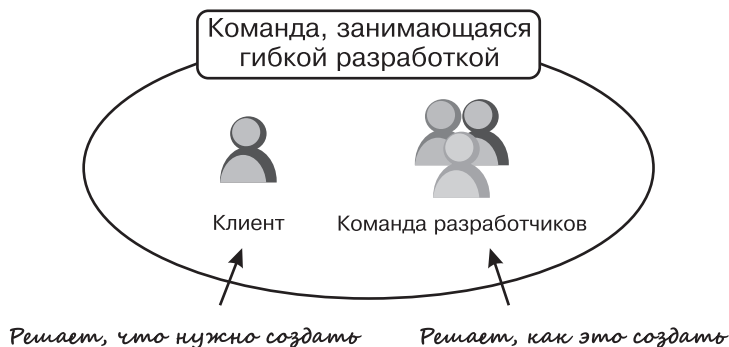
Разумеется, основным достоинством многофункциональной команды является скорость, с которой она способна работать. Ей не нужно ждать одобрения или договариваться о ресурсах с другими отделами, а можно с первого же дня начинать работу, не останавливая ее ни на день.

Конечно, вы должны обрисовать людям, чего от них ожидаете, и рассказать о результатах, которых хотите добиться, набирая свою команду.

А теперь поговорим о ролях.

2.3. Роли, которые встречаются в типичной команде

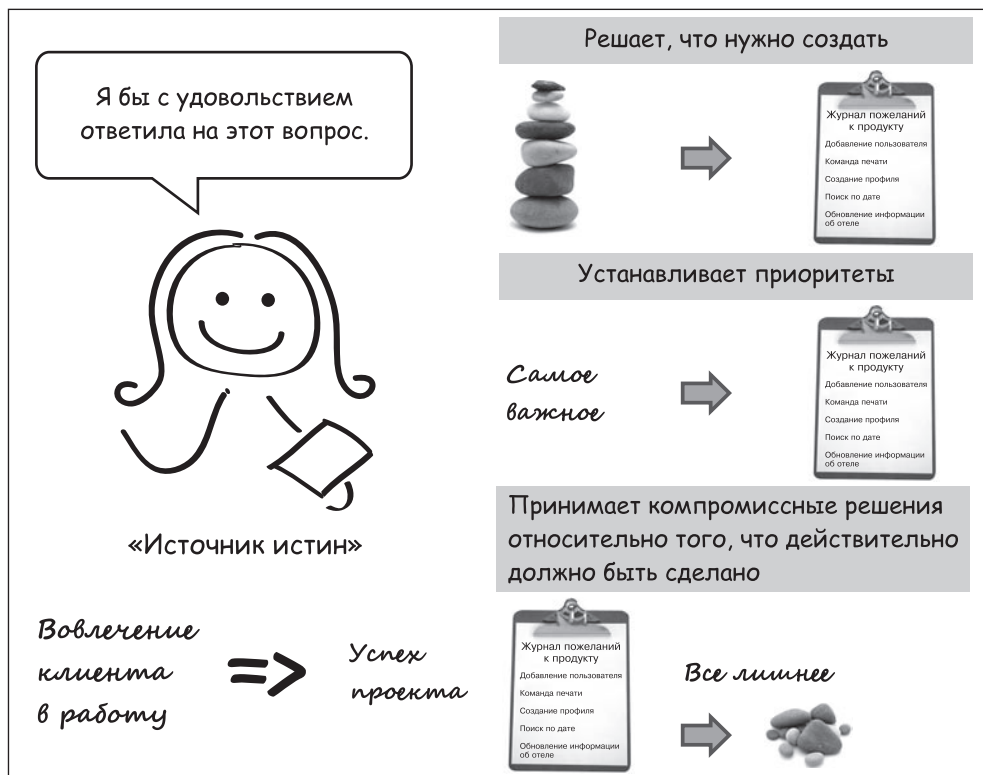
Гибкие методы, такие как скрам и экстремальное программирование, предусматривают при реализации проекта совсем немного формальных ролей. Есть люди, которые знают, что должно быть сделано (клиенты), и люди, знающие, как это сделать (команда разработчиков).



Если у вас возникает вопрос: «А где все программисты, тестировщики и аналитики?» — не волнуйтесь, они никуда не исчезли. Просто при гибкой разработке не так важно, кто именно играет конкретные роли.

Начнем с рассмотрения самой важной роли в гибком проекте — клиента.

Клиент гибкого проекта



«Источником истины» здесь является клиент, сотрудничающий с командой разработчиков, — от него поступают все требования к выполняемому проекту. Ведь именно для клиента пишется программа.

В идеале клиент должен ориентироваться в теме проекта. Это человек, глубоко погруженный в дело, ему действительно небезразлично, что делает программа, как она выглядит и функционирует. Кроме того, он с энтузиазмом направляет команду, отвечает на вопросы, словом, проявляет отдачу.

Кроме того, клиент устанавливает приоритеты. Он решает, что нужно создать и когда.

Все это не происходит в вакууме. Речь идет о совместной работе с командой разработчиков, ведь могут быть технические причины, по которым целесообразно сделать сначала одни элементы, а потом другие (иными словами, снизить технологический риск).

Однако обычно приоритеты расставляются с точки зрения бизнеса, а затем начинается работа с командой, которой нужно выполнить намеченный план, чтобы все получилось.

И именно клиентам приходится принимать решения относительно того, без чего можно обойтись, если поджимают сроки и начинают заканчиваться деньги.

Разумеется, чтобы все это получилось, требуется очень тесное сотрудничество клиента с командой (в идеале — в течение всего рабочего дня). В ранних версиях экстремального программирования было принято говорить о «заказчике в команде» (on-site customer). В скраме аналогичная роль называется «владелец продукта» (product owner).

Но не переживайте, если не удастся заполучить клиента на полный рабочий день, — это получается у мизерной доли команд. Даже без заказчика в команде вы можете сделать гибкий и весьма успешный проект. Не на всех проектах вообще требуется такой участник.

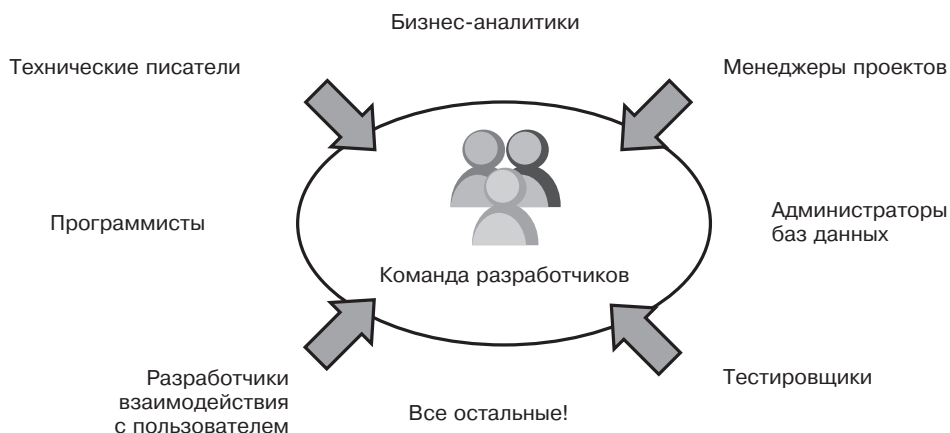
Еще важнее чувствовать тот дух, на котором строятся гибкие методы, подобные экстремальному программированию или скраму. Сущность этих

методов заключается в том, что чем более непосредственный контакт у вас с клиентом, тем лучше.

Итак, добейтесь настолько тесного сотрудничества с клиентом, насколько это возможно. Убедитесь, что клиент понимает важность своей роли. Нужно, чтобы клиент был полностью свободен в своих действиях и сам хотел принимать решения, необходимые для успешного завершения проекта.

Теперь поговорим о команде разработчиков.

Команда разработчиков



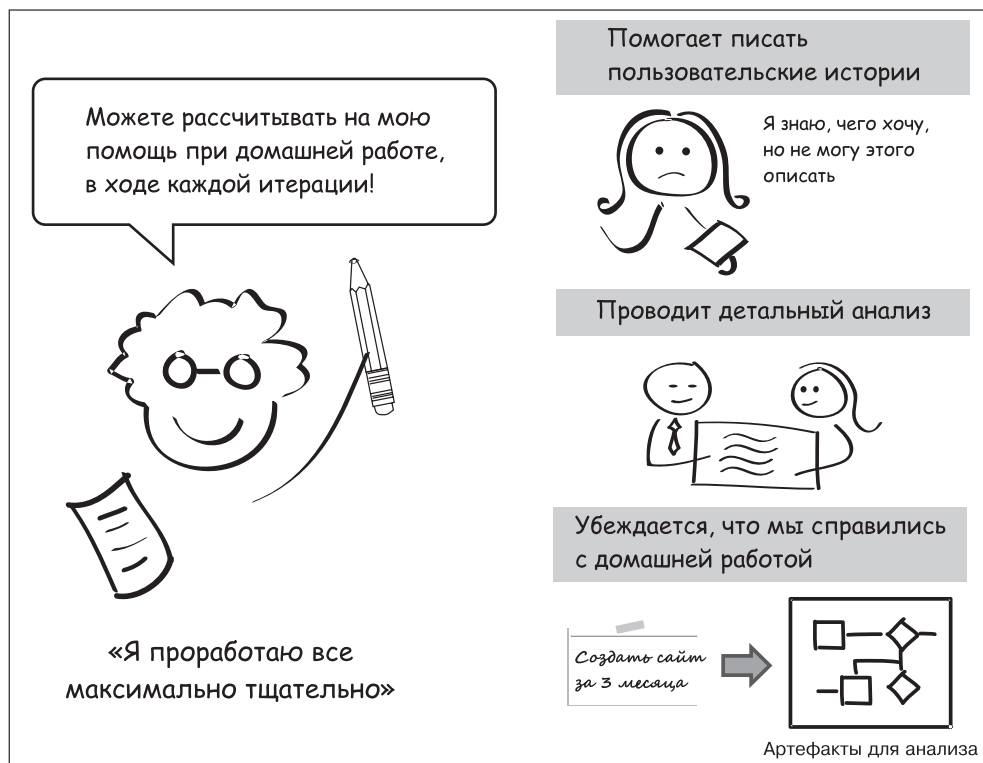
Гибкая команда разработчиков — это группа многофункциональных специалистов, которые могут работать с любой функцией, которую желает получить клиент, и превратить ее в готовый, рабочий программный компонент. В состав команды входят аналитики, разработчики, тестировщики, администраторы баз данных и все остальные специалисты, которые нужны для превращения пользовательских историй в программу, готовую для реального использования.

Как бы я ни обожал тот дух и те принципы, на основе которых работает гибкая команда, не имеющая формально распределенных ролей, должен признаться, что попытка пригласить глубоко консервативную команду разработчиков и сообщить ее членам, что им нужно «самоорганизоваться», на практике у меня никогда не срабатывала.

Чтобы чувствовать себя уверенно, нужна точность формулировок. Надо сразу ясно сказать, что в гибкой команде границы между отдельными ролями размыты и что от каждого участника требуется умение сражаться на нескольких фронтах. Но мне лучше удавалось придавать коллективам нужный вид, если я объяснял им гибкую методологию в понятных им выражениях.

Если ваша команда именно такая, то обратите внимание на следующие описания гибких ролей, которые помогут людям адаптироваться к новым условиям и понять, как изменятся их роли в гибком проекте.

Гибкий аналитик



Когда мы приступаем к разработке определенной функции, кто-то должен досконально с ней разобраться и описать все тончайшие детали ее работы. Это задача нашего гибкого аналитика.

Аналитик похож на неутомимого сыщика, задающего глубокие зондирующие вопросы и при этом испытывающего кайф от тесного сотрудничества

с клиентом. В ходе совместной работы они пытаются понять, что же нужно от программы.

Аналитик выполняет в гибком проекте множество задач: помогает клиенту писать пользовательские истории (см. главу 6); выполняет глубокий анализ, когда дело доходит до разработки; помогает создавать имитационные объекты (mock-ups) и прототипы; использует в ходе анализа все доступные ему инструменты, чтобы донести до разработчика сущность пользовательских историй.

Подробнее о функционировании гибкого анализа мы поговорим в разделе 9.4.

Гибкий программист



Пока не написан код, все сводится только к конструктивным намерениям. Написанием кода занимаются наши гибкие программисты.

Гибкий программист — это профессионал, так как он очень серьезно относится к качеству программы. Лучшие из таких программистов одновременно являются увлеченными тестировщиками, гордящимися своей работой и всегда старающимися написать максимально качественный код.

Поэтому существуют определенные вещи, без которых не обойтись программистам, желающим регулярно создавать отличные многофункциональные продукты.

- ❑ Они пишут много тестов и часто пользуются ими при разработке (см. главы 12 и 14).
- ❑ Они постоянно дорабатывают и оптимизируют архитектуру программы по мере работы (см. главу 13).
- ❑ Они уверены, что базовый код всегда готов к использованию и может быть развернут по первому требованию (см. главу 15).

Программист также тесно сотрудничает с клиентом и другими членами команды и гарантирует, что написанный код работает, что он максимально прост и что внедрение программы для практического использования не составит никакого труда.

Гибкий тестировщик



Гибкие тестировщики знают, что одно дело — написать продукт, а другое — гарантировать, что он работает. Поэтому такой тестировщик подключается к проекту уже на ранних этапах, заблаговременно обеспечивая успешное выполнение пользовательских историй и то, что программа, запущенная в практическую работу, действительно не подведет.

Поскольку в гибком проекте тестировать нужно буквально все, ни один этап работы не обойдется без участия тестировщика. Он будет работать бок о бок с клиентом, помогая ему оформить предъявляемые требования в виде тестов.

Что, если начинать каждый проект вот так?

Предположим, в начале каждого проекта вы вместе с командой пытаетесь ответить на четыре простых вопроса о себе.

- В чем я особенно хорош?
- Как я привык работать?
- Что я ценю?
- Каких результатов можно от меня ожидать?

Затем, получив новые идеи, задайте те же вопросы коллегам, чтобы они рассказали вам, в чем они хороши, как работают, что ценят и какой результат могут выдать.

Эта идея называется упражнением Друкера¹. При всей простоте оно очень помогает сплотить команду, наладить необходимую коммуникацию и уровень доверия, необходимый для любой высокопроизводительной команды.

Тестировщик будет тесно сотрудничать с разработчиками, помогая автоматизировать тесты, отыскивая «дырки» и выполняя широкое исследовательское тестирование, пытаясь проверить приложение под любым возможным углом.

Кроме того, тестировщик будет представлять себе целостную картину тестирования и никогда не упустит из виду тестирования с возрастающей нагрузкой, масштабируемости и всех остальных деталей, которые требуется учесть для создания высококлассного продукта.

¹ <http://agilewarrior.wordpress.com/2009/11/27/the-drucker-exercise>.

В книге Джанет Грегори и Лизы Криспин *Agile Testing: A Practical Guide for Testers and Agile Teams* [GC09] подробно описана важность роли тестировщика при гибкой разработке.

Подробнее о механизмах гибкого тестирования мы поговорим в разделе 9.6.

Гибкий менеджер проектов



Гибкий менеджер проекта знает, что успех невозможен без плодотворной работы всей команды. Вот почему хороший менеджер проекта сделает все возможное и невозможное, чтобы ничто не мешало его команде достичь успеха.

В частности, он займется планированием и перепланированием, а при необходимости — корректировкой курса (см. главу 8).

Кроме того, к его задачам относится улавливание ожиданий всех людей, занятых в работе над проектом: передача отчетов о состоянии дел владельцам (stakeholders), стимулирование взаимодействий внутри компании,

а также защита рабочей группы от неблагоприятных внешних воздействий. Все это — обязанности хорошего гибкого менеджера проектов.

Хороший менеджер проекта не рассказывает команде, чем ей заниматься, — этого просто не требуется. Он помогает создать такую среду, в которой команда будет чувствовать себя максимально независимо и продолжать отлично работать и в отсутствие менеджера проектов. На самом деле фирменной чертой хорошего гибкого менеджера проектов является умение исчезнуть и вернуться через две недели, но так, чтобы никто этого не заметил.

Подробнее об управлении проектами мы поговорим в главах 8 и 9.

Гибкий разработчик взаимодействия с пользователем



Разработчики уровня взаимодействия пользователя с программой сосредоточены на создании полезных, удобных и приятных функций, обеспечивающих такое взаимодействие. Специалист, интересующийся проблемой удобства работы с программой (юзабилити), будет стремиться понять, что нужно пользователю, а затем сотрудничать с оставшейся командой, чтобы

найти наилучшие способы удовлетворения пользовательских потребностей.

К счастью, многие практические методы, используемые юзабилити-экспертами, хорошо согласуются с духом гибкой разработки программ. Акцент на ценности продукта, быстрая коммуникация и максимальное удовлетворение потребностей клиента — общие цели как гибких разработчиков, так и специалистов по удобству программы.

Кстати, юзабилити-экспертам не в новинку работать постепенно, и они привычны к итерациям. Они проектируют и создают новые функции по мере того, как пишется код (а не пытаются сделать все заранее и кардинально опередить всю команду).

Если вы можете заполучить в ваш проект специалиста по юзабилити, считайте, что вам повезло. Такой человек может поделиться массой полезного опыта и знаний и очень поможет общему делу в области анализа и разработки уровня взаимодействия с пользователем.

Все остальные

Перечислю остальные важные роли и специалистов, которых не упомянул выше. Это администраторы баз данных, системные администраторы, технические писатели, инструкторы, специалисты по улучшению текущей деятельности, инфраструктуры и обслуживанию сетей. Все они — члены команды разработки и равноправные участники проекта.

В скраме есть роль *руководителя* (scrum master). В гибком проекте ему отводится место тренера и продюсера рок-звезды в одном флаконе. Такие тренеры могут быть очень полезны при «раскоচেгаривании» новых команд. Они умеют объяснить и привить принципы гибкой разработки и соответствующую философию, а также гарантировать, что команда не собьется с курса и не вернется к прежним вредным привычкам. Работа тренера хорошо описана в книге *Agile Coaching* [SD09].

Опытной команде, как правило, не требуется специальный тренер, но новому проекту такой специалист определенно не повредит.

И последнее: рассказывая об этих ролях, дайте людям понять, что в гибком проекте вполне нормально (и ожидаемо), что человек будет одновременно играть несколько ролей.

Иными словами, пусть аналитик знает, что разработчик имеет полное право беседовать с клиентом (это даже приветствуется). Пусть тестировщики не удивляются тому, что разработчику придется написать немало автоматизированных тестов. А если в вашем проекте не будет специального разработчика пользовательских интерфейсов, это не означает, что никто не собирается заниматься юзабилити и дизайном. Разумеется, все это будет сделано, только другими людьми, которые исполняют в гибком проекте сразу несколько ролей.

В завершение немного поговорим о том, как набирать людей в команду.

2.4. Советы по подбору команды для гибкой разработки

Хотя большинству людей должно понравиться работать в высокопроизводительной гибкой команде, есть некоторые вещи, на которые нужно обращать внимание при подборе квалифицированных профессионалов.

Ищите универсалов

Универсалы хорошо приживаются в гибких проектах, поскольку сама методология требует от людей систематически работать и использовать для этого все предоставляемые возможности. Если говорить о программисте, то нам нужен разработчик, который имеет представление обо всем технологическом стеке проекта. Аналитики и тестировщики должны соответственно проводить анализ и уметь работать с тестами.

Кроме того, универсал легко вживается в разные роли. Сегодня человек пишет код, завтра занимается анализом, а послезавтра — тестированием.

Люди, не боящиеся неопределенности

Если проект гибкий, это не означает, что в нем все пойдет как по маслу. Требования не будут преподноситься на блюдечке — нужно работать и самостоятельно выяснять их. Планы будут меняться, и вам придется приспосабливаться к этим изменениям.

Ищите людей, которые не боятся отбивать закрученные мячи, умеют держать удар и при необходимости изменять курс, не прекращая движения вперед.

Командные игроки, умеющие подавлять свое эго

Звучит избито, но для гибкой разработки лучше подойдут люди, которые умеют работать слаженно и подавлять собственное эго.

Не всем по вкусу расплывчатость ролей, принятая в гибкой методологии. Некоторые люди стараются защищать область, которую считают своей епархией.

Просто ищите тех, кто не имеет противоречий с самим собой, не боится делиться знаниями и искренне наслаждается взаимным обучением и ростом.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, я запутался. Если в гибком проекте нет predetermined ролей, то как же он работает?*

МАСТЕР: *Команда сделает то, что должно быть сделано.*

УЧЕНИК: *О да, Мастер, но если нет специальной роли тестировщика, как можно быть уверенным, что все необходимые тесты будут проведены?*

МАСТЕР: *Без тестирования никак не обойтись, поэтому команда будет им заниматься. Команда решает, сколько нужно тестов и сколько сил на это потратить.*

УЧЕНИК: *А если никто не хочет заниматься тестированием? Что, если всем нравится просто сидеть и писать код?*

МАСТЕР: *Тогда нужно найти людей, которым нравится тестировать, и самому убедиться, какими ценными членами твоей команды они станут.*

УЧЕНИК: *Благодарю тебя, Мастер. Я подумаю над этим.*

Что дальше?

Мы рассмотрели, как в гибких проектах исчезают четкие границы между ролями, почему команда будет работать наиболее успешно, если все соберутся в одном месте, и как, подыскивая людей в команду, находить специалистов-универсалов и тех, кто не боится неопределенности.

Теперь мы готовы сделать, пожалуй, один из важнейших шагов, чтобы отправить наш гибкий проект в свободное плавание. Поговорим об этапе, о котором почти ничего не сказано в большинстве гибких методологий, — как зарождается гибкий проект.

Из второй части книги вы узнаете, как с самого начала сориентировать свой проект на путь к успеху и гарантировать, что вы выбрали для работы нужных людей.

Часть II

**Концептуализация
проекта при гибкой
разработке**

Главное — никого не забыть



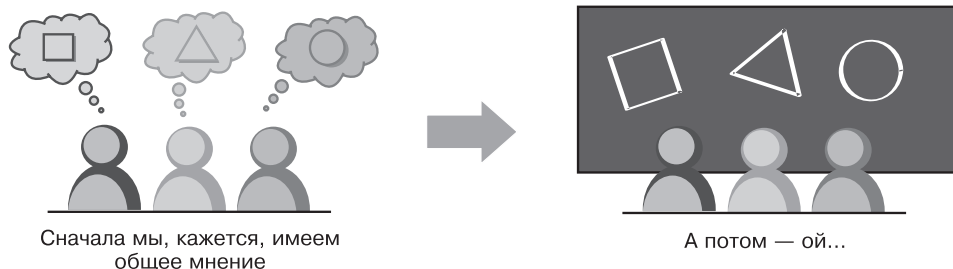
Многие проекты умирают в зачаточном состоянии. Обычно это происходит по одной из следующих причин:

- ❑ неумение задавать правильные вопросы;
- ❑ боязнь задавать сложные вопросы.

В этой части мы поговорим о мощной методике построения перспектив, которую условно назовем *стартовой колодой* (inception deck). Она помогает найти ответы на 10 вопросов, без которых лучше не начинать какой-либо софтверный проект. Испытав команду на данном этапе, вы узнаете, все ли нужные люди подобраны для проекта и в правильном ли направлении вы движетесь. Это произойдет еще до написания самой первой строки кода.

3.1. Из-за чего погибает большинство проектов

В начале любого нового проекта люди обычно имеют поразительно разные представления об общей цели.



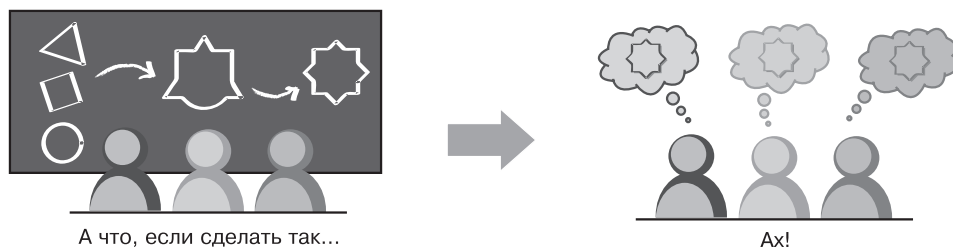
Для проектов это может быть губительно. Ведь, хотя мы и описываем наше видение общего дела на одном и том же языке, стоит нам приступить к работе — и мы понимаем, что думали о совершенно разных вещах.

И проблема не в том, что нам не удалось прийти к общему мнению уже на старте (это естественно). Проблема в том, что проекты начинаются *еще до того*, как найдены все нужные люди.

Ошибочное мнение о том, что консенсус достигнут там, где его нет и в помине, губит большинство проектов.

Нам нужно сформулировать план, который:

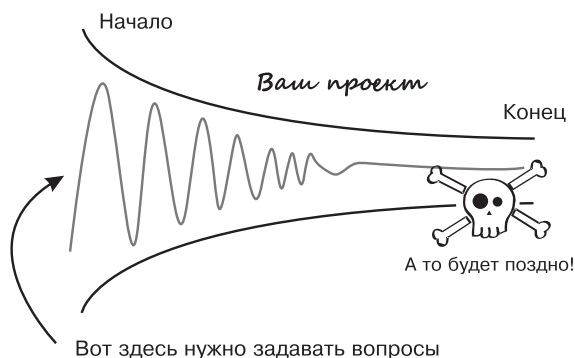
- ❑ позволяет сообщить команде цели, суть проблемы и контекст, в котором реализуется проект, так, чтобы при работе сотрудники могли принимать осознанные решения;
- ❑ предоставляет владельцам информацию, помогающую им решить, браться или не браться за дело, начинать проект или нет.



Единственный способ выстроить такой план — не бояться задавать вопросы.

3.2. Не избегайте сложных вопросов

Когда я работал в Новой Зеландии, мне представилась возможность сопровождать в поездке одного из крупнейших специалистов по маркетингу из компании ThoughtWorks — джентльмена по имени Кейт Доддс. Одна из многих вещей, которым научил меня Кейт, заключается в том, как важно уметь задавать самые сложные вопросы в самом начале любого нового предприятия.



Как видите, начиная любое новое дело (то есть проект), вы имеете большой простор для постановки вопросов и при этом ничего не теряете. Можно задавать общие вопросы, например, как приведенные ниже.

- ☐ Насколько опытна ваша команда?
- ☐ Занимались ли вы такими вещами ранее?
- ☐ Сколько денег у нас в распоряжении?
- ☐ Кто отдает приказы в этом проекте?
- ☐ Не смущает ли вас ситуация, когда в проекте участвуют два аналитика и тридцать разработчиков?
- ☐ Перечислите проекты, для работы над которыми вам пришлось пригласить команду молодых специалистов, практически незнакомых с объектно-ориентированным программированием, и успешно переделать устаревшую систему мейнфрейма для работы с Ruby on Rails — и сделать все это с помощью гибкой методологии.

Такие же вопросы необходимо задавать при запуске гибкого проекта. Нужно прояснить все щекотливые моменты с самого начала. Это нужно делать на этапе концептуализации проекта.

3.3. Знакомство со стартовой колодой



| | | |
|--|--------------------------------------|--------------------------------|
| Для чего мы собрались | Блицрезюме | Разработка оформления продукта |
| Список того, что мы не собираемся делать | Встреча с коллегами | Демонстрация решения |
| Что не дает покоя | Определяемся с длительностью проекта | Что необходимо сделать |
| Что для этого понадобится | | |

На этом уровне мы словно включаем прожектор, рассеивающий неясность и таинственность, окружающие ваш гибкий проект. Здесь мы прорабатываем 10 сложных вопросов и ситуаций, которые просто необходимо разрешить еще до начала проекта.

В компании ThoughtWorks такой подход часто используется для анализа той части первичных работ над проектом, о которой практически ничего не говорится ни в экстремальном программировании, ни в скраме. Речь идет о *закладке проекта* (project chartering). Мы знали, что глубокий шестимесячный анализ и упражнения в сборе требований нам не подходят, но не могли придумать какую-нибудь более легковесную альтернативу. И именно в таких условиях Робину Гиббонсу пришла в голову мысль о стартовой колоде: быстром, легком способе извлечения из проекта самой его сути и рассказа об общем понимании задачи всей команде и другим участникам проекта.

3.4. Как это работает

Смысл стартовой колоды заключается в том, что, если нам удастся собрать нужных людей в одной комнате и задать им правильные вопросы, это будет невероятно полезно для формулирования наших общих ожиданий, которые мы связываем с проектом.

Проведя в команде несколько упражнений и собрав их результаты в виде презентации (обычно — PowerPoint), можно совместно выработать достаточно хорошее понимание того, в чем заключается проект, чем он не является и что потребуется для его реализации.

В составлении стартовой колоды должны участвовать люди, непосредственно связанные с реализацией проекта. Речь идет о клиентах, владельцах, членах команды, разработчиках, тестировщиках, аналитиках — всех, кто способен сделать реальный вклад в эффективное выполнение проекта.

Отдельно подчеркну, как важно, чтобы в этом проекте участвовали владельцы. Ведь стартовая колода — это инструмент, предназначенный не только для нас, но и для них, и с ее помощью они могут принять решение о том, стоит ли вообще начинать проект.

На построение типичной стартовой колоды уходит от пары дней до примерно двух недель. Она равноценна примерно шести месяцам работ по планированию проекта и должна пересматриваться при внесении любых крупных изменений в суть или направление развития проекта.

Такой пересмотр необходим, поскольку стартовая колода — это живая, открытая система. Она не из тех вещей, которые можно один раз сделать и положить на полку. До самого завершения проекта схема должна висеть в офисе, где трудится команда, и напоминать о том, над чем идет работа и с какой целью.

Разумеется, вопросы и упражнения, представленные здесь, — это далеко не все. До начала проекта вам придется обдумывать и другие вопросы, разрабатывать упражнения и прояснять детали.

Итак, используйте такую схему в качестве отправной точки, но не следуйте ей слепо и не бойтесь корректировать ее под себя.

3.5. Сущность стартовой колоды

Ниже перечислены основные вопросы и упражнения, прорабатываемые на этапе концептуализации проекта.

1. **Зачем мы здесь собрались?** Это быстрое напоминание о нашей цели, наших клиентах, а также о том, почему мы решили заняться данным проектом в первую очередь.

2. **Составление блицрезюме.** Если бы у нас было 30 секунд, за которые нужно описать наш проект в двух фразах, что бы мы о нем сказали?
3. **Разработка оформления продукта.** Если бы мы быстро листали журнал и наткнулись на рекламу нашего продукта или услуги, то что бы она нам сообщила, и еще важнее — согласились ли бы мы за это заплатить?
4. **Составление списка того, что мы не собираемся делать.** Вполне ясно, что мы собираемся делать при реализации нашего проекта. Давайте еще точнее опишем ситуацию и подчеркнем, чего мы ни в коем случае делать не будем.
5. **Встреча с коллегами.** Сообщество специалистов, занятых в проекте, всегда больше, чем кажется. Почему бы не пригласить их на кофе, чтобы все могли познакомиться друг с другом?
6. **Демонстрация решения.** Давайте нарисуем общий концептуальный проект технической архитектуры, чтобы убедиться, что все мы одинаково представляем себе предстоящую работу.
7. **Что не дает нам покоя?** Иногда в ходе выполнения проектов происходят неприятные вещи. Но если поговорить о них и подумать, как их избежать, то, возможно, все будет не так плохо.
8. **Определение временных параметров.** Сколько времени займет проект: три месяца, а может, шесть или девять?
9. **Определение требуемого результата.** Проекты зависят от таких факторов, как время проекта, его функционал, бюджет и качество. Что наиболее и наименее важно для данного проекта в настоящий момент?
10. **Что нам требуется для достижения результата?** Сколько времени будет длиться проект? Сколько он будет стоить? И какая команда нам нужна для реализации этого проекта?

Мы изучим стартовую колоду в два этапа. В главе 4 поговорим о том, *зачем* мы беремся за проект, а в главе 5 рассмотрим, *как* его реализовать.

Общее представление о ситуации



Разработка программного обеспечения — один из уникальных видов деятельности, в которой сочетаются черты проектирования, конструирования, искусства и науки.

Ежедневно команды должны принимать тысячи решений и идти на не меньшее количество компромиссов. А без верного контекста и общего представления о ситуации такие решения не могут быть полностью осознанными или взвешенными.

При изучении первой части стартовой колоды мы должны досконально выяснить, *на чем* основан наш проект.

Для этого нужно ответить на ряд вопросов.

- ☐ Для чего мы здесь собрались?
- ☐ Каково блицрезюме нашего проекта?
- ☐ Как будет выглядеть реклама нашего продукта?
- ☐ Чего мы не собираемся делать?
- ☐ Кто работает с нами в команде?

В финале этой главы вы и ваша команда будете ясно понимать, какова цель проекта, что именно вы создаете и почему вы это делаете. Обо всем этом вы сможете с легкостью рассказать другим.

Но сначала зададим нашим спонсорам вопрос, зачем мы здесь?

4.1. Вопрос: зачем мы здесь?

Зачем мы здесь?

Чтобы спокойно отслеживать, какие работы
идут на нашей виртуальной стройке



Прежде чем любая команда сможет добиться успеха, она должна понять, *зачем* она занимается решением конкретной задачи. Поняв это, команда получает возможность:

- ☐ принимать оптимальные и осознанные решения;
- ☐ лучше выполнять работу, уравнивая противодействующие силы и идя на компромиссы;

- ❑ выдавать более инновационные и качественные решения, поскольку команде разрешено думать самостоятельно.

Задача сводится к тому, чтобы *понять намерение начальника* и сформулировать для себя, как *достичь поставленной цели*.

«Тойота»: компания, умеющая видеть и достигать

В замечательной книге *The Toyota Way* [Lik04] Джеффри Лайкер рассказывает историю о том, как однажды в 2004 году главному инженеру этой компании поручили переработать модель «Тойота-Сиенна» для североамериканского рынка. Чтобы почувствовать, как жители Северной Америки живут, работают и занимаются своими машинами, он с командой проехал на «Тойоте-Сиенна» по всем штатам США и Мексики, а также по всем провинциям Канады. И вот что выяснилось.

- Североамериканские водители больше едят и пьют в машине, чем японские автомобилисты (так как в Японии обычно приходится ездить на более короткие расстояния). Поэтому в любой «Тойоте-Сиенна» есть центральный лоток и 14 подставок для чашек.
- На канадских дорогах более высокий поперечный уклон, чем в США (выгнутый в середине), поэтому при вождении очень важно контролировать скольжение.
- Сильные ветры, дующие в провинции Онтарио, требуют особого внимания к устойчивости автомобиля к боковому ветру. Если вы поедете куда-нибудь, где сильные боковые ветры, вас приятно удивит устойчивость и легкость в движении новой «Тойоты-Сиенна».
- Если бы главный инженер мог всего лишь прочитать об этих проблемах в маркетинговом отчете, он не почувствовал и не осознал бы на собственном опыте суть данных проблем.

Идите и попробуйте сами

Одно дело — догадываться, зачем мы здесь, и совсем другое — знать об этом с определенностью. Чтобы действительно увидеть проблему с точки зрения клиента и понять, что ему нужно, требуется поставить себя на его место.

Пойти и посмотреть — означает растормошить команду и познакомить ее с той сферой, где будет происходить действие.

Например, если вы создаете систему обеспечения производственной безопасности для инженерной компании, которую предполагается использовать на промплощадке, — отправьтесь на место разработок. Поговорите

с офицерами службы безопасности. Посмотрите на тягачи. Познакомьтесь со стесненными условиями, неустойчивым интернет-соединением, а также с теми маленькими кабинетами, в которых будут работать ваши клиенты. Проведите день на этом месте и поработайте с людьми, которым ежедневно придется пользоваться системой, которую вы собираетесь разрабатывать.

Войдите в курс дела, задавайте вопросы и ненадолго превратитесь в своего клиента.

Как понять заказчика

Заказчик выражает свои намерения в краткой фразе, пожелании или формуле, которая обобщает цели и задачи вашего проекта. Эта краткая формулировка должна служить путеводной звездой, на которую можно взглянуть в последнюю минуту, в самом разгаре битвы и решить, атаковать или отступить.

В книге *Made to stick* [НН07] Чип и Дэн Хиты рассказывают, как в компании Southwest Airlines обсуждали, включить ли в меню одного из рейсов куриный салат «Цезарь».

Когда был задан вопрос, позволит ли это снизить стоимость билета (этого хотел добиться главный исполнительный директор Хербс Келлехер), стало понятно, что добавлять куриный салат абсолютно бессмысленно.

Итак, желание заказчика в начале проекта не обязательно должно сводиться к чему-то большому и пафосному. Оно может быть очень простым и совершенно не выходить за рамки вашего проекта.

Суть этого упражнения — заставить людей сказать о том, *что у них на уме*, а затем уточнить у клиента, действительно ли это — все, что требуется.

4.2. Создание блицрезюме



Блицрезюме

- Для [руководителей строительных работ],
- которые [должны отслеживать, какие работы проводятся на промплощадке],
- обеспечивают [соблюдение техники безопасности при проведении работ]
- в [системе выполнения требований техники безопасности]
- которая [создает, отслеживает и проверяет разрешения на выполнение работ].
- В отличие от [современных бумажных систем документооборота]
- наш продукт [основан на веб-технологиях, поэтому к нему можно обратиться когда угодно и откуда угодно].

10 причин взяться за выполнение вашего проекта

Недавно я выполнял это упражнение с командой, перед которой была поставлена задача создания инвойсов для нового отдела компании. Меня удивил разброс мнений, царивший в команде относительно того, зачем начался этот проект.

Некоторые считали, что смысл — снизить количество страниц в стандартном инвойсе и сэкономить таким образом бумагу. Другие думали, что так упростится заполнение инвойса и поэтому снизится нагрузка на колл-центр. Третьи полагали, что так реализуется возможность запуска целевых маркетинговых кампаний, задача которых — повысить продажи товаров и услуг.

Все это хорошие ответы, но ни один из них не оправдывал проект сам по себе. Понадобилось немало дискуссий и дебатов, чтобы перед командой вырисовалась истинная цель проекта и пришло ее понимание. На самом деле все делалось именно для упрощения инвойса и снижения нагрузки на колл-центр.

Поторопитесь! Венчурный инвестор, которого вы пытались выловить для разговора тет-а-тет на протяжении трех месяцев, только что вошел в лифт, и у вас есть 30 секунд, чтобы познакомить этого капиталиста с идеей вашего новоиспеченного проекта. Сможете — ваше предприятие получит столь необходимую поддержку. Не сможете — тогда снова придется питаться одним роллтоном.

Блицрезюме — это и есть ваша «речь в лифте». В нем вы должны сформулировать свою идею за крайне короткий промежуток времени. Блицрезюме предназначены не только для завлечения рискованных предпринимателей-авантюристов. В форме такого резюме еще удобно кратко и точно описывать новые софтверные проекты.

Хорошее блицрезюме помогает решить ряд очень важных для проекта задач.

1. **Вносит ясность.** Блицрезюме — это не попытка удовлетворить «и наших и ваших». Оно заставляет команду отвечать на сложные вопросы о том, чем является будущий продукт и для кого он предназначен.
2. **Заставляет команду задуматься о клиенте.** Делая акцент на том, что будет делать программа и как именно, команды приобретают ценные идеи, касающиеся особенностей данного продукта и того, почему клиент купит в первую очередь именно его.
3. **Помогает вникнуть в суть дела.** Блицрезюме, подобно лазеру, прореживает массу хлама и попадает в самое сердце проекта. Такая ясность помо-

гает расстановке приоритетов и значительно улучшает соотношение «сигнал — помеха», то есть количество того, что действительно важно.

Теперь давайте рассмотрим шаблон для составления блицрезюме.

Шаблон блицрезюме

- Для [адресат сообщения],
- который [утверждение о необходимости или о предоставляющейся возможности]
- это [название продукта]
- относится к [категория продукта]
- и при этом [основная выгода, убедительная причина приобрести].
- В отличие от [основное конкурентное предложение]
- наш продукт [объяснение основного положительного отличия].

Существует несколько способов преподнесения блицрезюме. Тот, который предпочитаю я, взят из книги Джефффри Мура *Crossing the Chasm* [Моо91].

- *Для* [адресат сообщения]. Объясняется, на кого ориентирован проект или кому он мог бы принести пользу.
- *Который* [утверждение о необходимости или о предоставляющейся возможности]. Расширенное представление проблемы или потребности, стоящей перед клиентом.
- *Это* [название продукта]. Проект начинается с присвоения имени. Название важно, так как оно помогает понять ваши намерения.
- *Относится к* [категория продукта]. Объясняется, чем, по сути, является данный продукт или услуга и какова полезная нагрузка проекта.

Быть кратким нелегко

Одна из причин, по которой блицрезюме оказывает такое сильное воздействие, — его краткость. Но не думайте, что написать короткое резюме так просто.

Вам и вашей команде потребуется немало времени для написания хорошей речи, поэтому не беспокойтесь, если все получится не сразу. Создать хорошее блицрезюме бывает непросто, но оно стоит затраченных усилий.

«Я написал бы вам еще короче, но у меня нет на это времени». — Блез Паскаль, «Письма к провинциалу», XVI.

- *И при этом* [основная выгода, убедительная причина приобрести]. Объясняется, почему клиент не отказался бы купить в первую очередь именно этот продукт.

- ❑ *В отличие от* [основное конкурентное предложение]. Говорится, зачем нужен продукт, если на рынке уже есть аналог.
- ❑ *Наш продукт* [объяснение основного положительного отличия]. Здесь мы помогаем почувствовать разницу и объясняем, что в нашем предложении особенного, чем оно лучше альтернатив, предлагаемых конкурентами. Это самое важное. Именно на данном этапе мы объясняем, почему стоит вложить деньги в наш проект.
- ❑ В двух красивых фразах, которые вы успеете сказать в лифте, должна быть заключена суть проекта или идеи. Необходимо рассказать, что собой представляет проект, для чего он нужен и почему стоит купить именно его.

Есть несколько способов составить вместе с командой такое блицрезюме. Можно напечатать шаблон, попросить каждого попробовать заполнить его самостоятельно, прежде чем собирать всех вместе.

Или, если вы хотите спасти елочку и не тратить бумагу, просто выведите шаблон на экран с помощью проектора и попробуйте заполнить его вместе со всей группой, по одному разделу за раз.

Когда блицрезюме будет готово, следует проявить творческие способности и перейти к созданию оформления для продукта.

4.3. Разработка оформления продукции

Система обеспечения
производственной безопасности
для инженерной компании

Идеальный вариант
для горнодобывающей промышленности



Обработывайте допуски к работе быстрее!
Обработывайте допуски к работе надежнее!
Отслеживайте рабочее время лучше!

Где вам это потребуется.
Когда вам это потребуется

Иногда компьютерные программы оказываются для компаний необходимым злом. Чтобы не принимать на себя весь риск и все неопределенные моменты, связанные с крупными проектами, многие клиенты предпочитают отправиться в местный супермаркет, вытащить карточку и просто закупить все, что надо.

Возможно, мы еще долго не увидим полок супермаркетов, уставленных программными продуктами, которые заботливо упакованы в пластиковые коробки и стоят шестизначные суммы. Но в связи с этим возникает интересный вопрос. Если бы можно было купить программу в супермаркете, как бы выглядела ее упаковка? И — еще важнее — купили бы мы ее?

Создавая оформление для своего проекта и задавая вопрос, почему клиент должен им заинтересоваться и купить его, вы сосредотачиваете внимание вашей команды на том, что более всего интересует вашего потребителя, и на основных преимуществах вашего продукта. При работе команда должна четко знать ответ и на первый, и на второй вопрос.


Как это работает

Представляю, о чем вы сейчас думаете. «Какой из меня креативщик. Я не силен в рекламе. Конечно, я не смогу разрекламировать наш продукт». Позвольте вас заверить — сможете! Я покажу вам, как это делается, за три простых этапа.

Этап 1. Мозговой штурм о достоинствах вашего проекта

Никогда не рассказывайте клиенту о характерных особенностях вашего продукта — это неважно. Людей интересует то, как ваш продукт сможет облегчить им жизнь, иными словами, какая от него польза.

Предположим, мы пытаемся убедить семью покупателей в том, что ей очень пригодился бы мини-вэн. Мы могли бы дать им подробное описание машины либо остановиться на том, как такой автомобиль полезен в обычной жизни.

| Функции |  | Польза |
|---|---|---|
| Двигатель 245 лошадиных сил | | Легкий ход машины по шоссе |
| Круиз-контроль | | Экономия денег |
| Антиблокировочная система | | Безопасное торможение вместе с любимой семьей |
| <i>Обязательно рассказывайте не о функциях, а о том, что в них полезного!</i> | | |

Чувствуете разницу?

Итак, первый шаг при оформлении продукции — это собраться вместе с командой и заказчиком и обсудить причины, по которым сотрудникам понравится с ней работать. Затем выберите три основные из них.

Этап 2. Создание слогана

Основа хорошего слогана — максимальное количество смысла в минимальном количестве слов. Не нужно описывать три перечисленные ниже компании, так как их слоганы говорят сами за себя.

- ❑ Acura — «За рулём или на ложе, а всё лучше помоложе»¹.
- ❑ FedEx — «Весь мир по расписанию»².
- ❑ Starbucks — «Рухни в прохладу»³.

Чувствуете эмоции, заложенные в этих слоганах? Теперь расслабьтесь. Эти слоганы — одни из лучших, и ваш совсем не обязательно должен быть таким же профессиональным. Просто соберитесь с командой, выделите 10 или 15 минут на придумывание слогана и проявите свои творческие способности. Помните — ни один слоган не бывает чрезмерно банальным!

Этап 3. Разработка упаковки

Отлично! Сформулировав три отличные причины купить ваш продукт и придумав для него запоминающийся слоган, вы готовы собрать все вместе.

The diagram shows a rectangular template for product packaging. At the top, it says "Здесь будет название продукта" (The product name will be here). Below this is a large rectangular area with a diagonal line forming an 'X' shape, labeled "Отличная картинка" (Excellent picture). Underneath the picture area is the text "Великолепный слоган" (Excellent slogan). At the bottom, there are three lines for benefits, labeled "Польза 1", "Польза 2", and "Польза 3" respectively, each followed by a horizontal line for text.

¹ В оригинале: The true definition of luxury. Yours («Истинное воплощение роскоши. Вашей»). — *Примеч. пер.*

² В оригинале: Peace of mind («Спокойствие и уверенность»). — *Примеч. пер.*

³ В оригинале: Rewarding everyday moments («Воздавая должное повседневности»). — *Примеч. пер.*

Выполняя эту часть работы, представьте себе, как покупатель заходит в ближайший софтверный магазин и замечает на полке вашу программу. А когда клиент возьмет с полки коробку, она настолько ему понравится, что он сразу купит 10 штук — для себя и своих друзей.

Итак, переходим к оформлению упаковки!

Не пытайтесь сразу создать шедевр. Просто возьмите ватман, фломастеры, бумажные наклейки, иголки и все, что еще может понадобиться. Озвучьте ваш слоган. Расскажите покупателям о пользе программы. Потратьте 15 минут на создание такой качественной упаковки, на которую вы только способны.

Прекрасно! Видите — не боги горшки обжигают. Надеюсь, вам понравился этот опыт (не каждый день приходится брать карандаши и рисовать выдающиеся изображения продукта). Такая практика отлично помогает сплотить команду и позволяет критически подойти к вопросу о том, *что* стоит за вашей программой.

Пришло время узнать, как дать клиенту представление о предполагаемом функционале нашего проекта.

4.4. Создание списка функций

| БУДЕТ РЕАЛИЗОВАНО В ПРОГРАММЕ | НЕ БУДЕТ РЕАЛИЗОВАНО |
|---|--|
| Создание нового допуска. Обновление/чтение/удаление имеющихся допусков. Функция поиска. Базовая система отчетности. Печать | Взаимодействие с устаревшей системой блокировки проезда. Возможность работы офлайн |
| НЕ РЕШЕНО | |
| Интеграция с системой отслеживания логистики. Система считывания электронных карт допуска | |

Знакома клиента с ожидаемым функционалом проекта, важно рассказать не только о том, что вы собираетесь сделать, но и о том, чего вы делать *не планируете*.

Создавая список функций, которые вы не собираетесь реализовывать, вы ясно указываете, что входит в проект, а что — нет. Поступая так, вы не только позволяете клиенту составить верное представление о том, на что следует

рассчитывать, но и гарантируете, что ваша команда сосредоточится при работе только на самом важном, абстрагировавшись от всего остального.

Как это работает

Список того, чего мы не собираемся делать, — отличный способ показать, что относится к нашему проекту, а что — нет. Как правило, нужно собраться с командой и клиентом и определиться со всеми невыясненными деталями, обсудив в режиме мозгового штурма основные функции, которые клиент хочет видеть в программе.

| ВХОДИТ | | НЕ ВХОДИТ | |
|---|--|---|-------------------------|
|  | Большие блоки, которые нам нужно сложить |  | Все, что нас не волнует |
| НЕ РЕШЕНО | | | |
| Вещи, с которыми еще нужно определиться | | | |

Все, что ВХОДИТ в проект, требует от нас глубокого внимания. Здесь мы имеем дело с большими блоками, из которых собираемся сложить наш проект. Это могут быть важнейшие функции (например, система отчетности) или общие задачи (например, масштабируемость в стиле Amazon).

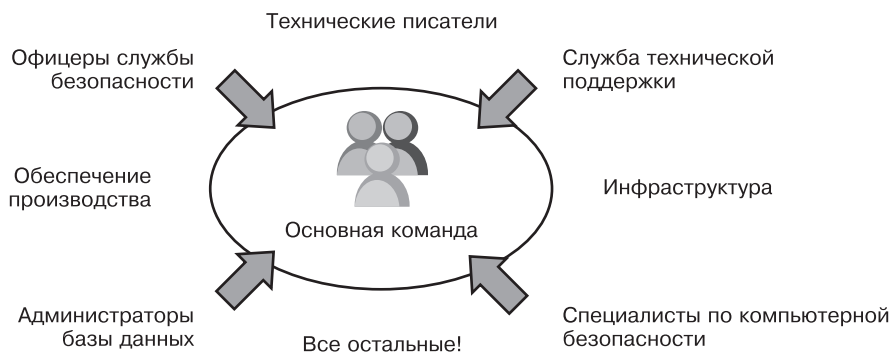
Все, что НЕ ВХОДИТ в проект, — детали, которые в данный момент нас не волнуют. Возможно, мы обратимся к ним в следующей версии или просто не будем включать их в проект. Но пока нас это совершенно не касается.

К НЕРЕШЕННЫМ задачам относится то, с чем еще предстоит определиться. Это очень важный раздел, отражающий истинную природу большинства софтверных проектов. У всех людей может быть свое мнение о том, без чего можно обойтись. В итоге все НЕРЕШЕННОЕ должно переключаться В проект или остаться ВНЕ его.

Прелесть этого визуального подхода заключается в том, как много можно мгновенно сообщить с его помощью. Перечислив важнейшие составляющие проекта слева, то, что к нему не относится, — справа и нерешенные вопросы — внизу, мы сразу понимаем, в каких границах находится функционал проекта.

Полностью определив объем работ по проекту, двигаемся дальше. Знакомимся с нашими коллегами.

4.5. Встреча с коллегами



Хорошие коллеги могут стать вашими лучшими друзьями. Они придут на помощь, когда вы захлопнете домашнюю дверь, а ключ забудете. Они помогут, когда вам понадобится какой-нибудь инструмент. И будет замечательно, если при необходимости вы поможете коллеге настроить дома беспроводную сеть.

Вопрос на миллион долларов



Как-то раз я участвовал в концептуализации проекта в большой канадской энергетической компании. И тогда вице-президент отдела спросил, как эта новая система будет интегрироваться с имеющимся, но устаревшим мейнфреймом.

В комнате стало так тихо, что можно было услышать, как жужжит муха. Вице-президент, тот самый человек, который выписывает чеки и несет полную ответственность за успех проекта, не понимал, что новая система вообще не должна интегрироваться со старой. Она должна была ее полностью заменить.

И только потому, что мы специально перечислили, чего не собираемся делать, нам удалось избежать крупного разочарования на более поздней стадии проекта. Лучше делать это сразу, чем пытаться объяснять подобные вещи, когда проект уже выполняется.

Необходимо учитывать, что при реализации любых проектов вам придется работать с коллегами. Но они помогут вам не тем, что сохраняют запасной ключ от вашего дома, и не тем, что одолжат нужный инструмент, а тем, что будут управлять базами данных, проводить проверку безопасности и поддерживать функционирование сетей.

Можно заранее выстроить отношения с коллегами, что потом будет оплачено сторицей. Встречаясь с коллегами, всегда несложно сказать «Привет» и лишь после этого бросаться решать неотложные проблемы. И что еще важнее, когда вы строите отношения с коллегами, то закладываете фундамент для успеха любого проекта — добиваетесь взаимного доверия.

Мой первый большой промах

Никто не застрахован от ошибок. Одну из самых больших профессиональных ошибок я совершил, когда работал руководителем команды в ThoughtWorks. Тогда мы выполняли один заказ для Microsoft.

Я взялся за работу и приступил к выполнению проекта, считая, что наш коллектив выглядит вот так:



И какое-то время все было нормально. Команда работала гибко. Мы регулярно делали хорошие программы и радовались жизни.

Затем, ближе к концу проекта, началось что-то странное. Откуда-то стали появляться группы людей или отдельные личности, которых я никогда раньше не встречал, и предъявлять мне и команде нелепые требования.

- ❑ Одни стремились пересмотреть архитектуру нашего продукта (как будто она вообще нуждалась в пересмотре!).
- ❑ Другие хотели убедиться, что мы соблюдаем действующие в компании правила обеспечения безопасности (о как!).

□ А третьи желали изучить нашу документацию (какую еще документацию!).

Кто все эти люди? Откуда они взялись? И почему они так стремились сорвать наше расписание?

За одну ночь прекрасная маленькая команда из шести человек стала гораздо больше и непонятнее.



Я, конечно, стал ругаться с незваными гостями по поводу того, что они вмешиваются в наше расписание. Но дело было как раз в том, что сначала я не понимал: *количество участников проекта всегда больше, чем вам кажется*¹.

Встречаясь с коллегами, вы начинаете понимать, кто, кроме вас, участвует в проекте, видеть ситуацию с их точки зрения и строить отношения еще до того, как ими придется воспользоваться. Таким образом, когда дело дойдет до тесных контактов, вы не будете совершенно незнакомы и эти люди будут гораздо больше расположены помочь вам.

Как это работает

Соберите команду и устройте мозговой штурм на тему «С кем нам придется иметь дело в ходе реализации проекта». Члены команды, которые давно работают с данной компанией, наверняка хорошо знакомы со всеми правилами, действующими в ней, и их опыт в преодолении организационных преград будет неоценим.

¹ *The Blind Men and the Elephant* [Sch03].

Кофе, пончики и чистосердечность

Когда речь заходит о выстраивании уважительных отношений с коллегами, нет ничего лучше, чем чашечка кофе и вкусный пончик.

Кофе — потому, что он подается в приятном горячем сосуде и коллеги, пьющие с вами кофе, будут ассоциировать вас с его теплом.

А если во время вашего рассказа о том, как вам нравится быть в кругу собравшихся, коллеги будут наслаждаться сахарной сдобой, они воспримут работу с вами как такую же сладость.

Но самый важный компонент налаживания отношений с коллегами — это, конечно же, чистосердечность.

Чтобы коллеги по-настоящему почувствовали ваше расположение и высокую оценку, вы должны это подчеркнуть. Неискренняя болтовня раскусывается сразу. Зато подлинное восхищение и искренняя благодарность невероятно помогают завоевать человека. И вам, и вашему проекту это пойдет только на пользу.

Широкое профессиональное сообщество

*Здесь перечислите членов
вашей основной команды*



*Люди, с которыми нужно
выстроить отношения*

Руководство (по борьбе с корпоративным и бухгалтерским мошенничеством).
Проверка безопасности.
Управление подготовки и обеспечения производства.
Управление по преобразованию бизнеса.
Служба по внесению изменений.
Администраторы базы данных.
Корпоративная архитектура.
Учебная группа.
Технические писатели.
Юридический отдел.
Служба поддержки.
Обеспечение сетевых взаимодействий/инфраструктура.
Служба по развитию лидерских навыков.
Управление рисками и соответствие требованиям.
Директора филиалов

И все остальные



Как только вы будете представлять себе «план местности», поговорите с каждой из групп и посмотрите, сможете ли вы приступить к подписанию новых контактов. Ваш проект-менеджер или кто-то другой, кто будет выстраивать эти внешние отношения, может подготовить план, который позволит заинтересовать коллег из этих групп.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, многие из твоих уроков требуют участия владельцев или спонсоров. Что, если они недоступны или просто очень заняты и не могут отвечать на подобные вопросы о проекте?*

МАСТЕР: *Тогда можешь себя поздравить. Ты только что обнаружил крупное уязвимое место своего проекта.*

УЧЕНИК: *А в чем заключается риск?*

МАСТЕР: *В незаинтересованности клиента. Если клиент не интересуется проектом, то этот проект подвергается опасности, еще не начавшись. Если у клиента нет времени рассказать, зачем ты пишешь для него эту программу, то, возможно, ее вообще не стоит писать.*

УЧЕНИК: *Ты имеешь в виду, что проект нужно остановить?*

МАСТЕР: *Я имею в виду, что для успешной реализации проекта необходим вклад клиента и владельца. Без этого проект уже гложет, хотите вы этого или нет.*

УЧЕНИК: *И если так, то что делать?*

МАСТЕР: *Нужно ясно и настойчиво рассказать клиенту, что требуется сделать, чтобы такой проект завершился успешно. Возможно, клиент действительно очень занят и запланировал слишком много дел. Если так, скажите, что можете поговорить, когда он освободится. До того вы поработаете с другими клиентами.*

УЧЕНИК: *Спасибо, сэнсэй. Я подумаю об этом.*

Что дальше?

Прежде чем переходить к следующей главе, остановимся и переведем дух.

Чувствуете?

Понимаете, что происходит?

Выполняя один за другим этапы концептуализации проекта, мы начинаем лучше понимать его смысл и функционал.

- ☐ Мы узнаем, *зачем* выполняем этот проект.
- ☐ У нас хорошее блицрезюме.
- ☐ Мы знаем, как будет оформлен наш продукт.
- ☐ Мы определяем объем работ по проекту.
- ☐ Мы начинаем достаточно хорошо понимать, с кем нам придется работать на этом проекте.

Знаю, о чем вы думаете. Хватит контекста! Когда же мы перейдем к делу и поговорим о том, как именно выполняется проект? И ответ будет дан прямо сейчас.

Из главы 5 вы узнаете, как выглядит техническое решение нашего проекта и как будет проходить его реализация.

Воплощение в реальность



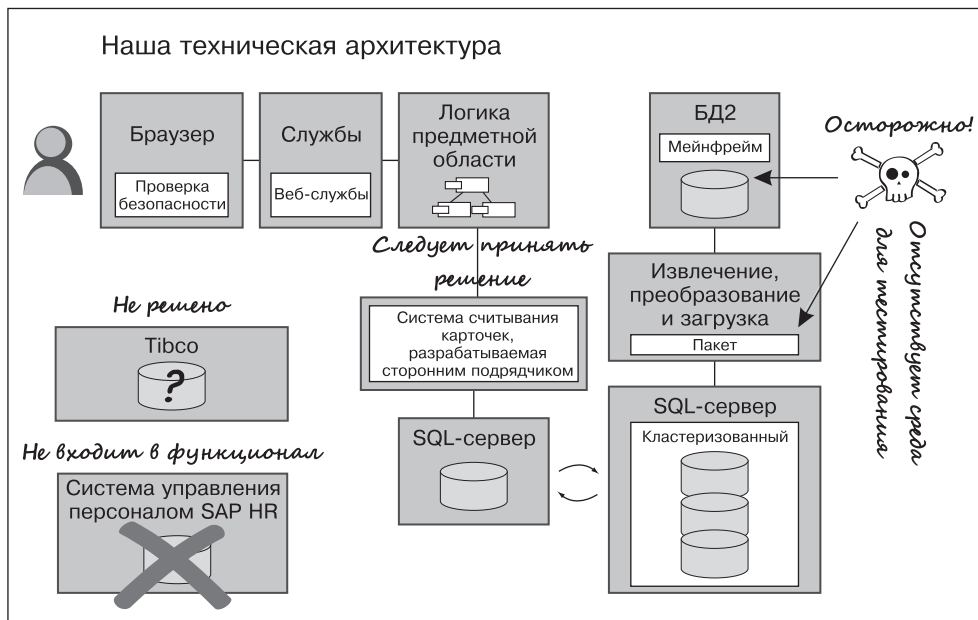
Теперь, когда мы знаем, *зачем* затевается проект, пришло время понять, *как* его реализовать. В этих разделах стартовой колоды мы более конкретно разберем наше решение и начнем разработку.

На данном этапе нам предстоит:

- ☐ представить техническое решение;
- ☐ рассмотреть некоторые риски;
- ☐ определиться с длительностью проекта;
- ☐ уяснить, каких именно результатов нужно достичь (так или иначе, ситуация обязательно решится);
- ☐ показать спонсорам, что необходимо для проекта.

Но сначала давайте продемонстрируем решение.

5.1. Демонстрация решения



Визуализация решения нужна для того, чтобы понять, с какими техническими нюансами придется столкнуться, и показать всем, как именно мы собираемся решить поставленную задачу.

Рассказывать о решении и представлять его на суд команды и заказчика полезно по нескольким причинам:

- ❑ мы рассказываем, с какими инструментами и технологиями придется работать;
- ❑ мы представляем наши предположения о том, каковы будут границы проекта и его функционал;
- ❑ мы сообщаем о возможных рисках.

Даже если вы полагаете, что другие участники проекта согласны с вашим решением, все равно покажите его всем. В худшем случае вы еще раз расскажете всем о том, что и так уже известно. В лучшем случае вы сэкономите массу нервов, не прилагая огромных усилий на выполнение проекта, который, оказывается, кому-то не подходит.

Как это работает

Вы просто собираетесь вместе с остальными технарями из вашей команды и рассказываете, как будет выполняться поставленная задача. Вы схемати-

чески изображаете архитектуру, прорабатываете сценарии «что, если...» и вообще пытаетесь дать людям представление о том, насколько объемна и сложна предстоящая работа.

Если вы размышляете о том, работать ли с проприетарными или свободно распространяемыми инструментами, поделитесь вашими соображениями с командой (в некоторых компаниях можно работать не со всеми свободными инструментами).

Архитектуру нужно подбирать одновременно с командой

Как говорится, если в руке большой молоток, все вокруг начинает напоминать гвозди¹.

Команда, хорошо умеющая работать с базами данных, конечно же, захочет реализовать самые сложные задачи на SQL, в то время как команда, сильная в объектно-ориентированном подходе, сделает основной упор именно на него.

Итак, подбирая команду, вы уже в значительной мере выбрали вашу архитектуру, хотите вы этого или нет.

Вот почему важно сообщать о своих технических предпочтениях как можно раньше — не потому, что ваше решение безупречно или у вас есть ответы на все вопросы, а потому, что на проект нужно подобрать подходящих людей и гарантировать, что они будут согласны с предложенным вами решением.

Суть именно в этом. Нарисуйте достаточно картинок и покажите, как вы собираетесь построить систему. Расскажите о наиболее рискованных областях и убедитесь, что все согласны с предложенным техническим решением.

5.2. Что нас беспокоит



Риски, связанные с проектом

- Мнение директора о возможности данной конструкции.
- Команда не работает в одном месте.
- Новая архитектура системы безопасности.
- Временные рамки разработки новой системы отслеживания логистики

¹ Афоризм принадлежит Марку Твену. — Примеч. пер.

Пока реализуется софтверный проект, многие менеджеры перестают спать по ночам — и не случайно. Оценки бывают чрезмерно оптимистичными. Клиенты могут постоянно менять намерения (и они действительно так делают). Всегда оказывается, что нужно сделать больше дел, чем предполагалось, и на все не хватает времени. И это только те риски, о которых нам известно!

Задавая вопрос о том, *что нас беспокоит*, мы начинаем здоровую дискуссию о некоторых проблемах, с которыми может столкнуться при работе команда, а также пытаемся определить, как предотвратить такие проблемы и ситуацию, в которой приходится работать не разгибаясь.

Почему говорить о рисках полезно



Обсуждение рисков, связанных с проектом, — одно из тех дел, от которых желает уклониться большинство людей, начинающих проект. Никто не хочет выглядеть как цыпленок, который носится и причитает, что небо вот-вот рухнет.

Но обсуждение рисков — отличный способ донести до людей, что нужно для успешного завершения проекта.

Возьмем, например, расположение работников в одном офисе. Для тех, кто не знает, как ведутся софтверные проекты, это может показаться мелочью. Однако в гибком проекте совместное расположение коллег правит бал, и именно при обсуждении рисков вы можете раскрыть карты и убедить

собеседников, что если следующие опасения верны, то проект просто не может увенчаться успехом:

- ❑ у нас нет команды, которая работает вместе;
- ❑ наш клиент не интересуется ходом проекта;
- ❑ мы не контролируем среду, в которой идет разработка;
- ❑ у нас нет чего-то еще, что требуется для успешной реализации проекта.

Блумберг о риске

Согласитесь, Майкл Блумберг кое-что знает о рисках. Как основатель финансовой компании Bloomberg и мэр Нью-Йорка он нередко уподоблялся штурману, ведущему судно среди рифов.

В книге *Bloomberg by Bloomberg* [Blo01] Майкл рассказывает о своем излюбленном методе, помогающем справляться с рисками.

1. Формулировать все проблемы, которые потенциально могут возникнуть.
2. Очень хорошо думать о том, как избежать таких проблем.
3. Затем окончательно с ними разобраться.

Философия Майкла Блумберга заключается в том, что никогда не удастся предусмотреть всего и что ни один план не совершенен. Жизнь преподносит сюрпризы, а подать иск против нее у вас не получится. Привыкните к этому. Или вы знаете, что происходит, или не знаете и никогда не узнаете. Что до остального, просто принимайте жизнь такой, какая она есть.

На этом этапе можно встать и попросить о том, что вам нужно. Вероятно, вы не получите всего, что хотите, но как минимум заявите об этом и объясните всем последствия, возможные при невыполнении вашей просьбы.

Приведу несколько хороших причин поговорить о рисках в самом начале проекта.

- ❑ Сообщать о проблемах, связанных с проектом, нужно на раннем этапе. Не ждите, пока мина замедленного действия рванет. Если у вас есть сложности или вы видите какие-то риски, неустранение которых может привести к срыву проекта, разобраться с ними нужно уже сейчас.
- ❑ Вы получаете шанс указать на явные ошибки. Если в ходе концептуализации проекта выдвигались какие-то безумные предложения, сейчас самое время сказать об этом.

- ❑ Это просто производит положительное впечатление. Когда делишься своими опасениями с другими и обсуждаешь такие опасения — это хорошо. Команда получает дополнительную возможность сплотиться, поделиться профессиональным опытом и узнать, как коллеги видят ситуацию.

Не забывайте, что здесь, в самом начале проекта, поле для маневра все еще очень широко, и это ваш шанс говорить начистоту. Используйте его.

Как определить риски, требующие тщательной проработки

Соберитесь всей командой (включая клиента) и устройте мозговой штурм по всем рискам, которые могут угрожать вашему проекту. Вы — меч, который клиент занес над проектом, поэтому расскажите клиенту обо всем, что может помешать вам рубить.

Затем, подготовив большой список, в котором будут перечислены все риски, разделите их на две большие категории: требующие и не требующие устранения.

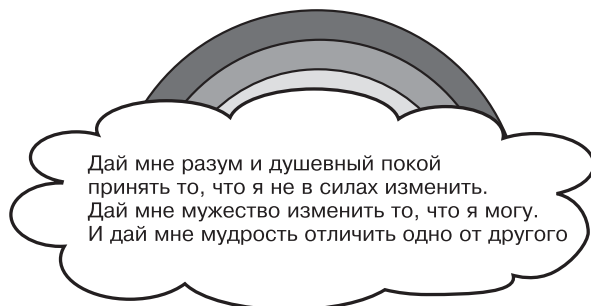
Риски, требующие устранения



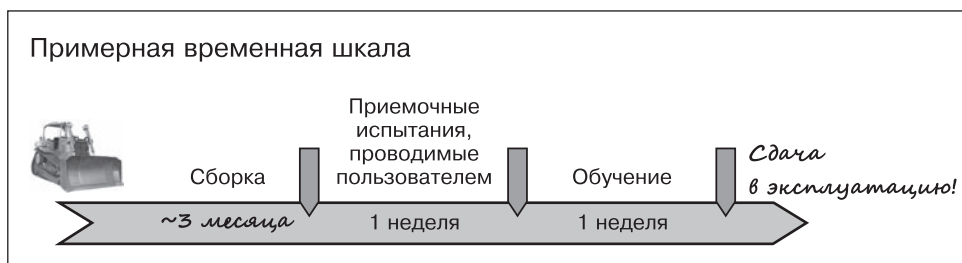
Например, если есть небольшой шанс, что реализация проекта губительно отразится на экономике компании и все останутся без работы, мы практически ничего не можем с этим сделать. Просто не беспокойтесь об этом.

Однако в условиях современного активного рынка труда мы вполне можем потерять ведущего программиста. Поэтому нужно гарантировать, что он будет делиться знаниями с командой и никто не будет излишне специализирован в определенной области.

В моменты, когда вы почувствуете себя сбитым с толку или будете пытаться решить, следует ли специально устранять ту или иную проблему, вам на помощь всегда придет молитва о душевном покое.



5.3. Сколько времени займет реализация проекта



Так мы пытаемся определить, сколько времени потребует запланированный проект — месяц, три или полгода. На данном этапе мы не можем указать сроки точнее, но все же должны дать спонсорам представление о том, когда они могут рассчитывать на готовую программу, даже если это всего лишь приблизительная догадка.

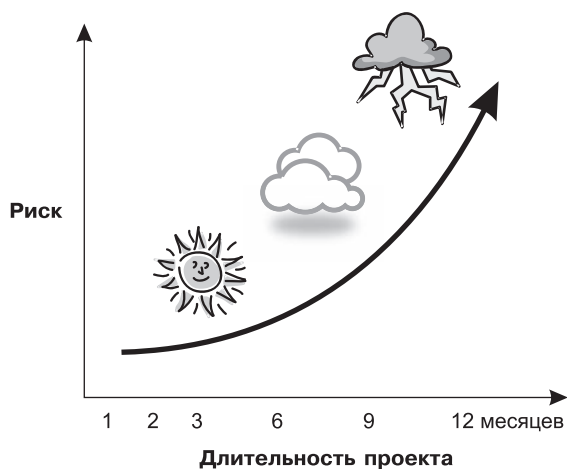
О том, как делается оценка при гибкой разработке, мы подробно поговорим в главе 7. А пока предположим, что команда уже оценила длительность проекта и здесь мы просто представляем результаты работы.

Однако прежде, чем перейти к этому, позвольте рассказать о том, как важно задумываться о мелочах.

Задумывайтесь о мелочах

Возможно, вы не слышали о Рэнди Мотте, но это легендарная фигура в мире Fortune 500¹. Он помог разработать знаменитую на весь мир систему хранения данных и управления запасами сети магазинов Wal-Mart. Она позволяет менеджерам в реальном времени отслеживать различные особенности: например, печенье с каким именно ароматом лучше всего продается в конкретном супермаркете, оборудованном такой системой. Рэнди внедрил подобную систему и в компании Dell, чтобы быстро фиксировать увеличение количества продукции на складе и предлагать скидки на товары, которых накопилось слишком много. Сейчас Рэнди работает директором по информационным технологиям в компании Hewlett-Packard и помогает проводить модернизацию внутренних систем компании, стоящую \$1 млрд.

Несомненно, Рэнди Мотт значительно помог этим компаниям стать такими гигантами рынка, какими они являются сейчас. Но особенно прославил Рэнди один из обнаруженных им секретов. Он утверждает, что ни один цикл разработки не должен занимать более шести месяцев. Чем дольше реализуется проект, тем сильнее риск его провала.



Проблема больших бессрочных проектов заключается в том, что они все время кажутся многообещающими и сравнительно несложными. Всегда думается, что стоит добавить еще одну функцию. И пока расходы не дости-

¹ Список 500 крупнейших промышленных корпораций США. — Примеч. пер.

гают угрожающих масштабов, оценкой никто не занимается, и проект рушится под собственным невыносимым весом.

Наиболее целесообразным сроком выполнения IT-проектов Рэнди считает полгода или меньше. Это не означает, что любая IT-инициатива, которую он начинал, могла быть реализована за шесть месяцев. Он просто не раз обжигался на этом и хорошо знает, что, если требуется создать что-то действительно большое, проект необходимо разбить на более мелкие детали, которыми удобнее управлять.

Рэнди Мотт и гибкая разработка приходят к одинаковому выводу при определении длительности IT-проектов: чем быстрее, тем лучше, предпочтительно — полгода или меньше.

Перспективы длительности проекта

Когда мы задаем временные рамки, то обычно принимаем во внимание сделанные оценки и представляем владельцу примерный план. Необходимо учитывать время на приемочные испытания, проводимые пользователем, обучение и все остальные этапы, которые требуется пройти перед запуском проекта.

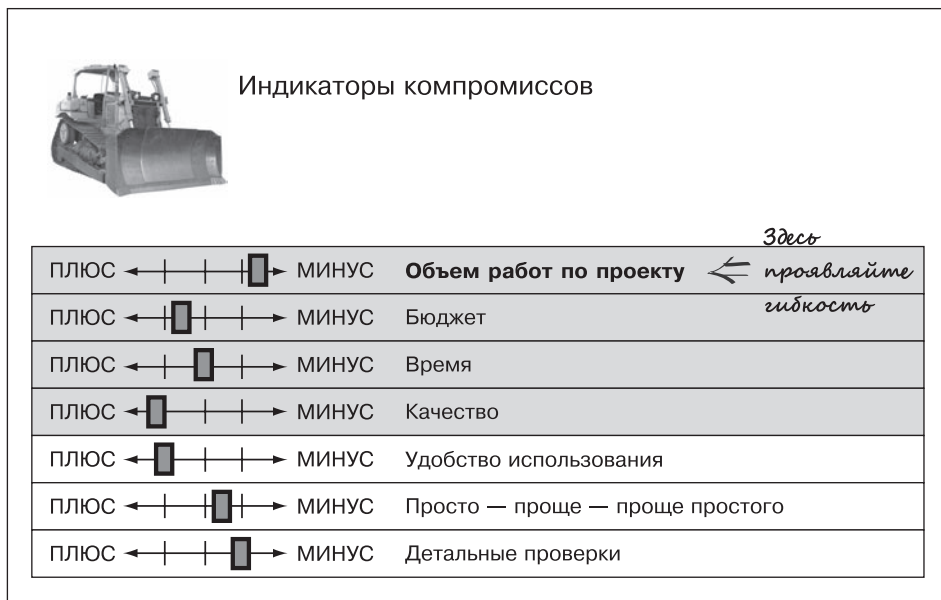
Но в действительности ваша работа на этом этапе сводится к тому, что вы представляете клиенту оптимальное предположение о том, как долго может продлиться проект и можно ли реализовать его в разумный период времени.

Есть несколько вариантов представления своего плана. Можно определить цель и сказать, что вы собираетесь выполнить проект в заданный срок. Или же пообещать реализовать основной набор возможностей, а с деталями определиться по ходу дела.

Разницу между этими вариантами мы рассмотрим, когда будем говорить о таком выборе в разделе 8.4.

Замечание: ни при каких условиях не позволяйте клиенту думать, что ваши планы — это жесткие обещания. Таких обещаний вы не даете. Это просто непроверенные примерные оценки, которые можно подтвердить или опровергнуть, лишь выполнив часть работы и измерив, сколько времени на это ушло, а затем скорректировать план в соответствии с полученными данными.

5.4. Чем можно пожертвовать



При реализации проектов действуют определенные законы и силы, которые необходимо учитывать.

Обычно бюджет и сроки являются фиксированными. Объем работ по проекту, кажется, постоянно безудержно растет, а качество всегда должно быть наивысшим.

Но часто эти силы вступают в конфликт. Чтобы стимулировать одно из направлений работы, требуется ослабить напор на других фронтах. Если проект слишком долго останется несбалансированным, одна из сил может возобладать над проектом и привести его к краху.

Чем-то придется жертвовать. Вопрос — чем?

В гибкой методологии существуют способы сдерживания этих неукротимых и опасных сил, и я перепоручаю вас Мастеру-сэнсэю, который покажет, как это делается.

Вместе с ним вы изучите, какие силы оказывают влияние на проекты, на какие компромиссы они заставляют пойти и как эти силы можно обратить на пользу проекта.

Испытание

1. Какие факторы наиболее важны при реализации софтверного проекта?
 - А. Качество.
 - Б. Время.
 - В. Объем работ.
 - Г. Бюджет.
2. Если требуется сделать слишком много, а времени на это недостаточно, то как лучше поступить?
 - А. Уменьшить объем работ.
 - Б. Привлечь к проекту дополнительных работников.
 - В. Отодвинуть дату завершения проекта.
 - Г. Пожертвовать качеством.
3. Что больше всего?
 - А. Идти по огню.
 - Б. Жевать битое стекло.
 - В. Танцевать «Макарену».
 - Г. Запрашивать у спонсора дополнительное финансирование.

Как бы вы ответили на эти вопросы?

Ловите ли вы себя на мысли «Смотря по обстоятельствам»?

На эти вопросы нет абсолютно правильных и неправильных ответов. Они просто призваны показать, что при выполнении проекта действуют определенные силы и, чтобы найти между ними необходимый баланс, требуется поработать.

Давайте научимся иметь дело с этими силами и укрощать их. Сейчас вы узнаете о секретах... Неистойой четверки.

Неистовая четверка



С незапамятных времен все проекты связываются воедино и управляются четырьмя переплетенными и взаимосвязанными силами. Назовем их Неистойой четверкой. Это время, бюджет, качество и объем работ по проекту.

Они сопровождают нас в каждом проекте, все время привнося в работу хаос и беды:

- ☐ расписание срывается;
- ☐ бюджеты урезаются;
- ☐ список багов (ошибок) растет;
- ☐ нам приходится слишком много делать.

Однако как бы ни была сильна Неистовая четверка, эти силы можно укротить. Рассмотрим каждую из них и разберемся, как можно гармонично использовать их в своих проектах.

Время

Время конечно. Его нельзя ни создавать, ни запасать. Нужно просто максимально использовать то время, которое у нас есть.

Вот почему адепты гибкой разработки стремятся разбивать имеющееся время на небольшие фрагменты и в каждый из них решать конкретную часть задач. Разработчик знает, что если постоянно отодвигать дату завершения проекта и задерживать выпуск ценной программы, то уменьшают-

ся дивиденды заказчика, а кроме того, возрастает риск вообще ничего не выпустить. Это самая горькая судьба для любого софтверного проекта.

Итак, при гибкой разработке нужно следить за временем.

Бюджет

Бюджет подобен времени. Он тоже фиксирован, конечен, и, как правило, его не хватает.

Одна из сложных задач, стоящих перед клиентом, — пойти к спонсору и попросить еще денег. Это случается, и такой опыт всегда неприятен.

Чтобы не сталкиваться с подобной ситуацией, будем считать, что бюджет, как и время, фиксирован.

Качество

Некоторые считают, что качеством можно пожертвовать во имя соблюдения сроков. Они неправы. Любой кратковременный выигрыш в скорости, получаемый путем снижения качества, является ложным, и со временем эта иллюзия рассеивается.

Это значит, что качество также имеет фиксированный уровень и должно стремиться к идеалу.

Объем работ

Приходится мириться с тем, что время, бюджет и качество — фиксированные величины, но остается одна сила, которая подчиняется ходу проекта: объем работ (функционал).

Если нужно сделать слишком много, гибкий разработчик сделает меньше. Если реальность не согласуется с планом, гибкий разработчик будет менять планы, а не реальность.

Некоторых моих учеников это напрягало. Многие приходили ко мне в додзё¹, будучи уверенными, что планы являются фиксированными, неизблемыми, окостенелыми, никогда не изменяются и не оптимизируются. Нет ничего более ошибочного, чем думать подобным образом.

Дата может быть фиксированной. Но план — нет.

¹ Зал для занятий восточными единоборствами. — *Примеч. пер.*

И поэтому, оказываясь наедине с противодействующими ему силами, умный разработчик будет фиксировать время, бюджет и качество, но гибко подходить к объему работ по проекту.

Теперь можно поговорить об индикаторах компромиссов.

Индикаторы компромиссов

Индикаторы компромиссов (trade-off sliders) — это полезное средство, которым разработчик может пользоваться при решении вопросов с клиентом, в частности при обсуждении влияния Неистойвой четверки на проект.

Обучение и разработка



Соотношение времени, затрачиваемого на обучение и на разработку, всегда было тем фактором, который мы в ThoughtWorks старались уравновесить. Хотя мы никогда не ставили педагогическую деятельность своей целью, она была очень полезна для наших специалистов по продажам, и именно с этого нам не раз удавалось начать хорошую работу.

Однако обучение — это одно, а разработка — совсем другое.

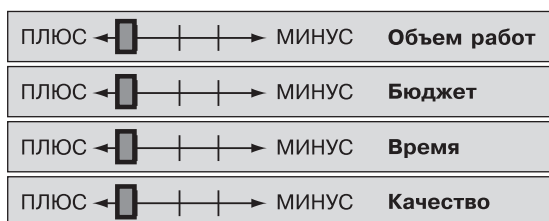
Используя схему скользящих индикаторов (slider board) и предлагая клиенту ранжировать противодействующие силы, мы можем понять, чего именно ожидает клиент, и действовать соответствующим образом.

Например, разработчик захочет понять, как клиент относится к времени, бюджету и качеству. Аналогично разработчик объяснит клиенту, как важно гибко подходить к работе и объему задач и не прикипать сильно ко всем функциям (пользовательским историям) в списке необходимых дел (журнале пожеланий).



Убедитесь, что клиент понимает: вы собираетесь гибко подходить к объему задач

Когда силы описаны и очевидны для всех участников проекта, разработчик просит клиентов оценить эти силы по их относительной важности. Никакие две силы не могут иметь одинаковый уровень приоритета (иными словами, все силы не могут быть первостепенной важности).



Все силы не могут быть первостепенной важности

Большинство клиентов понимают, что в ходе проекта чем-то приходится жертвовать. Если они начнут нервничать, определяя такие неприоритетные детали, напомним, что все элементы важны. То есть, если качество представляется не таким приоритетным, как время, это еще не значит, что качество будет неважно. Мы просто напоминаем, что в любом случае не можем отодвинуть дату выпуска продукта. Следовательно, время имеет более высокий приоритет.



Два фактора не могут иметь одинаковый приоритет




Время и бюджет — это еще не все

Зададимся некоторыми вопросами.

- ☐ Чем хороша компьютерная игра, если в нее неинтересно играть?
- ☐ Будет ли существовать сайт знакомств, если на нем никто не пытается флиртовать?
- ☐ Какой репертуар будет у интернет-радиостанции, если ее никто не слушает?

Неистовая четверка — это еще далеко не все важные факторы. Сознать это не менее важно, чем поддерживать баланс между четырьмя перечисленными выше силами. В наших проектах действуют и другие силы, причем не менее (если не более) важные.

Обязательно перечислите «неприкосновенные» факторы

| | | |
|--|-------|--|
| ПЛЮС ←  | МИНУС | Безумно интересная компьютерная игра |
| ПЛЮС ←  | МИНУС | Снижение нагрузки на колл-центр (20 % трафика) |
| ПЛЮС ←  | МИНУС | Пользовательская функция самообслуживания |

Вещи, от которых зависит судьба проекта

Возможно, эти факторы интуитивно отличались при работе со стартовой колодой. Или на них указывали участники семинаров по сбору пользовательских историй (см. раздел 6.4).

Представляя клиенту индикаторы компромиссов, исходите из тех «неприкосновенных» элементов, от наличия или отсутствия которых напрямую зависит судьба вашего проекта.

Лишь заостряя на них внимание и выставляя их на всеобщее обозрение, вы убеждаете клиента: «Я действительно понимаю, что наиболее важно в данной работе».

Вот вы и справились. Давайте посмотрим, как собрать все вместе в готовый план и рассказать нашим спонсорам, что предстоит сделать.

5.5. Определение средств, необходимых для достижения результата



Вы почти у цели!

У вас есть видение проблемы.

У вас есть план.

Теперь давайте сформулируем, что необходимо сделать и сколько это будет стоить.

В этом разделе вы представите все детали своим спонсорам, то есть поговорите с ними о команде, о плане и о том, сколько потребуется денег.

Начнем с команды.

Подбор команды

На данном этапе вы уже вполне представляете, какая команда требуется для выполнения задачи. Остается просто это озвучить.

| Количество | Роль | Умения/ожидания |
|------------|---|---|
| 1 | Разработчик уровня взаимодействия с пользователем | Умение быстро проектировать (бумажные прототипы). Каркасные представления и имитационные модели, разработка пользовательской навигации. Знание HTML/CSS — плюс |
| 1 | Проект-менеджер | Умение работать в неопределенных условиях, без непосредственного руководства и контроля |
| 3 | Разработчики | Опыт работы с C#, ASP.NET, MVC. Функциональное тестирование, разработка на основе тестирования, рефакторинг, непрерывная интеграция |
| 1 | Аналитик | Уверенные навыки динамического своевременного анализа. Разработка карточек с историями в стиле экстремального программирования. Желание участвовать в тестировании |
| 1 | Клиент | Доступность в течение одного часа в рабочий день для выяснения вопросов. Возможность раз в неделю встречаться с командой для обмена мнениями. Умение направлять, руководить и принимать решения о проекте |
| 1 | Тестировщик | Опыт автоматизированного тестирования. Способность к продуктивному взаимодействию с разработчиками и клиентом. Хорошие навыки исследовательского тестирования |

На этом этапе самое время поговорить о ролях и разделении ответственности (см. главу 2) и определить, какие ожидания возлагаются на всех участников данного проекта.

Роль, о которой я обычно рассказываю немного дольше, чем об остальных, — клиент. Во-первых, она исключительно важна, и во-вторых, эта роль не укоренилась в структуре большинства компаний. Здесь я предпочитаю посмотреть клиенту в глаза и убедиться, что он понимает, за что берется, решая запустить гибкий проект.

Сможет ли клиент уделить этому время?

Уполномочен ли клиент принимать необходимые решения?

Желает ли клиент направлять развитие проекта и руководить им?

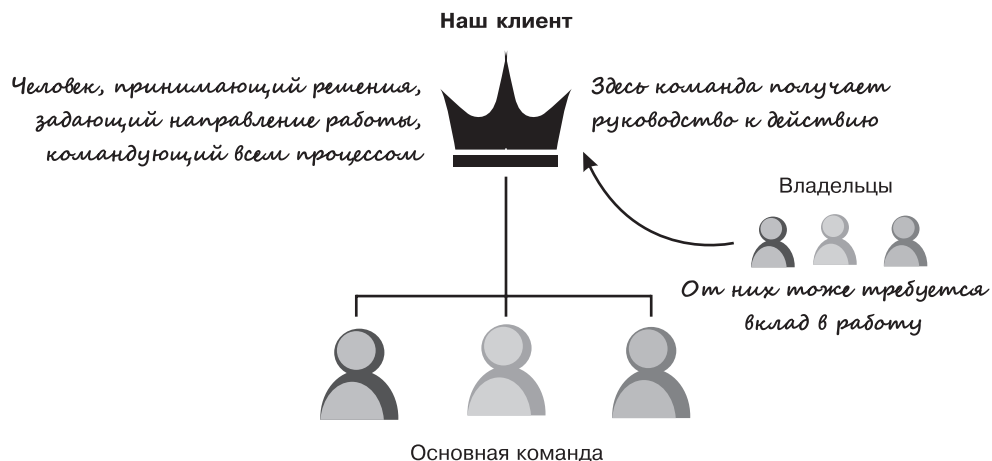
Разработчикам, тестировщикам и аналитикам обычно удастся разобраться с их новыми ролями. Но клиент гибкого проекта — для многих необычная роль, поэтому на ней стоит остановиться отдельно.

Еще одна деталь, которую требуется выяснить заранее (особенно если у проекта несколько владельцев), — кто командует.

Определение того, кто командует

Ничто не вносит в работу команды такой неразберихи, как незнание того, кто руководит работой.

Директор IT-отдела хочет опробовать новейшую технологию. Вице-президент по стратегии желает быть первым на рынке. Вице-президент по продажам просто открыто пообещал, что новая версия выйдет во втором квартале.



Нельзя, чтобы несколько владельцев по-разному представляли себе, в каком направлении должна идти работа, каковы приоритеты и над чем следует работать далее.

Напротив, нужно с самого начала четко определить, кто руководит проектом. Это совсем не означает, что другие владельцы будут лишены права голоса. Просто клиенты не должны противоречить друг другу при общении с командой.

Ставя этот вопрос сейчас, можно избежать значительной путаницы, дорогостоящих доработок и реорганизации на более поздних этапах.

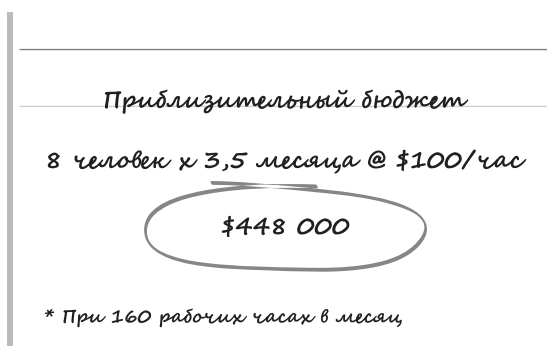
Даже если вы думаете, что знаете, кто командует, все равно задайте данный вопрос. Это не только позволит устранить всяческие сомнения, но и окончательно объяснит команде и всем владельцам, за кем на самом деле будет последнее слово.

А теперь поговорим о деньгах.

Оценка стоимости работ

Возможно, вам так и не придется говорить о деньгах в ходе реализации проекта. Бюджет уже может быть установлен, и вам просто скажут, сколько средств есть в распоряжении.

Если требуется составить приблизительный бюджет проекта, есть простой и быстрый способ обрисовать примерные цифры.



Просто умножьте количество членов команды на ориентировочную длительность проекта, с реальной процентной ставкой, и — вуаля — у вас есть бюджет.

Разумеется, вас ждут определенные расходы на программное обеспечение, и у компании может быть особая бухгалтерия для этих целей. Но, практически несомненно, основная ценность вашего проекта — специалисты, которые ходят на двух ногах и умеют работать с компьютером.

Теперь все суммируем и поможем клиенту принять решение — начинать или не начинать проект.

Все вместе

Данный этап презентации больше всего заинтересует владельцев, так как здесь остаются всего два вопроса, на которые действительно нужно ответить.

- ☐ Когда все будет готово?
- ☐ Сколько это будет стоить?

Строго говоря, мы не можем в данный момент ответить на них со стопроцентной точностью. Но мы неплохо подготовились и уже решили ряд фундаментальных вопросов. Просто на данном этапе у нас по-прежнему много неизвестных (например, как быстро сможет работать команда), чтобы считать любые цифры чем-то, кроме оптимистичных прогнозов.

Один из способов — представить цифры на слайде, как мы поступили в начале главы. Если мы имеем дело с целой программой проектов или чем-то более масштабным, то включим воображение и зададимся вопросом: «Что бы я хотел видеть в итоге, если бы это были мои деньги, если бы я был главным боссом и должен был решить, стоит ли овчинка выделки?»

Завершение формирования стартовой колоды

Поздравляю! Вы это сделали! Вы только что преодолели первый важнейший этап — определили цель, сплотили собранную команду и запустили собственный гибкий проект.

Просто посмотрите на картинку и истории, которые совместно сформулировали вы, ваша команда, ваш спонсор и ваш клиент. Вместе вы смогли узнать следующее:

- ☐ что мы будем делать и зачем;
- ☐ что выдающегося в нашем проекте;

- ❑ какие большие преграды нам придется преодолеть;
- ❑ кто работает вместе с нами;
- ❑ как будет выглядеть наше решение;
- ❑ с какими основными проблемами и рисками нам придется столкнуться;
- ❑ насколько велик реализуемый проект;
- ❑ какие участки работы допускают гибкий подход;
- ❑ приблизительное количество времени и денег, которые нам понадобятся.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

МАСТЕР: Расскажи мне, что ты уже усвоил о стартовой колоде.

УЧЕНИК: Теперь я понимаю, как важно задавать сложные вопросы и приходить к общему мнению еще до того, как проект начнется.

МАСТЕР: Очень хорошо. Что еще?

УЧЕНИК: Я понял, что формулирование проекта не обязательно требует нескольких месяцев анализа требований и планирования. Работая со стартовой колодой, мы можем анализировать требования и формулировать перспективы быстро, обычно на это уходит всего пара дней.

МАСТЕР: А что особенно важно в духе, объеме или назначении изменений, вносимых в проект? Что делается после них?

УЧЕНИК: Состав колоды уточняется. При каждом изменении колода прорабатывается заново, и мы убеждаемся, что общее мнение и единое понимание целей проекта сохранилось.

МАСТЕР: Очень хорошо. Мы можем продолжать путь.

Что дальше?

Определив *цели* проекта, давайте еще раз проработаем некоторые детали, кратко описанные в этой главе.

При гибком планировании следует периодически пересматривать все элементы плана. Оценка, списки пользовательских историй, скорость работы команды — обо всем этом мы поговорим.

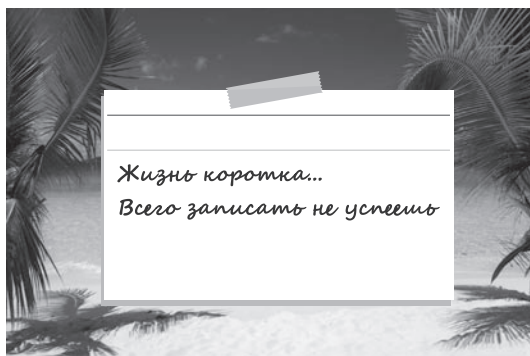
И первым делом рассмотрим предмет, с которого начинаются все гибкие проекты: обычная пользовательская история.

Часть III

Планирование проекта при гибкой разработке

Глава 6

Сбор пользовательских историй



В данной части мы познакомимся с основами планирования при гибкой разработке, поговорим о пользовательских историях, оценках и адаптивном планировании.

Научившись собирать требования в форме пользовательских историй, вы увидите, как удастся всегда выдерживать гибкие планы, почему они содержат только самую актуальную и важную информацию и обходятся без величайшего побочного продукта, обычного для большинства отраслей промышленности, — опережающего инженерного анализа.

Для начала рассмотрим, как собирать требования и какие проблемы связаны со стремлением все записать.

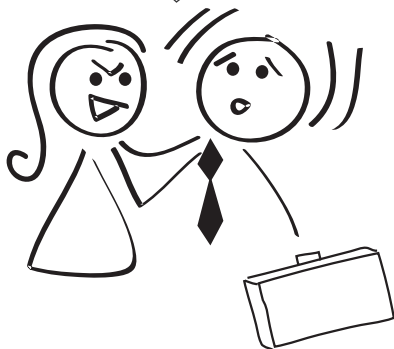
6.1. Проблема с документацией

Подробная документация как средство для всеобъемлющего охвата требований никогда особенно не приживалась в софтверных проектах. Клиенты редко получают то, что требуется. Команды редко формируются так, как надо. Огромное количество времени и сил тратится на обсуждение написанного — вместо того чтобы заниматься действительно нужными вещами.

Есть и другие проблемы, с которыми сталкиваются команды, чрезмерно полагающиеся на документацию при выполнении требований к программе.

Они не умеют справляться с изменениями

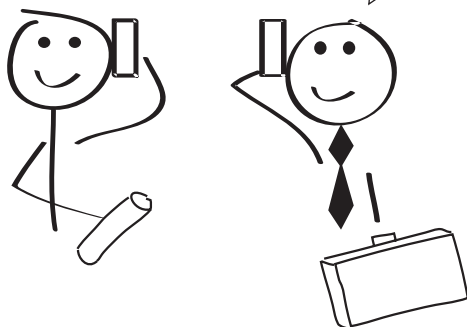
Я помню, о чем говорил,
но это было полгода назад!!!



Они пишут программу в соответствии со спецификацией,
а не с пользовательскими историями

Классная новость!
Я только что понял,
чего хотят клиенты.

Супер! Мы только что
закончили спецификацию,
сможешь с ней увязать?





А если просто написать побольше документации?

Проблема сбора требований в виде документации заключается не только в большом объеме этой документации, но и в сложности обмена информацией.

Во-первых, с документом (даже с самым интересным) нельзя поговорить. Во-вторых, очень легко неправильно понять то, что написал другой человек.

Я не говорил, что она брала деньги.

Я не говорил, что она брала деньги.

Я не **говорил**, что она брала деньги.

Я не говорил, что **она** брала деньги.

Я не говорил, что она **брала** деньги.

Я не говорил, что она брала **деньги**.

Я этого не говорил.

Я говорил что-то другое...

Но их мог взять кто-то другой!

Она просто их потратила.

Она просто похитила мое сердце и уехала в Сан-Франциско.

Слова неоднозначны!

На самом ли деле требование является требованием?

Адепты гибкой методологии не верят в существование требований. Понятие «требование» совершенно неверное, о чем в своей книге *Extreme Programming Explained: Embrace Change* [Вес00] рассказывает Кент Бек, один из величайших специалистов по гибкой разработке.

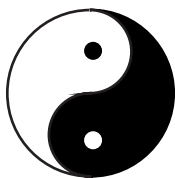
«Разработка ПО пошла в ложном направлении из-за слова “требование”. Требование — это нечто обязательное. Слово имеет коннотацию абсолютности и неизыблемости, а это мешает учитывать изменения. И слово “требование” просто совершенно неверное».

«Из тысяч страниц, на которых описываются требования, можно выбрать 5, 10 или 20 % полезной информации, которая позволит понять всю практическую пользу, которую должна обеспечивать система. Хорошо, а что насчет остальных 80 %? Это не требования, так как данная информация не является необходимыми условиями».

Помню, как Мартин Фаулер жаловался, что, даже потратив несколько лет на написание книги, он поражался тому, как часто люди не понимали того, что он хотел сказать.

В наихудших случаях грамматические ошибки могут обходиться компаниям в миллионы долларов¹. Но обычно письменные требования просто плохо описывают и заключают в себе то, что клиенты хотели бы видеть в своей программе.

Из этого вытекает один из важнейших принципов гибкой разработки.

**Принцип гибкой разработки**

Наиболее эффективный метод сообщения информации команде разработчиков и обмена ею внутри команды — это разговор лицом к лицу.

Итак, нам необходимо средство, которое позволило бы открыто обсуждать требования и схватывать суть желаний клиента. Это средство должно

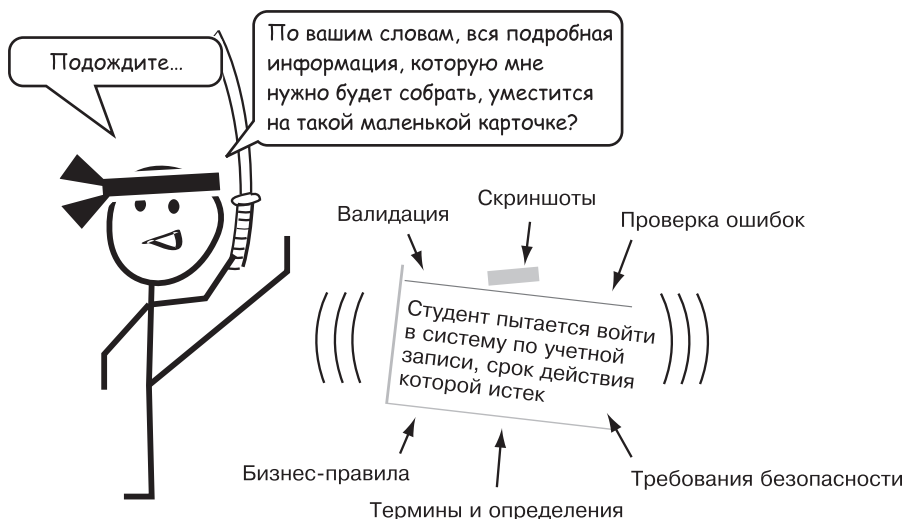
¹ <http://www.nytimes.com/2006/10/25/business/worldbusiness/25comma.html>.

обеспечивать сжатое, но достаточно подробное планирование, с помощью которого мы могли бы не забывать, о чем говорим.

6.2. Знакомство с пользовательскими историями

В гибкой разработке под пользовательскими историями (или пользовательскими пожеланиями) понимаются краткие описания функций, которые, по мнению пользователя, должны рано или поздно появиться в программе. Обычно истории записываются на небольших карточках, как в каталоге (как бы напоминающих нам, что не нужно пытаться записать абсолютно все). Такие карточки стимулируют нас собраться и продолжить разговор с клиентом.

Впервые столкнувшись с пользовательской историей (обычной для гибкой разработки), вы, возможно, захотите спросить: «В чем же суть?» Но не спешите — суть просто не сразу заметна.



Другими словами, все требованиям не обязательно должны уместиться на карточке.

Гибкая разработка стимулирует команды работать с карточками (небольшими рецептами), напоминающими, что изначальная цель сбора требований не сводится к выяснению всех деталей. Нужно просто записать несколько ключевых слов, передающих суть функции, и отложить карточку до тех пор, пока она не пригодится.

Почему же нужно записывать всего несколько слов, а не основательно приниматься за полное требование? Дело в том, что пока мы не знаем, как

именно будет реализована конкретная функция, и даже не уверены, что она нам понадобится! Возможно, до этой функции дело не дойдет и в течение нескольких месяцев. А к тому времени, как мы к ней приступим, наша программа, да и весь мир значительно изменятся.

Итак, чтобы сэкономить силы и время, которые потребовались бы на полную реализацию этой функции сейчас и на полную организацию этой же функции в будущем, мы обратимся к проработке мелких деталей позже (подробнее — в разделе 9.4).

То есть пользовательскую историю можно воспринимать как приглашение к беседе. Наступит время, и мы проработаем эту историю в деталях — но только тогда, когда будем уверены, что данная функция нам обязательно требуется.

6.3. Элементы хороших пользовательских историй

Первый полезный элемент пользовательской истории — ее суть. Что считать полезным? То, за что клиент готов заплатить.

Например, в какой ресторан клиент пойдет ужинать?

Техническая закусочная «У Эрни»

| | |
|--|--------|
| C++ Система будет написана на C++ | 3 дня |
| Объединенное подключение Все подключения к базе данных будут обрабатываться в ее пуле соединений | 2 дня |
| Паттерн «модель — вид — представление» (MVP) Система будет отделять логику представления от бизнес-логики | 5 дней |

Ну хорошо...
Не так уж я и голоден.



Бизнес-блинная «У Сэма»

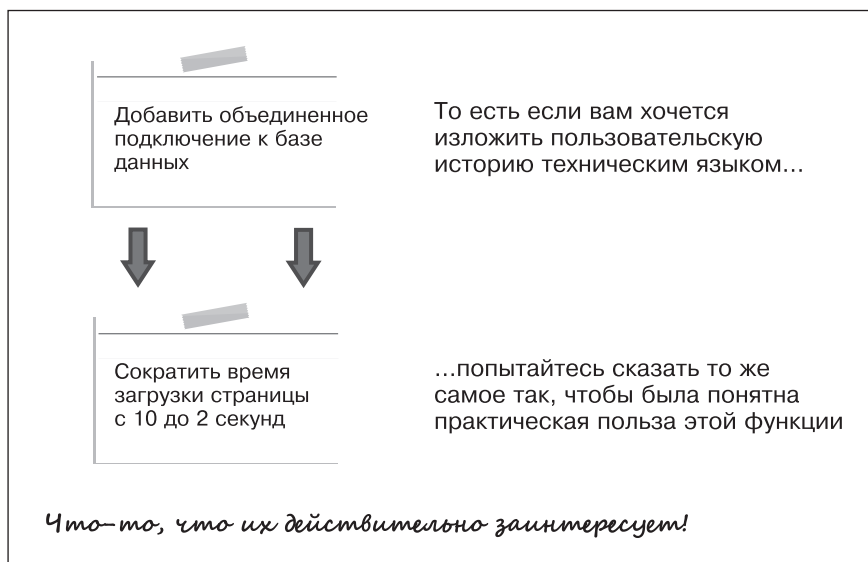
| | |
|---|--------|
| Создание пользовательской учетной записи Пользователи будут входить в вашу систему через персональные учетные записи | 3 дня |
| Уведомление пользователей о появлении на рынке новой недвижимости При появлении на рынке жилья новых предложений пользователи будут получать соответствующие уведомления | 2 дня |
| Отказ от рассылки В каждом письме будет предоставляться возможность отписаться от рассылки | 1 день |

М-м-м-м!
А можно добавки?



Пользовательские истории имеют смысл для бизнеса. Вот почему мы всегда пытаемся формулировать их просто, чтобы пользователь все понимал и не углублялся в технические дебри.

Это не означает, что при создании наших систем мы не можем использовать объединенное подключение или образцы проектирования. Это означает, что такие детали нужно излагать словами, понятными клиенту.



Вторая характеристика по-настоящему хорошей пользовательской истории заключается в том, что она описывает систему на всех уровнях, так сказать, «прорезает программу по всей толщине».



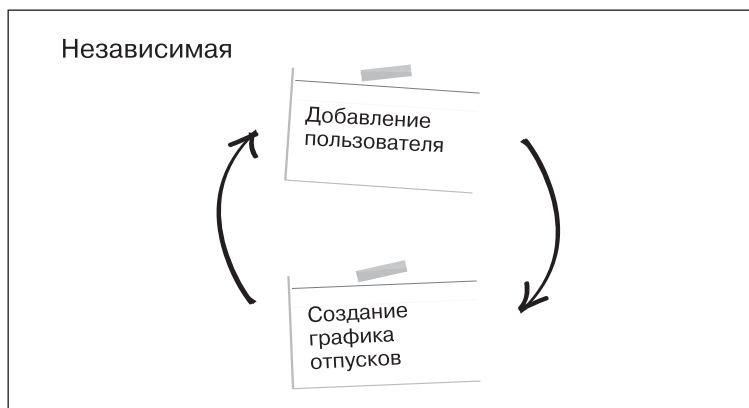
Пользовательский интерфейс (HTML, Ajax, CSS)
Средний уровень (C#, Java, Python)
Слой данных (Oracle, SQL Server)

М-м-м-м... тортик! Ням-ням-ням!

Большинству сладкоежек не понравится, если от торта им достанется только бисквит, без крема. Так и клиенту не хочется получать всего половину или треть решения. Вот почему хорошая пользовательская история явля-

ется комплексной, затрагивает все уровни архитектуры приложения и действительно имеет ценность.

Кроме того, хорошие пользовательские истории имеют следующие характеристики:



При реализации проектов обстановка меняется. То, что на прошлой неделе казалось очень важным, на этой неделе становится второстепенным. Если все пользовательские истории тесно взаимосвязаны и зависят друг от друга, компромиссы становятся довольно сложным делом.

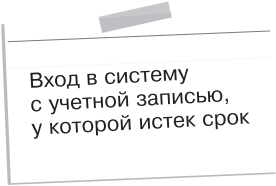
Такое разделение не всегда удастся (прежде чем создавать отчеты, нужно написать программу), но послышное препарирование историй и собирание пожеланий по конкретной функции позволяет считать абсолютное большинство историй независимыми и при необходимости корректировать функционал приложения.



Любую отдельно взятую историю всегда можно реализовать несколькими способами. Мы можем выполнить любую возможность в версиях, условно соответствующих «Форду-Фокус», «Хонде-Аккорд» и «Порше-911».

Истории, которые клиент готов обсуждать, хороши потому, что предоставляют нам поле для маневра, которое иногда очень требуется, чтобы уложиться в бюджет. Иногда обязательно нужен «Порше». В других случаях можно обойтись более непритязательным «Фордом».

Тестируемая




- Разрешение обычного входа в систему.
- Переадресация попыток входа с истекших учетных записей.
- Вывод соответствующего сообщения об ошибке.
- Работа с несуществующими пользовательскими учетными записями

Хорошо, если пользовательские истории поддаются тестированию (в противном случае их называют нетестируемыми (detestable)). Ведь мы должны знать, когда функция работает, а когда — нет. При написании тестов на основании пользовательских пожеланий мы даем ориентиры нашей команде разработчиков и помогаем понять, когда определенная функция готова.

Небольшая и поддающаяся оценке

Думаете, парни справятся с задачей за неделю?



И, если говорить о готовых функциях, как гарантировать, что конкретную историю удастся реализовать за отведенное нам время? Чтобы сроки соблюдались, истории должны быть краткими (каждая — не более чем на пять

дней работы). Так с ними будет легко работать в течение недельных или двухнедельных итераций, и наши оценки будут более уверенными.

Билл Уэйк придумал удобную аббревиатуру INVEST (Independent, Negotiable, Valuable, Estimatable, Small and Testable — независимые, доступные для обсуждения, ценные, поддающиеся оценке, небольшие и удобные для тестирования).

Далее попытаемся обобщить, чем пользовательские истории лучше обычной документации при сборе требований.

Пользовательские истории



- Аккуратные, точные, своевременные.
- Стимулируют очное общение.
- Упрощают планирование.
- Дешево, быстро, просто пишутся.
- Всегда актуальны.
- Основаны на самой последней информации.
- Обеспечивают реакцию клиента в реальном времени.
- Позволяют избежать ложного ощущения точности.
- Обеспечивают командное сотрудничество и инновационный подход

Спецификации

и документированные требования



- Громоздкие, неточные, неактуальные.
- Заставляют гадать (делать ложные предположения).
- Усложняют планирование.
- Дорогие, медленно и сложно пишутся.
- Всегда неактуальны.
- Основаны на скудной информации от пользователя или делаются вообще без такой информации.
- Исключают реагирование клиента в реальном времени.
- Дают ложное ощущение точности.
- Мешают открытому сотрудничеству и инновациям

Но хватит теории. Займемся делом и соберем несколько историй для того парня с предыдущей фотографии, который собрался заняться серфингом.

Добро пожаловать в магазин Дэйва по прозвищу Большая Волна, где можно приобрести все необходимое для серфинга

Эй, ребята! Не можете мне сделать сайт?



Дэйв Большая Волна

Помогать клиентам формулировать пожелания — это нормально

Не понимайте слишком буквально утверждение, содержавшееся в первых книгах по гибкой разработке, о том, что клиенты должны сами писать пользовательские истории. Этот совет верен по сути, но на практике все обстоит немного иначе.

Действительно, клиенты должны предоставлять информацию для пользовательских историй (так как только клиент знает, чего он действительно хочет). Однако на практике большую часть письменной работы придется выполнять именно вам.

Итак, не волнуйтесь, если окажется, что клиенту нужно подсобить. Просто добейтесь активного участия в работе со стороны клиентов и запишите их истории на карточки.

Дэйв несколько месяцев назад собрал команду, которая должна была помочь ему создать сайт, но весь бюджет ушел только на написание требований (а вы думали?), а к созданию самого сайта так и не приступили (черт возьми!). К счастью, Дэйв обратился за помощью к нам.

Давайте определим, что нужно Дэйву

Мы попросили Дэйва перечислить все функции, которые нужно реализовать на его сайте. Ничего сверхъестественного — просто общее описание тех возможностей, которые он хотел бы реализовать на сайте, и этапы, которые, по его мнению, для этого нужно выполнить.

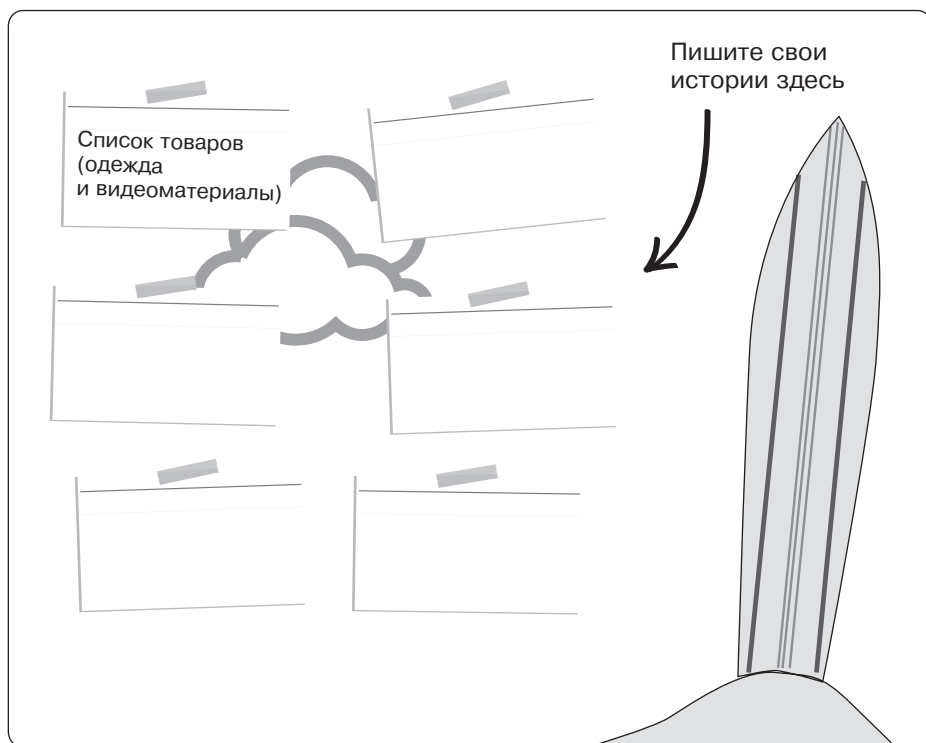
Во-первых, я хочу, чтобы сайт был местом сбора для здешних ребят. Чтобы дети могли заходить сюда и узнавать о планируемых мероприятиях: соревнованиях по серфингу, тренировках и т. п.

Во-вторых, нужно выделить место для торговли атрибутикой. Речь идет о досках, костюмах для подводного плавания, одежде, видео и подобных вещах. Но при этом сайт должен быть прост в использовании и действительно хорошо выглядеть.

В-третьих, хотелось бы установить веб-камеру, которая постоянно была бы направлена на пляж. То есть чтобы не приходилось ходить на берег и проверять, как там погода. Чтобы можно было просто открыть ноутбук, зайти на сайт и посмотреть, стоит ли выходить в море. Это также означает, что сайт должен быстро работать.



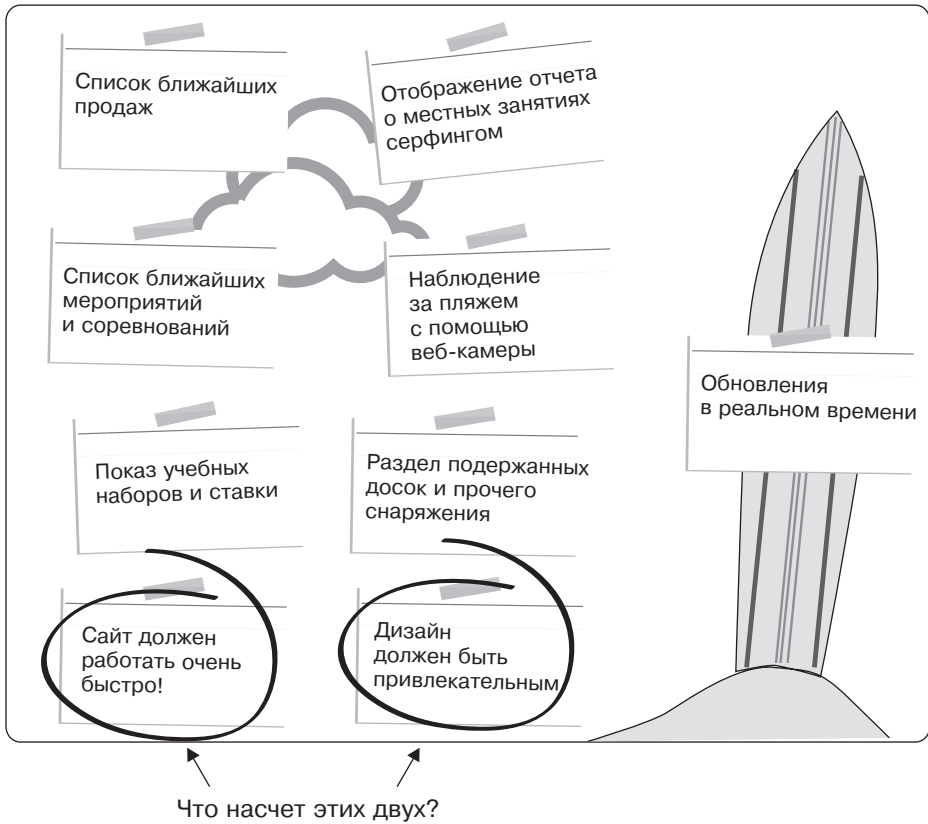
Посмотрим, можно ли извлечь из слов Дэйва о сайте шесть пользовательских историй. Не старайтесь написать идеальную историю. Просто предположите, что клиент хочет видеть в программе именно то, о чем говорит, и уже после этого формулируйте истории.



Теперь сравним все наши истории с критериями INVEST, упомянутыми выше.

Не беспокойтесь о том, что полученные истории несовершенны (они и не бывают совершенными). Просто постарайтесь схватить идею в таком виде, что она будет понятна вашему клиенту и клиент найдет ее ценной. Эту идею нужно записать на карточке.

Предположим, наш первый этап создания пользовательских историй будет выглядеть так:



Что вы думаете о двух последних историях из списка?

- ☐ Сайт должен работать очень быстро!
- ☐ Дизайн должен быть привлекательным.

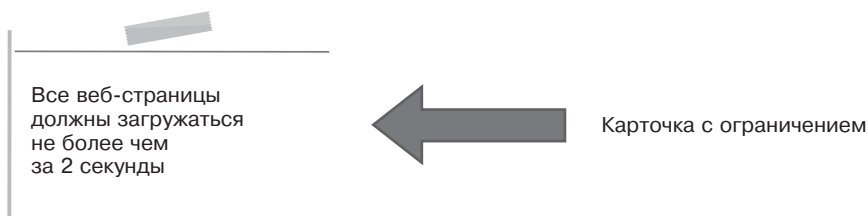
Качественные это истории или нет? И почему?

Если вам кажется, что пожелание «*Сайт должен работать очень быстро!*» — расплывчатое и неточное, то вы правы! «Очень» — это насколько быстро? А как удостовериться, что конкретный посетитель действительно считает дизайн сайта «*привлекательным*»?

Подобные пожелания мы называем *ограничениями* (constraints). Они не похожи на обычные пользовательские истории, реализуемые в течение недели. Но они тем не менее важны, так как описывают характеристики, которые клиент хочет видеть в заказанной программе.

Иногда ограничения удастся преобразовать в теории, поддающиеся проверке.

Например, пожелание «*Сайт должен работать очень быстро!*» можно выразить так:



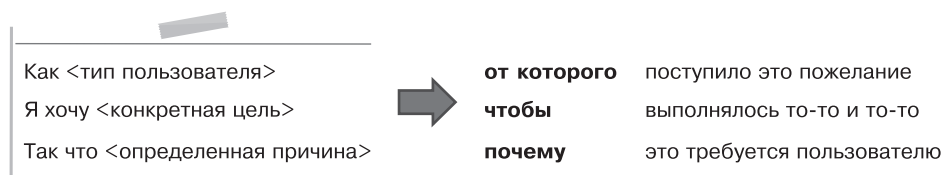
Такое пожелание, безусловно, становится яснее (теперь мы понимаем, что такое «очень быстро»). И, разумеется, такое пожелание уже можно тестировать.

Ограничения важны, но они не являются основной разновидностью пользовательских историй. Запишите их на карточках другого цвета. Убедитесь, что эти ограничения известны всем членам команды, и периодически проверяйте программу на соответствие ограничениям в процессе разработки.

Шаблон пользовательской истории

Нескольких кратких, правильно подобранных слов обычно хватает, чтобы напомнить команде, в чем заключается конкретная пользовательская история. Однако в некоторых командах требуется более широкий контекст.

Если ваша команда — из таких, попробуйте формулировать пользовательские истории по такому шаблону:



Например, с помощью такого шаблона можно придать некоторым историям Дэйва Большой Волны следующий вид:



«Я — серфингист, но люблю поспать. Поэтому мне бы хотелось проверять, подходит ли погода для серфинга, с помощью веб-камеры. Таким образом, я смогу не вылезать из кровати, если нет волны».

«Я — совершенно сухопутный канадский хоккеист. Я хочу взять пару уроков, которые поддадут мне адреналинчика и позволят почувствовать, каково это — балансировать на гребне волны».

«Я — начинающий, и мне нужен самый новый костюм для серфинга. Еще я хочу посмотреть все модные шорты для серфинга и другую атрибутику, чтобы летом выглядеть не хуже чемпиона».

Значительное достоинство такого шаблона пользовательских историй заключается в том, что он позволяет ответить на три важных вопроса, связанных с этой историей: «для кого?», «для чего?» и «зачем?». Он немного расширяет контекст и по-настоящему заостряет и фокусирует наше внимание на деле, что, конечно же, очень хорошо.

Серьезный недостаток такого шаблона — обилие лишних слов, из-за которых истории сложнее «раскладывать по полочкам» и улавливать суть пожелания. Некоторые люди любят дополнительный контекст, другие относятся к нему как к досадным помехам.

Попробуйте оба варианта и проверьте, какой вам больше по душе (причем варианты не исключают друг друга). Например, на этапе планирования можно использовать краткую формулировку «добавить пользователя», а на обратной стороне карточки написать развернутую версию этой истории, как в шаблоне. Позже при анализе развернутая версия вам пригодится.

6.4. Как организовать семинар по сбору пользовательских историй

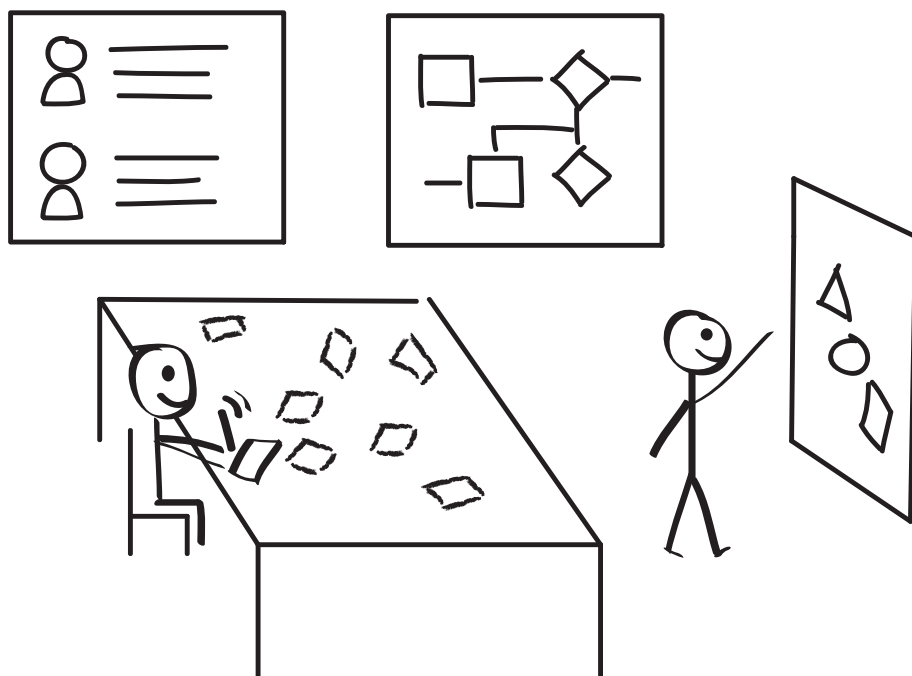
Прежде чем взяться за дело и составить план нашего гибкого проекта, нужно перечислить все функции, которые клиент хочет увидеть в готовой программе. Один из способов решения этой задачи — организовать *семинар по сбору пользовательских историй* (story-gathering workshop). Для клиента и команды разработчиков это отличная возможность собраться вместе и записать пользовательские истории о системе, которую требуется построить.

Цель такого семинара — получить широкое представление о задаче. Мы как будто закидываем невод и пытаемся выудить как можно больше функций. Вы не обязательно будете реализовывать все эти функции. Семинар проводится скорее для того, чтобы все истории были на виду и мы видели общую панораму того, с чем придется работать.

Список того, что мы не собираемся делать (см. раздел 4.4), составленный на этапе создания стартовой колоды, может помочь вам на таком семинаре. Но обычно все сводится к тому, что вы усаживаетесь вместе с клиентом, рисуете несколько картинок и по ходу беседы о системе записываете истории на карточки. Вот и все.

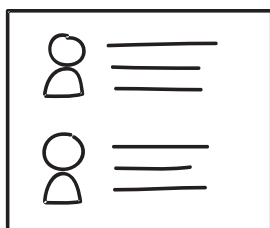
Позвольте дать несколько советов о том, как организовать хороший семинар по сбору пользовательских историй.

Этап 1. Найдите большое открытое помещение

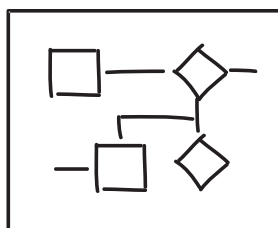


В этой комнате должно быть удобно перемещаться, должна быть возможность приклеивать картинки на стену и собирать карточки на большом столе. Необходимо обеспечить подходящую атмосферу для естественного творческого процесса.

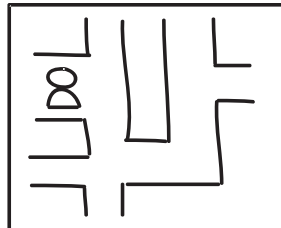
Этап 2. Нарисуйте много картинок



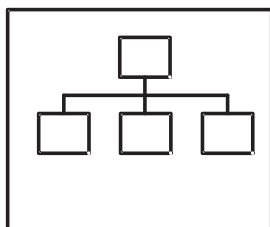
Персоналии



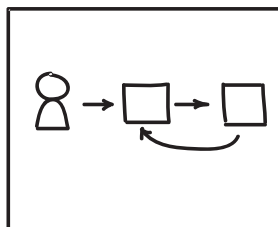
Функциональные схемы



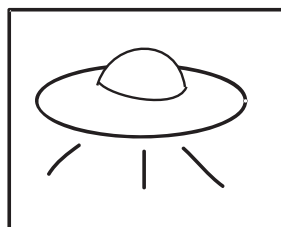
Сценарии



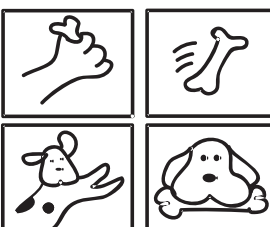
Системные карты



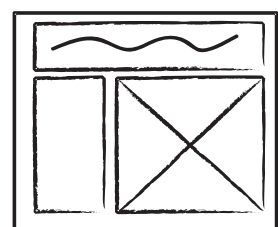
Технологические цепочки



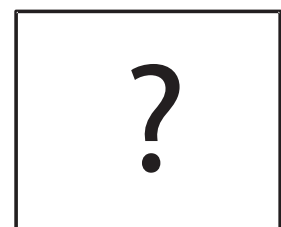
Концептуальные решения



Раскадровка



Бумажные прототипы



Ваша собственная

Картинки отлично помогают обсуждать идеи о системе в режиме мозгового штурма и являются настоящим сокровищем для обнаружения пользовательских историй.

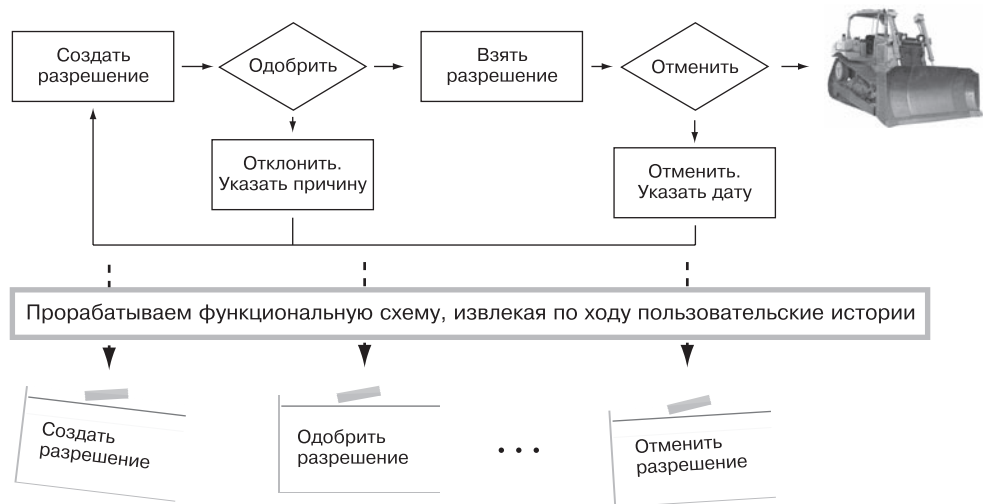
Персоналии (характеристики людей, которые будут работать с вашей системой) хороши для знакомства с клиентами. Функциональные схемы, технологические цепочки и сценарии очень пригодятся при ролевой игре и помогут по-настоящему ощутить, как должна работать система. Системные карты и схемы информационной архитектуры помогают организовать работу и разбить ее на составляющие. А концептуальные решения и бумажные прототипы представляют собой недорогие способы опробовать полученное содержимое и посмотреть, что работает.

Подчеркиваю, здесь нас интересует именно широкая перспектива. Старайтесь не углубляться в детали — картина должна оставаться обобщенной.

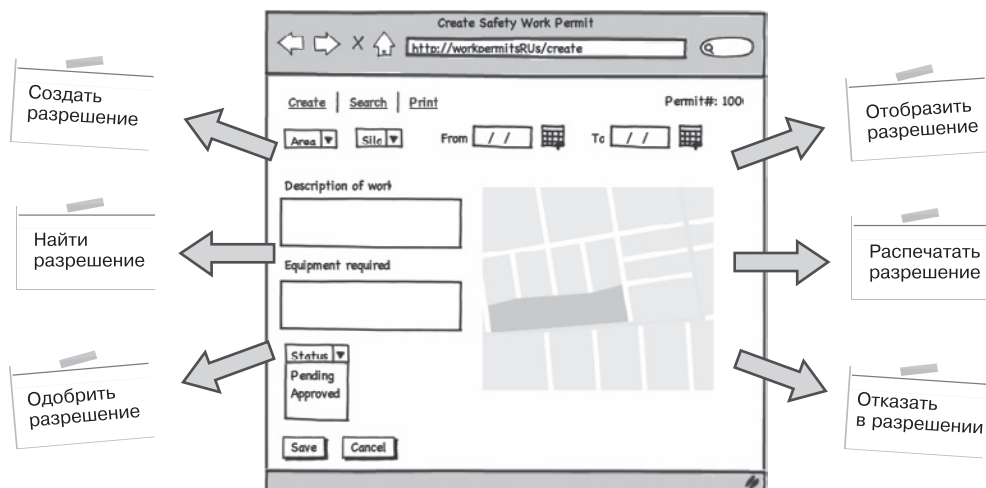
Но когда у вас будет несколько хороших картинок и вы будете понимать, как система должна работать, можете приступать к разработке картинок для конкретных историй.

Этап 3. Напишите много историй

С помощью новых схем и картинок проследите истории вместе с вашим клиентом, рассматривая отдельные пожелания по ходу работы.



Если большая часть вашей программы выводится на экран в одном-двух окнах, то возьмите их скриншоты и выделите на них небольшие функциональные элементы.



Из одной функциональной схемы, подкрепленной несколькими приблизительными бумажными прототипами, обычно можно получить большую часть ключевых историй для проекта.

Выделяя отдельные пользовательские истории, ищите небольшие самостоятельные элементы, касающиеся всех уровней программы (требующие для реализации от одного до пяти дней). Ничего страшного, если некоторые из историй будут сравнительно велики. В гибкой методологии они называются *эпиками* (epics) — это крупные пользовательские истории, на разработку которых уходит одна-две недели.

Эпики удобны при общем планировании, так как в них описываются крупные пожелания, за которые мы, возможно, и возьмемся, но пока уверенности в этом нет. Если у вас есть подобные крупные функциональные фрагменты, то считайте их обычными историями, а когда дело дойдет до разработки, просто разбивайте их на части.

В конечном счете от 10 до 40 пользовательских историй обычно заменяют 3–6 месяцев планирования. Если количество историй измеряется сотнями, то либо ваше планирование зашло слишком далеко, либо вы чрезмерно углубляетесь в детали. На данном этапе мы стремимся именно к ширине (а не к глубине) охвата материала. Поэтому поднимаемся на поверхность и не теряемся в дебрях.

Этап 4. Обсудите остальные вопросы в режиме мозгового штурма

Но как бы хороши ни были картинки, они не могут описать всего, что нам нужно для работы над проектом. В частности, еще нужно обсудить миграцию данных, нагрузочное тестирование, бумажную работу, связанную с недопущением корпоративного и бухгалтерского мошенничества, сопроводительную документацию по продукту, учебные материалы, двухнедельный срок на приемочные испытания, проводимые пользователем, и т. д. Требуется занести все эти детали (и не только) на карточки, расставить их приоритеты и работать с ними наравне с остальными отчетными данными по проекту.

Сейчас самое время вернуться к картинке, рассмотренной в начале раздела 4.5, и в режиме мозгового штурма обсудить все детали, которые требуются, чтобы проект получился максимально успешным.

Если еще остаются дела, с которыми необходимо разобраться (даже не относящиеся непосредственно к программной составляющей), выпишите их на отдельную карточку.

Этап 5. Подчистите список, чтобы он сиял

Как только будет готов первоначальный список, несколько раз просмотрите его и поищите элементы, которые случайно оказались продублированы. Посмотрите, чего не хватает, сгруппируйте логически близкие истории и подготовьте простой и понятный список задач, которые требуется решить. Поздравляю! У вас есть первоначальный план проекта!



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, если очная коммуникация — самый эффективный способ обмена информацией о системе, означает ли это, что я должен тратить больше времени на обсуждение требований с моим клиентом и меньше времени на их запись?*

МАСТЕР: *Да, именно так.*

УЧЕНИК: *А значит ли это, что при сборе требований я должен отказаться от документации?*

МАСТЕР: *Нет. Цель — не избавиться от всей документации, так как документация сама по себе не является ни хорошей, ни плохой. Цель — напоминать себе о том, какой способ обмена информацией наиболее эффективен.*

УЧЕНИК: *То есть при сборе требований какая-то документация все же может присутствовать?*

МАСТЕР: *Конечно. Просто не нужно ставить ее во главу угла. Вместо этого необходимо сконцентрироваться на клиенте и на том, что он хочет*

получить от программы. При описании программы создавай, что у документации есть границы. Пусть она будет вашим последним объяснительным средством. Но не первым.

УЧЕНИК: *Благодарю тебя, Мастер. Я подумаю над этим во время медитации.*

Что дальше?

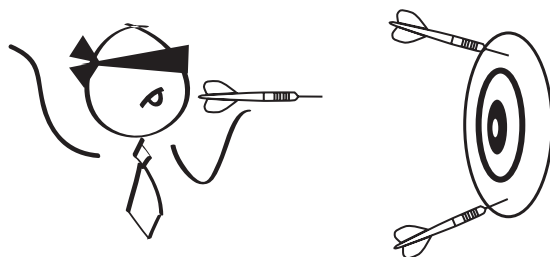
Мы хорошо потрудились, дружище! Теперь мы видим, что пользовательские истории — это просто краткие описания функций, которые клиенты хотят видеть в программе, и вы еще на шаг продвинулись к созданию вашего первого плана гибкой разработки.

Следующая глава посвящена оценке, в ней мы рассмотрим, как придавать историям такие размеры, при которых они смогут выдерживать неизбежные затруднения, возникающие в процессе работы.

Итак, двигаемся вперед, чтобы развеять ореол таинственности, окружающий науку и искусство гибкой оценки.

Глава 7

Оценка: филигранное искусство угадывания



Будьте готовы к тому, что процесс оценки должен в известной степени опираться на реальность. Гибкая разработка в данном контексте разрушает распространенные иллюзии и напоминает, чем на самом деле являются наши изначальные общие оценки — это всего лишь не самые удачные догадки.

Но, обучаясь гибкому оцениванию проекта, вы перестанете требовать от предварительных оценок того, чего они не смогут вам дать (в первую очередь, речь идет о точности), и сосредоточитесь на том, что действительно важно. То есть на выстраивании плана, с которым сможете работать вы и ваш клиент и в который будете верить как он, так и вы.

Эта глава поможет вам научиться гибкой оценке пользовательских историй. Кроме того, мы рассмотрим некоторые эффективные методы группового оценивания, помогающие определять временные рамки тех или иных задач.

7.1. Сложности, связанные с приблизительными оценками

Давайте займемся этой проблемой. Когда речь заходит о формулировании ожиданий, связанных с оценкой софтверных проектов, всегда возникает ряд сложностей.

Точка оценки

«Основная цель оценки при разработке программ — не предсказать результат проекта, а определить, являются ли цели проекта достаточно реалистичными, чтобы их можно было достичь, контролируя проект». — Стив Макконнелл, *Software Estimation: Demystifying the Black Art* [McC06].

Это не означает, что наши оценки обязательно будут ошибочными (хотя почти всегда именно так и случается). Дело скорее в том, чтобы люди не ждали от оценок того, чего они не могут дать в принципе, — точного прогноза.

Джонсон! Мне нужна
подробная оценка нашей...



...еще не специфицированной системы, создание которой запланировано на следующий год, с использованием нашей еще не определенной технологии, с учетом нашей еще не подобранной команды, в нашей еще не уточненной экономической обстановке (конъюнктуре).

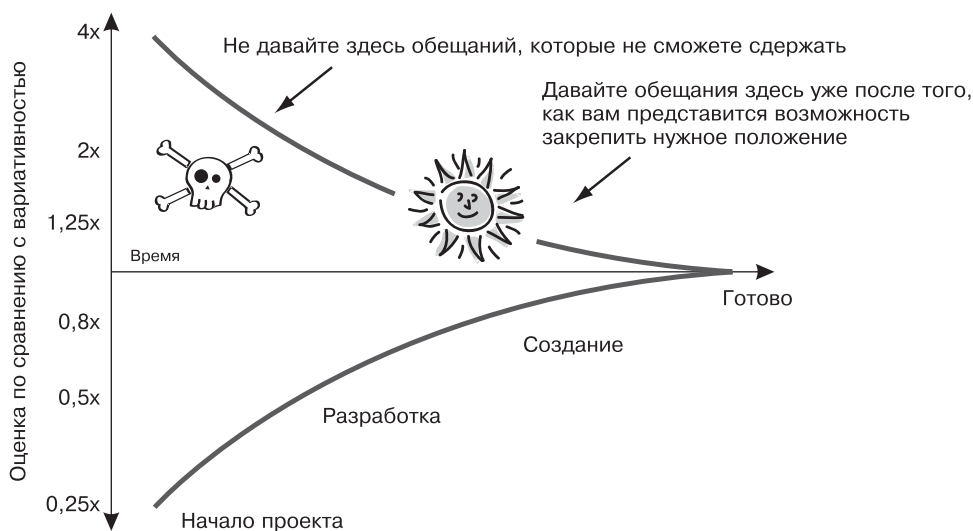
Ситуация такова, что в какой-то момент в ходе работы люди упускают из виду факт, что

ПРИБЛИЗИТЕЛЬНЫЕ ОЦЕНКИ — ЭТО ДОГАДКИ

(ОБЫЧНО СОВЕРШЕННО НЕУДАЧНЫЕ И, КАК ПРАВИЛО, ЧРЕЗМЕРНО ОПТИМИСТИЧНЫЕ)

Причем упускают его именно в тот момент, когда эти преждевременные, неточные, общие оценки недальновидно выдаются за четкие обещания, из-за которых во многих проектах начинаются неприятности.

Стив Макконнелл называет такое непродуктивное поведение *конусом неопределенности* (cone of uncertainty), напоминая о том, что на установочном этапе проекта приблизительные оценки могут варьироваться в диапазоне около 400 %. Настолько сильно могут отличаться наши оценки на различных этапах проекта.



Простая истина заключается в том, что *точные заблаговременные оценки невозможны*, и мы должны прекратить верить в их существование.

Единственный вопрос, ответ на который можно попытаться найти при предварительной оценке, следующий:

СТОИТ ЛИ ВООБЩЕ БРАТЬСЯ ЗА ЭТОТ ПРОЕКТ?!

(С УЧЕТОМ ВРЕМЕНИ И РЕСУРСОВ, КОТОРЫМИ МЫ РАСПОЛАГАЕМ)

Нам нужен способ оценки, который позволил бы решать следующие задачи:

- ☐ планировать на будущее;
- ☐ напоминать нам о том, что наши оценки — это догадки;
- ☐ учитывать сложности, присущие создаваемой программе.

7.2. Как сделать из лимонов лимонад

При гибкой разработке мы признаем, что не следует всецело доверять нашим первичным, обобщенным оценкам. Но мы также понимаем, что в самом начале проекта требуется рассчитать бюджет и обрисовать перспективы.

Чтобы этого добиться, мы поступаем так же, как и любой на нашем месте, кто желает получить правдоподобные оценки. Мы выполняем небольшой фрагмент работы, смотрим, сколько времени на это ушло, и используем полученный результат при дальнейшем планировании. Нам требуются две вещи:

- ❑ *сравнение* пользовательских историй (их размера) друг с другом;
- ❑ *балльная* система отслеживания прогресса.

Давайте рассмотрим каждый из этих аспектов детально и узнаем, как они помогают при планировании.

Относительные размеры

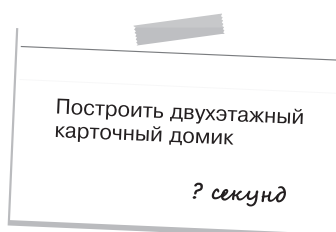
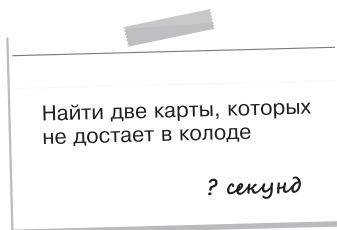
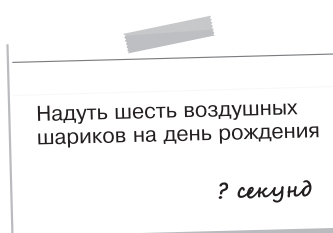
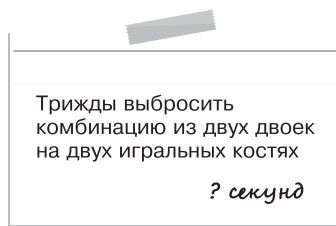
Предположим, нам известно, что одно шоколадное печенье можно съесть за 10 секунд. Перед нами стоит вопрос: сколько времени потребуется, чтобы съесть семь и четырнадцать таких печений (и еще запить стаканом молока)? Какова будет наша оценка?

Если на то, чтобы съесть одно печенье, нужно 10 секунд...

...сколько времени понадобится, чтобы съесть такую кучу?



А теперь предположим, что вас попросят оценить что-то другое — простое, но то, что вам еще не приходилось оценивать. Сколько, по-вашему, понадобится времени на решение следующих простых задач?



Если вы среднестатистический человек, то оценка в случае с печеньем не составит для вас труда (это юмор), а остальные задачи окажутся значительно сложнее.

Если на одно печенье уходит 10 секунд,
то на семь печений — $10 \text{ секунд} \times 7 = 70 \text{ секунд}$,
а на четырнадцать — $10 \text{ секунд} \times 14 = 140 \text{ секунд}$



*Или в два раза больше,
чем на семь*

Разница между двумя упражнениями заключается в том, что с печеньями мы применяем относительную оценку, а с подсчетом карт — абсолютную.

Научные данные свидетельствуют о том, что относительная оценка удастся любому представителю человеческого вида достаточно хорошо. Если мы кладем перед собой два камешка, то с достаточно большой точностью можем угадать, насколько один камень больше другого.

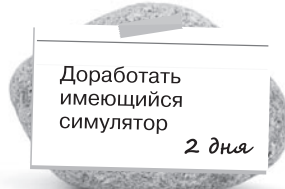
Этот камень



на глаз в два раза больше



этого



Большие проблемы начинаются тогда, когда мы пытаемся сказать, на сколько именно один камень больше другого (то есть дать абсолютную оценку).

Этот простой принцип является одной из основ гибкой оценки и гибкого планирования. Сравнивая пользовательские истории друг с другом и измеряя, насколько быстро мы сможем работать, мы получаем все элементы, необходимые для создания гибкого плана.



Отдельная проблема, связанная с относительной оценкой, заключается в том, что один день в наших оценках не всегда равен одному дню в наших планах. Команда будет работать медленнее или быстрее, чем мы запланировали.

1 ОТНОСИТЕЛЬНЫЙ ДЕНЬ \neq 1 календарный день.

Чтобы учесть это неравенство и избежать необходимости постоянной переоценки всех историй, при гибкой разработке для оценки применяется балльная система.

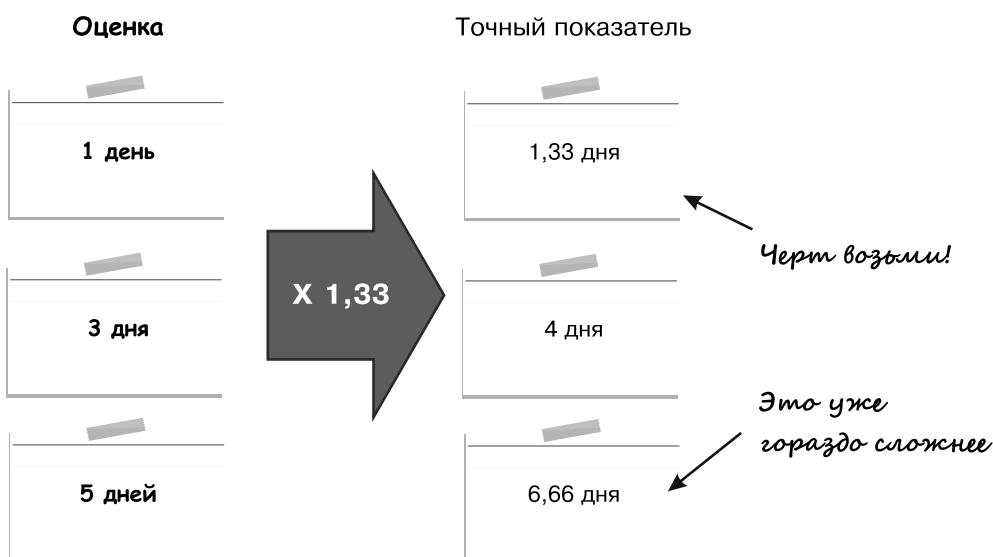
Балльная система оценки

Балльная система помогает отслеживать прогресс и давать относительные оценки без необходимости сравнения текущих обстоятельств с нашими оценками.

Допустим, исходя из первоначальной оценки, мы решили, что на реализацию данной истории должно уйти три дня, тогда как в реальности понадобилось около четырех.



Тогда можно попытаться скорректировать все полученные нами оценки на 33 %.



Но кому понравится работать с такими числами, как 1,33 или 6,66 дня? Такие величины дают не только ложное ощущение точности, но и ставят перед нами следующую проблему: вдруг после реализации еще нескольких историй, мы обнаружим, что наша оценка 1,33 на самом деле ближе к 1,66. Будем оценивать заново?

Чтобы оторваться от такого постоянного и бесконечного уточнения чисел, при гибкой разработке рекомендуется привязывать свои оценки к простой в использовании балльной системе и не соотносить их с обычными календарными сроками.



При работе с балльной системой единицы измерения уже не играют роли. Мы работаем с относительными величинами, а не с абсолютными.



Все, что мы пытаемся сделать, — это выразить *величину* задачи в числовом виде и относительно других задач. Если вам будет удобнее, можете считать гибкую оценку попыткой отсортировать истории по тому же принципу, по которому сортируются майки: маленькие, средние или большие.

Кроме того, наперстывая наши оценочные показатели и стараясь не отставать от них, нужно признать, что в итоге будут важны не оценки, а именно результат. При сравнении историй по размеру друг с другом мы можем переоценивать одни и недооценивать другие, так что в итоге картина выравнивается.

При использовании балльной системы мы достигаем определенных результатов, а исследования показывают, чем она особенно хороша:

- ❑ напоминает нам, что оценки — это всего лишь догадки;
- ❑ измеряет только длительность работы (не затухающую с течением времени);
- ❑ это быстрый и простой способ.



Да, действительно. Прежде чем мы начали заниматься гибкой оценкой, всякий раз, когда речь заходила об оценке, я говорил о *днях*, хотя на самом деле должен был говорить о *баллах*. Я допустил такое разночтение по двум причинам. Во-первых, нам было бы тяжело обсуждать саму концепцию оценки, говоря о баллах. Во-вторых, поскольку в некоторых гибких командах применяется оценка в днях, они просто называют их *идеальными днями*.

Идеальный день — просто еще одна разновидность балла для оценки истории. Под идеальным понимается совершенный рабочий день, когда вы ни на что не прерываетесь и имеете возможность работать восемь часов без всяких помех.

Разумеется, на работе идеальных дней не бывает, но некоторым командам эта концепция кажется полезной.

В принципе, идеальными днями можно оперировать, но лично я предпочитаю баллы. В основном потому, что при оценке в баллах становится очевиден сам факт оценки, а кроме того, при использовании баллов я могу не думать о том, совпадает ли мой идеал с вашим.

Далее в книге считайте понятия «день» и «балл» равнозначными. Далее я буду пользоваться в основном баллами, но иногда будут встречаться и дни.

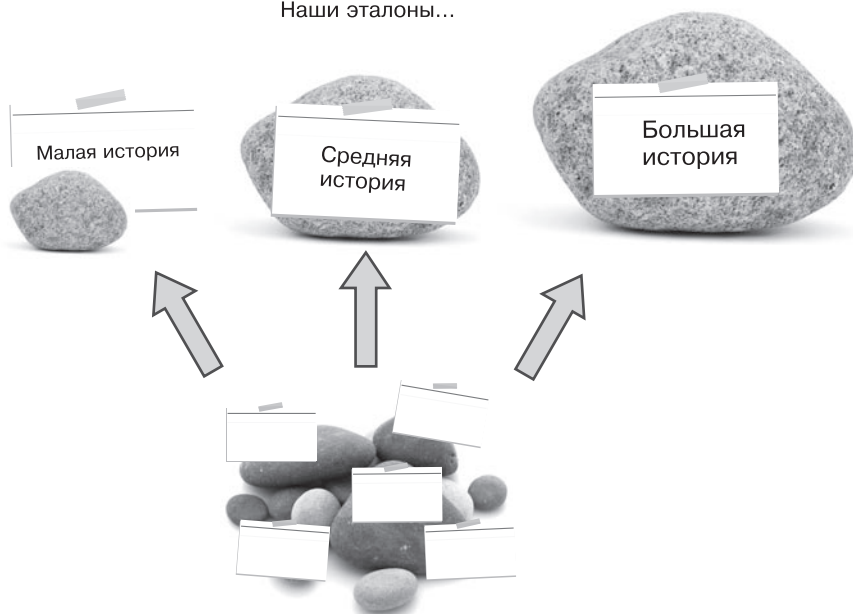
7.3. Как это работает

Но хватит разговоров. Пора возвращаться на грешную землю. Далее мы обсудим два простых метода оценки, которые вы и ваша команда можете использовать для правильного определения размеров историй при гибком планировании.

Триангуляция

Триангуляция — это метод, при котором берется несколько ориентировочных историй, а остальные измеряются по ним, как по эталону.

Наши эталоны...



...с которыми мы сравниваем наши оставшиеся истории

Представим себе, например, местный велосипедный магазин, в котором только что установили новую систему управления запасами (inventory system). Команда уже справилась с «домашней работой» и составила неплохой список пользовательских историй. Где им действительно может понадобиться помощь — так это при оценке.

Велосипедный универсам «У Майка»

Нужен велик? Обратись к Майку!

Настройка
пользовательских
учетных записей

Обеспечение
ежемесячных мероприятий
по стимулированию сбыта

Настройка работы
с карточками
MasterCard

Создание
торгового отчета

Загрузка банковской
информации

Обработка новой
продажи велосипеда

Создание панели
наблюдения
за запасами

Получение учетной
информации
со склада

Эй, ребята! Вы правда думаете,
что вам удастся помочь нам с оценкой
длительности всех этих работ?



Настройка
работы с чеками
American Express

Обработка операций
возврата, обмена
и сдачи старого товара
в счет оплаты нового

Настройка работы
с карточками Visa

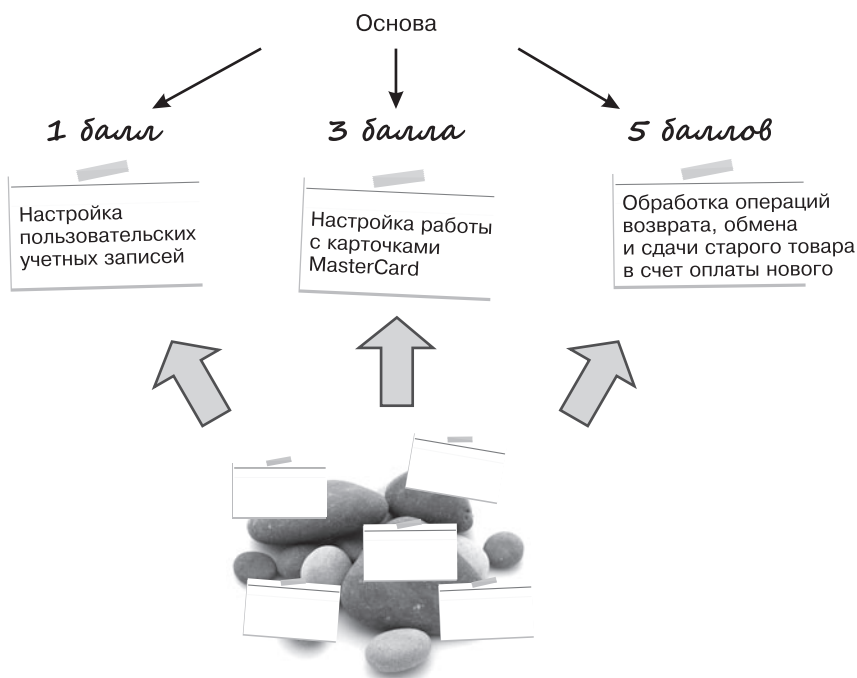
Написание
учебного мануала
для сотрудников

Давайте внимательно рассмотрим список и выясним, есть ли в нем кандидаты на роль хороших эталонных историй. В идеале нам нужна маленькая история, средняя история и достаточно большая история, которая при этом сможет уместиться в одной итерации (то есть на выполнение которой может уйти от одной до двух недель). Кроме того, обратим внимание на следующие моменты:

- ❑ построение логических групп;
- ❑ выделение историй, охватывающих всю систему целиком (это нужно, чтобы содержательно наполнить ее архитектуру);
- ❑ поиск типичных черт, которые характерны для проекта на всех этапах его реализации.

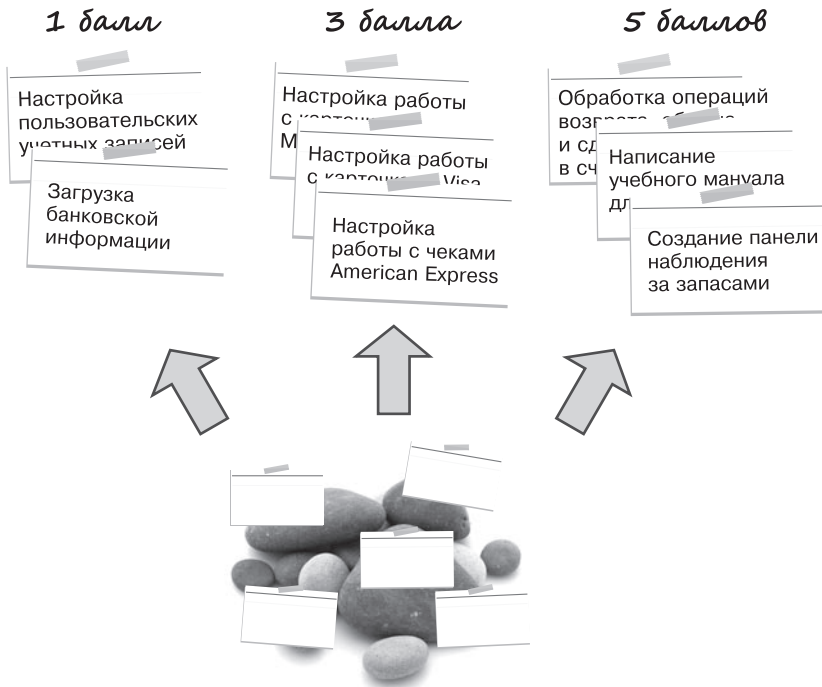
Есть вещи, которые нужно держать в уме при поиске таких историй-кандидатов. Это должны быть самые обычные истории, с которыми обязательно придется столкнуться при работе.

После изучения списка начнем, например, со следующих трех историй.



Оставшиеся истории, длительность которых необходимо оценить

Теперь нам есть с чем сравнивать, поэтому мы можем заняться оставшимися историями и сравнить их с данными вариантами.



Здесь может возникнуть вопрос: нужно ли постоянно оценивать заново имеющиеся истории? Ответ — да. Если вы начнете выстраивать несколько историй и среди них попадутся такие, длительность реализации которых определена неверно, то вы, конечно же, должны заново оценить такие экземпляры, так как иначе вам все время придется пересматривать скорость работы команды. Из-за этого планирование становится несколько неопределенным, потому что на различных запланированных этапах у вас получится разная скорость работы.

Кроме того, если вам придется заниматься работой, с которой вы никогда ранее не сталкивались, и вы не будете знать, как определить необходимое на нее выполнение время, проведите *предварительное исследование* (spike¹). Под ним в гибкой разработке понимается эксперимент, для выполнения которого ставятся жесткие временные рамки. В ходе такого эксперимента мы успеваем провести ровно столько исследований, сколько требуется для формирования оценок, а затем эксперимент останавливается (в данном случае сама история до конца не доводится).

¹ В литературе также встречается перевод этого понятия как «выброс» или «проба». — *Примеч. пер.*

Мудрость толп

В книге Джеймса Суловицки *The Wisdom of Crowds* [Sur05] рассказывается следующая история. В 1906 году британский ученый Фрэнсис Гальтон был шокирован результатом эксперимента, проведенного на сельской ярмарке.

Предположив, что профессиональный мясник сможет точнее угадать вес забитого быка, он был удивлен и разочарован тем, что толпа обывателей (практически не имевших опыта рубки мяса) смогла не просто угадать вес туши, но и сделать это с точностью до фунта.

Данный факт опровергал мнение сэра Фрэнсиса о том, что эксперты всегда правы и им не составляет труда превзойти в мастерстве толпу.

Разыгрывая покер-планирование, мы просто проверяем на прочность мудрость толпы, сравнивая прогнозы людей с нашими оценками. Мы спорим, что группе удастся предложить более точную догадку, чем любому отдельно взятому человеку.

Обычно предварительные исследования продолжаются не дольше пары дней и отлично подходят для того, чтобы гибко оценить что-либо и получить достаточное количество информации, позволяющее сообщить клиенту, сколько примерно будет стоить та или иная работа. И клиент может решить, стоит ли вкладывать деньги в эту работу.

Прежде чем закончить главу, рассмотрим еще один удобный инструмент командной оценки, помогающий прийти к общему мнению, — так называемое *покер-планирование*.

Покер-планирование

Покер-планирование — это игра, в ходе которой члены команды сначала оценивают пользовательские истории индивидуально (при помощи карточек с номерами — например, 1, 3 и 5 баллов), а затем вместе сравнивают результаты с получившимися у коллег.

В случае если все оценки примерно одинаковы, то общая оценка считается верной.

Но если возникают различия, команда обсуждает их и после прихода к общему мнению вновь проводит оценку.



Покер-планирование эффективно, так как оценки выставляют люди, выполняющие работу. Здесь я упомянул разработчиков, но оценки могут выставлять также администраторы баз данных, дизайнеры, технические писатели и другие специалисты, отвечающие за реализацию истории.

Сильной стороной данного метода является дискуссионная составляющая. Если кто-то говорит, что история совсем невелика, а другой считает ее, напротив, очень большой, не так важно, кто прав, а кто виноват (это выяснится на практике). Дело в том, что начинается ценная дискуссия, и вот это действительно важно.

Хотелось бы оговориться: покер-планирование — это не голосование (то есть три голоса молодых разработчиков не перевешивают голос одного опытного специалиста). Но так люди могут высказать свое мнение, что в идеале приводит к оптимальной оценке ситуации.

И не ведитесь на имеющиеся на рынке колоды для покер-планирования с номерами 8, 13, 20, 40 и 100 — они вам не понадобятся.

Будьте проще. Измеряйте истории в малых величинах (1, 3, 5 баллов — если случайно попадется эпик). Не гонитесь за ложным чувством точности, которое придают такие большие числа. На самом деле они только мешают работать.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, верно ли то, что при гибкой разработке мы не гонимся за точностью при оценке, и все, что нам нужно, — это относительная оценка?*

МАСТЕР: *При оценке всегда нужно давать самую точную оценку, которую ты можешь дать. Поэтому было бы неправильно говорить, что гибкой разработке чуждо стремление к точности.*

УЧЕНИК: *То есть, оценивая пользовательские истории, мы должны стремиться как к точности, так и к относительности?*

МАСТЕР: *Да. Оценивай настолько точно, насколько можешь, только помни, что абсолютной точности все равно не достигнешь. Только при сравнении пользовательских историй друг с другом и при верном измерении скорости работы команды ситуация сложится для нас благоприятно и наши планы окрепнут.*

УЧЕНИК: *То есть я должен стараться дать максимально точную оценку, но еще больше обращать внимание на то, чтобы истории, с которыми я работаю, сравнивались друг с другом?*

МАСТЕР: *Да. При оценке небольшое количество работы дает далеко идущие последствия. Не стремись к чрезмерной точности оценок. Сравнивай истории друг с другом. Принимай их такими, какие они есть, и соответствующим образом корректируй ожидания.*

УЧЕНИК: *Спасибо, Мастер. Я подумаю об этом.*

Что дальше?

Поздравляю! Научившись оценивать пользовательские истории относительно друг друга по балльной системе, вы получили все необходимое для создания своего первого гибкого плана.

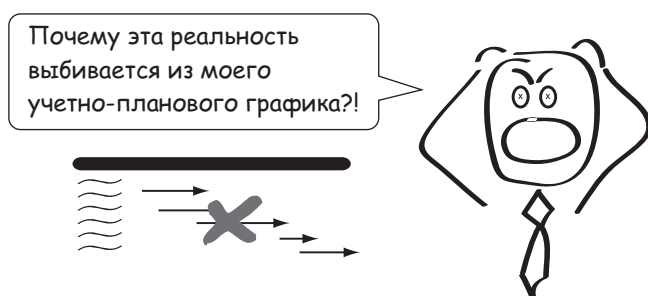
При планировании гибких проектов мы остановимся на всех инструментах, которые нужны для прогнозирования, отслеживания и создания плана проекта, которому будете доверять и вы, и ваш клиент.

Затем, располагая планом и стартовой колодой, мы будем готовы перейти к главной части работы — реализации гибкого проекта.

Итак, приготовьтесь познакомиться с секретами гибкого планирования.

Глава 8

Гибкое планирование: столкновение с реальностью



Привыкни к этому, друг. Законы Мерфи не знают пощады, если речь заходит о разрушении самых стройных планов. Если у тебя нет стратегии, которая позволяет бороться с изменениями, твой проект сожрет тебя заживо.

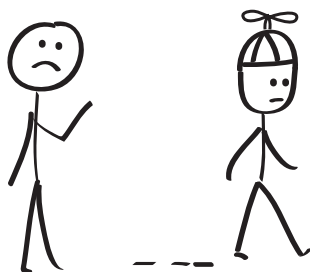
В этой главе рассказано о том, как создавать реалистичные планы, позволяющие вам и вашей команде выполнять взятые на себя обязательства.

Изучая гибкое планирование проектов, мы будем спать спокойно, будучи уверенными, что наш план всегда актуален, мы можем честно и открыто давать обещания и не стоит опасаться изменений, а следует использовать их в качестве орудия в конкурентной борьбе.

8.1. Проблемы, связанные со статичным планированием

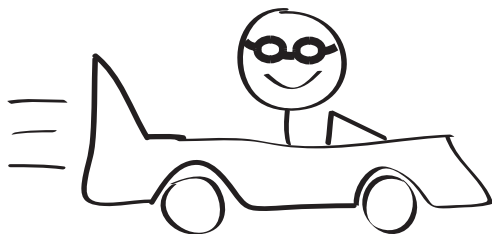
Случалась ли с вами такая история: проект запускается без всяких проблем, у вас отличная команда, правильно подобранная технология и безупречный план. В течение первых двух недель реализации проекта кажется, что все просто превосходно. И вдруг откуда ни возьмись... бам!

Ваша команда меняется...

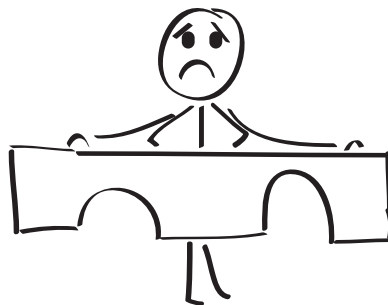


Ваш ведущий разработчик с головой погружается в другой проект — считается, что этот второй проект стратегически очень важен (странно, еще недавно такое же значение придавалось вашему проекту). «Ладно, время терпит, — думаете вы, — справимся». И вот внезапно все идет насадку...

Вы осознаете, что работа идет не так быстро, как вы планировали...



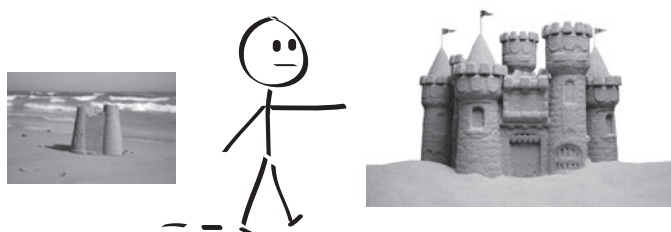
Как вы планировали



Как происходит на практике

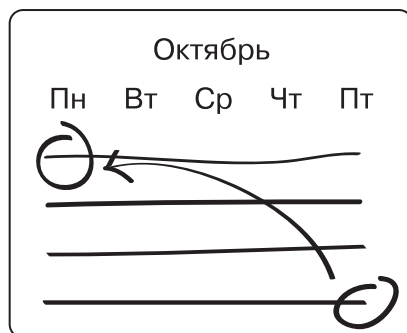
То, чего вы ожидали от команды, и то, что ей удастся, — это, как говорится, большая разница. И потом, на полпути к завершению проекта...

Ваш клиент осознает, чего он действительно хочет...



Это простое, легко выстраиваемое веб-приложение вдруг оказывается гораздо более сложным и обескураживающим. То, что, казалось бы, было проще пареной репы, теперь представляется невыполнимой задачей, учитывая, сколько времени и ресурсов у нас осталось. А потом начинается катастрофа.

Время на исходе

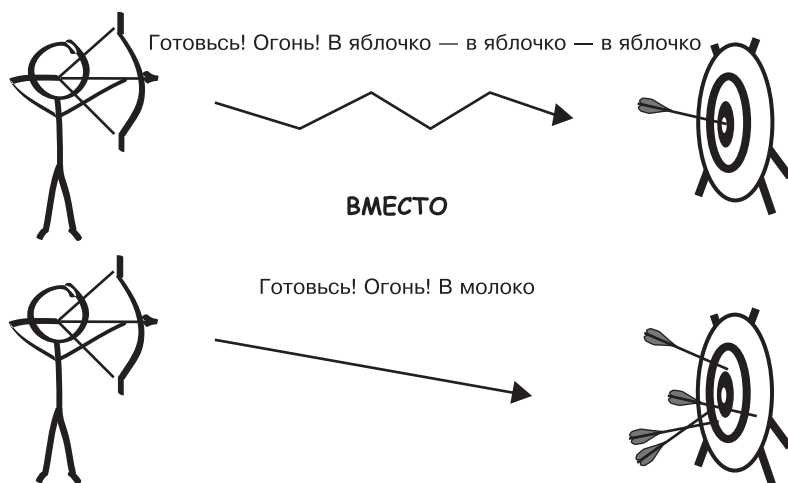


Выясняется, что программа требуется заказчику как можно быстрее. Из всех сил вы пытаетесь уложиться в новые сроки и из-за этого отказываетесь от тестирования. И когда программа в итоге доводится до конца, она оказывается настолько некачественной, что пользоваться ею невозможно. Она пополняет ряды мертворожденных, превысивших бюджет, неудавшихся IT-проектов.

Если эта история задевает вас за живое, утешьтесь — вы не одиноки в своей беде. Команды меняются, планы урезаются, требования постоянно корректируются — в интересном софтверном проекте это обычное дело.

Чтобы справиться с такой реальностью, нам нужно составить план, который обладал бы следующими качествами:

- ❑ представлял ценность для наших клиентов;
- ❑ был максимально наглядным, открытым и честным;
- ❑ позволял нам сдерживать данные обещания;
- ❑ позволял адаптироваться к ситуации и при необходимости допускал изменения.



Итак, мы очертили контекст, в котором обычно происходят изменения. Теперь рассмотрим гибкий план.

8.2. Знакомство с гибким планированием

В простейшей форме гибкое планирование — это просто измерение скорости, с которой команда может превращать пользовательские истории в работающие программы, готовые к реальному использованию, и последующее ориентирование на эту скорость при прогнозировании того, что мы собираемся сделать.

История о том, как меня один раз попросили уйти с проекта



Однажды мне довелось работать с клиентом, вместе с которым мы пытались написать программу для бухгалтерского учета в нефтегазовой сфере.

Объективно система стоила \$2 млн, но мы пытались управиться, имея \$700 тыс. Когда стало очевидно, что бюджет более чем в два раза меньше, чем необходимо, компания стала закручивать гайки, требуя от нас, чтобы мы работали сверхурочно и выполнили проект «в соответствии с планом».

Можете себе представить, как это выглядело. Каждый раз, когда мы собирались для планирования следующей итерации, от нас требовали удвоить текущую скорость работы, и нам приходилось отказываться.

В один совсем не прекрасный день наступил критический момент. Начальник отвел меня в сторонку и заявил, что если мы не согласимся работать больше, то просто угробим кредит доверия, который зарабатывали у клиента в течение года, и мои услуги в этом проекте более не понадобятся.

В итоге я потерял этого клиента. Честно говоря, мы допустили несколько крупных ошибок: не сделали стартовой колоды в начале проекта, не совсем внятно объяснили, как происходит гибкое планирование.

Но в данном случае важна культура сотрудничества. Не всем нравятся открытость и прозрачность, которые присутствуют при гибкой разработкой. Перед началом работы убедитесь, что клиент понимает, как происходит гибкое планирование, и уточните, где можно проявить эту гибкость, если реальность перестанет укладываться в план.



Как вы помните, при гибком планировании список того, что нужно сделать, называется *журналом пользовательских пожеланий* (бэклогом). В нем перечисляются все функции, которые клиент хотел бы видеть в заказанной программе.

Скорость, с которой пользовательские истории преобразуются в готовые программные функции, в гибкой методологии называется *скоростью работы команды* (team velocity). С ее помощью измеряется производительность команды, она же помогает делать прогнозы на будущее.

Рабочий процесс в гибкой методологии строится из так называемых *итераций*, или спринтов, — недельных, иногда двухнедельных периодов, в течение которых мы реализуем пользовательские истории в виде программных функций.

Чтобы приблизительно спрогнозировать сроки, в которые продукт будет готов, мы берем общий объем работы по данному проекту, делим его на предполагаемую скорость команды и подсчитываем, сколько итераций должно пройти до завершения проекта. Так у нас получается план проекта.

$$\text{Количество итераций} = \frac{\text{Общий объем работ}}{\text{Примерная скорость работы команды.}}$$

Например:

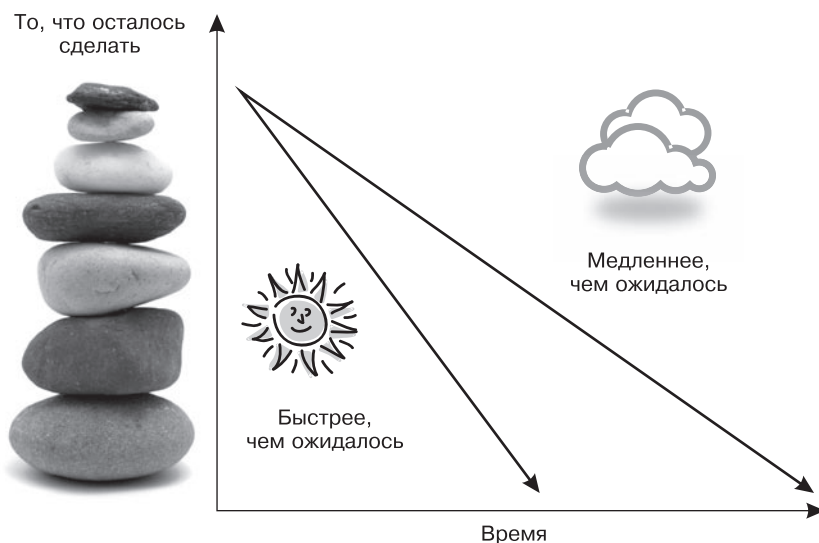
$$\begin{aligned} \text{Количество итераций} &= 100 \text{ баллов} / 10 \text{ баллов} \\ &\text{на итерацию} = 10 \text{ итераций.} \end{aligned}$$

Очень важно понимать, что составленный нами план проекта — это не непреложное обещание, а прогноз. В начале проекта мы не знаем, какова будет скорость работы команды. Уточнить ее можно лишь тогда, когда мы закончим работу над одним из компонентов и узнаем, сколько на это потребуется времени. После уже можно будет говорить о реалистичных датах.

Если считать первичные прогнозы обещаниями, проект можно похоронить еще до того, как он начнется.

Теперь, когда мы приступили к разработке, произойдет одно из двух. Мы обнаружим, что:

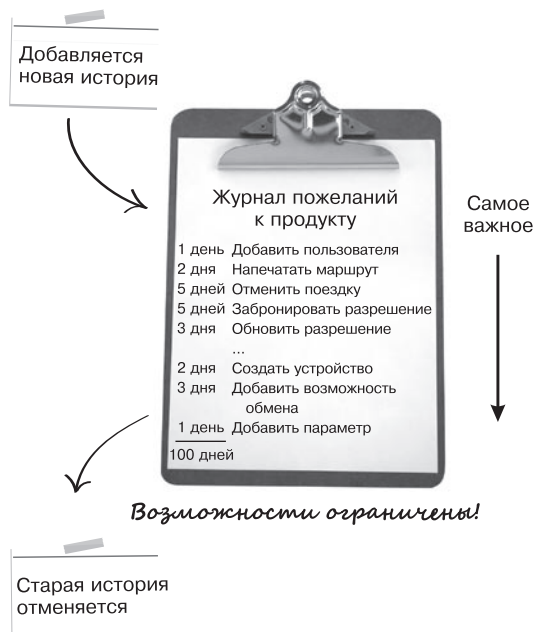
- ❑ работа идет быстрее, чем ожидалось;
- ❑ работа идет медленнее, чем ожидалось.



Быстрее, чем ожидалось, — это ситуация, в которой вы и команда работаете с опережением плана. Медленнее, чем ожидалось (и это бывает чаще), означает, что вам еще много нужно сделать, а времени на все не хватает.

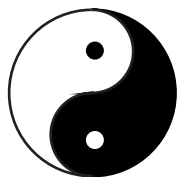
Сталкиваясь с ситуацией, когда сделать нужно слишком много, гибкие команды пытаются уменьшить объем работы (так поступаем все мы, когда, например, обнаруживаем, что слишком много запланировали на выходные). Вместо того чтобы упрямо следовать первоначальному плану, его нужно менять, как правило — снижать объем работ.

8.3. Изменение объема работ



Благодаря изменению объема работ гибким командам удастся выполнять стоящие перед ними планы.

Гибкие команды настаивают на том, что, если в ходе работы клиент добавляет новую историю, он должен отказаться от одной из имеющихся, но дают клиенту возможность подумать и изменить мнение (не выплачивая заоблачную цену).



Принцип гибкой разработки

Положительно относитесь к изменению требований, даже на поздних этапах разработки. Гибкая разработка не боится изменений, превращая их в преимущества клиента при конкурентной борьбе.

Таким образом, у клиента не создается иллюзии, что нужно расширяться в лепешку, собирая требования (просто уменьшается объем лишней работы), а команда по мере работы осознает, что невозможно составить совершенный план заранее.

Строго говоря, клиент теперь не обязательно откажется от старой истории, если у него появится новая. Например, если старая история описывает

функцию, которая действительно потребуется при работе и за которую клиент готов заплатить.



* Рекомендуется

Чего действительно не следует ожидать клиенту, так это возможности добавить в список дополнительный пункт без отказа от какого-то из имеющихся пунктов. Это принятие желаемого за действительное, и таким поступкам нет места в гибком планировании.

Если встает дилемма — отодвинуть дату релиза или изменить объем работ, сторонники гибкой разработки обычно предпочитают второй вариант. Ничто не происходит в нашем деле так легко, как постоянное сдвигание дат выпуска. А вот делать готовые программы вовремя — это гораздо сложнее.

Но независимо от того, есть ли у вас жесткая дата окончания разработки или вы ведете работу до того, как будет готов основной функционал, гибкий подход к объему работ — та концепция, которую клиент должен очень хорошо усвоить. Только тогда ваши планы останутся реалистичными, а ваша команда не будет брать на себя больше, чем сможет сделать. Здесь может возникнуть вопрос: а что делать, если клиент не хочет гибко подходить к плану и настаивает на том, чтобы вы и ваша команда работали сверхурочно?

В таком случае есть два варианта.

- ❑ Можете продолжать лгать, смотреть на работу сквозь пальцы и придерживаться старого плана, как это делают все остальные. Или же дать

чрезмерно оптимистичную оценку, раздуть расчеты, игнорировать скорость работы команды и молиться, чтобы в конце все разрешилось само собой (то есть рассчитывать на чудо).

- Или, если ничего не помогает, можно рассказать о ситуации без прикрас, какова она есть на самом деле, и посидеть в наступившей неловкой тишине, пока клиент не поймет, что вы не собираетесь уступать. Вы не будете больше создавать видимость и участвовать в распространении той величайшей лжи, которая существует в информатике на протяжении уже более 40 лет. Никто и не говорил, что самураем быть легко.

А теперь давайте рассмотрим, как составить первый гибкий план.

8.4. Ваш первый план

Создание первого гибкого плана не особо отличается от подготовки к выходным, на которые запланировано много дел. Все начинается с хорошего списка.

Этап 1. Подготовка журнала пользовательских пожеланий



Список пользовательских пожеланий — это набор пользовательских историй (с описанием функций), которые клиент хочет видеть в программе. Клиент расставляет приоритеты для отдельных историй, создавая, таким образом, основу для плана вашего проекта.

Обычно на проработку хорошо составленного журнала пожеланий требуется от одного месяца до полугода. Нет смысла отслеживать истории, реализация которых планируется на более поздний срок, так как:

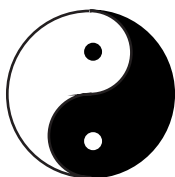
- ❑ вы не знаете, что произойдет в мире через полгода;
- ❑ возможно, этими историями так и не придется заниматься, поэтому сейчас думать о них точно незачем.

Иногда приходится реализовать все функции, перечисленные в списке, но обычно так не происходит, ведь времени и денег на это, как правило, не хватает.

Поэтому, чтобы спрогнозировать, что, скорее всего, будет сделано, а что — нет, гибкая команда обычно берет из журнала пожеланий набор историй и называет его *релизом*.

Определение релиза. Релиз — логически объединенная группа историй, которая представляет ценность для вашего клиента. То есть это набор, который стоит разрабатывать и впоследствии развертывать. Кроме того, релиз иногда называют *минимальной коммерчески ценной функциональностью* (Minimal Marketable Feature set, MMF¹).

Первая «М» в аббревиатуре MMF, означающая «минимальная», напоминает нам, что требуется в кратчайшие сроки начать выдавать полезный результат (и о том, что зачастую 80 % ценности системы обеспечивается всего 20 % ее функций). Итак, мы хотим выбрать наименьшее количество функций, которые принесут наибольшую пользу в первой версии нашей программы.

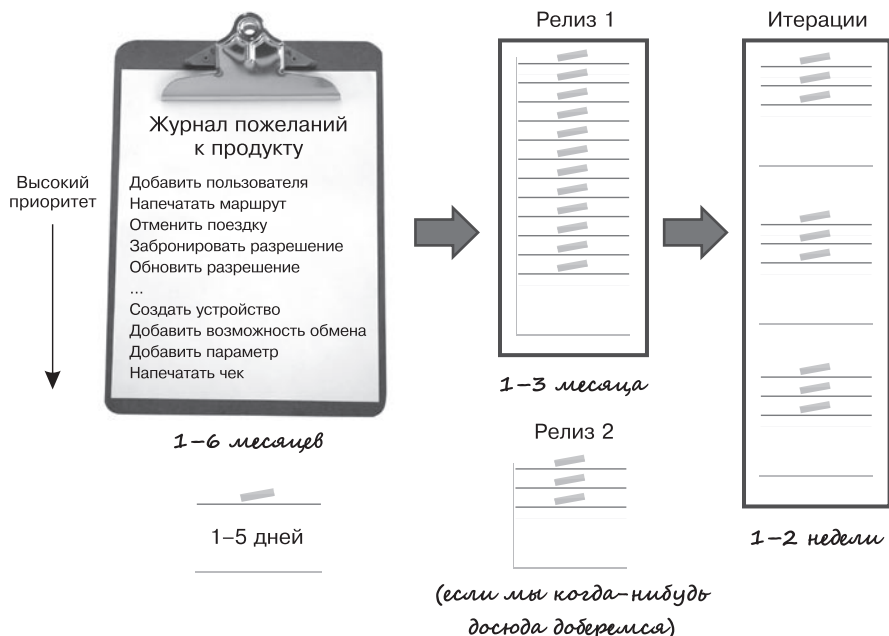


*Принцип гибкой
разработки*

Очень важна простота, то есть умение максимально уменьшать объем работы.

Обратите внимание на слово «коммерчески» в этом термине. Оно означает, что функциональность должна иметь ценность для клиента (иначе он просто не будет ею пользоваться). Итак, минимализм и коммерческая ценность — два основных параметра, по которым следует формировать набор историй для первого релиза.

¹ *Software by Numbers: Low-Risk, High-Return Development* [DCH03].



Когда вы определитесь с журналом пользовательских пожеланий и с тем, что войдет в первый релиз, нужно прикинуть, сколько времени потребуется на разработку.

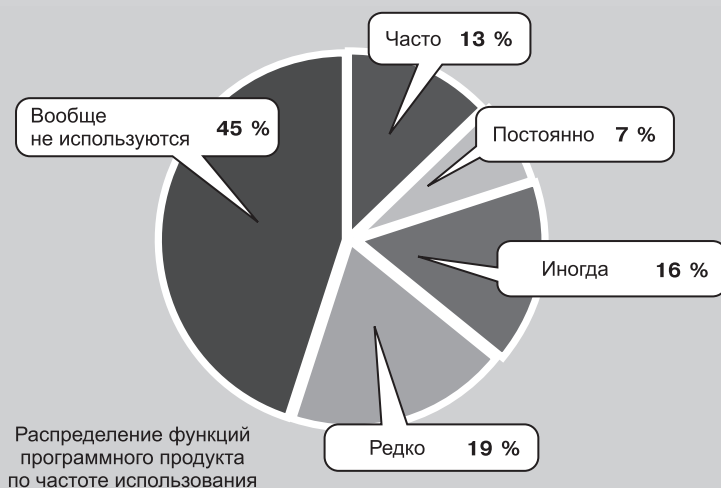
Этап 2. Определение времени, необходимого на разработку

В главе 7 мы рассмотрели методы, применяемые гибкими командами для оценки времени, которое потребуется на разработку той или иной функции.



Основная причина траты времени в ходе проектов

А вы знали, что 64 % функций любой программы используются редко или вообще не используются? Невероятно, но факт!¹



Подумайте об этом. Сколько функций вы используете в программе Microsoft Word? Пять процентов? Десять процентов? Может быть, процентов двадцать, если вы очень опытный пользователь.

Когда мы просим клиента сосредоточиться только на том, что действительно важно, а все остальное оставить в стороне, мы экономим ему массу времени и денег, так как можем выполнить его заказ максимально быстро.

На данном этапе мы пытаемся понять, насколько велика стоящая перед нами задача и сколько времени понадобится на ее решение — один, три, шесть или девять месяцев.

Когда рамки списка задач определены, можно переходить к расстановке приоритетов.

¹ Данные исследования аналитической компании Standish Group по докладу ее руководителя Джима Джонсона, прочитанному на XP2002.

Этап 3. Расстановка приоритетов

В любой момент может случиться что-то неожиданное (то есть проект может быть отменен или сокращен), поэтому в первую очередь нужно реализовать самые важные функции.

Если клиент расставит в журнале пожеланий приоритеты в соответствии с нуждами своего бизнеса, это гарантирует, что он вложит деньги максимально выгодным образом.



Хотя за клиентом остается последнее слово по поводу того, что и когда будет создаваться, вы также должны выдвигать предположения относительно функций, которые следует реализовать в первую очередь, чтобы архитектура программы получилась более надежной.

Например, хорошими кандидатами на раннюю разработку являются истории, которые важны для клиента и позволяют испытать архитектуру. Сопоставляя факты на раннем этапе и прорабатывая систему на всех уровнях, можно исключить множество рисков, а также получить ценнейшие данные о том, как лучше всего организовать данную систему. Поэтому не стесняйтесь высказываться — ваш профессионализм и опыт имеют большое значение.

Имея список, в котором расставлены оценки и приоритеты, мы практически готовы говорить о датах. Но перед этим еще нужно сориентироваться, насколько быстро способны работать вы и ваша команда.

Скорость — это командная черта!

Когда мы строим планы исходя из скорости работы нашей команды, за исход мы ручаемся тоже как команда. Мы говорим: «Как команда мы чувствуем, что сможем выполнять вот такой объем работы, от итерации к итерации».

Этот подход значительно отличается от измерения индивидуальной продуктивности и открывает перед нами неизвестную сторону управления проектами.

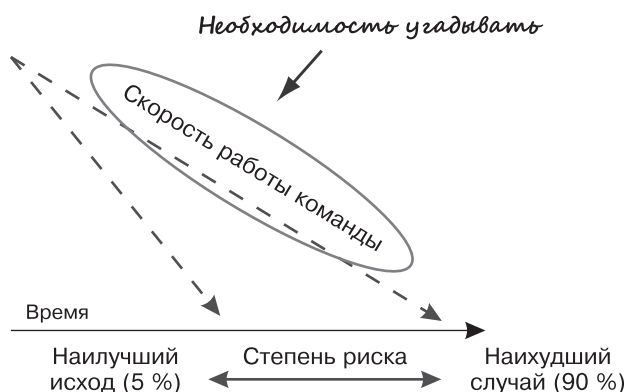
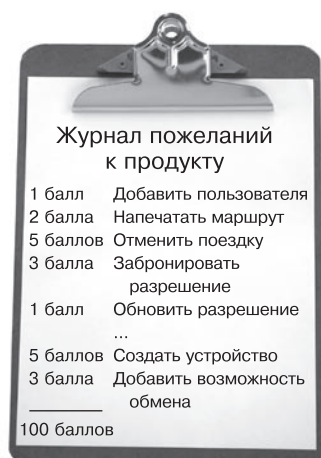
Если вам нужно больше ошибок, больше материала для переработки, больше непонимания, меньше сотрудничества, умения и обмена знаниями, то во что бы то ни стало стимулируйте, подчеркивайте и вознаграждайте в ходе разработки высокую индивидуальную продуктивность.

Но помните, что, поступая так, вы убиваете сам дух и природу наших проектов. Мы стремимся к обмену идеями, взаимопомощи и отслеживанию деталей, которые обычно остаются незамеченными.

Этап 4. Оценка скорости работы команды

Гибкое планирование действует потому, что мы строим планы на будущее, но основываемся при этом на результатах, которых смогли достичь в прошлом.

И поскольку в начале проекта мы еще не знаем, насколько быстро сможет работать наша команда, нам приходится угадывать.



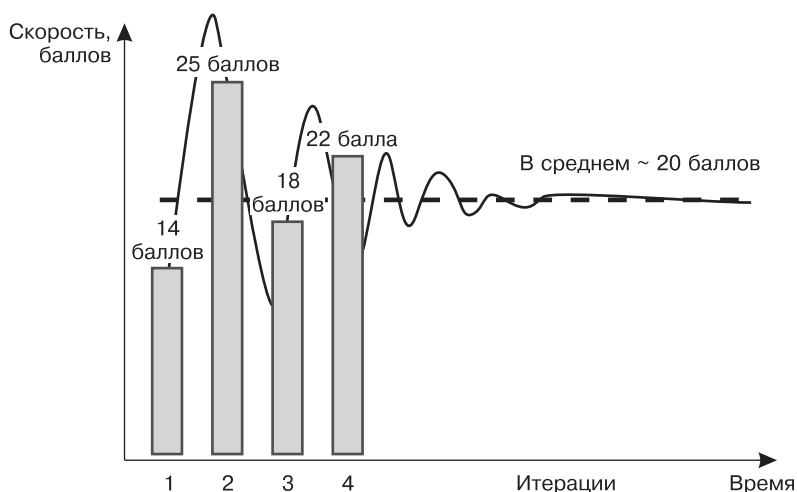
Теперь, если все ваши истории имеют одинаковые размеры, эту схему можно представить в упрощенном виде:

$$\text{Скорость работы команды} = \text{Выполненные истории} / \text{Итерация}.$$

Однако обычно размеры историй значительно различаются, и в таком случае скорость работы команды будет выражаться следующей формулой:

$$\text{Скорость работы команды} = \text{Балльная оценка выполненных историй} / \text{Итерация}.$$

Сейчас, в самом начале проекта, скорость работы будет колебаться, и пусть вас это не волнует. Это нормально, пока команда не освоится с работой и не найдет наилучший способ взаимодействия.



Но через 3–4 итерации скорость должна начать выравниваться и вы начнете понимать, насколько быстро команда справляется с проектом.

Не существует безоговорочных правил, которые позволили бы оценить скорость работы команды. Спросите коллег, сколько, на их взгляд, удастся сделать за итерацию, а также обязательно учитывайте такие детали, как доступность заказчика и возможность организации работы команды в одном офисе.

Также напомните команде, что значит «сделано» (см. раздел 1.3), и еще раз поясните, что при гибкой разработке под реализацией истории понимается анализ, тестирование, проектирование и написание кода. Всё вместе.

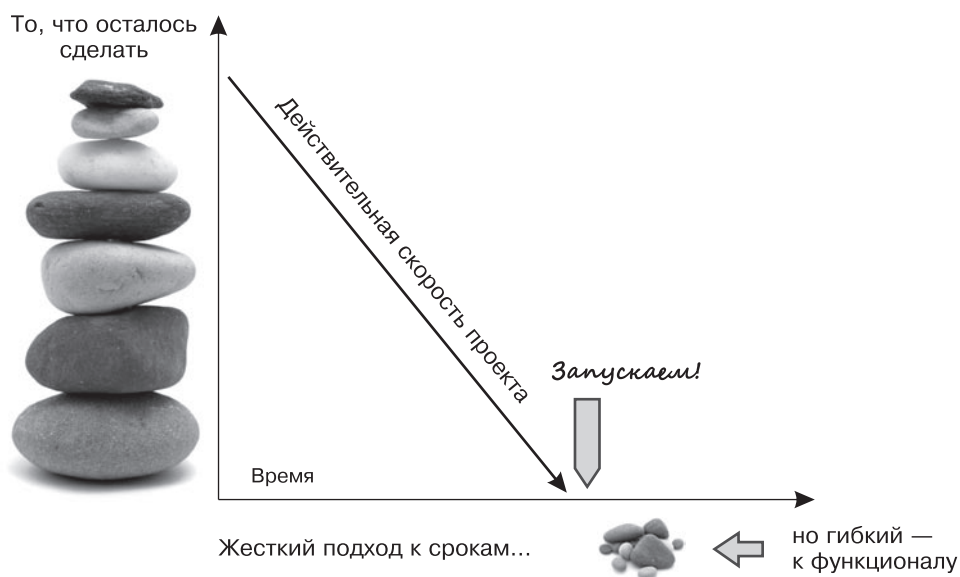
Кроме того, лучше не переусердствовать с первоначальными оценками. Секрет успеха — в заниженных ожиданиях, а если задать слишком высокую планку, то вам предстоит пережить неприятный разговор, если эта планка не будет достигнута. Поэтому будьте скромны в оценках, напоминайте владельцу продукта, что оценки — это догадки, и начинайте измерение скорости с первого дня работы.

Имея готовый список и оценку скорости работы, мы можем приступать к определению сроков.

Этап 5. Определение сроков

Есть два способа примерно спланировать сроки работы. Можно выполнять проект *к определенному сроку* (by date) или *до реализации основного набора функций* (by feature set).

Выполнение к определенному сроку



Когда проект выполняется к определенной дате, мы как будто проводим линию на песке и говорим: «Проект будет закончен к этому моменту во что бы то ни стало».

Когда обнаруживаются новые важные пользовательские истории, то более ранние, не менее объемные, но менее важные, отбрасываются.

Поэтому мы вынуждены принимать жесткие решения о компромиссах заранее (компромиссы будут касаться, например, функционала приложения), но достаточно категорично даем понять, что дело не терпит отлагательств.

Можно более гибко подходить к дате, уделяя внимание основному набору возможностей, то есть работать до реализации основного набора функций.

Выполнение до создания определенного набора функций



Мы выбираем основные функции и разрабатываем их до полной готовности.

Гибкий подход к функционалу приложения по-прежнему актуален (поскольку в ходе работы будет обнаруживаться, что нужны и новые функции), но суть в данном случае такова, что команда должна разработать несколько крупных блоков, а дата уже не так важна — главное, чтобы весь набор основных возможностей был реализован.

Преимущество реализации до определенного набора функций заключается в том, что уже в самом начале проекта вы знаете, что именно нужно сделать обязательно, и можете оценить риски, связанные со сроками выполнения. Ваши клиенты и спонсоры определяют, приемлем ли для них этот риск.

Именно так и создается гибкий план! Вы выстраиваете взвешенный журнал пожеланий, в котором учтены все приоритеты, оцениваете скорость работы вашей команды и осознанно выбираете дату окончания работ.

Пока мы только начинаем работу, нужно поговорить еще об одном отличном инструменте выстраивания перспектив — о графике прогресса разработки (burn-down chart). Затем обратимся к искусству гибкого планирования.

8.5. График прогресса разработки (burn-down chart)

Хотя формально мы еще не обсуждали график прогресса разработки, вам уже приходилось сталкиваться с ним на страницах книги. Это график, показывающий, насколько быстро мы как команда перерабатываем пользовательские истории. Кроме того, он позволяет прогнозировать, какова будет скорость работы в дальнейшем.

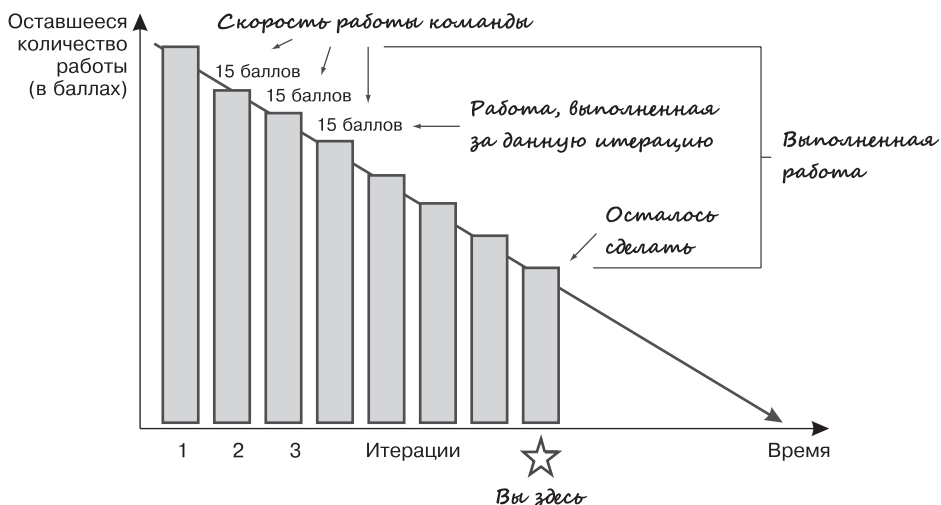


По оси Y мы отслеживаем количество оставшейся работы (в рабочих днях или баллах), а по оси X — время, отпущенное на итерацию. Просто запишите объем работы (количество баллов), остающийся на каждую из итераций, и отобразите это на графике. Угловой коэффициент (уклон прямой линии) — это скорость команды (позволяющая судить, сколько работы команда способна выполнить за итерацию).

График прогресса разработки — отличное средство, позволяющее узнать, в каком состоянии находится ваш проект. Просто взглянув на него, можно узнать следующую информацию:

- ☐ сколько работы сделано;
- ☐ сколько работы остается сделать;
- ☐ скорость команды;
- ☐ ожидаемую дату завершения работы.

Каждый столбец (итерация) на следующей схеме представляет собой объем оставшейся работы по данному проекту. Мы завершаем работу, когда высота столбца становится нулевой.



В идеальном мире скорость работы вашей команды была бы постоянной. Линия началась бы на отметке 15 баллов, аккуратно спустилась слева направо и к концу проекта оказалась бы в правой нижней части.

Однако в реальности график прогресса разработки обычно выглядит так:



Дела редко идут по плану. Скорость работы команды колеблется. Обнаруживаются новые истории. От некоторых старых историй приходится отказываться.

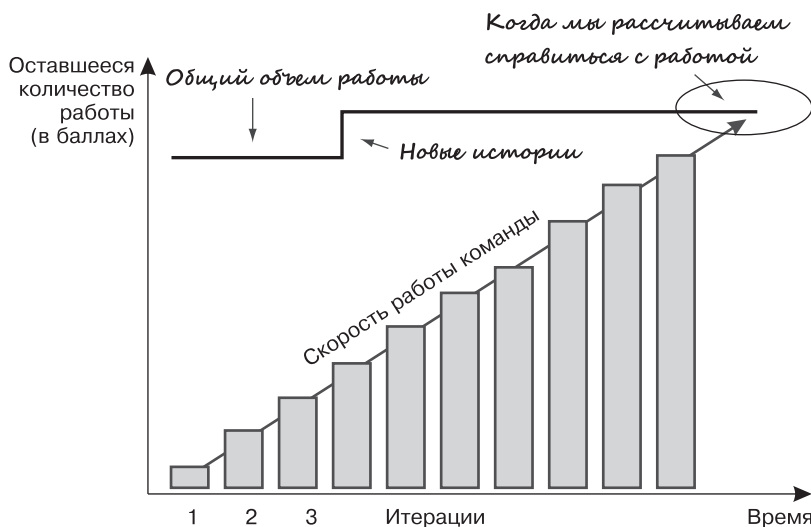
На графике прогресса разработки можно отобразить все эти события, происходящие в ходе проекта. Если клиент решает добавить в проект дополнительные задачи, вы сразу же увидите, как это скажется на дате готовности проекта. Если работа команды замедляется из-за того, что ее покинул ключевой сотрудник, данное обстоятельство также отразится на графике скорости работы команды.

У графика прогресса разработки есть не только количественная, но и качественная сторона. Если на графике что-то выявится, это поможет нам в беседе с владельцем проекта на данную тему, а также будет полезно при принятии решений, принципиально влияющих на работу.

Графики прогресса разработки описывают ситуацию как она есть. Это крайне репрезентативная часть гибкого планирования. Мы ничего не прячем и не украшаем реальность. Регулярно пересматривая график прогресса разработки вместе с клиентом, мы можем открыто и честно делать прогнозы, причем все будут понимать, к какому сроку мы рассчитываем завершить работу.

График объема работы

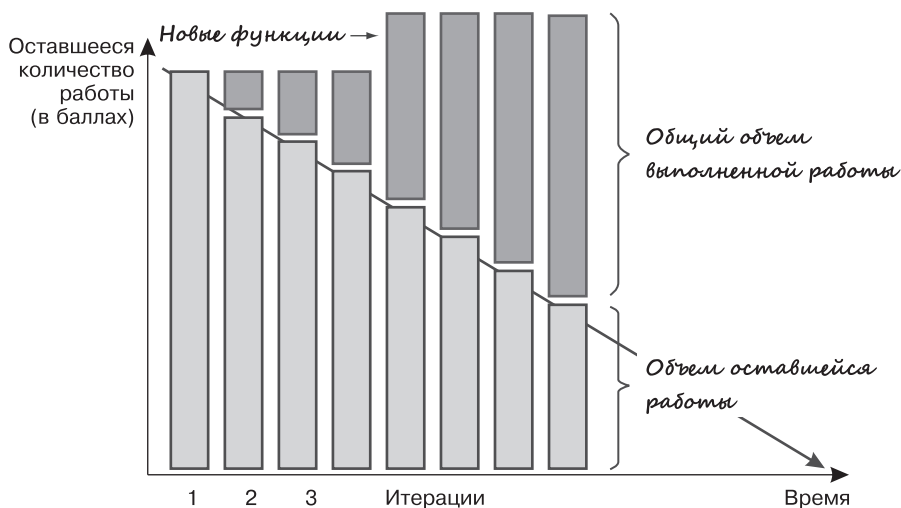
Кроме графика прогресса разработки, часто используется и обратный график (burn-up chart). На нем отображается динамика объема работы.



Некоторые специалисты предпочитают пользоваться именно графиком объема работ, так как на нем отображается процесс появления новых историй. Если провести в его верхней части непрерывную линию, можно сразу

увидеть любой рост объема работ, и отслеживать этот рост во времени становится несколько проще.

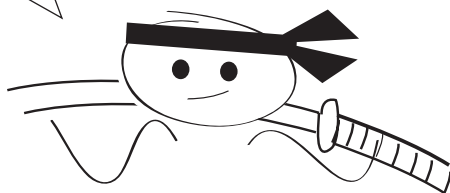
Если вам нравится наглядность графика объема работ, но вы не хотите отказываться от простоты и самой концепции графика прогресса разработки, можно объединить эти две схемы. Просто отслеживайте общий объем работ для каждой итерации на графике прогресса разработки, а одновременно с этим следите за количеством оставшейся работы.



Каким именно графиком пользоваться, полностью зависит от вашего выбора. Просто убедитесь, что у вас есть простой и наглядный способ сориентироваться, сколько работы еще остается и когда с ней удастся разобраться.

8.6. Перевод проекта в режим гибкой разработки

Что, если вам достанется запутанный проект?
Как придать гибкость такому недоработанному проекту?



Если у вас уже есть начатый проект, существует немало вариантов его перевода в режим гибкой разработки. Возможно, вы желаете сделать проект гибким по следующим причинам:

- ❑ то, чем вы сейчас занимаетесь, не работает;
- ❑ вам срочно нужно выдать какой-то результат.

Если проблема заключается в выходе на единый курс, создайте стартовую колоду (см. главу 3).

Вероятно, вам и не понадобится полная колода, но вы должны убедиться, что всем известны ответы на следующие вопросы.

- ❑ Зачем мы здесь собрались?
- ❑ Чего мы пытаемся достичь?
- ❑ Кто наш клиент?
- ❑ Какие крупные проблемы мы должны решить?
- ❑ Кто командует?

Если по этим или по другим вопросам из стартовой колоды существуют сомнения, возьмите карточку, с которой связаны разночтения, задайте конкретные вопросы и придите к общему мнению.

Когда нужно быстро подготовить какую-то часть работы, откажитесь от текущего плана и создайте новый, который покажется вам реалистичным. Действуйте так же, как при создании с чистого листа нового гибкого плана: составьте список того, что необходимо делать, прикиньте сроки работы, установите приоритеты и разработайте минимальный функциональный фрагмент, который представляет интерес для клиента.

Если необходимо продемонстрировать прогресс, но ставится условие — работать по старому плану, начните выдавать небольшие ценные результаты каждую неделю. На каждую неделю выбирайте одну-две полезные функции и просто доводите их до конца — полностью. Как только вы покажете, что можете выдавать результат (и заслужите, таким образом, определенный кредит доверия), постепенно перерабатывайте план и определите, что войдет в релиз, на основе уже измеренной скорости работы команды, с учетом того, сколько остается сделать.

Затем просто продолжайте работу до тех пор, пока у вас не будет что-нибудь готово. В процессе работы уточняйте план, действуйте решительно. Если у вас на пути будут какие-то преграды, просто устраняйте их, аргументируя свои действия неотложностью работы, которой приходится заниматься.

Рассмотрим, как некоторые из перечисленных аспектов выглядят на практике.

8.7. Практическая реализация

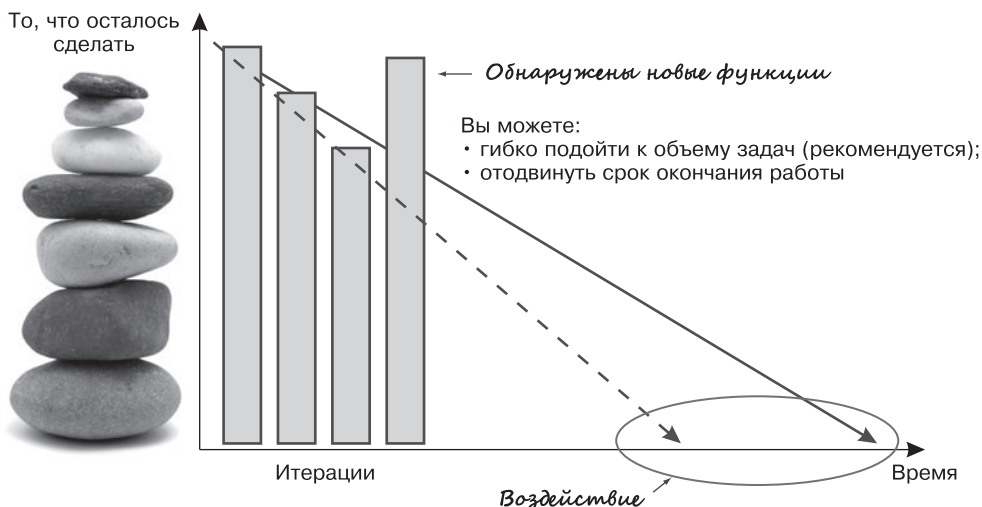
Мы хорошо поработали и знаем уже немало теории. Теперь перейдем к практике и пересмотрим четыре проблемы, с которыми столкнулись в начале главы. Затем поговорим о том, как с ними можно справиться в рамках нового гибкого плана.

Блаженное неведение



Помню, как однажды я спросил вице-президента о том, что он думает о гибком планировании. Он сказал: «Это называется “ни с тобой, ни без тебя”». С одной стороны, ему нравилась прозрачность гибких проектов, но в то же время она действовала ему на нервы. Раньше он мог просто закрыть глаза на происходящее и сказать, что все нормально. Но теперь проект на виду. Каждый день. В истинном виде. И от него не скрыться. Проект постоянно напоминает о том, как много еще остается доработать, и это, приходится согласиться, совсем не плохо.

Сценарий 1. Клиент обнаруживает новые требования

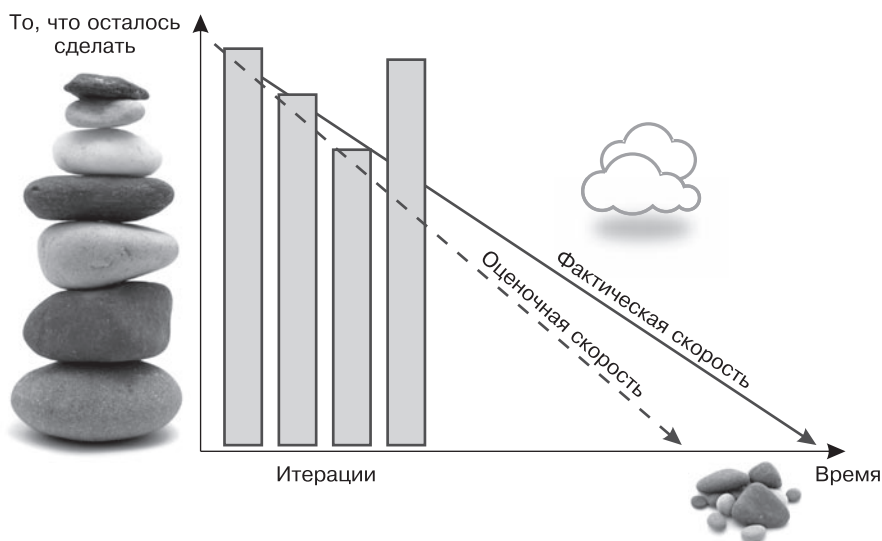


Когда клиент осознает, чего же именно он хочет от создаваемой программы, спросите его, что бы он сделал: отодвинул дату выпуска продукта (под этим подразумевается, что проект потребует дополнительных затрат) или отказался от каких-то не самых важных функций (более предпочтительный вариант).

В ходе разговора сдерживайте эмоции. Не вы сейчас принимаете сложное решение. Вы просто посредник, сообщающий, что является или может стать причиной самого плачевного исхода работы. Ваша ответственность заключается в том, чтобы осведомить клиента о важности его действий и предоставить ему всю необходимую информацию для того, чтобы решение было продуманным.

Если клиент действительно хочет все и никак не меньше, составьте список функций, которые неплохо было бы реализовать, и скажите, что если в конце проекта останется время, то в первую очередь вы займетесь реализацией данного списка. Но это необходимо четко озвучить. В настоящее время список того, что «неплохо бы иметь», не рассматривается и не входит в состав основного плана.

Сценарий 2. Работа движется медленнее, чем ожидалось



Если после трех-четырех итераций вы заметите, что скорость работы не такая, как вы надеялись, не паникуйте — такое случается. Мы делаем лишь ориентировочные прогнозы и просим клиента не считать исходные планы

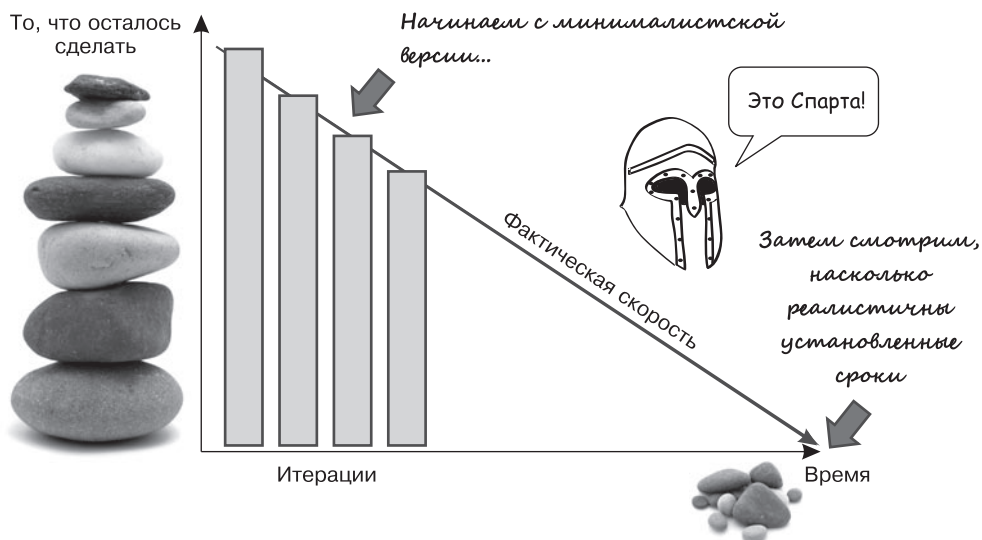
истиной в последней инстанции. Все не так плохо, поскольку теперь мы знаем о том, что ожидания не оправдались, и можем соответствующим образом скорректировать курс.

Гибкий подход к объему задач — это предпочтительный способ для восстановления равновесия. Кроме того, можно рассмотреть возможность вовлечения дополнительных ресурсов (поначалу это замедлит работу) или отодвинуть дату релиза (оба варианта неидеальны).

Важно пойти на такой разговор и предложить клиенту несколько вариантов. Да, это может быть нелегко, но речь идет о вещах, которые вы не сможете скрыть. Гибкая методология требует сообщать дурные вести как можно раньше.

Теперь мы уже не будем полностью беззащитны, когда придется отвечать на вопрос, достаточно ли у нас времени. Существует определенная стратегия, гарантирующая, что при разговоре на тему «еще очень много работы, времени не хватает» ваши доводы будут совершенно честными, прозрачными и непротиворечивыми.

Путь спартамца основан на простейшем положении. Если мы не можем сделать «урезанный», минималистский вариант приложения, имея выделенный объем ресурсов, то план никуда не годится и его необходимо менять.



Нужно работать так: берем одну-две по-настоящему важные функции из нашего проекта (что-нибудь основное, касающееся всей архитектуры при-

ложения) и измеряем, сколько времени потребуется на создание минималистского варианта этой функции, так сказать, ее «скелета».

Затем сравниваем полученный результат с пользовательскими историями, длительность которых измерена относительно друг друга, и определяем, можно ли написать минимальную пригодную для работы версию приложения, располагая имеющимися ресурсами.

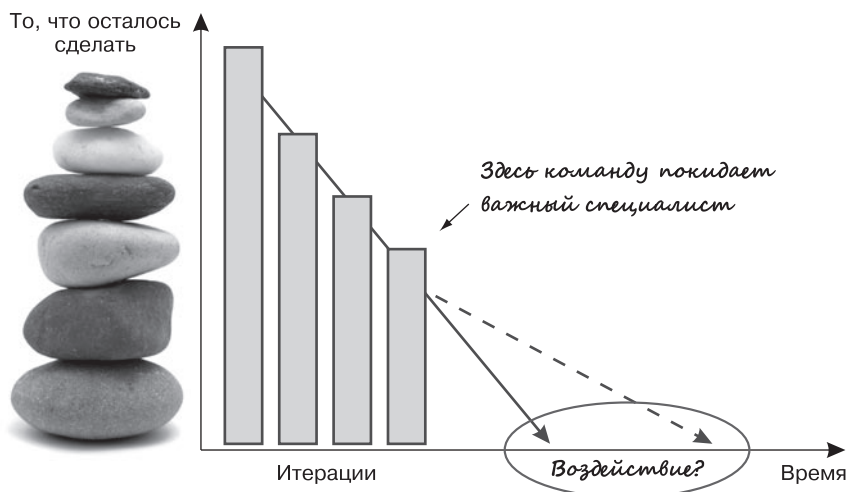
Если даты выглядят реалистично, то все в порядке! Продолжайте следить за процессом.

Если же оказывается, что поставленные сроки явно ошибочны, — тоже неплохо! По крайней мере, вы об этом узнали.

Спартанский подход позволяет вам начать разговор на тему «план нужно пересмотреть» с сильных и непротиворечивых позиций. Вы не выдаете желаемое за действительное. Никаких эмоций. Только факты. И об этом лучше сказать сейчас, чем позже.

Располагая такой информацией, вы теперь можете начать конкретный разговор с клиентом о том, какие функции будут относиться к «спартанской» версии приложения, а какие потребуют несколько более тонкой работы. Затем план проекта можно откорректировать так, чтобы деньги были вложены максимально эффективно и вы могли решить задачу, укладываясь в выделенный бюджет.

Сценарий 3. Команда теряет важного члена



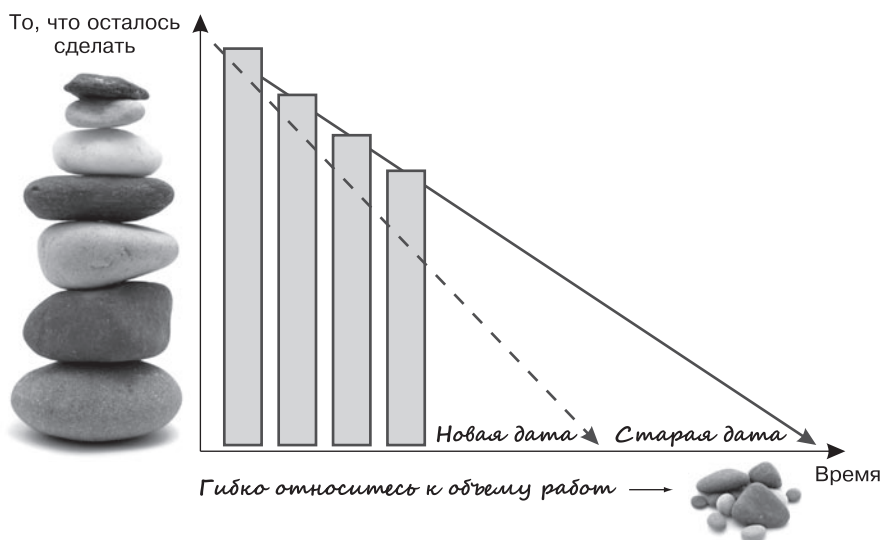
Оценить урон от потери командой одного из ключевых членов никогда не бывает просто. Вы знаете, что несете потери, вопрос лишь в том, насколько серьезные.

Когда дело доходит до прогнозирования изменений, связанных с составом команды, эти прогнозы по определению не будут точными и обоснованными. Просто дайте клиенту знать, что проект понес потери (если можете — предположите, насколько значительные). И когда вы сможете измерить, как потеря повлияла на скорость работы команды (на измерение этого воздействия понадобится две-три итерации), то опишите эту потерю точнее.

Разумеется, ваш менеджер может заупрямиться и сказать, что новый сотрудник, которого только что наняли, не хуже (а может быть, и лучше) человека, покинувшего вашу команду, а значит, никакого снижения скорости быть не должно.

Может быть, он прав. Но лучше не рассчитывайте на это. Новый человек может не влиться в команду. Или на собеседовании о новом сотруднике сложилось неверное впечатление — слишком уж кадровикам понравилось его великолепное резюме и крепкое рукопожатие. Верьте в человека после того, как увидите его в деле. А до тех пор проявляйте здоровый скептицизм и учитывайте это при прогнозах.

Сценарий 4. Время на исходе



Тривиальный ответ в данном случае — гибко подходите к объему работ. Если вы наполовину сокращаете сроки, вам просто придется в два раза сократить количество функций, которые вы собираетесь разработать. Ничего сложного.

Нетривиальный ответ — встретьтесь с клиентом, побеседуйте и придумайте инновационный способ, который поможет справиться со сложившейся ситуацией.

Может быть, на повестке дня есть истории, которые могут быть реализованы в минималистском, «спартанском» варианте программы. Вероятно, двадцать статических отчетов можно заменить только одним динамическим, но по-настоящему хорошим отчетом.

Помогая клиенту тогда, когда он испытывает затруднения, вы делаете огромную работу по налаживанию с ним необходимых отношений. Вы хотите, чтобы в вас видели надежного консультанта, и один из способов добиться этого — предлагать клиенту варианты.

Просто не будьте чрезмерно жестки или бесцеремонны и не давайте обещаний, которые не сможет сдержать ваша команда. Это никому не принесет пользы. Просто будьте честны и расскажите клиенту, что потребуется для выполнения задачи.

А теперь Мастер-сэнсэй снова встретится с вами, чтобы узнать, чему вы научились.

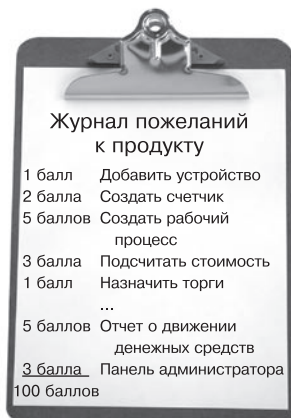


*Мастер-сэнсэй
и воин,
постигающий
искусство*

Добро пожаловать, ученик. Рад, что ты все еще жив. Я хочу обсудить с тобой один невыдуманный сценарий, с которым пришлось столкнуться другому моему ученику в реальных боевых условиях.

Сценарий: все параметры проекта жестко заданы, и изменить план нельзя.

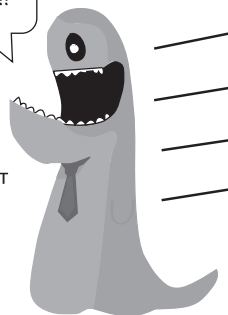
Государственная компания с жестко регламентированным управлением



Изменения не приветствуются!

Фиксированный объем работ
Жестко заданный срок
Фиксированный бюджет

P-p-p-p!!!



Как задействовать в этом проекте гибкую методологию? Изменение

МАСТЕР: Это проект для крупной государственной организации. Поскольку данная организация тратит деньги налогоплательщиков, все затраты подвергаются тщательному аудиту и компания не может позволить себе никаких изменений объема работ, стоимости и установленных сроков. Все четко определено. Следует ли в данном проекте рассмотреть возможность гибкой разработки?

УЧЕНИК: Если объем работ, сроки и бюджет действительно фиксированы и их нельзя изменить, как нельзя изменить и план, то я не знаю, как в данной ситуации может применяться гибкая разработка, Мастер.

МАСТЕР: Если в ходе проекта предпринимаются попытки жестко зафиксировать время, бюджет и объем работ, то скоро выясняется, что проект не может справиться с Неистойвой четверкой. Где-то приходится идти на уступки, так как изменения происходят всегда. Выбор заключается лишь в том, собираетесь ли вы открыто сказать об изменении или попытаться скрыть его.

УЧЕНИК: Но как можно и рассказать об изменении, и продолжать следовать плану, не допускающему изменений?

МАСТЕР: Здесь воин должен показать все свое мастерство. Что, если будет достаточно просто продемонстрировать аудиторам «склад» старых историй, которые уже не входят в объем работ, чтобы стало ясно, какие из-

менения произошли в проекте? Так руководство получит возможность отслеживания различий между исходным и фактическим планом, при этом план не потеряет целостности, которая была у него на старте проекта.

УЧЕНИК: *Итак, Мастер, ты говоришь, что независимо от желаний клиента план изменится.*

МАСТЕР: *Именно так.*

УЧЕНИК: *А при простом документировании изменений разработчики, возможно, смогут соблюсти требования аудитора и при этом будут создавать продукт, устраивающий клиента.*

МАСТЕР: *Верно.*

УЧЕНИК: *Спасибо, Мастер. Я подумаю об этом во время медитации.*

Мораль урока: от изменений никуда не деться. Просто иногда нужно творчески подходить к представлению этих перемен и управлению ими.

Что дальше?

Мы хорошо поработали, дружище. За плечами осталась стартовая колода. Вы освоили искусство и науку создания пользовательских историй и оценки. Теперь вы знаете, как составить гибкий план работы из разнообразных элементов.

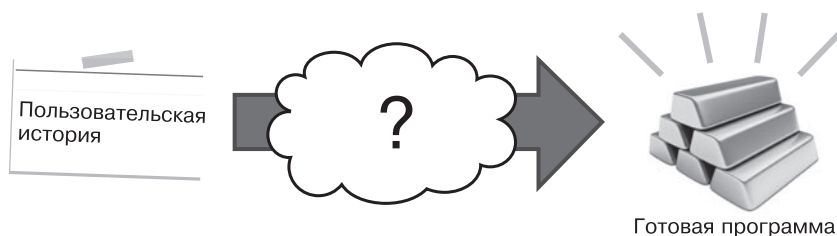
На очереди следующий отрезок пути — реализация гибкого проекта. В следующих главах будет рассказано, как благие намерения и планы становятся реальностью — рабочей, протестированной, функциональной программой.

И все начинается с самой обычной итерации.

Часть IV

Выполнение гибкого проекта

Управление итерациями



Добро пожаловать в часть IV! Здесь мы обратимся к планам, подготовленным в частях II и III, и превратим эти благие намерения во что-то, с чем смогут работать наши клиенты, то есть в готовые программы.

В данной главе, посвященной управлению итерациями, я покажу вам, что происходит за кулисами, и продемонстрирую, как потенциал, заложенный в итерациях, помогает решать задачи, встающие перед нами на гибких проектах.

Затем, в главе 10, будет рассмотрено, как функционирует обычная гибкая итерация. Там же будет рассказано о различных собраниях и точках синхронизации, с помощью которых гибкие команды обеспечивают работу проекта. Далее, в главе 11, я расскажу, как несколько простых изменений в офисе, где вы работаете, помогут вам достичь еще большей ясности и сосредоточения.

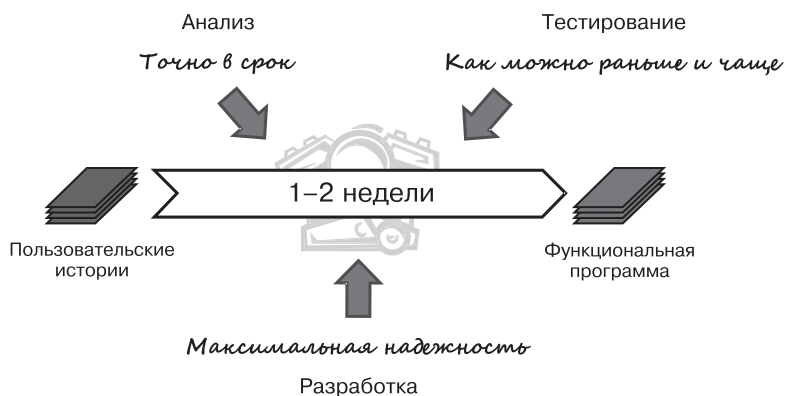
9.1. Как создавать что-то ценное каждую неделю

Итак, у вас есть план. Вы знаете, зачем собрались, и готовы работать. А что дальше? Как превратить карточку, на которой написано несколько слов, в готовую функциональную программу?

Во-первых, у вас не хватит времени, чтобы записать все подробности. Вам нужен простой и точный способ проведения анализа, применимый именно к тому материалу, который вас интересует, причем тогда, когда это требуется.

Во-вторых, применяемые вами способы разработки должны быть максимально надежными. Нет времени постоянно возвращаться и исправлять ошибки в коде. Он должен работать с самого начала. Это означает, что нужно создавать хорошо спроектированный, протестированный и полностью интегрированный код.

В-третьих, тестирование ни на шаг не должно отставать от разработки. Вы не можете ждать до окончания проекта и только потом проверить, все ли работает. Необходимо поддерживать целостность и исправность системы с самого первого дня проекта.

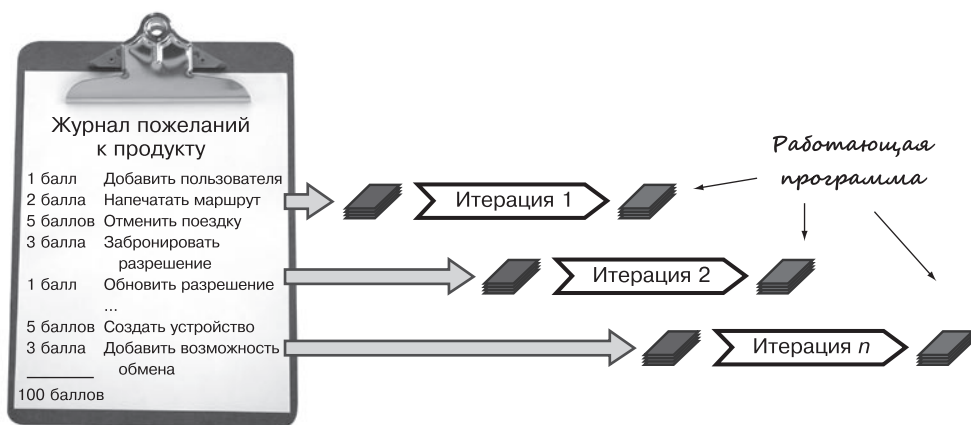


И если удастся соблюдать три этих правила, то каждую неделю у вас будет получаться какой-то положительный результат. А один из наиболее выгодных и правильных путей к работе в таком ключе — использование гибких итераций.

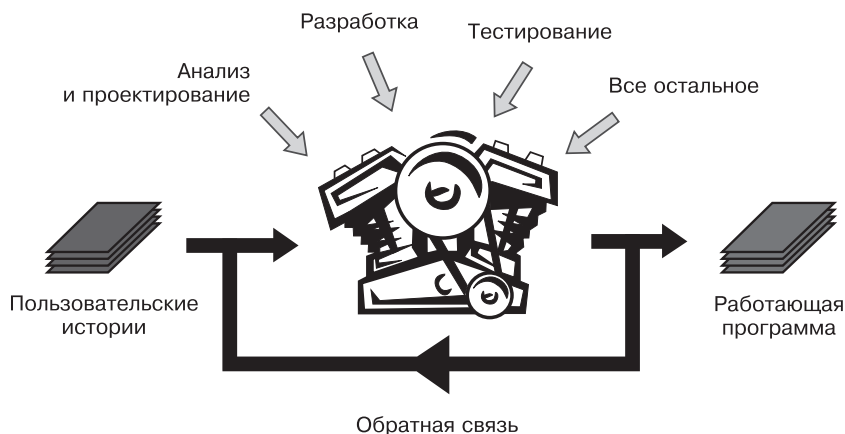
9.2. Гибкая итерация

На данный момент вы, вероятно, уже хорошо представляете себе, как выглядит гибкая итерация. Это ограниченный во времени период (недельный

или двухнедельный), в который мы берем основные истории наших клиентов и превращаем их в готовые программы.



Именно так достигается результат в гибком проекте. Наша цель — выдавать что-то полезное всякий раз, когда мы завершаем итерацию. Это означает, что мы любой ценой должны создать за итерацию рабочий, протестированный код.



Кроме того, итерации помогают при необходимости скорректировать курс. Если приоритеты изменяются либо случается какой-нибудь форс-мажор, мы можем изменить ход работы по окончании очередной итерации. Обычно истории не изменяются в ходе итерации (это было бы слишком разрушительно для команды). Как вы увидите в главе 10, при необходимости существует возможность изменить приоритеты прямо в ходе итерации.

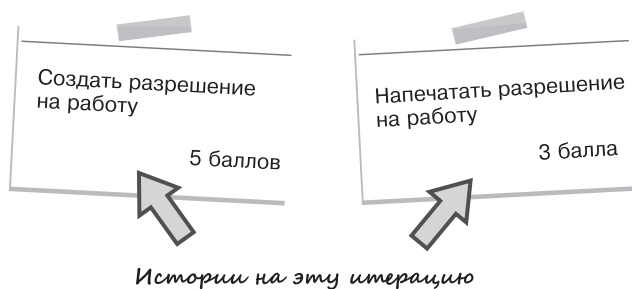
Но хватит разговоров. Лучший способ понять, как действует итерация, — увидеть ее на практике. Давайте возьмем пользовательскую историю и посмотрим, что нужно для превращения ее в готовую программу.

9.3. Требуется помощь

Помогите! Дата сдачи строительного проекта BigCo только что была перенесена на месяц назад, и вашему другу мистеру Келли нужен сайт, на котором подрядчики смогут оформлять разрешения по безопасности.



Разумеется, мы не сможем создать целый сайт всего за одну итерацию, но мистер Келли был бы очень признателен, если бы за следующие две недели мы могли реализовать две такие истории.



Чтобы все получилось, каждая пользовательская история проходит три этапа обработки перед тем, как станет программой.

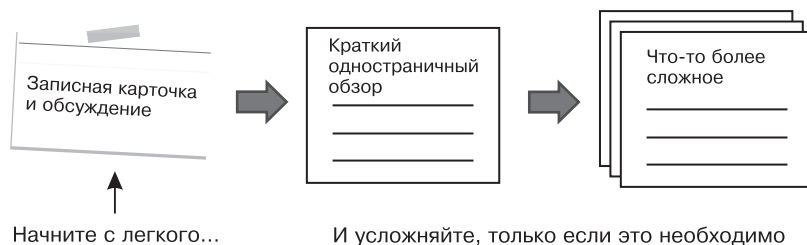
1. Анализ и проектирование (подготовка к работе).
2. Разработка (выполнение работы).
3. Тестирование (проверка работы).

Рассмотрим каждый из этих этапов более подробно.

9.4. Этап 1. Анализ и проектирование (подготовка к работе)

Гибкий анализ имеет две основные характеристики: «столько, сколько нужно» и «точно в срок». Под достаточным анализом понимается анализ всего того, что требуется для запуска работы, — ни больше ни меньше.

Выполняйте столько анализа, сколько требуется для работы



Небольшая команда, работающая в одном офисе, с которой сотрудничает представитель заказчика, не нуждается в большом количестве официальной документации. Гораздо лучше подойдет одностраничный обзор задач, разбивка на подзадачи и список необходимых критериев.

Название истории: создание разрешения на работу

Описание

Чтобы подрядчики могли на законных основаниях работать на месте строительства, им требуется разрешение на работу. Этот документ они берут с собой на место строительства, когда готовы приступить к работе.

Задачи

1. Создание страницы с образцом разрешения.
2. Сохранение разрешения в базе данных.
3. Обеспечение базовой валидации (проверки).
4. Вопросы безопасности игнорировать (на данном этапе).

Критерии проверки

1. Пользователь, заходящий на сайт, может сохранить образец разрешения.
2. Разрешение сохраняется в базе данных.
3. Неправильно заполненные разрешения отклоняются.
4. По умолчанию разрешение выдается со следующей недели.



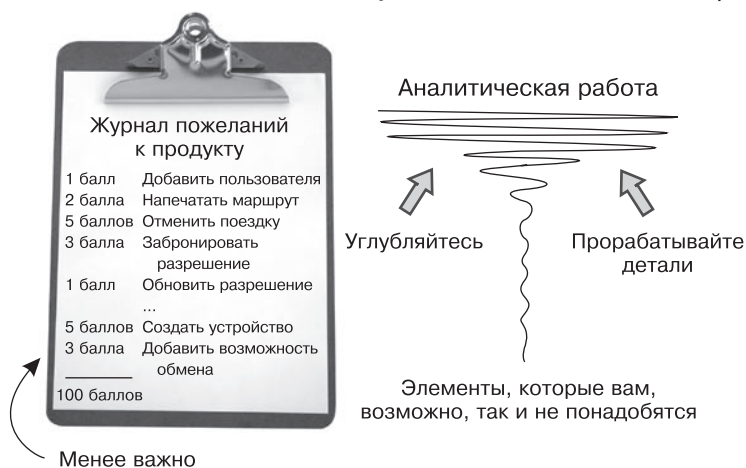
Если проект очень большой и выполняется удаленной командой, работающей в Чикаго, Лондоне и Сингапуре, то нам определенно потребуется нечто большее, чтобы все работали слаженно и двигались в нужном направлении.

Дело в том, что в гибком проекте нет какого-то однозначно подходящего уровня детализации. Существует лишь уровень, подходящий для вас и вашего проекта.

Позже в случае необходимости проект всегда можно усложнить, но если нести с собой ненужный лишний груз, то это вас только замедлит. Поэтому начинайте с легкого и усложняйте проект только при необходимости.

Еще одна ключевая характеристика гибкого анализа — «точно в срок».

Выполняйте гибкий анализ именно тогда, когда требуется



Под своевременным анализом понимается глубокий анализ пользовательской истории прямо перед тем, как вам потребуются его результаты (обычно — в ходе предыдущей итерации).

Здесь выполняется анализ...



историй, разработкой которых вы собираетесь заняться здесь



Мы ведь не можем с уверенностью сказать, как будут обстоять дела через месяц. Все меняется. Поэтому попытки забежать вперед и заранее спланировать все, что можно, обычно оборачиваются лишь крупной потерей времени. Лучше не приступать к глубокому анализу истории до того момента, пока этот анализ не станет необходимостью.

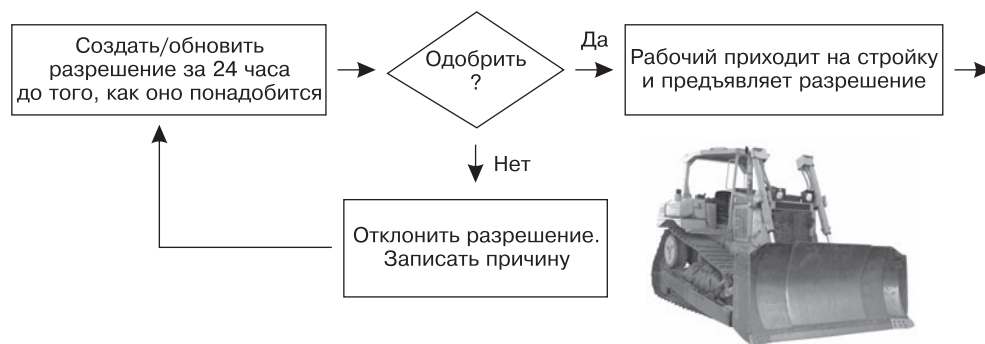
В случае работы подобным образом гарантируется следующее:

- ❑ при анализе учитывается только новейшая и ценнейшая информация;
- ❑ вы и клиент получаете возможность учиться и внедрять инновации по ходу работы;
- ❑ вам не приходится многое переделывать.

Если ваш проект действительно очень сложный и требует много времени на проработку — смиритесь с этим. Сделайте все, что необходимо для выполнения проекта. Просто не забегайте далеко вперед, поскольку все равно вам придется многое пересматривать, так как обстановка обязательно изменится.

Итак, давайте рассмотрим анализируемые компоненты истории «Создание разрешения на работу». Лучше всего начать с хорошей блок-схемы.

Начнем с хорошей блок-схемы



Блок-схемы очень хороши, так как они демонстрируют работу системы простым и наглядным образом. На них мы видим этапы, которые следует выполнить специалистам. Кроме того, на блок-схеме можно сделать пометки обо всем, что представляет интерес с точки зрения технологического процесса.

Кроме того, вы, возможно, лучше поймете пользователей вашей системы и то, как они собираются обращаться с *персоналиями*.

Теперь создадим несколько персоналий

Администратор



Аманда

Администратор должен уметь добавлять пользователей в систему и удалять учетные записи.

Требуется уверенный навык работы с компьютером.

Задача — обеспечение работы офиса (через Аманду распределяются все разрешения на работу для новых сотрудников-строителей)

Заказчик



Роберт

Руководитель строительства или инженер, который будет запрашивать разрешения по заявке сотрудника.

Детально разбирается в работе.

Отвечает за то, чтобы разрешения своевременно запрашивались

Утверждающее лицо



Мистер Келли

Отвечает за технику безопасности и предотвращение потерь.

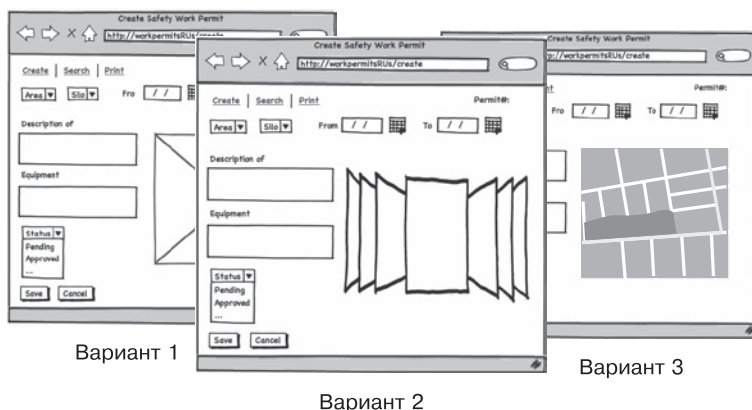
Обязан рассматривать все разрешения перед их выдачей.

Принимает окончательное решение о выдаче разрешения или об отказе в выдаче

Персоналии — это обычные описания ролевых стереотипов, характерных для людей, которые будут работать с вашей программой. Персоналии помогают немного «одушевить» систему. Мы начинаем видеть перед собой реальных людей со своими проблемами и понимать, как они работают и что им нужно.

Затем, когда мы приступаем к проектированию, вместо того чтобы просто замкнуться на первом проекте, который придет на ум, мы быстро делаем несколько моделей различных проектов и вариантов. Для этого используются дешевые и легко изготавливаемые бумажные прототипы.

*Быстро опробуйте несколько проектов,
пользуясь бумажными прототипами*



Прелесть сплочения вашей команды и совместного проектирования на бумаге заключается в том, что итог такой коллективной работы практически всегда оказывается лучше, чем результат творчества одного человека.

После разработки проекта можно встретиться с клиентом и записать определенные критерии тестирования, чтобы не осталось сомнений в том, как должна выглядеть успешно реализованная история.

*Затем определите успех, описав несколько
приемо-сдаточных испытаний
...на обратной стороне карточки*



*Запишите три функции, относящиеся к истории,
которые, по вашему мнению, подходят для ее тестирования
(если не знаете, то просто предположите)*

На данном этапе происходит встреча с клиентом, в ходе которой вы спрашиваете: «А как мы узнаем, что эта функция работает?»

Здесь вы можете обсудить детали настолько глубоко (или настолько поверхностно), насколько считаете нужным. Вы можете начать с самого важного

и убедиться, что команда понимает, какие основные функциональные элементы должны работать, чтобы выполненную историю можно было считать успешной.

Если данная история по природе очень технологична, содержит много бизнес-правил и деталей, возможно, вам понадобится дополнительное время на описание всех этих тонкостей (еще лучше, если вам удастся выразить их в форме какого-то автоматизированного теста).

А есть ли другие инструменты и методы, которые можно использовать при тестировании? Конечно! В вашем распоряжении — раскадровка, диаграммы параллельного исполнения (concurrency diagrams), карты процессов, каркасные представления (wireframes) и другие полезные средства, известные человечеству (другие идеи, связанные с анализом, описаны в разделе 6.4).

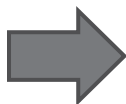
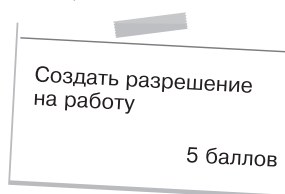
Помните, никого не учили в школе, как выполнять такую работу. Проявляйте творчество. Путь к цели далеко не один.

Кстати, совсем забыл рассказать, что стало с историей о распечатке (см. раздел 9.3). Оказалось, что она нам просто не нужна. Для первого релиза вполне достаточно распечатать разрешение из браузера, поэтому от отдельной функции печати мы отказались. Хорошо, что в свое время не затрачивали усилия на ее анализ!

Когда анализ закончен, можно приступать к работе.

9.5. Этап 2. Разработка (выполнение работы)

Здесь мы возьмем наш завершенный вовремя анализ и попытаемся сделать из него чистое золото (то есть в нашем случае программу, готовую к использованию).



```
if (userAccountExists)
    RedirectToLoginPage();
else
    DenyAccess();
```

Java, C#, Ruby, Python, HTML, CSS



*Материал, готовый
к развертыванию*

В наше время хорошие программы действительно могут цениться на вес золота. Для их создания требуется много работать, соблюдать жесткую дисциплину и достигать технического совершенства.

Несколько слов о парном программировании

Вряд ли найдется другой элемент гибкой разработки и экстремального программирования, который привлекал бы больше внимания и вызывал бы больше противоречий, чем парное программирование (pair programming).

Метод заключается в том, что двое разработчиков сидят у компьютера и вместе работают над пользовательской историей.

Разумеется, картина «два ценных специалиста сидят за одним компьютером и работают» заставит понервничать любого менеджера. Менеджер считает, что производительность команды при этом снижается вдвое, и это действительно так, если программирование сводится к обычному набору кода на клавиатуре.

Но парное программирование — не такое. Более того, эта идея часто избавляет команду от массы работы и переработок. Применяя парное программирование, вы организуете в команде обмен навыками и умениями, большее количество ошибок удастся обнаружить на ранних этапах, а качество кода повышается потому, что каждую строку проверяют два человека.

Такой режим подойдет не каждому, и, конечно же, нужно с пониманием относиться к тому, как люди привыкли работать. Но если ваша команда готова попробовать работать попарно (то же касается анализа и тестирования), то такой опыт чаще оправдывается, чем нет.

Например, в гибком проекте не обойтись без определенных вещей:

- ☐ нужно писать автоматизированные тесты;
- ☐ нужно постоянно улучшать и дорабатывать элементы проекта;
- ☐ нужно постоянно интегрировать код, чтобы получались готовые программы;
- ☐ нужно следить за тем, чтобы система была локализована под тот язык, на котором вы обсуждали ее с клиентом.

К сожалению, объем книги не позволяют обсудить все хорошие методы разработки программ, но то, о чем я собираюсь поговорить, условно назы-

вается *джентльменским набором* (то есть человек в здравом уме не может от них отказаться).

Далее, в частности в главах 12–15, мы подробно обсудим рефакторинг, разработку на основе тестирования (test-driven development) и непрерывную интеграцию и увидим, как все эти методы помогают получить код, готовый к практическому использованию.

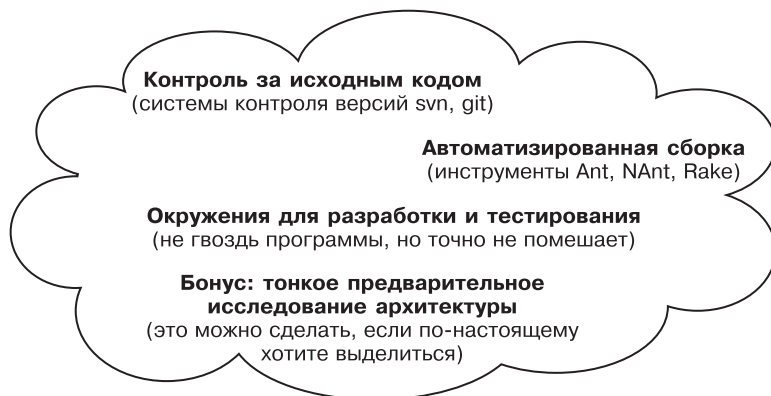
Пока лишь скажу, что никакая «магия» гибкой разработки не произойдет без опоры на серьезную работу по написанию кода, которая исключительно важна, пусть и остается на заднем плане.

Теперь давайте обратимся к самой необычной итерации вашего проекта — первой (ее также называют *нулевой итерацией*).

Начинаем с нулевой итерации

В зависимости от трактовки, итерацию можно считать либо началом, либо прологом вашей работы. В ходе ее происходит подготовка.

Если бы проект находился в разгаре, то в новой итерации мы бы просто углубились в работу над определенной историей (предварительно ее проанализировав). Но раз мы лишь приступаем к новому проекту, есть моменты, которые необходимо уладить перед началом работы.



*Задачи, которые обязательно нужно решить
перед тем, как приступать к работе над историейми*

Нулевая итерация сводится к приведению рабочей среды в порядок. В ходе ее, в частности, обеспечивается контроль версий, организуется автоматическая сборка, а также подготавливаются среды для разработки и тестирования

(а если возможно, то и среда для производства). На этом фоне мы можем приступить к внедрению продукта.

Если вы хотите по-настоящему щегольнуть, реализуйте в базовой версии одну из уже известных историй (которая затрагивает систему на всех уровнях и позволяет протестировать архитектуру).

После окончания разработки все, что нам остается, — это проверить выполненную работу.

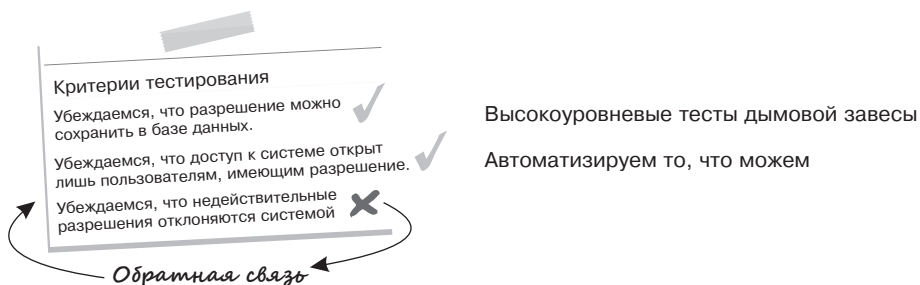
Коллективное владение кодом

В гибком проекте ни один отдельный участник не обладает правами на код. Код принадлежит команде. Это означает, что любому члену команды в любое время разрешено (и даже рекомендуется) вносить в код изменения, необходимые для завершения общей работы.

В экстремальном программировании эта практика называется *коллективным владением кодом* (collective code ownership). Она соответствует принципам, которые в гибкой разработке касаются обмена информацией, единообразия архитектуры и стандартов написания кода по всей базе кода.

9.6. Этап 3. Тестирование (проверка работы)

Было бы крайне неприятно, если в результате проделанной работы то, что у нас получилось, не функционировало бы. Тестирование — тот этап, на котором мы можем убедиться, что справились с поставленной задачей. И такой вывод нам позволит сделать реакция клиента.



При демонстрации программы клиенту можно затронуть критерии тестирования и при их рассмотрении показать, что программа работает. Еще лучше, если вы позволите клиенту попробовать поработать с демонстраци-

онной версией, а сами посидите рядом и посмотрите, как *он* использует программу.

Знаю, о чем вы подумали. С учетом всего тестирования, которым пронизан гибкий проект, нуждаемся ли мы в пользовательских тестах приемки (user acceptance test, UAT) перед сдачей программы в эксплуатацию? Ответ — да, нуждаемся. И вот почему.

Вы как гибкий разработчик (и как любой другой член команды) хотите свести значение приемочных тестов к минимуму. То есть вы сами проводите тестирование так хорошо и получаете при разработке настолько исчерпывающую отдачу от клиента, что, когда дело доходит до приемочных тестов, пользователю действительно очень сложно найти в системе какие-либо огрехи.

Немногим командам удастся достигать при работе такого уровня качества, особенно на первом проекте (а некоторые не достигают его никогда). Поэтому мой совет — не отказываться от приемочного тестирования, пока вы не докажете спонсорам и сами себе, что вы и ваша команда способны писать код такого качества, что в официальных, полномасштабных приемочных тестах нет необходимости. До тех пор такое тестирование не помешает.



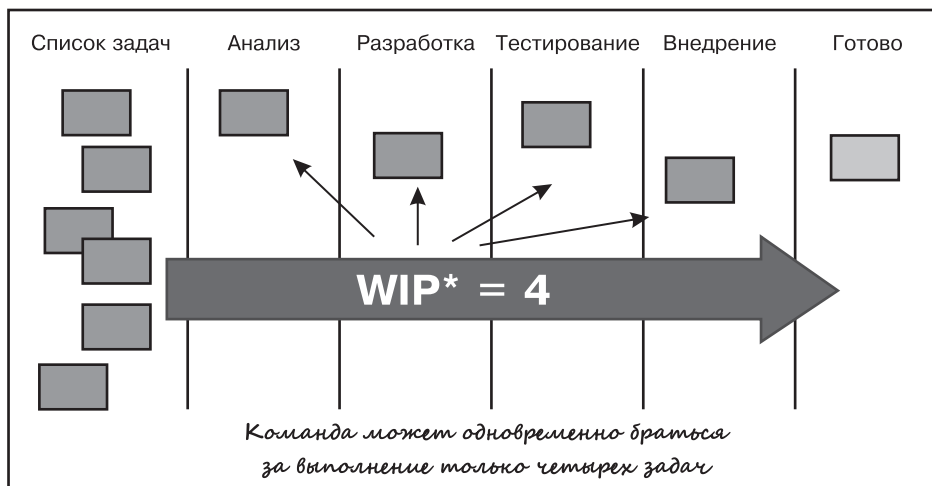
Конечно. Существует стиль гибкой разработки, лучше подходящий для такого стиля работы — в частности, для технической поддержки. Такая разновидность гибкой методологии называется *канбан* (Kanban).

9.7. Канбан

Канбан — это основанная на использовании карточек сигнальная система, разработанная компанией «Тойота» для координации заказа запасных частей. Она применяется на линиях поточной сборки. Эта система очень

напоминает привычную раскадровку, но отличается от нее несколькими ключевыми характеристиками.

Пример канбан-доски



Итерация не имеет фиксированной длины

История/задача не имеет фиксированной длины

* WIP (Work in Progress) — работа, выполняемая в данный момент

Что, если вам не разрешат отслеживать ошибки?

Предположим, нам больше не разрешено отслеживать ошибки в собственных проектах. Что делать, чтобы изменить ситуацию в свою пользу?

Во-первых, ошибки придется устранять прямо на месте, так как отслеживать их мы не сможем.

Во-вторых, нам понадобится быстрый и дешевый способ регрессивного тестирования, чтобы гарантировать, что найденная ошибка устраняется раз и навсегда и уже никак не проявится в вашей системе.

В-третьих, если ошибок наберется много, нужно остановиться и выяснить, что вызвало такие большие проблемы, а затем устранить их основную причину.

Такое отношение к работе и соответствующую практику нужно привить в своей команде. Мы не обсуждаем, какой системой отслеживания ошибок вы будете пользоваться. Речь о том, чтобы научиться писать программы таким образом, чтобы вам вообще не потребовалось отслеживать ошибки.

Работа в канбане ограничивается концепцией, называемой *«работа, выполняемая в данный момент»* (work in progress, WIP). В каждый момент времени команде разрешено синхронно заниматься решением ограниченного количества задач.

Например, если команда может одновременно работать над четырьмя задачами, показатель WIP этой команды становится равен 4. Все остальное, что требуется сделать, откладывается на потом, но с расстановкой приоритетов, и работа над этим остальным начинается только тогда, когда решены более насущные проблемы.

Еще одна характерная черта канбана заключается в том, что он не требует итераций. Можно просто переходить к следующей по важности задаче из списка по мере готовности.

Цель канбана — поточное производство. Требуется выполнять задачи, перечисленные на доске, настолько быстро, насколько это возможно при одновременной обработке нескольких задач. Перечислю несколько достоинств такого принципа работы.

- ❑ Вам не приходится волноваться об итерациях. Если вы разрываетесь между рутинной деятельностью и проектом, то при использовании канбана можете не волноваться о том, что ваша работа будет прервана посреди итерации (например, из-за проблем, связанных с технической поддержкой), так как итерации как таковые отсутствуют. Когда вы справитесь с текущими делами и вернетесь к проекту, то просто перейдете к следующей задаче, и вам не нужно будет корректировать прогнозы, связанные с итерациями.
- ❑ В процессе работы вы не ограничены выполнением задач, относящихся только к одной итерации. Хотя в целом было бы правильно разбивать крупные задачи на более мелкие подзадачи, так как в некоторых случаях задача может быть слишком велика и потребуются пара недель, чтобы она прошла весь путь по доске канбана слева направо. Но ничего страшного.
- ❑ Это удобный способ управления ожиданиями. Работая по принципу канбана, большинство команд все же занимается оценкой в той или иной форме, соизмеряя друг с другом задачи, описанные на канбан-доске (как минимум относительная оценка рекомендуется, если вы хотите обрисовать, как собираетесь достигать своей цели).

Но в этом отношении канбан характеризуется очевидной простотой. Ситуация примерно такова, как если бы вам сказали: «Понимаешь, дружище, у нас тут дел невпроворот. Мы бы не отказались взяться за твой проект, но работать сможем только над четырьмя вещами одновременно». Никаких баллов. Нет оценок, которые нужно объяснять. Проза жизни. Одновременно мы можем заниматься только таким-то количеством задач.

И если сейчас вам все это кажется безумием (хотя бы потому, что на протяжении всей книги мы говорили о том, как великолепны итерации), не волнуйтесь.

Гибкие итерации очень сильны сами по себе, и если вы выполняете проектную работу, основанную на определенных ограничениях (в частности, на фиксированных сроках и бюджете), то в нынешнем мире поточного производства с ежегодными бюджетами итерации — это метод что надо.

Но гибкая разработка не сводится к одним только итерациям. Проявлять гибкость — означает делать все, что может принести пользу. Так, если вам больше подходит работа без итераций, то работайте без них. Канбан отлично годится для команд, занимающихся обслуживанием оборудования или технической поддержкой, которые должны быстро реагировать на изменяющуюся обстановку и не могут позволить себе такой роскоши, как итерации с фиксированной длительностью.

Я все же советую работать с итерациями. Если вы только начинаете деятельность и ваша работа представляет собой проект, то вам пойдут на пользу дисциплина и строгость, которые развиваются в условиях регулярного предоставления клиенту готовых программ, неделя за неделей.

Но если вы занимаетесь рутинной работой, попробуйте канбан. Принципиально он не отличается от обычной гибкой разработки. Немного разнится только ход работы.

Подробнее познакомиться с новейшими и наиболее важными фактами, связанными с канбаном, можно на сайте <http://finance.groups.yahoo.com/group/kanbandev/messages>.

А теперь вы готовы к новой встрече с сэнсэем.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, я работаю над проектом, связанным с созданием хранилищ данных. Наша команда занята составлением финансовых отчетов для высшего руководства. Мы никак не можем производить что-то новое и ценное каждую неделю. На одно только создание хранилища данных уходит около месяца. Как мне использовать при работе итерации?*

МАСТЕР: *Чтобы регулярно выдавать что-то ценное, нужно сосредоточиться на небольших функциональных фрагментах, которые затрагивают приложение на всех уровнях. Вместо того чтобы от начала и до конца выстраивать хранилище данных, возьми небольшой подраздел одного из твоих отчетов и выстрой только те элементы инфраструктуры, которые требуются в этом фрагменте.*

УЧЕНИК: *Но что, если, сделав это, мы столкнемся с настолько большой задачей, что она никак не уместится в одной итерации?*

МАСТЕР: *Ну, заладил: «Не уместится, не уместится»! Возьмите столько итераций, сколько вам требуется для создания инфраструктуры, и двигайтесь дальше. Просто не забывайте, что вас интересует участие клиента. А если ему сказать, что вы собираетесь исчезнуть на три месяца и заняться настройкой хранилища данных, клиент потеряет интерес. И для вас, и для него будет гораздо лучше, если вы найдете способ выполнения работы маленькими фрагментами и постройте процесс в форме итераций.*

УЧЕНИК: *Спасибо, Мастер. Я подумаю над этим.*

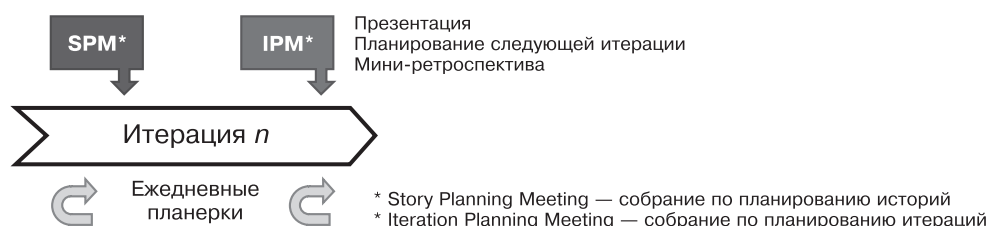
Что дальше?

Вот и все. Анализ, разработка и тестирование, сплетены воедино, чтобы каждую неделю ваша работа давала ощутимый результат. Помните, эта работа может выполняться несколькими способами, и артефакты, а также способ вашей работы обязательно будут изменяться от проекта к проекту. Поэтому не бойтесь экспериментировать и применять разные способы.

Разобравшись со всем этим, мы готовы поговорить о коммуникации в гибких командах и координации действий, разворачивающихся одновременно в ходе итерации.

Глава 10

Создание гибкого плана коммуникации



Кроме того, что нужно обустроить пространство для совместной работы и регулярно выдавать готовые программы клиенту, гибкая разработка практически не содержит указаний о том, как организуется работа в ходе итерации. Такой организационной работой занимаетесь вы и ваша команда, для этого вы общаетесь, консультируетесь друг с другом и вместе налаживаете рабочий процесс.

В этой главе мы поговорим о важнейших аспектах любого гибкого плана, обеспечивающих обмен информацией в команде, а также о том, как наладить комбинацию, которая действительно поможет в работе вам и вашим сотрудникам.

К концу этой главы у вас не только будет план, но и зародятся определенные ритм и ритуал, которые позволят регулярно создавать в ходе проекта ценные программы.

10.1. Четыре вещи, которые необходимо сделать в ходе любой итерации

Две неотъемлемые характеристики гибкого проекта — это формирование ожиданий и обеспечение отдачи со стороны клиента.

Непрерывное формирование ожиданий необходимо потому, что обстановка все время будет меняться. Регулярные встречи с клиентом должны войти в привычку, как и постоянный пересмотр проекта.

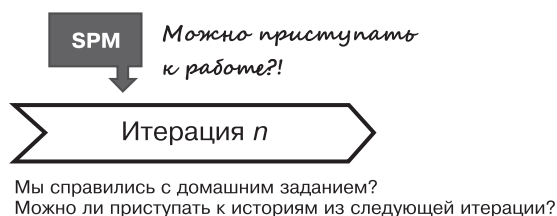
Поскольку простой акт передачи клиенту полезных программ изменяет требования, вы, конечно же, хотите обеспечить сильный «контур связи», чтобы гарантировать, что вы движетесь прямо к цели.

Есть четыре вещи, которые потребуется сделать, чтобы каждая итерация приобрела свой ритм и характерный ход:

- ❑ убедитесь, что подготовлена работа на следующую итерацию (проведено собрание по планированию историй);
- ❑ получите реакцию на истории, выполненные в ходе последней итерации (презентация);
- ❑ спланируйте, как будет протекать следующая итерация (планирование итерации);
- ❑ постоянно ищите способы оптимизации работы (мини-ретроспектива).

Сначала рассмотрим, как убедиться в том, что подготовка к следующей итерации закончена.

10.2. Собрание по планированию историй



Это собрание служит точкой отсчета для нашего динамического анализа. В ходе собрания по планированию историй (SPM) мы пересмотрим критерии тестирования для предстоящих историй вместе с клиентом, заново рассмотрим оценки с разработчиками и убедимся, что справились с домаш-

ней работой над следующей партией историй, которые будут реализовываться в ходе итераций.

Без ошибок не обойтись, так что не заморачивайтесь на них!

Как-то раз я занимался историей, связанной с функцией распечатки документов на строительном проекте. Я избрал спартанский путь — решил создать минимальную реализацию, пригодную для работы. Как только я ее продемонстрировал, стало ясно, что клиента она не устраивает.

Из вежливости клиент мне ничего не сказал, но я почувствовал его разочарование и понял, что это далеко не лучшая моя работа.

Но в тот момент мне пришлось это проглотить, и я спросил заказчика, могу ли попытаться переделать работу. Он согласился.

Если бы в течение нескольких недель перед презентацией я не работал в поте лица, то ответ мог бы быть другим. Но когда клиент видит, что вы вкалываете на него каждую неделю, то прощает вам некоторые промахи, которые могут случиться с каждым.

Поэтому не бойтесь пробовать. Метод проб и ошибок и взятие на себя инициативы — все это части одной игры.

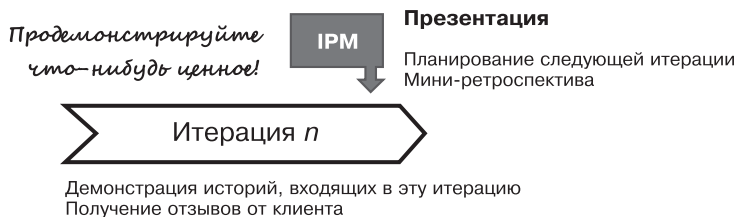
Иногда оказывается, что история больше, чем вы полагали на первый взгляд. Ничего страшного. Просто разбейте ее на фрагменты, которые будут помещаться в отдельные итерации, уточните план и продолжайте работать. Хорошая новость заключается в том, что бывает и обратная ситуация (некоторые истории оказываются меньше, чем мы думали).

Ни в одном методе гибкой разработки собрания по планированию историй специально не описываются. Они просто являются одним из способов, который показался полезным мне и другим специалистам, так как позволяет избежать трат времени, возникающих, когда итерация начинается с непроанализированной истории.

Но в этом и заключается красота гибкой разработки. Любую задачу можно решить разными способами. Если вам что-то нужно, сделайте это сами (несмотря на то что пишут в разных книгах).

Еще одна деталь, без которой не должна проходить ни одна итерация, — это реакция клиента.

10.3. Презентация



У вас получилось! Вы сделали что-то ценное. Вы знаете, сколько проектов длятся неделями, месяцами, а иногда и годами без получения какого-либо полезного результата? Много.

Презентация (showcase) — это возможность похвастаться той работой, которой занимались вы и ваша команда, и получить определенные отзывы клиента, сказанные честно и откровенно.

В ходе презентации вся команда демонстрирует истории, выполненные в ходе последней итерации. То есть вы показываете реальный действующий код, развернутый на тестовом сервере. Это не просто картинки или конструктивные намерения. Это творение, которое можно использовать на практике, и работа при необходимости может начаться прямо с сегодняшнего дня.

Предполагается, что презентации должны быть интересными, и вообще, они представляют собой отличный вариант завершения разработок по конкретной итерации. Отпразднуйте это! Принесите печенье или конфет. Хвастайтесь. Выслушивайте отзывы. Позвольте клиенту поэкспериментировать с демоверсией и посмотрите, как он работает с программой.

Далее поговорим об одном из видов собраний, которые рекомендуется проводить, занимаясь скрамом или экстремальным программированием. Речь пойдет о собрании по планированию итерации (IPM).

10.4. Планирование следующей итерации



Именно на собрании по планированию итерации вы встречаетесь с клиентом и обсуждаете, какую работу нужно выполнить за следующую итерацию. Вы пересматриваете скорость работы команды, изучаете новые поступившие истории, после чего команда может переходить к выполнению следующей итерации.

Кроме того, собрание по планированию итерации отлично подходит для экспресс-проверки состояния проекта.



Ясно, безоблачно

- Все идет гладко, без проблем.
- Ничто не замедляет работу.
- Дела идут как нельзя лучше



Облачно, возможен дождь

- Работаем.
- Испытываем небольшие проблемы.
- Но ничего такого, с чем бы мы не могли справиться

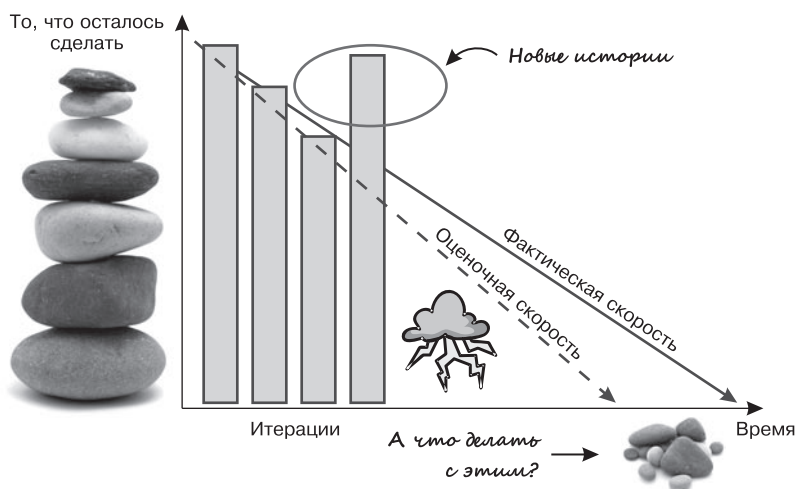


Гроза

- Хьюстон, у нас проблемы¹.
- Предстоят большие сложности.
- На помощь!

Здесь можно быстро сделать «прогноз погоды», касающийся протекания проекта. Если вам что-то нужно или имеется особо сложная проблема, требующая обсуждения, то именно здесь вы можете ее озвучить, предложить варианты решения и порекомендовать, как, с вашей точки зрения, следует работать далее.

Когда разговор заходит о датах, используйте график прогресса разработки. Он абсолютно честен и сух, и по нему клиент может оценить, насколько реалистично выглядят планируемые сроки.



¹ Считается, что это сообщение появилось в связи с серьезной аварией на корабле «Аполлон-13». — Примеч. пер.

Это видимая сторона гибкой разработки. Нужно вести себя максимально открыто по отношению к клиентам и владельцам. Один из принципов гибкой разработки — сообщать плохие новости как можно раньше.

Как получаются конструктивные отзывы

Существуют два способа налаживания коммуникации. Можно подать ее прямо и холодно:

«Сьюзи, я заметил, в последнюю итерацию ты хорошо поработала над модулем печати, но твоё тестирование компонентов никуда не годится».

Или можно выразиться дипломатичнее:

«Сьюзи, ты классно поработала над модулем печати! Если бы твоё тестирование компонентов было таким же подробным, ты скоро стала бы специалистом мирового класса».

Чувствуете разницу? Можно совершенно изменить и тон, и наполнение такого сообщения.

Я не призываю вас переслащивать такие моменты. Но иногда, немного изменив слова, можно сильно повысить качество работы.

Эффективная коммуникация подробно описана в классической книге Дейла Карнеги «Как завоевывать друзей и оказывать влияние на людей» [Car90].

Последнее, что требуется сделать перед окончанием любой итерации, — спросить себя, есть ли вещи, которые могли бы получиться у нас лучше?

10.5. Как выполнить мини-ретроспективу



Принцип гибкой разработки

С определенными интервалами команда обсуждает, как повысить эффективность работы, а затем соответствующим образом корректирует свою деятельность.

Ретроспектива может быть большим и пышным мероприятием на целый день и проводиться в конце крупного релиза или ближе к окончанию проекта. Но я хочу поговорить не об этом.

Такие ретроспективы представляют собой быстрые адресные решения, требующие от 10 до 15 минут. На ретроспективах вы регулярно собираетесь с командой и обсуждаете, что у вас получается наиболее хорошо, а какие участки работы можно было бы и улучшить.

Первое железное правило для проведения хорошей ретроспективы — убедиться, что все справляются со своими задачами. Если такого ощущения достичь не удастся, нужно сослаться на основное правило проведения ретроспективы и напомнить людям, для чего состоялась эта встреча.

Основное правило ретроспективы

Независимо от того, что мы обнаружим, мы понимаем (и действительно верим), что все выложились по максимуму, учитывая информацию, которой располагал при работе каждый специалист, его навыки и способности, доступные ресурсы и сложившуюся ситуацию

Иными словами, это не охота на ведьм



Затем можно начать разогрев, задав первый вопрос ретроспективы.

1. Что у нас получается по-настоящему хорошо?

«Джимми, классно ты поработал над тестированием компонентов!»

«Сьюзи, могу только восхититься тем, как ты сделала руководство по стилю и переработала таблицы стилей. Благодаря твоей работе все разделы приложения выглядят согласованно и единообразно».

Подчеркивая достижения и хваля людей, которые заслуживают признания, можно воодушевить их и стимулировать именно такой подход к работе, который мы желаем видеть в наших проектах.

В другой части этого уравнения находятся показатели, которые мы можем улучшить.

2. Что нам требуется делать лучше?

«Народ, при реализации последней партии историй допущено много ошибок. Давайте немного сбавим темп, подтянемся и убедимся, что уделяем достаточно внимания тестированию компонентов».

«В последней базе кода было много дубликатов. Не забывайте при работе отводить время на рефакторинг кода».

«Я полностью запарол эту историю с функцией печати. Разрешите, я попытаюсь переделать ее в следующей итерации — обещаю, следующая версия будет гораздо лучше».

В чем бы ни была проблема, организация ретроспективы и обмен мыслями с товарищами по команде — отличный способ приободриться и подстегнуть команду в тех областях, где требуется поддержка. Затем можно будет сформулировать тему для ближайших итераций, подчеркнуть и отследить те области, которые вы хотели бы улучшить.

Подробное руководство по организации ретроспектив дается в книге *Agile Retrospectives: Making Good Teams Great* [DL06].

Отлично. Теперь немного поговорим о ежедневной планерке, во время которой мы быстро приводим все планы к общему знаменателю.

10.6. Как не нужно проводить планерку

*Быстрая ежедневная синхронизация,
позволяющая прийти к согласию*



Координация действий с остальными членами команды
Короткая (менее 10 минут)
Участники не должны сидеть

Ежедневная планерка нужна для быстрого обмена важной информацией с командой. Это собрание, которое должно стать первым и последним. Оно длится 5–10 минут, и на нем никто не сидит (чтобы напомнить людям, что планерка должна быть краткой). Как правило, в ходе планерки вы уточняете, над чем нужно поработать, и сообщаете все, о чем, по вашему мнению, следует знать остальной команде.

В большинстве пособий по гибкой разработке указано, что при проведении планерки нужно встать в круг и попросить каждого из членов команды сказать:

- ❑ что он делал вчера;
- ❑ что он собирается делать сегодня;
- ❑ есть ли что-нибудь, замедляющее его работу.

Что ж, это небесполезная информация. Но она не слишком вдохновляет. Вместо этого попробуйте встречаться с командой в начале каждого рабочего дня и рассказывать о следующем:

- ❑ что вы вчера сделали, чтобы изменить мир к лучшему;
- ❑ как вы собираетесь справиться с делами сегодня;
- ❑ как вы намерены преодолеть все преграды, которые оказались у вас на пути.

Ответы на эти вопросы полностью меняют динамику планерки. Вместо того чтобы просто постоять вместе и уточнить положение дел, вы выкладываете все начистоту и сообщаете миру о своих намерениях.

После этого может произойти одна из двух вещей: либо вы сделаете все, что нужно, и достигнете результата, либо нет. Это полностью зависит от вас.

Но могу вас заверить: если каждый день вы будете приходить на работу и открыто говорить коллегам, чего собираетесь достичь сегодня, это радикально повысит ваши шансы на успех.

10.7. Делайте все, что считаете целесообразным

От вас зависит и ответ на вопрос, должны ли все эти собрания быть отдельными или их можно объединить?

Чтобы свести количество собраний к минимуму, некоторые команды предпочитают совмещать презентацию, планирование следующей итерации и ретроспективу в одно мероприятие и проводить все это примерно за час (я работаю именно так, и это показано выше, в разделе о собрании по планированию итерации (IPM)).

Другие считают, что нужно разделять планирование и презентацию, а ретроспективу проводить с развлекательным элементом ближе к концу недели.

Иногда команды налаживают такой хороший контакт с клиентом, что они даже не проводят специальных собраний по планированию историй (SPM). Они просто беседуют каждый день, а при необходимости организуют сессии по проектированию.

Помните: для выполнения данной работы также существует несколько способов. Если что-то не идет на пользу проекту, откажитесь от этого. Попробуйте другие варианты и посмотрите, что вам подходит.

Просто нужно гарантировать, что в определенный момент в ходе итерации у вас будет возможность встретиться с клиентом, показать ему работающую программу, сделать очередной прогноз и искать способ улучшить эту ситуацию.

Ой-ой. Кажется, сэнсэй хочет проверить, весь ли материал вы усвоили. Идите лучше к нему в додзё и проверьте, имеет ли все это смысл. Удачи!



*Мастер-сэнсэй
и воин,
постигающий
искусство*

С возвращением, ученик. Я подготовил три задания, чтобы проверить твоё умение на нескольких реальных сценариях, которые могут случиться при итерациях. Прочти внимательно каждое задание, прежде чем отвечать.

Сценарий 1. Неполная история

МАСТЕР: Однажды на собрании по планированию итерации выяснилось, что одна из историй готова лишь наполовину. Молодой проект-менеджер хотел продемонстрировать прогресс, поэтому предложил учесть половину баллов одной истории при расчете скорости команды в данной итерации, а остальные баллы причислить уже к результатам следующей итерации. Хорошая ли это идея?

УЧЕНИК: Пожалуй, если половина истории действительно выполнена, не вижу ничего плохого в том, что менеджер приплюсовал ровно половину баллов этой истории к завершённой итерации и учёл это при расчете скорости команды, а остальные баллы учёл в следующей итерации, когда история была полностью завершена.

МАСТЕР: *Так ли это? Ответь мне. Может ли крестьянин везти рис на телеге с одним колесом? Может ли человек есть только одной палочкой? Может ли клиент воспользоваться функцией, если она готова лишь наполовину?*

УЧЕНИК: *Нет, Мастер.*

МАСТЕР: *Для адепта гибкой разработки пользовательская история не бывает выполнена наполовину, на три четверти или на четыре пятых. История либо готова, либо нет. Поэтому при расчете скорости конкретной итерации учитываются только полностью готовые истории. Все незавершенные истории переносятся на следующую итерацию.*

Сценарий 2. Ежедневные планерки не помогают

МАСТЕР: *Я знаю одну команду, руководитель которой боролся за то, чтобы все сотрудники присутствовали на ежедневных планерках. Члены команды думали, что планерки особо не нужны и лучше сразу приступать к работе, а при необходимости советоваться друг с другом. Как поступить команде?*

УЧЕНИК: *Руководитель команды должен напомнить всем о том, как важно, чтобы работа шла согласованно, и как необходимы для этого ежедневные планерки.*

МАСТЕР: *Да. Команда может пересмотреть цели ежедневной планерки, а также уточнить, зачем она нужна в первую очередь. Но что, если, несмотря на это, команда ощущает ненужность планерок?*

УЧЕНИК: *Не вполне понимаю тебя, Учитель. Как быстрая ежедневная ориентировка и настройка всех на общий лад может казаться тратой времени?*

МАСТЕР: *Несмотря на все достоинства, которыми обладает хорошая ежедневная планерка, это всего лишь один из возможных способов. Если вся небольшая команда работает в одном офисе и все ее члены тесно сотрудничают на протяжении проекта друг с другом и с клиентом, ежедневная планерка вообще может оказаться ненужной.*

УЧЕНИК: *Вы хотите сказать, что некоторым командам ежедневная планерка не требуется?*

МАСТЕР: *Я хочу сказать, что команда должна продолжать проводить планерки, если видит в этом смысл. Если польза не ощущается — нужно изменить формат планерки или вообще отказаться от нее.*

Сценарий 3. Итерация, за которую не было создано ничего ценного

МАСТЕР: *Однажды команда прошла целую итерацию, но не смогла по ее завершении выдать какой-либо ценный результат. Неудача была полностью на их совести. Они не смогли разработать план, слишком поздно начали и вообще были ленивы. Зная, что им предстоит преподнести клиенту дурную новость, они отменили презентацию. Мудро ли они поступили?*

УЧЕНИК: *Хотя я и могу себе представить, как команда могла бы с достоинством выйти из ситуации, пусть ей и не удалось предъявить никакого ценного результата, мне кажется, что если им действительно было нечего показать, то отмена презентации была оправданна. Но лучше бы они честно рассказали, почему возникла такая ситуация.*

МАСТЕР: *О, ученик, ты хорошо схватываешь. Иногда случается, что тебе не удастся сделать за итерацию чего-то ценного, но обычно ненамеренно или из-за лени. Как команда может побороть такую лень?*

УЧЕНИК: *Ты полагаешь, Мастер, презентацию не следовало отменять? А следовало встретиться с клиентом и признать, что предъявить ему нечего?*

МАСТЕР: *Вот именно! Иногда угрызения совести — лучший урок. Встретившись с клиентом и ощутив, каково это — ничего ему не показать, — ты узнаешь, как велико отвращающее действие такой встречи. Пережив стыд, хорошая команда приложит все усилия, чтобы подобное больше не повторилось.*

УЧЕНИК: *Спасибо, Мастер. Я обдумаю твой урок.*

Не пытайтесь сбежать от неприятных ситуаций в вашем проекте, если они произойдут. Иногда они могут многому вас научить. Признавайте свои ошибки, делитесь горьким опытом с другими и двигайтесь дальше.

Что дальше?

Имея готовый план коммуникации и хорошо понимая, как происходит итеративная разработка, мы переходим к изучению того, как хорошие гибкие команды могут подняться над собой, когда нужно вложиться в работу.

Далее мы изучим секреты визуального оформления рабочего пространства и поговорим о том, что нужно делать, чтобы команда все время оставалась энергичной и сосредоточенной.

Визуальное оформление рабочего пространства



Расписания авиарейсов просто великолепны. Достаточно беглого взгляда на такое расписание — и уже понятно, какой самолет прибывает, какой отправляется, а какой рейс вообще отменен.

Почему бы не сделать такое же расписание для своего проекта?

Научившись создавать визуальное оформление рабочего пространства, вы и ваша команда никогда не забудете, что делать дальше или где можно максимально повысить эффективность работы. Такой подход принесет в работу большую ясность и сосредоточенность. В свою очередь, полученная прозрачность поможет вам делать прогнозы с достаточной уверенностью.

И раз уж речь зашла о таких схемах, вот они.

11.1. В бой вступает тяжелая артиллерия!

На предприятии шла коренная реорганизация. Бюджеты урезались. Сроки реализации проектов сокращались. Все процессы должны были стать лучше, быстрее и дешевле.

В результате вас просили делать больше с меньшими усилиями. Отдел менеджмента требовал, чтобы вы выполняли то же количество работы, что и раньше, но оставили для этого только половину команды, причем с месячным опережением сроков. Иначе...

Такой распорядок должен был установиться раз и навсегда, и завтра планировалось собрание, на котором вы должны были подтвердить, что согласны с новым планом.

Брр! Что же делать? Выдвигаемые требования совершенно безумны. Вы это знаете. Команда это знает. Кажется, что ситуацию не понимает только начальство.

Что же делать, чтобы доказать, что вы и сами очень бы хотели создавать прежний объем материала, располагая всего половиной ресурсов, но это попросту невозможно?

Организация брифинга с управленцами

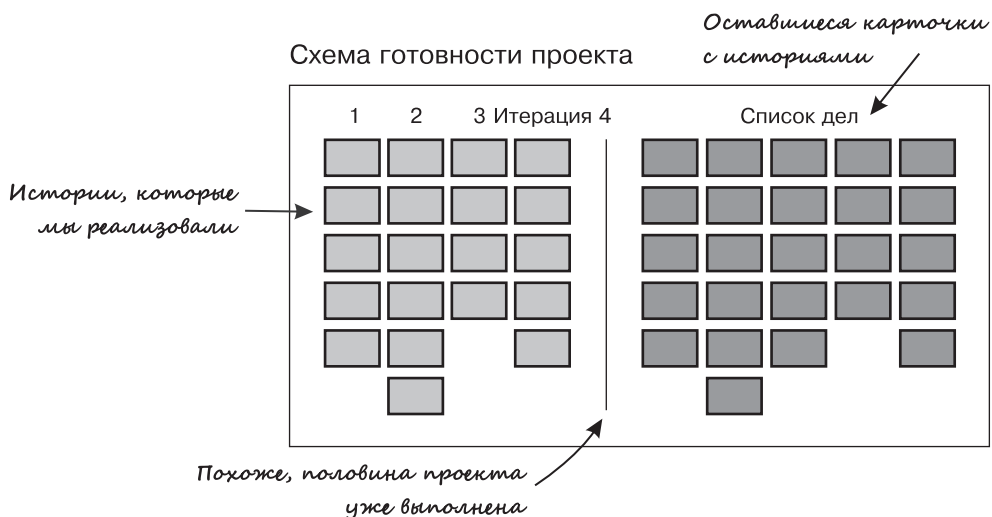
Не нужно организовывать официальное собрание и защищать свою точку зрения с помощью презентации PowerPoint. Гораздо лучше пригласить менеджеров прямо на место работы и позволить им самостоятельно убедиться, в каком состоянии находится проект.

Сначала вы показываете гостям стартовую колоду проекта — карточки теперь красиво наклеены на стену.



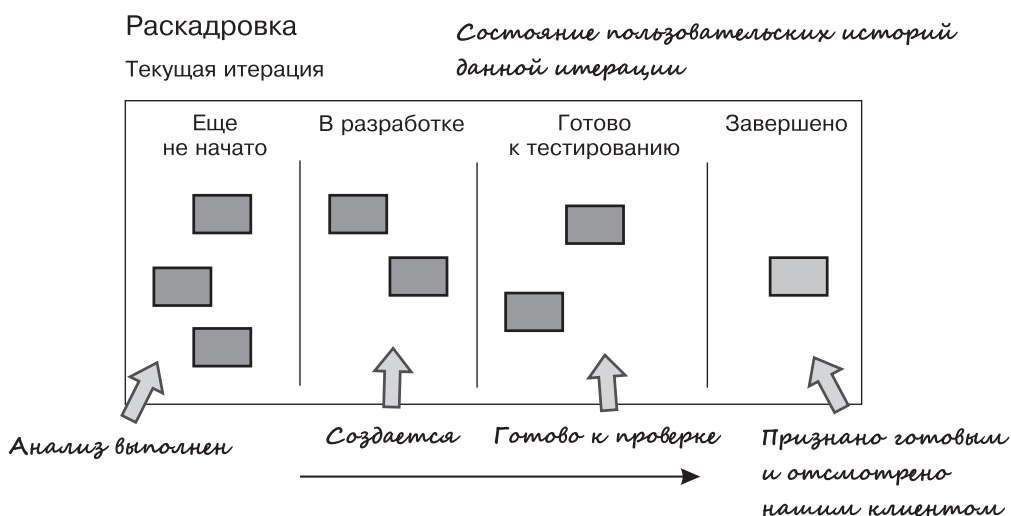
Вы объясняете, что стартовая колода — это инструмент, с помощью которого вы и команда всегда в курсе, на какой стадии находится реализация проекта. Благодаря наглядности этой схемы вы не забываете, кто заказчик проекта, чего требуется достичь и, самое важное, почему клиент пожелал потратить деньги на этот проект в первую очередь.

Впечатленные менеджеры подходят поближе и спрашивают, на каком этапе проекта вы находитесь. Чтобы ответить на вопрос, вы обращаете их внимание на *схему готовности* (release wall).



По схеме готовности вы и команда отслеживаете, что уже сделано и что осталось сделать. В левой части схемы находятся функции, которые полностью проанализированы, разработаны, протестированы и проверены клиентом (то есть готовы к использованию). Справа расположены истории, которые еще нужно разработать.

Рассказывая о задачах, решаемых командой в этой итерации, вы показываете менеджерам раскадровку данной итерации.

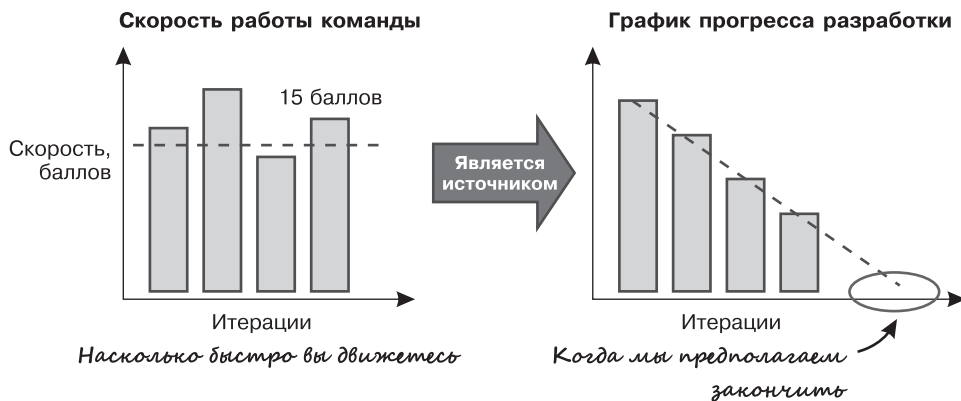


Раскадровка позволяет отслеживать состояние функций, входящих в состав итераций (функции — это те же пользовательские истории). Функции, которые еще нужно разработать, находятся слева, а полностью реализованные и одобренные клиентом — справа.

По мере того как история проходит разные стадии разработки, она движется по схеме слева направо. И только когда история полностью разработана, протестирована и одобрена клиентом, она оказывается в столбце «Завершено».

Посматривая на часы, менеджеры переходят к сути дела и спрашивают, когда, *по вашему мнению*, проект будет завершен.

Чтобы ответить на этот вопрос, вы показываете им всего две диаграммы, которых они еще не видели. Это график скорости работы вашей команды и схема прогресса разработки.



Вы объясняете, что скорость работы команды — наилучший инструмент, который есть у вас в распоряжении для измерения уровня продуктивности. Измеряя, сколько работы удастся выполнить за неделю, и используя эти данные в качестве основы для дальнейшего планирования, команда может точно прогнозировать, когда ожидается завершение проекта. Эти данные отображаются на графике прогресса разработки.

График прогресса разработки (см. раздел 8.5) учитывает скорость работы команды и экстраполирует скорость, с которой команда перерабатывает список пожеланий пользователя. Проект готов, когда команда реализует все задачи из списка или когда закончатся деньги (в зависимости от того, что наступит раньше).

Описав, таким образом, сложившуюся обстановку, вы спокойно озвучиваете мысль, которая и так уже ясна всем присутствующим. Если наполовину сократить команду разработчиков, это в два раза снизит производительность труда.

Менеджеры, впечатленные тем, как вы контролируете ситуацию, благодарят вас за то, что вы нашли время с ними побеседовать, и отправляются на следующее проектное собрание.

Через несколько недель вы получаете электронное сообщение о том, что, поскольку компания выбирает новое стратегическое направление развития, ваш проект отменяется (да, такова жизнь).

Но есть и хорошие новости. Компания оказалась настолько впечатлена вашей работой, что хочет дать вам руководящую роль в новом проекте!

Это всего один гипотетический пример того, как визуальное оформление рабочего пространства помогает обрисовывать владельцам проекта его перспективы и делает реальное положение дел очевидным. Но основной козырь визуального оформления — подспорье, которое команда получает при выполнении работы и при сосредоточении на ней.

Рассмотрим несколько идей о визуальном оформлении рабочего пространства.

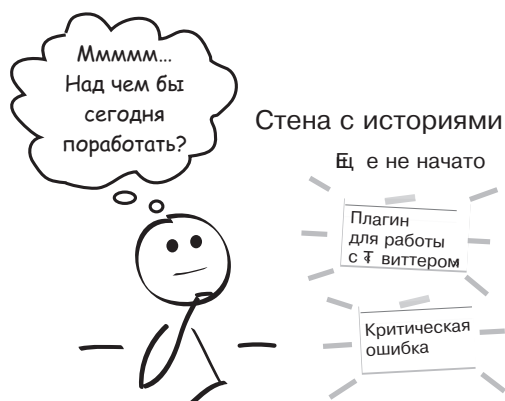
11.2. Как оформить визуальное рабочее пространство

Качественно оформить рабочее пространство совсем не сложно. Если ваша команда еще только знакомится с гибкой разработкой, то обычно рекомендуется обустроить следующие элементы:

- ❑ стену с карточками историй;
- ❑ схему готовности;
- ❑ графики скорости работы команды и прогресса разработки;
- ❑ стартовую колоду, если для нее найдется место.

Стартовая колода полезна, так как напоминает команде, какова общая цель и чем команда на самом деле занимается (если вы с головой ушли в проект, то потерять ориентацию довольно легко).

Стена, на которой наклеены карточки с историями, хороша тем, что кто угодно каждое утро может подойти к ней и посмотреть, что именно нужно делать дальше.



На стене с историями видны все узкие места вашей системы, а также направления, по которым вы можете направлять ресурсы.

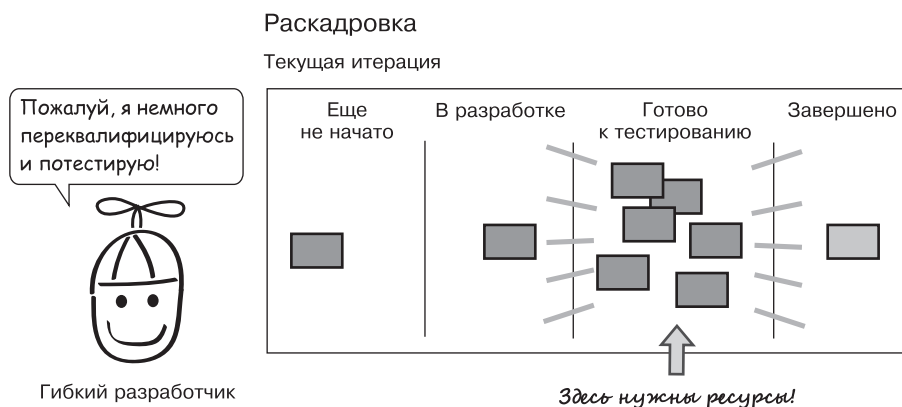


Схема готовности — это очень красивая вещь, так как она позволяет любому, кто войдет в комнату, одним взглядом оценить состояние проекта. Вот то, что сделано. Вот то, что осталось. Не требуется никакой изощренной математики или таблиц Excel.

Возвращаясь к подробному обсуждению гибкого планирования, отмечу, что нет лучшего средства для прогнозирования, чем хороший график прогресса разработки. Такой график обязательно должен висеть на стене в вашем офисе, и вы всегда будете видеть, насколько реалистичны сроки и какова тенденция развития проекта.

Разумеется, все это только начало. Если у вас есть другие изображения, имитационные модели или схемы, которые помогут в работе вам и вашей команде, развесьте их по стенам так, чтобы они всем были видны.

Вот еще несколько идей по визуальному оформлению рабочего пространства.

11.3. Демонстрация намерений

Рабочее соглашение заключается в том, чтобы поставить командный ориентир и сказать: «Мы — команда, поэтому будем работать именно так». Это способ сформировать у всех членов команды представление о том, как предстоит трудиться и что ожидается от людей, которые будут подключаться к вам в проекте.

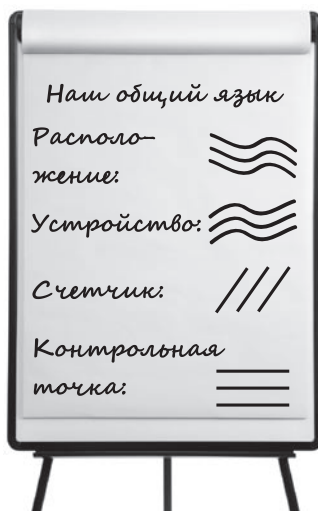
Общепризнанные ценности имеют такое же значение, но они более ощутимы, чем обычное заявление. Если ранее команде доводилось «обжигать»

ся», так как ее вынуждали идти на компромиссы по вопросам качества, и коллеги больше не хотят иметь дурную славу людей, которые удушатся за копейку и пишут низкокачественные поделки, команда может во всеуслышание заявить о своих ценностях в форме специального плаката.

| Рабочие соглашения | Общие ценности |
|--|--|
| <ul style="list-style-type: none">• Основное время работы — с 9:00 до 16:00.• Ежедневные планерки — ровно в 10:00.• Готово — означает протестировано.• Обращайте внимание на сборку.• Если кто-то просит вас о помощи — не отказывайте.• Еженедельные демонстрации результатов — по вторникам в 11:00.• Представитель заказчика работает с командой с 13:00 до 15:00 | <ul style="list-style-type: none">• На работе не экономить.• Окна не разбивать.• Каждый имеет право на несогласие.• Мы знаем, как обращаться с истиной.• Не предполагайте — спрашивайте.• Сомневаетесь — напишите тест.• Стремитесь получить отзыв.• Свое «я» можете выставить за дверь |

Еще одна вещь, которая должна быть общей у всех участников проекта, — это язык.

11.4. Создание и совместное использование общего рабочего языка



Если термины, используемые в вашей программе, не совпадают с обозначениями тех же понятий в бизнесе, то могут случиться всевозможные неприятности.

- ❑ В программе будут реализованы неверные абстракции (например, заказчик будет понимать под *расположением* одно, а разработчики интерпретируют это слово иначе).
- ❑ Программа будет хуже поддаваться изменениям (так как слова, появляющиеся на экране, не совпадают с теми, которые используются для сохранения информации в базе данных).
- ❑ Будет появляться больше ошибок, и соответственно возрастут расходы на техническую поддержку (так как команде придется усердно работать при внесении изменений в программу).

Чтобы обойтись без такой несогласованности, нужно создать общую терминологию, которую будете неуклонно соблюдать и вы, и пользователь. Она будет применяться при написании пользовательских историй, моделей, изображений и кода.

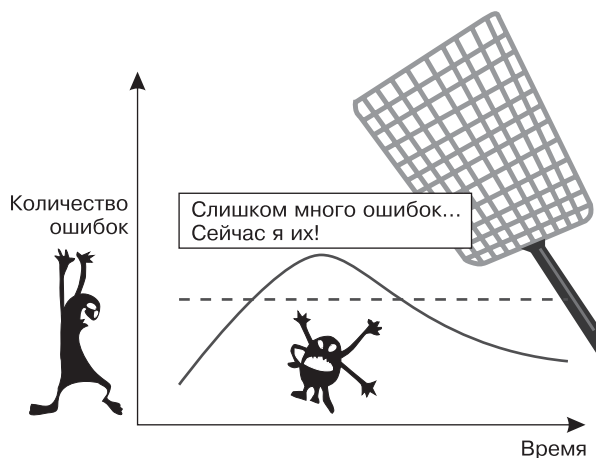
Например, если при обсуждении системы вы и клиент используете несколько ключевых слов, выпишите их, выработайте для них определения, которые устроят обе стороны, а затем гарантируйте, что содержание программы соответствует этим определениям (речь идет о том, что видно на экране, о коде и о столбцах базы данных).

Сделав так, вы не только сведете к минимуму количество ошибок и элементов, требующих переработки, но и облегчите диалог с клиентом, так как ваш код всегда будет точно соответствовать пониманию бизнеса с точки зрения клиента.

К сожалению, объем книги ограничен и мы не можем тщательно обсудить эту тему. Однако данному вопросу посвящена отличная книга Эрика Эванса *Domain-Driven Design: Tackling Complexity in the Heart of Software* [Eva03]. Очень рекомендую ее прочитать.

Наконец перейдем к работе над ошибками.

11.5. Отслеживание ошибок (bugs)



Чтобы гарантировать, что жизнь не преподнесет вам и вашей команде неприятный сюрприз в виде нашествия ошибок перед самой сдачей готового продукта, отслеживайте и записывайте возникающие неполадки с первого дня проекта.

Если это поможет, выделите 10 % каждой итерации на устранение ошибок и отдачу технического долга. Просто разбирайтесь с ошибками сразу же после их появления и не допускайте, чтобы увеличение количества ошибок приобрело стихийный характер.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, а что делать, если не получается визуально оформить мое рабочее место?*

МАСТЕР: Действительно, в некоторых офисах командам, занимающимся проектами, запрещается наклеивать на стены рабочие схемы. Если столкнешься с таким противодействием, смирись с ним и подумай, как работать в сложившихся условиях.

УЧЕНИК: Да, Мастер. Но должен ли я бороться за визуальное оформление? Или следует просто свыкнуться с тем, что у меня на работе его не будет?

МАСТЕР: Это зависит от тебя. Можно идти на компромиссы. Можно уступить. Или можно бороться. Временами и в определенных условиях может быть оправдан любой из этих вариантов. Слушай свое сердце, ищи единомышленников и решай, стоит ли ввязываться в этот бой.

УЧЕНИК: Если это действительно очень важный участок работы, как можно идти на компромиссы?

МАСТЕР: Сталкиваясь с подобными ситуациями, некоторые умельцы научились делать складывающиеся раскладовки, которые позволяют не захламлять офис и обеспечивают для команды возможность свободного общения на протяжении рабочего дня. Другие пользуются онлайн-инструментами и виртуальными раскладовками, чтобы делиться важной информацией, а также обеспечивать слаженную работу команды.

УЧЕНИК: Значит, визуальное оформление рабочего пространства может быть и виртуальным?

МАСТЕР: Нет. Реальные наглядные материалы всегда лучше, но иногда ты просто не можешь себе их позволить.

УЧЕНИК: А если я решу бороться? Что делать тогда?

МАСТЕР: Можешь просто начать готовить визуальное оформление, ежедневно пользоваться им при реализации проекта и надеяться, что в результате переговоров и объяснений клиент поймет, как полезно это оформление.

УЧЕНИК: А если не поймет?

МАСТЕР: Тогда не исключено, что основная причина несогласия имеет эмоциональную природу. Возможно, в компании действуют силы, диаметрально противоположные тому, чего вы хотите достичь. Попробуй прочувство-

вать и понять дух, стоящий за противодействующими тебе силами. Быть может, с помощью переговоров вы сможете найти решение, которое устроит обе стороны. Твоими лучшими союзниками здесь будут время и терпение.

Что дальше?

Наше путешествие близится к концу. Вы подыскиали всех нужных людей (см. главу 3), составили план (см. главу 8) и знаете, что нужно для его исполнения.

Следующая часть книги посвящена основным практикам гибкой разработки программного обеспечения, которыми доведется пользоваться вам и вашей команде, чтобы вся эта система работала.

Данный материал необходимо прочитать каждому, кто «режет» код, а также тем, кто когда-либо планировал гибкий проект или руководил им. Никакие методики не будут работать, если в основе их не лежит обстоятельный технический подход. Хотя каждую из следующих четырех глав можно было бы расширить до размеров отдельной книги, они помогут получить представление о том, как выглядит гибкая разработка с практической точки зрения и почему технические аспекты важны для общей гибкости нашей команды.

Начнем с того, что лучше всего экономит время в любом софтверном проекте, — с автоматического тестирования компонентов.

Часть V

Создание гибкого ПО

Тестирование компонентов: убеждаемся в том, что все работает



Неважно, сколько времени было потрачено на планирование и управление ожиданиями, — гибкие процессы могут работать лишь в том случае, если они опираются на надежные практики разработки программного обеспечения. Некоторые такие практики (например, работа в паре, принятая в экстремальном программировании) вызывают противоречивые отзывы, а другие (такие как автоматическое тестирование компонентов) получили широкое распространение.

В четырех последних главах книги мы поговорим о том, что я считаю элементарными составляющими гибкого программирования:

- ❑ тестирование компонентов;
- ❑ рефакторинг;
- ❑ разработка на основе тестирования;
- ❑ непрерывная интеграция.

Я просто познакомлю вас с этими концепциями, чтобы вы в достаточной степени понимали механизмы, обеспечивающие работу вашей команды.

Все примеры написаны на языке Microsoft .NET C# (хотя сами концепции применимы к любому языку программирования). Не беспокойтесь, если вы не технарь. Вам все равно не помешает прочитать обо всем этом, и по мере повествования я буду отдельно подчеркивать самые важные места.

Начнем с практики, на которой основана большая часть гибкого программирования, — тщательного и обширного тестирования компонентов.

12.1. Добро пожаловать в Вегас, малыш!

Вы счастливчик! Вы только что присоединились к команде разработчиков, которая пишет новый симулятор игры в блек-джек! Ваша задача — разработать колоду карт.

Вот первый фрагмент кода, описывающего колоду карт¹:

```
public class Deck
{
    private readonly IList<Card> cards = new List<Card>();
    public Deck()
    {
        cards.Add(Card.TWO_OF_CLUBS);
        cards.Add(Card.THREE_OF_CLUBS);
        // ... оставшиеся трефы
        cards.Add(Card.TWO_OF_DIAMONDS);
        cards.Add(Card.THREE_OF_DIAMONDS);
        // ... оставшиеся бубны
        cards.Add(Card.TWO_OF_SPADES);
        cards.Add(Card.THREE_OF_SPADES);
        // ... оставшиеся пики
        cards.Add(Card.TWO_OF_HEARTS);
    }
}
```

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/src/Deck.cs>.

```
cards.Add(Card.THREE_OF_HEARTS);  
// ... оставшиеся червы  
// джокер  
cards.Add(Card.JOKER);  
}
```

Напишите заведомо неуспешный тест перед тем, как исправлять ошибку

Когда вы обнаруживаете в программе ошибку, существует соблазн сразу ее исправить. Не делайте этого. Лучше отследить ошибку, написав заведомо неуспешный тест компонента (failing unit test), и только потом ее исправить. При этом вы:

- понимаете природу ошибки;
- убеждаетесь, что действительно ее исправили;
- удостоверяетесь, что эта ошибка больше никогда не появится в вашей программе.

Наступает время равноправного анализа (peer review). Все, кажется, идет хорошо, и перед самой сдачей продукта отдел контроля качества находит ошибку.

Оказывается, в колоде для блек-джека нет джокеров! Вы исправляете ошибку, передаете в отдел контроля качества новую сборку, и программа сдается заказчику.

Затем, через пару недель, вы получаете от менеджера из отдела контроля качества жесткое письмо, в котором он сообщает вам, что прошлым вечером в программе проявилась серьезная ошибка. Пришлось выплатить несколько десятков тысяч долларов, так как кто-то поместил абстракцию джокера в класс `Card`!

«Что? — восклицаете вы. — Это невозможно. Я лично исправил эту ошибку две недели назад». Но, разобравшись в проблеме подробнее, вы обнаруживаете, что практикант, работавший летом под вашим руководством, понял вас буквально, когда вы попросили его убедиться, что ваша колода карт ведет себя точно так же, как настоящая бумажная колода, с которой вы попросили сверять программу.

Видимо, он из лучших побуждений вернул джокера в колоду, считая, что нашел ошибку.

Пристыженный и ошеломленный практикант просит прощения у вас и у всей команды. Наедине он спрашивает вас, что нужно делать, чтобы подобных проколов больше никогда не случилось.

Что вы ему скажете? Что можете сделать вы (или он), чтобы гарантировать, что ошибка с джокером не вкрадется в базу кода после того, как ее устранят?

В свете всего сказанного давайте рассмотрим самый простой тест компонента.

12.2. Приступаем к тестированию компонентов

Тесты компонентов — это небольшие тесты на уровне метода (method-level tests), которые разработчик пишет всякий раз, когда вносит в программу изменения и проверяет, работает ли программа так, как он задумывал.

Например, мы хотим проверить, на самом ли деле в нашей колоде 52 карты (а не 53). Можно написать тест компонента, который будет выглядеть примерно так¹:

```
[TestFixture]
public class DeckTest
{
    [Test]
    public void Verify_deck_contains_52_cards()
    {
        var deck = new Deck();
        Assert.AreEqual(52, deck.Count());
    }
}
```

Сразу хочу оговориться: приведенный код — это не фрагмент настоящего кода, который работает в написанном нами симуляторе игры блек-джек. Это тестовый код, доказывающий, что наш реальный код работает так, как нужно.

Всякий раз, когда у нас появляется сомнение в функциональности кода или мы хотим удостовериться, что код работает правильно, мы пишем тест компонента (данный конкретный тест показывает, что в нашей колоде ровно 52 карты).

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/test/DeckTest.cs>.

Тесты компонентов просто неоценимы, так как при автоматизации и упрощении тестирования мы можем проводить тест всякий раз, когда вносим изменения в программу. В итоге мы сразу же узнаем, не сломалось ли что-нибудь (подробнее об этом читайте в главе 15).

Как правило, гибкий проект требует сотен, если не тысяч тестов компонентов. Потребуется препарировать приложение и проверить все — от бизнес-логики программы до того, можем ли мы сохранять пользовательскую информацию в базе данных.

В написании большого количества таких тестов и подобной проверке базы кода есть немалая польза.

- ❑ Вы мгновенно узнаете, какова ситуация. Если вы внесли изменения в свой код, а тестирование компонента не проходит, вы узнаете об этом сразу (а не через три недели после того, как программа начнет использоваться на практике).

Тестируйте все, что может сломаться

В экстремальном программировании есть постулат «тестируй все, что может сломаться». Это напоминание разработчикам: если им начинает казаться, что есть определенная вероятность сбоя в системе, то вызывающий опасения фрагмент системы нужно проверить автоматическим тестом.

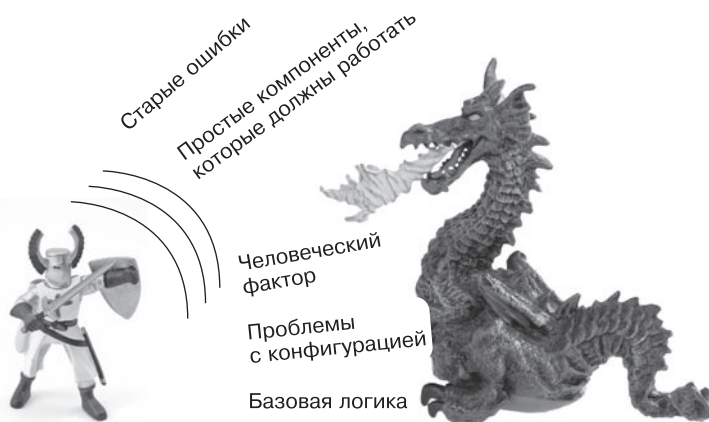
Никогда не удастся протестировать все, но на практике можно понять философию того, как команды, занимающиеся гибкой разработкой, должны относиться к тестированию. Тестируйте до тех пор, пока не будете уверены, что ваша программа работает без ошибок, и, опираясь на здравый смысл, проверяйте, где тестирование может принести максимальную пользу.

В главе 14 описано, как такой метод разработки помогает максимально выгодно использовать средства, выделенные на тестирование, а также найти золотую середину между стремлением протестировать все и вся и потребностью тестировать лишь то, что необходимо.

- ❑ Радикально снижаются затраты на регрессивное тестирование. Вместо необходимости заново тестировать все вручную перед выпуском нового релиза, мы экономим уйму времени, автоматизируя самые легкие операции, а сами занимаемся тестированием лишь того, что действительно представляет сложность.

- ❑ Значительно сокращается длительность отладки. Когда тест компонента не удастся, вы знаете наверняка, где искать проблему. Больше не нужно запускать отладчик и пробираться сквозь сотни строк кода, отыскивая фрагмент, вызывающий сбой. Тесты компонентов подобны лучу лазера, прорезающему туман, они избавляют вас от долгого поиска проблемы.
- ❑ Тесты компонентов позволяют вам чувствовать себя уверенно при внедрении программы.
- ❑ Вы гораздо спокойнее будете переходить к коммерческой разработке, зная, что ваша работа подстрахована набором автоматизированных тестов. Они не могут полностью исключить возможность неверного применения, но зато освобождают время на ручное тестирование более сложных/интересных частей вашей системы.

Представьте, что тесты компонентов — это доспехи, в которые вы облачаетесь перед битвой. Они становятся своего рода исполняемой спецификацией, которая навсегда остается в вашем коде, защищает вас от стрел, находящихся и воображаемых драконов и, что самое важное, от вас самих.



Предупреждаю: вам придется сталкиваться со случаями, когда написать автоматизированный тест непросто. Например, сложно написать тест, проверяющий, можем ли мы перетасовать колоду карт (поскольку результат теста является случайным и всякий раз изменяется). Кроме того, тестирование конкурентных и многопоточных приложений, мягко говоря, может представлять проблему.

Если вы попадаете в подобные ситуации — не отчаивайтесь. Это скорее исключения, чем правила. В подавляющем большинстве случаев вы

сможете инстанцировать объект и применить правила к вызываемым методам. В настоящее время подобные операции еще более упрощаются благодаря современным имитационным фреймворкам тестирования компонентов.

В редких случаях, когда не удастся без проблем перейти к тестированию определенного элемента, сложности могут заключаться в строении вашего кода (см. главу 14). Или, возможно, вы просто наследуете какой-то устаревший код, который действительно очень сложно тестировать.

В таком случае пусть все остается как есть. Признайте, что все протестировать не удастся. Убедитесь, что вы провели по-настоящему качественное ручное и исследовательское тестирование, и двигайтесь дальше.

Просто не сдавайтесь! Всегда пытайтесь автоматизировать тестирование того или иного элемента кода, так как, если кольчужка будет чуть потолще, это может сослужить хорошую службу, когда придет требование срочно исправить ошибку и необходимо будет ускорить выпуск продукта.

Можете также почитать книгу Майкла Фэзерса *Working Effectively with Legacy Code* [Fea04], в которой содержится масса бесценных советов о том, как работать с устаревшим кодом и легко вносить в него изменения.



Поставим себя на место тестировщика. Какие тесты, по вашему мнению, можно написать для проверки функционирования класса колоды карт, учитывая следующие требования? Как нам надежно предотвратить повторное проявление этой злосчастной ошибки с джокером?

Запишите здесь тестовые требования



Если ваш тест выглядит примерно так, вы на правильном пути. Нужно протестировать все, что теоретически может сломаться либо сработать неправильно. Сделайте глубокий вдох и выдох и напишите тест¹.

```
[TestFixture]
public class DeckTest2
{
    [Test]
    public void Verify_deck_contains_52_cards()
    {
        var deck = new Deck();
        Assert.AreEqual(52, deck.Count());
    }
    [Test]
    public void Verify_deck_contains_thirteen_cards_for_each_
suit()
    {
        var Deck = new Deck();
        Assert.AreEqual(13, Deck.NumberOfHearts());
        Assert.AreEqual(13, Deck.NumberOfClubs());
        Assert.AreEqual(13, Deck.NumberOfDiamonds());
        Assert.AreEqual(13, Deck.NumberOfSpades());
    }
    [Test]
    public void Verify_deck_contains_no_joker()
    {
        var Deck = new Deck();
        Assert.IsFalse(Deck.Contains(Card.JOKER));
    }
    [Test]
    public void Check_every_card_in_the_deck()
    {
        var Deck = new Deck();
        Assert.IsTrue(Deck.Contains(Card.TWO_OF_CLUBS));
        Assert.IsTrue(Deck.Contains(Card.TWO_OF_DIAMONDS));
        Assert.IsTrue(Deck.Contains(Card.TWO_OF_HEARTS));
        Assert.IsTrue(Deck.Contains(Card.TWO_OF_SPADES));
        Assert.IsTrue(Deck.Contains(Card.THREE_OF_CLUBS));
        Assert.IsTrue(Deck.Contains(Card.THREE_OF_DIAMONDS));
        Assert.IsTrue(Deck.Contains(Card.THREE_OF_HEARTS));
        Assert.IsTrue(Deck.Contains(Card.THREE_OF_SPADES));
        // остальные
    }
}
```

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/test/DeckTest.cs>.

Гуманитариям скажу, что в этом фрагменте кода выполняется тестирование компонентов, позволяющее решить следующие задачи:

- ❑ проверить, на самом ли деле в колоде содержится 13 карт каждой масти;
- ❑ убедиться, что в колоде нет джокеров (присутствие джокеров — это ошибка, замеченная нами ранее);
- ❑ проверить каждую карту из колоды (всего их 52).

Где подробнее изучить эту тему

Здесь мы лишь бегло коснулись тестирования компонентов, хотя на эту тему можно рассказать гораздо больше. К счастью, тестирование компонентов в софтверных проектах становится настолько общепринятой практикой, что большинство современных языков содержит специальные фреймворки для тестирования компонентов, которые можно бесплатно скачать. Кроме того, в открытом доступе есть руководства, показывающие, как начать работу.

Любому разработчику, желающему понять принципы этого метода, рекомендую для начала познакомиться с классической статьей Кеннета Бека¹.

Кроме того, отсылаю вас к книгам *Pragmatic Unit Testing in C# with NUnit* [НТ04] и *Pragmatic Unit Testing in Java with JUnit* [НТ03].



Мастер-сэнсэй
и воин,
постигающий
искусство

УЧЕНИК: Мастер, а разве тестирование компонентов не замедляет работу команды? Я имею в виду, что нам ведь требуется писать вдвое больше кода.

¹ <http://junit.sourceforge.net/doc/testinfected/testing.htm>.

МАСТЕР: Если бы программирование сводилось к набору кода на клавиатуре, то ты был бы прав. Но тесты компонентов нужны для того, чтобы гарантировать следующее: по мере того как мы вносим в программу изменения, все в целом по-прежнему работает именно так, как мы того ожидаем. И мы экономим время, избавляя себя от регрессивного тестирования всей системы при внесении каждого изменения.

ВОПРОС: Да, Мастер. Но не делают ли тесты компонентов код непрочным? Как быть уверенным в том, что тесты компонентов не будут давать сбои всякий раз, когда я изменяю свой код?

ОТВЕТ: Действительно, существует возможность написать некачественный тест, основанный на жестко закодированных данных, характеризующийся тесным связыванием и плохой структурой, но когда ты привыкнешь руководствоваться тестированием при разработке (см. главу 14), то обнаружишь, что твои тесты не дают сбоев и действительно оптимизируют общую структуру приложения. Самые современные интегрированные среды разработки (*integrated development environments*, IDE) также облегчают работу с изменениями и тестирование кода. Можно переименовать метод во всей базе кода, просто нажав несколько клавиш. Таким образом, тестирование и написание кода идут нога в ногу.

ВОПРОС: Должны ли я и моя команда стремиться к стопроцентному тестированию компонентов?

ОТВЕТ: Нет. Основная цель тестирования компонентов заключается не в полном охвате. Такое тестирование обеспечивает команде достаточную уверенность в том, что программа готова к реальному использованию.

ВОПРОС: В таком случае насколько полное тестирование компонентов должны обеспечивать я и моя команда?

ОТВЕТ: Это решаете вы сами. Некоторые языки и фреймворки позволяют с легкостью достичь при тестировании хорошего охвата кода. В других случаях добиться такого охвата сложнее. Если вы только начинаете работу, не придавайте охвату чрезмерного значения. Просто напишите максимальное количество хороших тестов так, чтобы вам это не мешало.

Что дальше?

Мы хорошо поработали. Теперь вы знаете об одной из важнейших основ, на которых базируются все остальные практики гибкой разработки. Без слаженного тестирования компонентов все они могут сорваться.

Далее мы рассмотрим, как на базе тестирования компонентов выстроить и осуществить настолько важную работу, пренебрежение которой превратит наш продукт в еще один переоцененный, непригодный для поддержки сгусток кода, в который решительно невозможно внести какие-либо изменения.

Эта важнейшая практика называется рефакторингом.

Глава 13

Рефакторинг: погашение технического долга



Подобно дому, купленному в ипотеку, любая программа имеет долги, которые постоянно приходится отдавать.

В этой главе мы рассмотрим практику рефакторинга и увидим, как, регулярно отдавая технический долг, мы поддерживаем программу легкой и гибкой и в этом «доме» становится приятно жить и работать.

Прочитав данную главу, вы увидите, как рефакторинг помогает снизить расходы на техническую поддержку, позволяет сформировать общий словарь, обеспечивающий совершенствование кода, и дает возможность добавлять новые функции, не снижая темпа работы.

Теперь давайте окунемся в мир рефакторинга и посмотрим, что требуется, чтобы «развернуться на пятачке».

13.1. Как «развернуться на пятачке»

Кажется, конкуренты только что выпустили «детскую» версию создаваемого вашей компанией онлайн-симулятора игры блек-джек. И эта программа продается на ура.

Чтобы ответить на этот выпад со сторон конкурентов, вы и команда немедленно беретесь за работу, и какое-то время дела идут нормально. Но затем начинается что-то странное. То, что на первый взгляд не составляло никакого труда, теперь кажется по-настоящему сложным.

Значительная часть кода попала в базу путем копирования и вставки. Таким образом, вносить изменения становится сложно, так как при каждом исправлении в одном месте его нужно продублировать в десятках других мест.

Вдобавок ко всему, код, который вы с командой написали в спешке, стремясь уложиться в поставленные сроки, теперь не дает вам покоя. Он очень запутанный, и с ним действительно тяжело работать. И, что еще хуже, программист, который является автором кода, уже покинул проект.

Вот пример такого проблемного кода¹:

```
public bool DealerWins(Hand hand1)
{
    var h1 = hand1; int sum1 = 0;
    foreach (var c in h1)
    {
        sum1 += Value(c.Value, h1);
    }
    var h2 = DealerManager.Hand; int sum2 = 0;
    foreach (var c in h2)
    {
        sum2 += Value(c.Value, h2);
    }
    if (sum2 >= sum1)
    {
        return true;
    }
    else
        return false;
    return false;
}
```

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrape/code/Refactoring/src/BlackJack.cs>.

Не переживайте, если не можете понять этот код (я, например, тоже не могу). Но все же вам требуется его изменить. И данный код приходится поддерживать. Именно его вы (ага!) скоро должны будете использовать на практике.

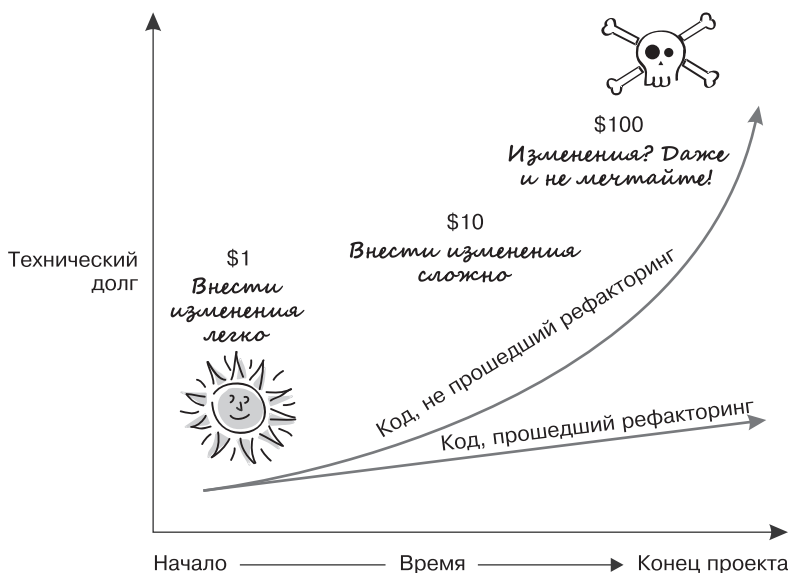
Работа с этой базой кода и внесение в нее изменений, оказывается, даются сложнее и стоят дороже, чем вы поначалу думали.

Быстро становится очевидно, что на то, чтобы все сделать правильно, вам понадобится не менее двух недель всего лишь для подчистки имеющейся базы кода — и только потом можно будет добавлять новый функционал. К сожалению, начальник говорит, что этих двух недель у вас нет.

Что же пошло не так? Может ли код, который был красивым, простым и легким в использовании, превратиться в нечто настолько громоздкое, уродливое и неудобное?

Давайте рассмотрим феномен, который называется *техническим долгом* (technical debt).

13.2. Технический долг



Технический долг — это непрерывное накопление сокращений, уловок, дублей и других ошибок, которыми мы наполняем базу нашего кода во имя повышения скорости и соблюдения расписания.

У вас в коде всегда найдется какой-нибудь долг (если его нет, это означает, что вы не пробуете инновационных или нетривиальных подходов). Вы должны сознавать, что если его накопится слишком много, то нечто интересное, легкое и простое однажды станет нудным, тяжелым и сложным.

Технический долг — это не просто код



Большая часть технического долга завязана на коде, однако он также может проявляться в данных, файлах сборки и файлах конфигурации.

Мне доводилось участвовать в масштабной переработке системы интерфейса базы данных, в которой имелось название города, записанное двумя разными способами. Стоимость внесения этих, казалось бы, незначительных изменений была огромной. Выяснилось, что пренебречь разницей в двух названиях города нельзя, поэтому пришлось написать и внедрить дополнительный код (а значит, усложнить весь код в целом) на весь срок использования системы. Поскольку речь идет о мейнфрейме, данный срок может быть очень длительным.

Технический долг может принимать разнообразные формы (неструктурированная программа, излишняя сложность, дублирование в коде и общая небрежность), но настоящую опасность представляет долг, который способен заставить вас врасплох. Любое нарушение, недавно допущенное в базе кода, может показаться небольшим или незначительным. Но, как и любой долг, технический долг накапливается и усугубляет проблему.

Нам необходим способ систематического погашения нашего технического долга по мере работы. Мы должны последовательно улучшать и поддерживать целостность программы и ее структуры, чтобы решать текущие задачи и уверенно встречать проблемы, которые еще не известны и могут возникнуть в ближайшем будущем.

В гибкой разработке такая методика называется *рефакторингом*.

13.3. Погашение долга с помощью рефакторинга

Рефакторинг — это практика непрерывного внесения в структуру программы небольших постепенных изменений, не требующих изменения всего ее внешнего поведения.

Занимаясь рефакторингом кода, мы не добавляем новые функции и даже не исправляем ошибки. Мы просто делаем код понятнее, облегчая его чтение и упрощая внесение изменений.

На первый взгляд они могут показаться мелкими и незначительными. Но если их последовательно и неуклонно вносить в базу кода, это поможет радикально повысить качество и упростить поддержку программы. Например, когда вы переименовываете неудачно названный метод или переменную, чтобы облегчить их чтение и понимание, вы занимаетесь рефакторингом.

decimal sal; → decimal salary; [Переименование переменной] ← Рефакторинг

public decimal Calc() → public decimal CalculateTotalTaxes() [Переименование метода]

Обратите внимание на следующие фрагменты кода и задайте себе вопрос: какой из них проще читается и более понятен?

```
if (Date.Before(SUMMER_START) || Date.After(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

ИЛИ

```
if (NotSummer(date))
    charge = WinterCharge(quantity);
else
    charge = SummerCharge(quantity);
```

Рефакторинг: [Извлечение метода]

Вам не обязательно быть опытным разработчиком и знать язык C#, чтобы увидеть, что второй пример проще и понятнее, чем первый. Писать код — все равно что писать хорошую прозу. Нужно, чтобы код был чистым и ясным и чтобы было без труда понятно, что в нем происходит.

Рефакторинг — это секретный рецепт, с помощью которого специалисты по объектно-ориентированному программированию решают такие задачи. Правильно подбирая имена для методов и переменных и скрывая от зрителя ненужные детали, хороший программист умеет предельно ясно выразить свои намерения, его код становится понятен и прост для изменения.



Принцип гибкой разработки

Постоянное стремление к техническому совершенству и хорошей структуре повышает гибкость проекта.

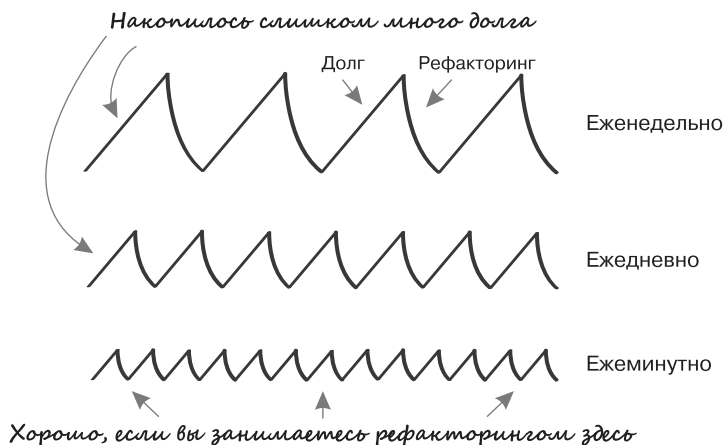
В сущности, это все, в чем заключается рефакторинг: напоминание самому себе о том, что написанием кода и его поддержкой занимаются такие же люди, как и мы с вами. И если мы не сможем сделать нашу программу простой для изменения и удобной для работы, нам всегда будет очень сложно вносить в нее изменения или добавлять новые функции.

Занимайтесь рефакторингом на совесть и непрерывно

Если вы настойчиво занимаетесь рефакторингом, то к концу проекта работа будет не замедляться, а ускоряться. Ведь, поддерживая структуру вашего кода актуальной, вы выполнили массу сложной работы. Новые функции строятся на более старых, хорошо проработанных. Теперь вы можете справиться с самой сложной работой и насладиться плодами того, что заблаговременно привели все в порядок.

Настойчивый рефакторинг не означает, что нужно сохранять все версии переработанного кода до конца итерации. Рефакторинг продолжается непрерывно, каждый день.

Если все сделано правильно, то рефакторинг почти незаметен. Его этапы настолько малы, а шаги оптимизации так миниатюрны, что вы едва ли сможете указать, где коллега выполнил рефакторинг кода и добавил новую функцию.



Но хватит теории. Давайте напряжем мозг и опробуем рефакторинг сами.



Какие изменения можно внести в «детскую» версию игры блек-джек, которая была рассмотрена в начале этой главы?

```
public bool DealerWins(Hand hand1)
{
    var h1 = hand1; int sum1 = 0;
    foreach (var c in h1)
    {
        sum1 += Value(c.Value, h1);
    }
    var h2 = DealerManager.Hand; int sum2 = 0;
    foreach (var c in h2)
    {
        sum2 += Value(c.Value, h2);
    }
    if (sum2 >= sum1)
    {
        return true;
    }
    else
        return false;

    return false;
}
```

Можем ли мы переименовать какие-либо переменные?

Нет ли дублей в описании функций программы?

Нет ли ненужной логики или кода?

Любой рефакторинг стоит начинать с проверки того, насколько хороши названия всех методов и переменных. И именно это нужно делать в первую очередь при приведении кода в порядок.

```
public bool DealerWins(Hand playerHand) {
    int playerHandValue = 0;
    foreach (var card in playerHand)
    {
        playerHandValue += DetermineCardValue(card, playerHand);
    }

    var dealerHand = DealerManager.Hand;
    int dealerHandValue = 0;

    foreach (var card in playerHand)
    {
        dealerHandValue += DetermineCardValue(card, dealerHand);
    }

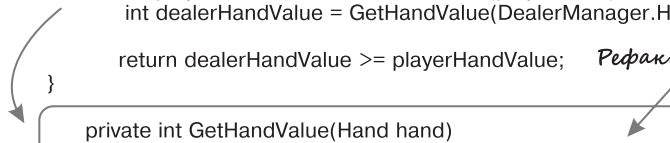
    return dealerHandValue >= playerHandValue;
}
```

Рефакторинг: [Переименование переменной]

Рефакторинг: [Переименование метода]

Рефакторинг: Упрощение кода

Что ж, уже несколько лучше. Читать код стало проще. Но мы еще сделали не все. В коде по-прежнему много дублей. Что, если попытаться вычленить некоторую схожую логику в специальный метод?



```

public bool DealerWins(Hand playerHand)
{
    int playerHandValue = GetHandValue(playerHand);
    int dealerHandValue = GetHandValue(DealerManager.Hand);

    return dealerHandValue >= playerHandValue;
}

private int GetHandValue(Hand hand)
{
    int handValue = 0;

    foreach (var card in hand)
    {
        handValue += DetermineCardValue(card, hand);
    }

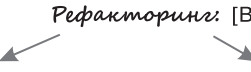
    return handValue;
}

```

Рефакторинг: [Извлечение метода]

Вы только посмотрите! После извлечения метода `GetPlayerHandValue` наш метод `DealerWins` сократился всего до трех строк кода. Теперь хорошо видно, какие функции выполняются в этом методе. Читать его стало гораздо проще. А если мы захотим узнать, как вычисляются карты, попадающие на руки пользователю, эта информация всегда найдется в методе `GetPlayerHandValue`.

Код совершенно ясен. Но если есть желание усовершенствовать его еще на порядок, потребуется сделать нечто подобное:



```

public bool DealerWins(Hand playerHand)
{
    return GetHandValue(DealerManager.Hand) >= GetHandValue(playerHand);
}

```

Рефакторинг: [Встраивание переменной]

Итак, в процессе рефакторинга мы выполнили всего три простые операции:

- ❑ переименовали переменные и методы;
- ❑ встроили переменную;
- ❑ извлекли метод,

но при этом значительно облегчили чтение и поддержку кода.

Для любого менеджера, читающего данную книгу, эта информация очень важна, так как теперь, если команде потребуется срочно исправить ошибку

или внести в программу принципиальные изменения, это получится сделать быстрее, лучше и дешевле, чем ранее.

Чтобы не тратить бесконечные часы, пытаясь понять, что делает этот код, можно прямо приступить к работе и вносить изменения.

Таким образом, следует всячески стимулировать в своей команде привычку активного рефакторинга и постоянного погашения технического долга.

Дурная слава рефакторинга



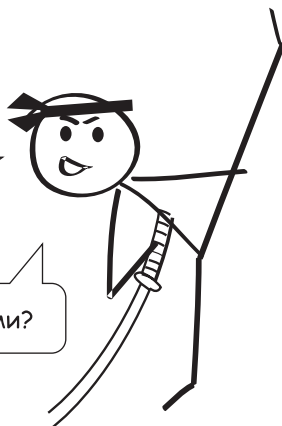
Однажды при написании программы для обслуживания торговли энергоресурсами наша команда выполнила несколько крупных серий рефакторинга базы кода и за несколько недель не добавила в программу практически никаких новых функций.

Неудивительно, что менеджеры не долго думая стали презирать понятие «рефакторинг» (для них он сводился к переработке старого без добавления чего-либо нового), и вскоре сверху стали приходить приказы бросить заниматься рефакторингом.

Желаю, чтобы с вами такого не произошло. Не прекращайте рефакторинг по ходу работы. Технический долг погашается тем хуже, чем позднее вы начинаете это делать, и последнее дело — считать рефакторинг чем-то неуместным.

Как насчет по-настоящему больших серий рефакторинга, на которые уходят недели?

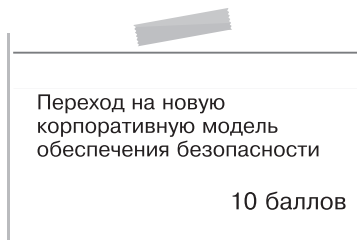
Как нам быть с ними?



Отличный вопрос. Иногда в программу требуется вносить и более серьезные изменения, чем обычное переименование нескольких переменных. Возможно, нужно заменить фреймворк или библиотеку, интегрировать новый инструмент. А быть может, мы слишком много ожидали от старого раз-рекламированного инструмента и теперь, когда дело дошло до сложной работы, должны перейти на новый инструмент.

Независимо от причин, крупный рефакторинг случается время от времени, и необходимо уметь им заниматься.

Если причина внесения изменений находится вне команды и мы сталкиваемся с чем-то, что нам приказывают сделать, то рефакторинг считается обычной пользовательской историей. Оцените ее, определите ее приоритет, покажите, сколько это будет стоить и какое влияние окажет на весь проект.



Крупный рефакторинг

Сложнее справляться с многочисленными субъективными ситуациями, с которыми может столкнуться наш брат-наемник. Но затраты на внесение таких изменений могут сторицей окупиться на дальнейшем пути.

Если вы планируете крупный рефакторинг, относящийся к такой «серой» области, ответьте на два вопроса, прежде чем приниматься за него.

- ☐ Близок ли конец проекта?
- ☐ Можно ли сделать необходимые изменения постепенно?

Обычно не стоит заниматься крупным рефакторингом в конце проекта, так как плодами своей работы вы все равно уже не воспользуетесь. То есть по окончании проекта от такой идеи лучше отказаться.

В постепенном рефакторинге клиента убедить проще, так как эта работа не будет отбирать у вас и у команды все рабочее время. Клиент будет радоваться новым функциям своего приложения, а вы тем временем будете понемногу выделять время на рефакторинг.

Действуйте в зависимости от ситуации. Посмотрите, что нужно сделать. И если считаете, что рефакторинг сэкономит вам немало времени, то начинайте его выполнять — возможно, вы действительно не пожалеете.

Где подробнее изучить эту тему

Мы лишь слегка коснулись очень важной темы рефакторинга. И в этой небольшой главе, конечно, не рассмотрены все вопросы, заслуживающие внимания.

По данной теме вам действительно стоит прочитать книгу Мартина Фаулера *Refactoring: Improving the Design of Existing Code* [FBB+99].

Другая стоящая книга принадлежит перу Майкла Фэзерса и называется *Working Effectively with Legacy Code* [Fea04].



Мастер-сэнсэй
и воин,
постигающий
искусство

УЧЕНИК: Мастер, бывают ли случаи, когда вообще не требуется заниматься рефакторингом кода?

МАСТЕР: Не считая масштабных серий рефакторинга, о которых мы говорили выше, — нет. Обычно рефакторинг кода нужно делать всякий раз, внося изменения в программу.

УЧЕНИК: Нужно ли что-то исправить, если мне регулярно приходится тратить целую итерацию на один только рефакторинг?

МАСТЕР: Нет. Просто такая ситуация неидеальна. Старайся выполнять рефакторинг небольшими порциями, и тогда не придется тратить на эту работу много времени. К тому же без неудач не обойтись, и иногда тебе потребуются крупные изменения. Но на них нужно идти только в крайнем случае. Не стоит заниматься ими регулярно.

Что дальше?

Мы хорошо поработали. Тестирование компонентов и рефакторинг вместе образуют сильнейшую связку, которой не сможет противостоять большинство плохо сработанных программ.

Но есть и еще одна практика, о которой необходимо знать. Она не только помогает улучшить структуру программы, но и указывает, сколько тестов потребуется.

Переходите к следующей главе и начинайте знакомиться с искусством тестирования через разработку. Вы узнаете, как написание тестов помогает в те минуты, когда смотришь на чистый лист и думаешь, с чего бы начать.

Глава 14

Разработка на основе тестирования



Вы забуксовали. Оказались в тупике. Вы целый день взираете на один и тот же фрагмент кода и просто не знаете, как его разобрать и даже с чего начать.

Если бы вы писали код, как Эрик!

Его код — это что-то особенное. Он работает, и все. Где бы и когда бы вы ни использовали любой фрагмент его кода, кажется, что Эрик заранее прочитал ваши мысли. Все, что вам нужно, — на месте. Причем это подкреплено целым комплектом автоматизированных тестов компонентов.

Как у него это получилось? И почему у вас не получается?

Чувствуя себя все неудобнее, но сознавая, что без помощи не обойтись, вы наконец набираетесь смелости и идете к столу Эрика. «Как ты смог написать такой хороший, чистый код?»

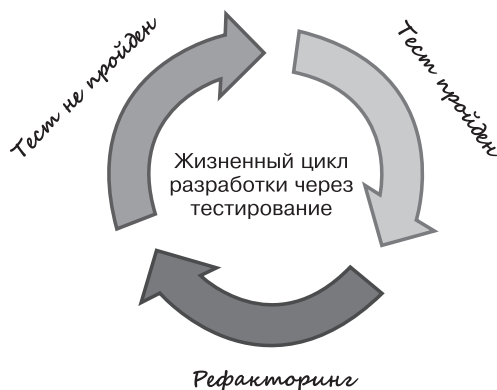
«Да все просто, — отвечает он. — Сначала я пишу тесты».

14.1. Сначала пишем тесты

Разработка на основе тестирования (test-driven development, TDD) — это техника разработки, в ходе которой применяются очень краткие рабочие циклы, обеспечивающие постепенное создание программы.

Вот как она строится.

1. *Красный*: прежде чем написать для системы любой новый код, пишется заведомо неуспешный тест компонента, демонстрирующий ваши намерения, которые вы собираетесь реализовать в новом коде. Здесь вы критически подходите к структуре кода.
2. *Зеленый*: затем вы делаете все, что необходимо для выполнения этого теста. Если вы уже видите всю реализацию целиком, добавьте новый код. Если нет — сделайте столько, сколько требуется для прохождения этого теста.
3. *Рефакторинг*: после этого вы возвращаетесь и подчищаете код, устраняя все погрешности, которые допустили, пытаясь выполнить тест. На этом этапе убираются дубли, и вы убеждаетесь, что код максимально экономичен, хорош и ясен.



Вы спрашиваете Эрика, как он узнает, что пора заканчивать с тестом. Эрик отвечает: «Я продолжаю писать тесты до тех пор, пока не убеждаюсь, что код выполняет все функции, упомянутые в пользовательской истории» (обычно это означает соответствие всем приемочным критериям данной истории).

Кроме того, Эрик руководствуется несколькими железными правилами, которые помогают ему не сбиться с пути.

Правило 1: не писать никакого нового кода, пока имеющийся код не пройдет созданный тест.

Правда, Эрик признает, что не следует этому правилу в 100 % случаев (некоторые вещи очень сложно протестировать заранее — таковы, например, пользовательские интерфейсы).

Но сущность метода, объясняет он, не в том, чтобы написать значительно больше кода, чем требуется. При написании теста мы неизбежно задумываемся о том, какую полезную нагрузку мы добавляем к программе, и не перегружаем программу техническими сложностями.

Правило 2: тестировать все, что может сломаться.

Следовать этому правилу не означает в буквальном смысле тестировать *все*, ведь на это может уйти целая вечность. Ключевое слово здесь — «*может*». Если существует вероятный шанс того, что что-то может сломаться, или мы хотим продемонстрировать, как программа будет действовать в определенных условиях, для такой ситуации нужно написать тест. Затем Эрик показывает пример того, над чем он сейчас работает.



«Знаешь, тут в Вегасе у нас есть несколько по-настоящему отчаянных игроков, — объясняет Эрик. — Наши ребята из отдела по обслуживанию хранилища данных хотят смоделировать самых влиятельных игроков. Они выясняют, что им нравится, а что нет, какие блюда и напитки они заказыв-

вают, а также все остальные детали, которые помогли бы завлечь этих людей в казино».

«Вот у нас в системе уже есть объекты с профилями определенных клиентов. Нужно выяснить, как сохранить такую информацию о профиле в базе данных».

Первым делом Эрик пишет тест. Здесь он предполагает, что код, который требуется протестировать, уже существует, и создает тест просто для того, чтобы самостоятельно убедиться, что этот код работает¹:

```
[Test]
public void Create_Customer_Profile()
{
    // настройка
    var manager = new CustomerProfileManager();
    // создание нового профиля клиента
    var profile = new CustomerProfile("Scotty McLaren" , "Hagis"
);
    // подтверждение того, что в базе данных еще нет такого
профиля
    Assert.IsFalse(manager.Exists(profile.Id));
    // добавление этого профиля
    int uniqueId = manager.Add(profile); // получение id из
базы данных
    profile.Id = uniqueId;
    // подтверждение того, что профиль был добавлен
    Assert.IsTrue(manager.Exists(uniqueId));
    // очистка
    manager.Remove(uniqueId);
}
```

Эрик уверен, что тест позволит судить о том, можно ли спокойно добавить профиль нового пользователя. Поэтому он ненадолго изменяет подход и начинает думать, как добиться выполнения данного теста.

Здесь ясно видно, что нужно для этого сделать (взять информацию о профиле пользователя и сохранить ее в базе данных). Эрик продолжает работать и добавляет новые функции².

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/test/Customer-ProfileManagerTest.cs>.

² Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/src/Customer-ProfileManager.cs>.

```
public class CustomerProfileManager
{
    public int Add(CustomerProfile profile)
    {
        // предположим, этот код сохраняет профиль
        // в базе данных, а потом возвращает его реальный
id
        return 0;
    }
    public bool Exists(int id)
    {
        // код, проверяющий существование клиента
    }
    public void Remove(int id)
    {
        // код, удаляющий клиента из базы данных
    }
}
```

Теперь Эрик запускает тест и видит, что он выполняется. Ура!

Рефакторинг — это последний этап разработки на основе тестирования. Программист возвращается и все перепроверяет (код теста, код команд, файлы конфигурации и все остальное, необходимое для выполнения теста), после чего проводит серьезный рефакторинг (см. главу 13).

После рефакторинга Эрик вновь возвращается к самому началу и задает себе вопрос: протестировал ли он все, что может сломаться? Одно из требований этой истории предписывает убедиться, что в коде нет никаких дублей.

Итак, программист повторяет тот же процесс. Пишет неуспешный тест, делает все, чтобы он стал успешным, а затем проводит рефакторинг.

Иногда приходится сталкиваться с проблемой вроде «что было раньше: курица или яйцо?» (прежде чем проверить, работает ли вставка, нужен код, проверяющий, существует ли уже в системе профиль нужного пользователя).

В такой ситуации Эрик просто приостанавливает текущий тест, добавляет новые функции (разумеется, после тестирования), а затем возвращается к работе, которой занимался ранее.

Вы благодарите Эрика за то, что он познакомил вас с разработкой на основе тестирования, и возвращаетесь на рабочее место, размышляя о тестах, рефакторинге и написании кода.

В чем суть предыдущего примера?

Давайте остановимся на минуту и подумаем, что сейчас произошло и почему это так важно.

При разработке на основе тестирования мы сначала пишем тесты, а потом пытаемся их выполнить. Кажется, что мы действуем в обратном порядке. Разумеется, в школе нас учили поступать иначе.

Но задумайтесь об этом ненадолго. Есть ли лучший способ спроектировать программу, чем представить себе, что она уже существует!

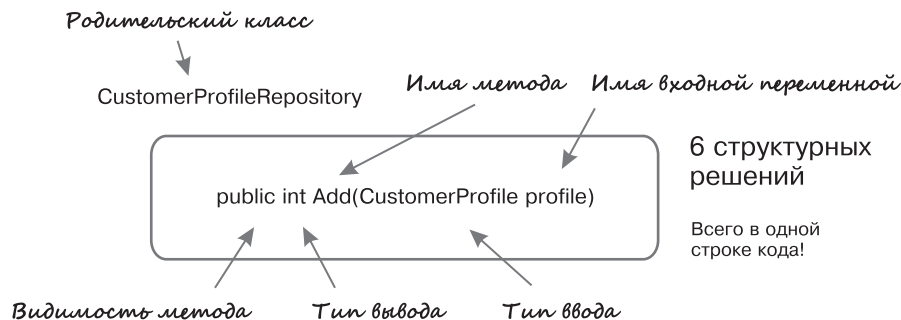
Именно этим мы и занимаемся при обработке на основе тестирования. Программист пишет необходимый код так, как если бы он уже существовал, а потом проводит тесты, удостоверяющие, что код работает. Это отличный способ гарантировать, что вы напишете только те функции, которые вам нужны, и при этом убедитесь, что все они работают.

И еще — не расстраивайтесь, если команда не может сразу адаптироваться к разработке на основе тестирования. Это довольно продвинутая технология, которая базируется на тестировании компонентов и рефакторинге. И, если честно, возможны такие случаи, когда разработка на основе тестирования неприменима и вам придется просто сидеть и думать, в чем заключается проблема.

Но, когда вы приобретете некоторый опыт и поймете, какой ритм и потенциал придает работе процесс написания самого маленького теста, который затем нужно пройти и переработать, вам обязательно понравится, как выглядит и тестируется ваш код.

14.2. Использование тестов для разрешения сложных ситуаций

При написании кода разработчик сталкивается с многочисленными сложностями. Посмотрите, сколько решений пришлось принять Эрику, пока он наполнял программный интерфейс своего приложения «Создание пользовательского профиля».



Пересчитайте. Перед вами шесть решений, компромиссов и развилок, о которых должен подумать разработчик, — и все это в одной строке кода! Неудивительно, что иногда от внимания разработчика что-то ускользает.

Разработка на основе тестирования помогает справиться с солидным количеством сложных случаев, встречающихся вашей команде каждый день. Вы пишете тесты и убеждаетесь, что существует неуспешный тест, еще *до того*, как добавляете код в программу.

Кроме того, разработка на основе тестирования обеспечивает надежность процесса проектирования. Сосредотачиваясь на единственном тесте и добиваясь его выполнения, вы избавляетесь от необходимости думать о сотне вещей сразу.

Можно заняться одной небольшой проблемой, постепенно разобраться с тем, как лучше всего ее решить, и немедленно получить ответ на вопрос, помогающий понять, в правильном ли направлении вы движетесь.

Есть и другие причины начинать работу с тестов.



Благодаря этому становится гораздо проще поддерживать и модифицировать базу кода. Меньше кода — проще программа. А простой дизайн программы облегчает изменения.

Но хватит разговоров. Давайте проведем пару тестов и посмотрим, как работать на таких испытаниях.



Эрик приглашает вас написать с ним на пару код, который сравнивает старшинство двух карт. Он считает, что функции должны выполняться в классе `Card`, и поможет вам составить тест.

Напишите выбранное вами имя метода в классе `Card`. Этот метод должен сравнивать две карты и определять, какая из них старше.



```
public void Compare_value_of_two_cards() {
```

```
    Card twoOfClubs = Card.TWO_OF_CLUBS;
    Card threeOfDiamonds = Card.THREE_OF_DIAMONDS;
```

Здесь напишите
ваш тест

```
}
```

Представьте, что код уже существует, — просто наберите его!

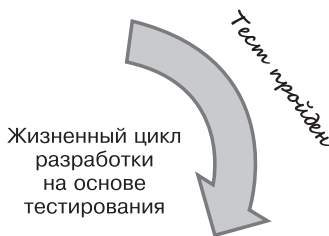
Допустим, программа выглядит так¹:

```
[Test]
public void Compare_value_of_two_cards()
{
    Card twoOfClubs = Card.TWO_OF_CLUBS;
    Card threeOfDiamonds = Card.THREE_OF_DIAMONDS;
    Assert.IsTrue(twoOfClubs.IsLessThan(threeOfDiamonds));
}
```

¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrap/code/tdd/test/CardTest.cs>.

Эрик вручает вам клавиатуру и спрашивает, что бы вы сделали, чтобы тест удалось пройти. У вас получается нечто следующее¹:

```
public bool IsLessThan(Card newCard)
{
    int thisCardValue = value;
    int newCardValue = newCard.value;
    return thisCardValue < newCardValue;
}
```



После того как тест пройден, Эрик спрашивает, не хотите ли вы провести здесь рефакторинг. Вы вносите изменения. Когда тест и метод доработаны, ваш код приобретает вид, показанный ниже.

Код из файла CardTest.cs:

```
[Test]
public void Compare_value_of_two_card()
{
    Assert.IsTrue(Card.TWO_OF_CLUBS.IsLessThan(Card.THREE_OF_
DIAMONDS));
}
```

Код из файла Card.cs:

```
public bool IsLessThan(Card newCard)
{
    return value < newCard.value;
}
```



¹ Ссылка для скачивания: <http://media.pragprog.com/titles/jtrapp/code/tdd/src/Card.cs>.

Когда этап цикла разработки на основе тестирования закончен, Эрик улыбается и говорит: «Думаю, ты все понял!» Вы, полный решимости испробовать такие методы на собственном коде, благодарите Эрика и отправляетесь на рабочее место, чтобы самостоятельно написать несколько тестов.

Где подробнее изучить эту тему

Чтобы прочувствовать смысл разработки на основе тестирования, рекомендую прочитать книгу Кеннета Бека *Test Driven Development: By Example* [Вес02], в которой дается немало хороших советов и объясняются более глубокие механизмы разработки на основе тестирования. Из нее вы также узнаете, как задействовать эти механизмы себе на пользу.



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, я не совсем понимаю разработку на основе тестирования. Как мне писать тесты для проверки кода, который еще не существует?*

МАСТЕР: *Пиши тесты так, как если бы необходимый код уже был у тебя в распоряжении.*

УЧЕНИК: *Но как мне узнать, что именно нужно тестировать?*

МАСТЕР: *Тестируй то, что требуется.*

УЧЕНИК: *То есть я должен просто писать тесты для проверки необходимых компонентов, и все, что нужно, появится в системе как по волшебству?*

МАСТЕР: *Да.*

УЧЕНИК: *Можно ли поподробнее узнать о том, как именно происходит вся эта магия?*

МАСТЕР: *Нет никакой магии. Ты просто выражаешь свои потребности в виде тестов. Создавая код таким образом, ты гарантируешь, что напи-*

шешь только то, что действительно требуется. Ты пользуешься тестами как инструментом, помогающим осознать твои намерения. Вот почему разработку на основе тестирования часто относят к методам проектирования, а не тестирования.

УЧЕНИК: *То есть разработка через тестирование действительно касается больше проектирования, чем тестирования?*

МАСТЕР: *Это чрезмерное упрощение. Тестирование — основная часть данного метода, поскольку именно с помощью тестов мы доказываем, что создаваемый нами код работает. Но мы не можем закончить тестирование, не выполнив предварительного проектирования и не отразив наших намерений в коде.*

УЧЕНИК: *Спасибо, Мастер. Мне нужно подумать об этом на досуге.*

Что дальше?

Вы уже чувствуете, как выстраиваются практические навыки? Тестирование компонентов позволяет удостовериться, что создаваемые нами элементы функционируют. Рефакторинг помогает не усложнять код. А разработка на основе тестирования представляет собой мощный инструмент, помогающий справиться со сложностью проектирования.

Осталось только собрать все вместе и убедиться, что ваш проект полностью готов к практической эксплуатации.

А теперь перейдем к заключительной главе и испытаем силу непрерывной интеграции!

Непрерывная интеграция: обеспечение готовности к работе



Приготовьтесь выдать результат, пригодный для практического использования. Научившись непрерывной интеграции своих программ, вы будете избавляться от ошибок практически сразу после их появления, стоимость внесения изменений в программу уменьшится и вы сможете уверенно ее внедрять.

Приступить к изучению непрерывной интеграции нужно прямо сейчас!

15.1. Презентация

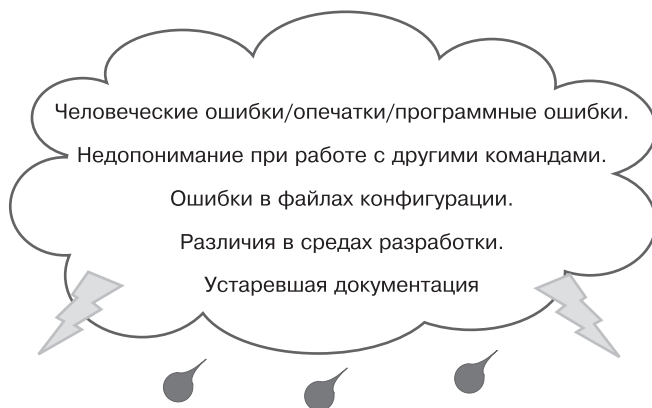
Начнем с хорошего. Директор приглашает нескольких влиятельных инвесторов и хочет продемонстрировать им последнюю версию вашего флагманского продукта — симулятора игры блек-джек. А теперь плохая новость: гости прибудут через час!

Таким образом, у вас меньше 60 минут на то, чтобы создать стабильную сборку, отправить ее на тестовый сервер и подготовить к демонстрации.

Что делать?

Прежде чем ответить на этот вопрос, обдумайте, что может пойти не так при развертывании программы.

Проблемы, которые могут возникнуть при развертывании программы



Это неприятности, которых требуется избежать либо хотя бы свести к минимуму в процессе непрерывной интеграции программы. Нам требуется развить культуру управления подготовкой продукта к производству и умение продемонстрировать программу кому угодно, когда угодно и где угодно.

Давайте рассмотрим два способа добиться этого.

Сценарий 1. Аврал

Один час! Прямо скажем, времени в обрез. Вы подаете сигнал тревоги, собираете команду и начинаете задавать вопросы со скоростью пулемета.

- ❑ У кого новейшая сборка?
- ❑ Чья настольная сборка самая стабильная?

❑ Кто может максимально быстро представить пример и запустить его? Как известно, если хочешь сделать что-то хорошо — сделай это сам. Вы сообщаете команде, что интеграция будет проходить на вашей машине и у всех есть 15 минут, чтобы вставить сделанные изменения в вашу ветку кода.

Начинается интеграция кода, и возникают новые проблемы. Интерфейсы в основных классах изменились. Файлы конфигурации были модифицированы. Файлы из старой системы претерпели рефакторинг, и в них явно чего-то не хватает. Сведение всех элементов воедино сразу превращается в кошмар.

Проклиная про себя директора за то, что он не дал вам достаточно времени на подготовку, вы приказываете команде откомментировать и исключить все, что мешает осуществить интеграцию.

Затем, в последние 5 минут вдруг появляется свет в конце тоннеля — пошла компиляция!

Но надежда сменяется катастрофой — инвесторы явились на 5 минут раньше срока. На тестирование нет времени.

Вы скрещиваете пальцы и разворачиваете программу, запускаете демонстрационную версию и... она отказывается работать. Вы быстро устраняете проблему и запускаете программу, но она «падает» еще раз, как только вы проходите вводную заставку.

Директор в некотором замешательстве вынужден признать, что демоверсия пока не работает. Тогда он спрашивает, не может ли команда показать вместо программы демонстрационные модели.

Сценарий 2. Ничего особенного

Вы знаете, что до демонстрации у вас есть еще целый час, поэтому сообщаете команде, что по истечении этого времени работу предстоит предъявить инвестору. Вы говорите, что были бы очень благодарны, если бы все успели быстренько закругиться и проверить то, что уже готово.

После того как все сохранили сделанное, вы проверяете последнюю версию кода, проводите тесты, убеждаетесь, что все работает, и отправляете программу на тестовый сервер. На все уходит не более 5 минут.

Инвесторы прибывают раньше времени. Демоверсия работает отлично. А ваш босс благодарит вас за то, что вы, практически не имея времени на подготовку, смогли показать такой класс, и вручает вам что-то, о чем вы

давно мечтали, — например, ключи от VIP-санузла. Ладно, допустим, вам сто лет не нужен этот санузел, но идею вы, полагаю, поняли.

Подготовка к презентации и подготовка кода к реальному использованию — не обязательно нервное, трудоемкое и выводящее из себя событие.

Нужно, чтобы сборка, интеграция и развертывание вашей программы не были чем-то из ряда вон выходящим. А для этого необходимо наладить качественный и гладкий процесс интеграции и воспитать культуру обеспечения готовности продукта к производству.

15.2. Культура подготовки продукта к производству

В экстремальном программировании есть выражение, гласящее, что производство начинается с первого дня проекта. С того момента, как вы запишете первую строку кода, начинается подготовка проекта к реальному использованию, и все, что вы делаете, — просто вносите изменения в живую систему.

Из этого вытекает совершенно нетипичный подход к коду. Вы воспринимаете производство и внедрение не как события, относящиеся к какому-то отдаленному будущему, а понимаете, что уже здесь и сейчас вы и команда занимаетесь производством и это требует от вас необходимого уровня ответственности.

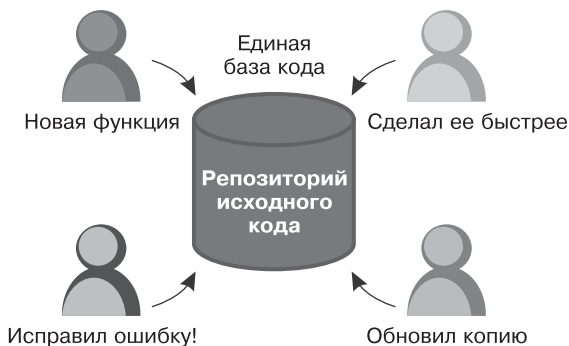
Сторонникам гибкой разработки нравится такое отношение к производству, так как оно учитывает, что для производства программы нужно гораздо больше, чем просто написать код, и прививает командам ощущение, что все изменения вносятся в систему, подготавливаемую к реальному использованию.

Правда, развитие культуры обеспечения готовности продукта к производству требует определенных усилий. В частности, такая культура невозможна без строгой дисциплины, а соблазн повременить с подготовкой кода к реальному использованию, но зато соблюсти сроки, может быть очень велик.

Но те, кто начинает готовиться к выпуску заранее, могут при реализации проектов буквально «развернуться на пятачке». Развертывание проходит легко, изменения в систему вносятся регулярно и уверенно, а реагировать на потребности клиента удастся быстрее, чем конкурентам.

В развитии такой культуры может помочь непрерывная интеграция.

15.3. Что такое непрерывная интеграция



Непрерывная интеграция — это процесс постоянного учета изменений, которые разработчики вносят в программу, и интеграция этих изменений в единое целое изо дня в день.

Данный процесс можно сравнить с написанием книги. Предположим, что вы с соавтором работаете вместе над главой и вам нужно согласовывать изменения. Неплохо было бы объединить несколько простых фрагментов, в которых изменяется пара фраз.

Буряя лиса перепрыгнула через ленивую собаку



Ерунда!



Буряя лиса перепрыгнула через ленивую *черную* собаку

Если достаточно долго не сводить варианты воедино, могут начаться проблемы.

Буряя лиса перепрыгнула через ленивую собаку. Но потом собака вытворила тот еще трюк! Она испекла порцию шоколадных печений и стала угощать ими всех, кого встречала на улице. Разумеется, это заметили кошки. Они разозлились и решили достойно ответить на собачью инициативу с печеньем собственной затеей — взяли одарить всю улицу шоколадным сырным пирогом!



А теперь найдите 6 отличий



Буряя лиса перепрыгнула через ленивую собаку. Но потом собака вытворила тот еще трюк! Она испекла порцию шоколадных маффинов и стала угощать ими всех, кого встречала на проспекте. Разумеется, это заметили кошки. Они разозлились и решили достойно ответить на собачью инициативу с маффинами собственной затеей — взяли одарить всю улицу ванильным сырным пирогом!

При написании программ возникает аналогичная ситуация. Чем дольше вы работаете, не объединяя сделанные изменения с программами коллег, тем сложнее будет впоследствии интегрировать все созданные компоненты воедино.

Давайте посмотрим, как это происходит на практике.

Работать надо быстро



Однажды я работал над проектом, в рамках которого нужно было создать крупный инструмент для автоматизированного написания тестов записи и воспроизведения. На самом деле инструмент был отличный, все стали писать с его помощью свои тесты, а о низкоуровневых и быстрых тестах компонентов просто забыли.

Некоторое время все шло хорошо. Но по мере накопления тестов, созданных автоматически, время сборки постепенно выросло с 10 минут до более чем трех часов.

Нас это просто погубило. Люди прекратили пользоваться сборкой. Они начали пренебрегать тестированием, и неисправные сборки стали в нашем проекте нормой.

Не повторяйте нашей ошибки и не позволяйте сборкам вырастать до огромных размеров. Сборка должна занимать не более 10 минут — это железное правило. В некрупных проектах удастся добиться, чтобы время сборки не превышало 5 минут.

15.4. Как это работает

Для настройки системы непрерывной интеграции требуется несколько вещей:

- ❑ репозиторий исходного кода;
- ❑ процесс постановки кода на учет;
- ❑ автоматизированная сборка;
- ❑ готовность обрабатывать небольшие фрагменты.

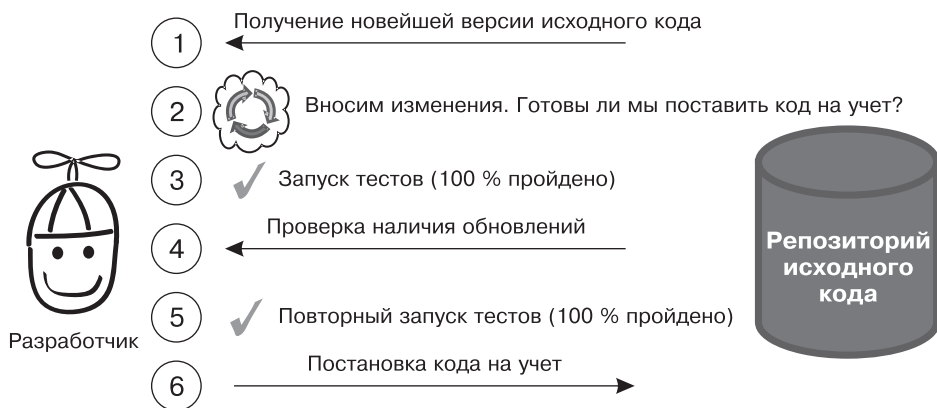
В репозиториях исходного кода хранится ваша программа и ведется контроль ее версий. Именно здесь команда «учитывает» код. Это точка интеграции всего кода, и здесь же содержится его контрольный экземпляр. В данном случае вам очень пригодятся свободные репозитории исходного кода, такие как Git или Subversion.

Просто убедитесь, что в команде не происходит пессимистического блокирования (так называется ситуация, при которой в каждый момент над определенным файлом может работать только один человек). Это нервирует разработчиков и мешает команде совместно владеть базой кода.

Гораздо интереснее типичный процесс постановки кода на учет (check-in process). Рассмотрим, как он может строиться в обычной гибкой команде.

15.5. Организация постановки кода на учет

Как правило, для разработчика в составе гибкой команды процесс постановки кода на учет выглядит примерно так.



1. *Получение новейшей версии исходного кода из репозитория.* Прежде чем приступить к какой-либо новой работе, нужно убедиться, что у вас в распоряжении новейший и самый лучший код из репозитория. На данном этапе вы проверяете, какова новейшая сборка, и начинаете работу с чистого листа.
2. *Внесение изменений.* Затем вы выполняете свою работу: добавляете в программу новую функцию, исправляете ошибку или делаете что-то еще, что собирались.

3. *Проведение тестов.* Чтобы убедиться, что внесенные вами изменения не повредили другие фрагменты базы кода, вы повторно запускаете все тесты и убеждаетесь, что они по-прежнему выполняются.
4. *Проверка на наличие обновлений.* Удостоверившись, что измененная программа работает, вы берете из репозитория еще одно обновление, чтобы проверить, не внес ли кто-нибудь в код новые изменения, пока вы работали.
5. *Повторное проведение тестов.* После этого еще раз прогоняются все тесты — нужно гарантировать, что внесенные вами изменения не конфликтуют с другими изменениями, сделанными коллегами с тех пор, как вы начали работать над последними задачами.
6. *Постановка кода на учет.* Итак, все работает. Сборка происходит. Тесты выполняются. У нас в руках — новейшая версия. Ее можно ставить на учет.

Кроме описанного процесса постановки на учет, существует еще несколько вещей, которые необходимо и недопустимо делать для безупречной сборки.

| Необходимо | | Недопустимо |
|---|--|--|
| Проверять наличие обновлений |  | Портить сборку |
| Проводить все тесты | | Ставить код на учет поверх поврежденных сборок |
| Регулярно ставить код на учет | | Оформлять в виде комментариев неуспешные тесты компонентов |
| В качестве наивысшего приоритета рассматривать исправление дефектной сборки | | |

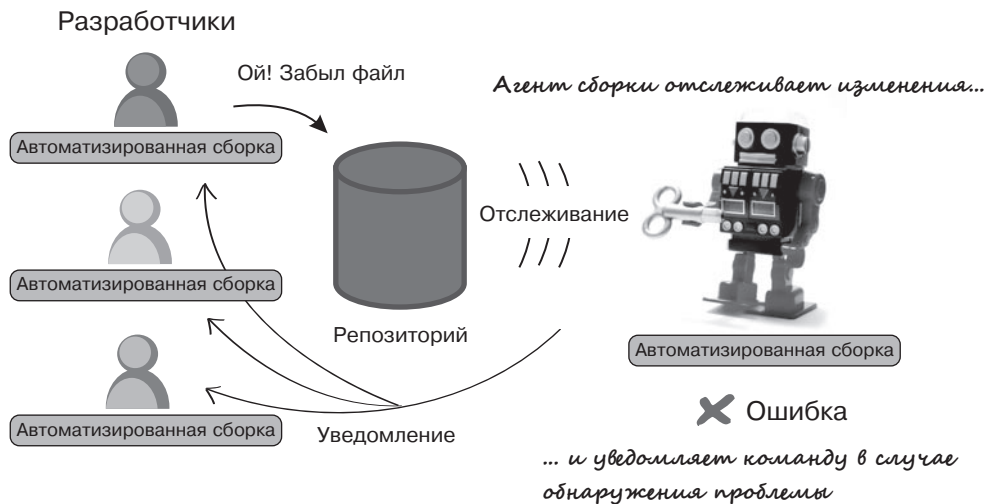
В конечном счете все сводится к внимательному отношению к сборке. Нужно гарантировать, что она всегда находится в рабочем состоянии, и помогать товарищам, если она ломается (что время от времени происходит).

15.6. Организация автоматизированной сборки

Следующий этап — организация автоматизированной сборки. На самом деле именно она образует каркас процесса непрерывной интеграции, проводимого вашей командой.

Хорошая автоматизированная сборка компилирует код, выполняет тесты и вообще делает все, что регулярно требуется в процессе сборки проекта.

В процессе разработки на основе тестирования программисты постоянно пользуются такой сборкой, а агенты сборки (например, CruiseControl¹) применяют данный механизм всякий раз, когда обнаруживают изменения в репозитории с исходным кодом.



Подобные сборки также позволяют автоматизировать развертывание программы для промышленного использования и избавиться от множества человеческих ошибок.



Любая сборка лучше проходит в автоматическом режиме, то есть при минимальном участии человека. Кроме того, сборка должна быть быстрой, так как вы и ваша команда постоянно будете ее запускать, не реже нескольких раз в день (поэтому хорошая сборка завершается не более чем за 10 минут).

В большинстве современных языков программирования есть собственные фреймворки автоматизированной сборки (automated build frameworks): Ant

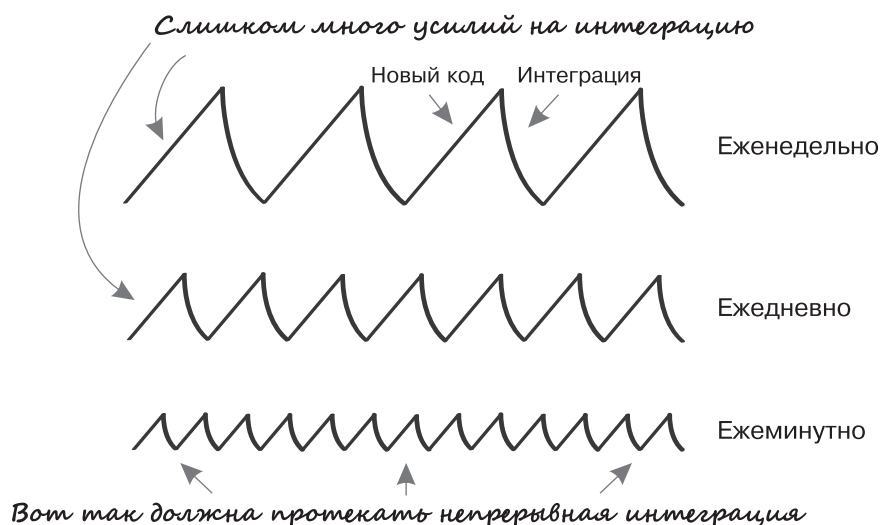
¹ <http://cruisecontrol.sourceforge.net/>.

в Java, NAnt или MS-Build в .NET, rake в Rails. Если в языке, с которым вы работаете, такого фреймворка нет, вы обычно можете сами его создать с помощью BAT-файлов DOS или скриптов Unix.

Прелесть постановки кода на учет и автоматизированных сборок ясна. Но для того, чтобы они работали, вы должны уметь обращаться с небольшими фрагментами.

15.7. Работа с небольшими фрагментами

Как и тестирование при разработке на основе тестирования, так и интеграция протекают гораздо лучше, если делать все небольшими фрагментами.



Зачастую команды могут работать днями и неделями, не интегрируя своей работы. Это слишком большие промежутки. Необходимо интегрировать код каждые 10–15 минут (во всяком случае, не реже раза в час).

Не волнуйтесь, если не можете ставить код на учет с такой же частотой. Просто учитывайте, что чем чаще вы выполняете интеграцию, тем проще она идет.

Итак, если объединять код заранее и часто, можно надежно избавиться от неприятностей, связанных с крупными интеграциями.

Где подробнее изучить эту тему

Непрерывная интеграция стала такой распространенной практикой, что почти любую информацию о ней можно найти в Интернете.

Об этом методе хорошо рассказано в статье англоязычной «Википедии»¹, а одна из первых статей о непрерывной интеграции размещена на сайте Мартина Фаулера².



*Мастер-сэнсэй
и воин,
постигающий
искусство*

УЧЕНИК: *Мастер, мы определенно не сможем подготовить весь код к реальному использованию за одну лишь первую итерацию. Что же на самом деле понимается под «реальным использованием»?*

МАСТЕР: *Это отношение к работе. Когда вы пишете готовый к использованию код, вы сегодня же тестируете и интегрируете свою программу. Если вы замечаете ошибку, то исправляете ее сразу же. Вы не откладываете ее в долгий ящик и не думаете, что еще сможете вернуться к ней когда-нибудь потом. Вы относитесь к программе так, как будто она должна работать уже сегодня, а не в далеком будущем. Да, после первой итерации у вас, конечно же, не будет проработано все и вся и вы можете не приступать к развертыванию, пока в программе не наберется определенного количества функций. Но если перед вами стоит перспектива развертывания и вы знаете, что ваша программа работает, то вы признаете, что большую часть своего существования эта программа проведет не в разработке, а в действии. Так вы привыкаете к мысли, что все изменения вносятся в практически значимую систему.*

УЧЕНИК: *Что, если я не могу построить целостную систему, так как моя работа — лишь часть большого проекта?*

¹ http://en.wikipedia.org/wiki/Continuous_integration. Русскоязычный вариант: http://ru.wikipedia.org/wiki/Непрерывная_интеграция.

² <http://martinfowler.com/articles/continuousIntegration.html>.

МАСТЕР: *Тогда собирай, тестируй и развертывай то, что можешь. Однажды тебе потребуется интегрировать свой фрагмент со всем остальным проектом. Приложи все усилия, чтобы твой фрагмент был готов к использованию — тогда ты сможешь внести необходимые изменения в тех частях программы, к которым имеешь доступ. Но пусть тот факт, что ты работаешь всего лишь над небольшим фрагментом, не заставит тебя отказаться от автоматизации сборки и непрерывной интеграции программы.*

Вот и все, народ!

Итак, вы прочитали книгу. Мы превосходно поработали над важнейшими практиками гибкой разработки программ.

- ❑ Тестирование компонентов — чтобы убедиться, что написанная нами программа работает.
- ❑ Рефакторинг — искусство простоты и сохранения кода чистым и удобочитаемым.
- ❑ Разработка на основе тестирования — для проектирования и работы со сложными деталями.
- ❑ Непрерывная интеграция — регулярное приведение работы к общему знаменателю и обеспечение готовности программы к реальному использованию.

Без этих методов в гибком проекте мало что будет работать, и мы быстро вернемся в программистский каменный век, когда все работали по принципу «кодируй и исправляй».

15.8. Что дальше?

Поздравляю! Теперь вы вооружены и очень опасны, так как обладаете знаниями и умениями, необходимыми для запуска, планирования и выполнения вашего первого гибкого проекта.

Дальнейший путь выбираете вы сами.

Если вы только начинаете проект, попробуйте составить для него стартовую колоду (см. главу 3). Соберите всех и двигайтесь в нужном направлении, не стесняясь задавать неудобные вопросы с самого начала проекта.

Или, если проект уже в разгаре (а вам совершенно ясно, что план никуда не годится), вы можете устроить перезагрузку, организовав семинар по сбору

пользовательских историй (см. раздел 6.4). Выберите несколько самых важных историй и посмотрите, сможете ли вы реализовывать по нескольку из них каждую неделю. Затем на основе собранного материала постройте новый план.

Если вы испытываете трудности на инженерном фронте, может потребоваться пересмотреть некоторые практики разработки, убедиться, что вы не халтурите при тестировании и регулярно погашаете технический долг.

Карты нет. Вам придется самостоятельно определить, что лучше всего подойдет для вас и вашего проекта. Но знайте, что необходимые инструменты у вас есть, — и я готов поспорить, что вы уже представляете, что делать.

Так чего же вы сидите?

Действуйте!

И напоследок



Это вопрос выбора.

Никто не может запретить вам создавать высококлассные программы. Никто не мешает вам заранее прорабатывать детали и честно говорить с клиентом о состоянии проекта и о том, что нужно сделать.

Не поймите меня неправильно — я ведь не говорю, что это просто. Против нас — десятилетия истории и накопленного опыта. Но необходимо понять, что принцип вашей работы и ее качество зависят от вас и только от вас.

Не проповедуйте.

Не учите людей жить.

Вместо этого показывайте им личный пример, признавайте, что иногда придется рассчитывать только на себя, и делайте то, что нужно.

Да, чуть не забыл...

Не зацикливайтесь на гибкой разработке

Во многих командах, обращающихся к этой методологии, часто возникает вопрос: «А мы уже стали гибкой командой?»

И этот вопрос совершенно оправдан. Когда делаешь что-то впервые, то, разумеется, хочешь знать, как ты это делаешь и все ли получается «по науке».

И все бы хорошо, но нужно осознать, что по гибкой разработке нет какой-то книги, содержащей непреложную истину. Если бы она даже и была, то не содержала бы ответов на все вопросы. Не существует такого перечня литературы по гибкой методологии, который можно прочитать от корки до корки и стать настоящим самураем.

Это путешествие, а не рейс. Вы никогда его не завершите.

И не забудьте о том, что смысл заключается не в том, чтобы быть гибким, а в том, чтобы писать высококлассные программы и оказывать клиенту безупречные услуги.

Будьте осторожны: если у вас появляется ощущение, что вы все поняли и всего достигли, — это означает, что вы потеряли гибкость.

Поэтому не зацикливайтесь на определенных практиках. Возьмите из этой книги то, что сможете, и реализуйте приобретенные навыки в вашей уникальной ситуации и в вашем контексте. И вместо того, чтобы задаваться вопросом, «гибко» ли вы работаете, задайте себе следующие вопросы:

- ☐ удастся ли нам каждую неделю создавать что-то ценное;
- ☐ есть ли у нас постоянное стремление к совершенству?

Если вы утвердительно ответили на оба этих вопроса, то вы достигли гибкости.

Приложения

Приложение А. Принципы гибкой разработки

Здесь приведены Манифест гибкой разработки¹ и 12 основополагающих принципов гибкой разработки, взятые с сайта этого Манифеста².

А1. Манифест гибкой разработки

Разрабатывая программное обеспечение и помогая другим делать это, мы стараемся найти наилучшие подходы к разработке. В процессе этой работы мы пришли к тому, что:

- ❑ *личности и их взаимодействие* важнее процессов и инструментов;
- ❑ *работоспособное программное обеспечение* важнее обширной документации;
- ❑ *сотрудничество с заказчиком* важнее согласования условий контракта;
- ❑ *умение реагировать на изменения* важнее следования плану.

Таким образом, хотя и существует ценность в понятиях, стоящих в правой части этих сравнений, мы все же больше ценим понятия, стоящие слева.

¹ <http://agilemanifesto.org>.

² <http://agilemanifesto.org/principles.html>.

A2. 12 принципов гибкой разработки

1. Нашим главным приоритетом является удовлетворение заказчика посредством ранней и непрерывной поставки работоспособного программного обеспечения.
2. Изменение требований приветствуется даже на поздних стадиях разработки. Гибкие процессы используют изменения как средство обеспечения конкурентных преимуществ для заказчика.
3. Работоспособное программное обеспечение следует выпускать часто: от раза в несколько недель, до раза в несколько месяцев, отдавая предпочтение коротким интервалам.
4. Представители бизнеса и разработчики должны работать вместе в течение всего проекта.
5. Проекты необходимо строить вокруг мотивированных профессионалов. Предоставьте свободу и поддержку, в которой они нуждаются, и доверьте им самим делать работу.
6. Наиболее эффективным способом передачи информации команде проекта (а также внутри команды) является непосредственное живое общение.
7. Основным показателем прогресса проекта является работоспособное программное обеспечение.
8. Гибкие процессы поощряют разработку с постоянной скоростью. Спонсоры проекта, разработчики и пользователи должны быть способны поддерживать постоянную скорость на неограниченной дистанции.
9. Непрерывное внимание к техническому совершенству и хорошему дизайну увеличивает степень гибкости.
10. Простота — искусство минимизации лишней работы — является существенным фактором.
11. Наилучшие требования, архитектура и дизайн создаются самоорганизующимися командами.
12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Приложение Б. Интернет-ресурсы

Есть много отличных новостных групп, ресурсов и других сайтов, которые вы можете посетить в ходе вашего путешествия. Ниже приведено несколько ссылок, на которых я рекомендую остановиться подробнее, почитать о гибкой разработке программ и о том, как она строится.

- ❑ <http://tech.groups.yahoo.com/group/extremeprogramming>.
- ❑ <http://groups.yahoo.com/group/scrumdevelopment>.
- ❑ <http://tech.groups.yahoo.com/group/leanagile>.
- ❑ <http://finance.groups.yahoo.com/group/kanbandev>.
- ❑ <http://tech.groups.yahoo.com/group/agile-testing>.
- ❑ <http://tech.groups.yahoo.com/group/agile-usability>.
- ❑ <http://finance.groups.yahoo.com/group/agileprojectmanagement>.

Приложение В. Список литературы

- [Bec00] *Kent Beck*. Extreme Programming Explained: Embrace Change. — Addison-Wesley, Reading, MA, 2000.
- [Bec02] *Kent Beck*. Test Driven Development: By Example. — Addison-Wesley, Reading, MA, 2002.
- [Blo01] *Michael Bloomberg*. Bloomberg by Bloomberg. — John Wiley & Sons, Hoboken, NJ, 2001.
- [Car90] *Dale Carnegie*. How to Win Friends and Influence People. — Pocket, New York, 1990.
- [DCH03] *Mark Denne and Jane Cleland-Huang*. Software by Numbers: Low-Risk, High-Return Development. — Prentice Hall, Englewood Cliffs, NJ, 2003.
- [DL06] *Esther Derby and Diana Larsen*. Agile Retrospectives: Making Good Teams Great. — The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.
- [Eva03] *Eric Evans*. Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley Professional, Reading, MA, first edition, 2003.
- [FBB+99] *Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts*. Refactoring: Improving the Design of Existing Code. — Addison-Wesley Longman, Reading, MA, 1999.
- [Fea04] *Michael Feathers*. Working Effectively with Legacy Code. — Prentice Hall, Englewood Cliffs, NJ, 2004.
- [GC09] *Lisa Gregory and Janet Crispin*. Agile Testing: A Practical Guide for Testers and Agile Teams. — Addison-Wesley, Reading, MA, 2009.
- [HH07] *Dan Heath and Chip Heath*. Made to Stick: Why Some Ideas Survive and Others Die. — Random House, New York, 2007.
- [HT03] *Andrew Hunt and David Thomas*. Pragmatic Unit Testing In Java with JUnit. — The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.
- [HT04] *Andrew Hunt and David Thomas*. Pragmatic Unit Testing In C# with NUnit. — The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2004.

- [Joh98] *Spencer Johnson*. Who Moved My Cheese? An Amazing Way to Deal with Change in Your Work and in Your Life. — Putnam Adult, New York, 1998.
- [Lik04] *Jeffrey Liker*. The Toyota Way. — McGraw Hill, New York, 2004.
- [McC06] *Steve McConnell*. Software Estimation: Demystifying the Black Art. — Microsoft Press, Redmond, WA, 2006.
- [Moo91] *Geoffrey A. Moore*. Crossing the Chasm. — Harper Business, New York, 1991.
- [Sch03] *David Schmaltz*. The Blind Men and the Elephant. — Berrett-Koehler, San Francisco, 2003.
- [SD09] *Rachel Sedley and Liz Davies*. Agile Coaching. — The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.
- [Sur05] *James Surowiecki*. The Wisdom of Crowds. — Anchor, New York, 2005.

Расмуссон Дж.
Гибкое управление IT-проектами.
Руководство для настоящих самураев
Перевод с английского О. Сивченко

Заведующая редакцией
Руководитель проекта
Ведущий редактор
Научный редактор
Художник
Корректор
Верстка

*К. Галицкая
Д. Виноцкий
М. Моисеева
П. Хасанов
Л. Адуевская
Е. Павлович
О. Богданович*

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 28.03.12. Формат 70×100/16. Усл. п. л. 21,930. Тираж 2000. Заказ 0000.

Отпечатано с готовых диапозитивов в ГППО «Псковская областная типография».
180004, Псков, ул. Ротная, 34.





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!**

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

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным
направлениям: история и публицистика, экономика и финансы, менеджмент
и маркетинг, компьютерные технологии, медицина и психология.

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. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: yashny@yandex.ru

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 229-68-09; e-mail: pitvolga@samara-ttk.ru, pitvolga@mail.ru


УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: gv@minsk.piter.com

 Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых
партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: rodionova.tatyana@piter.com

 Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

 Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: fokina@piter.com

 Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225

Для заметок
