



Постигая Agile

ЦЕННОСТИ, ПРИНЦИПЫ, МЕТОДОЛОГИИ

Эндрю Стеллман
Дженнифер Грин

Эту книгу хорошо дополняют:

Scrum. Революционный метод управления проектами

Джефф Сазерленд

Управление продуктом в Scrum

Роман Пихлер

Канбан

Дэвид Андерсон

Andrew Stellman
Jennifer Greene

Learning Agile

Understanding Scrum, XP, Lean, and Kanban

O'Reilly
2015

*Эндрю Стеллман
Дженнифер Грин*

Постигая Agile

Ценности, принципы, методологии

Перевод с английского Светланы Пасерба

Москва
«Манн, Иванов и Фербер»
2017

УДК 658.5.012.1
ББК 65.291.217
С79

*Издано с разрешения O'Reilly Media, Inc.
На русском языке публикуется впервые*

Благодарим за помощь в подготовке издания компанию ScrumTrek
в лице Алексея Пименова, Сергея и Александры Липчанских

Стеллман, Эндрю

С79 Постигаая Agile. Ценности, принципы, методологии / Эндрю Стеллман, Дженни-
фер Грин ; пер. с англ. С. Пасерба. — М. : Манн, Иванов и Фербер, 2017. — 448 с.

ISBN 978-5-00100-614-5

Эта книга рассказывает о самых популярных agile-подходах — Scrum, XP (экстремальное программирование), Lean (бережливое программирование) и Канбан. Она познакомит вас с методами, работающими в повседневной жизни, а также с базовыми ценностями и принципами, которые помогут вашей команде полностью изменить свой подход к работе над проектами. Вы начнете лучше разбираться в конкретных agile-подходах и сможете сразу внедрить их на практике. А главное, вы поймете, как превратить группу сотрудников, добавляющих в свою работу Agile, в настоящую команду, которая действительно улучшает способ создания продукта и добивается выдающихся результатов.

УДК 658.5.012.1
ББК 65.291.217

*Все права защищены.
Никакая часть данной книги не может
быть воспроизведена в какой бы то ни было форме
без письменного разрешения владельцев авторских прав.*

Authorized Russian translation of the English edition of Learning Agile, ISBN 9781449331924. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls rights to publish and sell the same.

ISBN 978-5-00100-614-5

© Andrew Stelman and Jennifer Greene, 2015
© Перевод на русский язык, издание на русском языке,
оформление. ООО «Манн, Иванов и Фербер», 2017

Оглавление

Предисловие партнера	11
Предисловие	13
Глава 1. Обучая Agile	15
Что такое Agile?	16
Кому следует прочитать эту книгу	21
Цели, которые мы ставим в этой книге	22
Продвижение Agile в ваше сознание любыми необходимыми средствами	23
Структура книги	27
Глава 2. Понимание ценностей Agile	31
Руководитель команды, архитектор и менеджер проекта заходят в бар... ..	33
Серебряной пули не существует	36
Agile для спасения! (Правильно?)	39
Раздробленное видение	44
Agile-манифест помогает командам видеть цели применения каждой практики ..	51
Понимание слона	58
С чего начинать при работе с новой методологией	65
Глава 3. Agile-принципы	71
12 принципов гибкой разработки программного обеспечения	72
Клиент всегда прав, не так ли?	73
Реализация проекта	76
Общение и совместная работа	86
Выполнение проекта — перемещение по проекту	97
Постоянное совершенствование проекта и команды	101
Agile-проект: объединение всех принципов	105
Глава 4. Scrum и самоорганизующиеся команды	111
Правила Scrum	113
Акт I. Я могу использовать Scrum?	116
Каждый член scrum-команды — владелец проекта	119
Акт II. Обновления статуса — это только для социальных сетей!	135
Вся команда принимает участие в scrum-митингах	137
Акт III. Спринтерский забег в тупик	147
Спринты, планирование и ретроспективы	148
Акт IV. Собака ловит автомобиль	159

Глава 5. Планирование и коллективные обязательства в Scrum	167
Акт V. Не вполне ожидаемая неожиданность	168
Пользовательские истории, скорость работы команды и общепринятые практики Scrum	171
Планирование и выполнение спринта с использованием историй, очков, задач и доски задач	184
Акт VI. Круг почета	190
Переосмысление scrum-ценностей	191
Глава 6. XP и охватывающие изменения	210
Акт I. Сверхурочная работа	212
Основные практики XP	213
Акт II. План игры изменился, но мы все равно проигрываем	224
XP-ценности помогают команде изменить мышление	227
Эффективное мышление начинается с ценностей XP	233
Акт III. Динамика изменений	239
Понимание принципов XP поможет вам принять изменения	240
Глава 7. XP, простота и инкрементальная архитектура	259
Акт IV. Работа в сверхурочное время, часть 2: снова сверхурочные	260
Код и архитектура	262
Принимать решения по коду и проекту в последний ответственный момент	277
Инкрементальная архитектура и целостные XP-практики	287
Акт V. Окончательный результат	302
Глава 8. Lean, ликвидация потерь и целостная картина	314
Бережливое мышление	315
Акт I. И вот еще что...	325
Создание героев и магическое мышление	327
Ликвидируйте потери	330
Постарайтесь глубже понять продукт	336
Поставляйте как можно быстрее	344
Глава 9. Канбан, поток и постоянное совершенствование	364
Акт II. Игра в догонялки	366
Принципы Канбана	368
Совершенствование процесса с Канбаном	375
Измерение и управление потоком	390
Эмерджентное поведение с Канбаном	411
Глава 10. Agile-коуч	425
Акт III. И вот еще что (опять?!)	427
Коучи понимают, почему люди не всегда хотят меняться	428
Коучи понимают, как люди учатся	433
Коучи понимают, что делает методологию работающей	438
Принципы коучинга	441
Об авторах	445
Несколько слов об обложке	446

*Нише и Лизе,
которые были очень терпеливы*

Предисловие партнера

Книга удачно сочетает две важные темы — идеологические принципы и практические методы Agile. Каждая из этих тем подробно изложена и дополнена интересными примерами из практики. Книга вдохновляет и дает рабочие инструменты. В результате вы научитесь правильно распоряжаться человеческими ресурсами, что приведет к экономии времени и денег при работе над любым проектом.

Agile не просто обобщает гибкие методы разработки программного обеспечения и заявляет о новом подходе к управлению ИТ-проектами. Употребляя этот термин, авторы книги говорят скорее о новой концепции мировоззрения.

В эпоху технического прогресса принципы Agile смотрятся свежо. Они очеловечивают бизнес, заставляя многие неповоротливые компании меняться изнутри. Подобные радикальные перемены — это не дань моде. При всей своей простоте методики Agile действительно эффективны.

Даже если специфика вашей работы напрямую не связана с высокими технологиями, книга даст возможность взглянуть по-новому на управленческие вопросы и поможет перезарядить батарейки всем сотрудникам.

Благодаря итерационному подходу к результатам гибкий метод работы над проектом помогает избежать многих рисков, связанных с актуальностью, социальной пользой, дефицитом финансирования.

Отдельно отмечу, что методы Agile работают фрагментарно. Если не получается следовать всем советам сразу, то польза от частичного внедрения этих методик все равно будет ощутимой.

Для меня как руководителя Университета Иннополис важно, что Agile предполагает активное вовлечение в работу всех членов команды, в том числе на этапах планирования и обсуждения промежуточных

результатов. Благодаря такому подходу проекты получают всесторонний анализ. Кроме того, развиваются горизонтальные связи в коллективе и он становится крепче и целеустремленнее. Как раз ради этого книгу стоит прочесть каждому проектному менеджеру, не говоря уже о руководителях.

Вывод из книги простой — будь гибким, чтобы не сломаться. Но пусть вас не смущает простота этого тезиса. Книга не просто отвечает на вопрос, зачем быть гибким, но и наглядно показывает как.

Кирилл Семенихин, директор Университета Иннополис

Предисловие

Похоже, людям постоянно нужно о чем-то спорить. С кем была успешнее группа Van Halen — с Дэвидом Ли Ротом или с Сэмми Хагаром? Что лучше — пепси-кола или кока-кола? Кто, Леннон или Маккартни? Кошки или собаки? В начале развития agile-подхода спорили на тему «принципы или практики». Первые сторонники Agile выделили набор принципов, отраженный в Agile-манифесте, а практики разбрелись по многочисленным гибким подходам. Однако ожесточенные споры, должна ли команда сначала понять принципы гибкой разработки ПО или же приступить к выполнению практической части до их полного усвоения, продолжались.

Сторонники немедленного перехода к практике считали, что постепенно придет и понимание. Если команда будет действовать гибко, то она станет гибкой. Усвоив такие практики, как парное программирование, автоматизация тестирования, непрерывная интеграция, использование итераций, тесное сотрудничество с заказчиком и прочее, команда непременно обретет понимание принципов гибких методологий.

Сторонники первичности принципов утверждали, что практика без принципов — ничто. Побуждение к действию без понимания того, зачем это нужно, не приведет к гибкости. Ведь ее смысл в ориентации на постоянное совершенствование. Аргументы этой стороны состояли в том, что команда не сможет постоянно развиваться, если не понимает сути того, что делает.

Авторам книги «Постигая Agile» удалось невероятное: они подчеркивают важность принципов гибкости, не принижая при этом практики. Стеллман и Грин отмечают, что бездумное следование практикам приведет, скорее всего, лишь к позиции «это все же лучше, чем ничего». Иными словами, внедрение практики без понимания методологии хотя и полезно, но значительно уступает по результатам тому, что обещает подлинное усвоение принципов гибкой методологии.

Я впервые встретился с Эндрю и Дженнифер шесть лет назад, когда они брали у меня интервью для своей книги «Идеальные команды.

Вдохновляющие и предостерегающие рассказы ветеранов тимлидинга»*. Хотя в названии ничего не говорится о гибкости, во многом отношении эта книга именно о ней. Команда, которая приняла принципы гибкой разработки, овладела необходимыми методиками и отказалась от тех практик, которые посчитала ненужными, — действительно идеальная команда. В книге «Постигая Agile» Эндрю и Дженнифер сосредотачивают свое внимание на гибких методах разработки ПО и трех наиболее распространенных сейчас вариантах этого подхода — Scrum, Extreme Programming и Канбан. Вы увидите, как их общие принципы легли в основу различных практик в рамках каждого подхода. Например, здесь вы найдете ответ на вопрос, почему в Scrum требуется ретроспективный анализ после рывка, а в экстремальном программировании — нет.

Присоединяясь к Эндрю и Дженнифер посредством изучения Scrum, Extreme Programming и Канбан, вы прочтете множество рассказов. И это логично: в конце концов, во многих agile-командах принято слушать истории пользователей системы, чтобы знать, что именно они желают от нее получить. Вы увидите команды, которые пытаются создать правильную функциональность, тратя слишком много времени на выполнение прошлых годовых требований, ошибочно принимая за гибкий подход лишь иную форму командно-административного управления, и, будучи подвергнутыми переменам, не могут принять их и многое другое. Но более важно то, что вы узнаете, как эти команды преодолели проблемы. Сможете это и вы.

Книга «Постигая Agile» раз и навсегда решает вопрос о том, что первично — методы или принципы. Увлекательные рассказы и обсуждения, изложенные в этой книге, иллюстрируют простую истину: в гибкой методологии не может быть разделения между принципами и методами. На этих страницах вы обретете более глубокое понимание того, как начать свой путь к идеальной команде — или вернуться на него.

*Майк Кон, автор книги «Scrum. Гибкая разработка ПО»***

* Стеллман Э., Грин Дж. Идеальные команды. Вдохновляющие и предостерегающие рассказы ветеранов тимлидинга. М. : Символ-Плюс, 2010.

** Кон М. Scrum. Гибкая разработка ПО. М. : Вильямс, 2016.

ГЛАВА 1

Обучая Agile

Самая важная установка, которая может быть сформирована, — это желание учиться.

Джон Дьюи. Опыт и образование*

Сейчас самое время быть гибкими! Впервые наша отрасль нашла реальный и устойчивый способ решения проблем, над которыми бились поколения разработчиков программного обеспечения. Вот лишь некоторые примеры, которые возможны с Agile:

- Agile-проекты завершаются вовремя, что отлично подходит для тех команд, которые стремятся закончить работу в срок и не превысить смету.
- Agile-проекты обеспечивают высокое качество программного продукта, а это важно для команд, уставших создавать неэффективное, полное ошибок программное обеспечение.
- Код, написанный agile-командами, хорошо сделан и прост в обслуживании. Это большое облегчение для команд, привыкших поддерживать извилистый и запутанный спагетти-код.
- Agile-команды делают потребителей счастливыми, в этом их огромное отличие от разработчиков сложных программ, суть которых пользователи не понимают.
- Но главное, разработчики в эффективной agile-команде трудятся только в рабочее время, поэтому могут проводить вечера и выходные с семьей и друзьями — возможно, впервые за долгие годы.

* Джон Дьюи (1859–1952) — американский философ и педагог, представитель философского направления «прагматизм». Автор более 30 книг и 900 научных статей по философии, социологии, педагогике и другим дисциплинам. *Прим. ред.*

Agile-методологии популярны, потому что многие перешедшие на них команды сообщают об отличных результатах: они создают качественное программное обеспечение, успешнее работают вместе, удовлетворяют запросы своих пользователей и добиваются всего этого в спокойной рабочей обстановке. Некоторые agile-команды даже продвинулись в решении проблем, которые десятилетиями беспокоили программистов. Итак, каким образом команды используют Agile для создания хороших программ? Или, точнее говоря, как *вы* можете при помощи Agile добиться подобных результатов?

Из этой книги вы узнаете о двух самых популярных agile-методологиях — Scrum и XP (экстремальном программировании). Вы прочтете также о Lean (бережливом программировании) и Канбан (Kanban), о том, как они помогают понять принципы создания программ и развить свои навыки. Вы увидите, что четыре школы Agile сосредотачивают внимание на разных аспектах разработки, но у них есть нечто общее: они направлены на **изменение образа мыслей команды**.

Именно изменение образа мыслей превращает группу сотрудников, добавляющую в свою работу несколько agile-методов, в настоящую команду, которая действительно улучшает способ создания ПО. Цель книги — познакомить читателя с обеими сторонами Agile: методами, работающими в повседневной деятельности, а также ценностями и принципами, которые помогают вашей команде полностью изменить свой подход к разработке программ.

Что такое Agile?

Agile — это набор **методов и методологий**, которые помогают вашей команде эффективнее мыслить, работать и принимать решения.

Эти методы и методологии охватывают все области традиционного программирования, включая управление проектами, дизайн и архитектуру ПО, а также оптимизацию процессов. Все методы и методологии состоят из **процедур**, максимально четких и оптимизированных, которые легко применить.

Кроме того, Agile — это **мировоззрение**, поскольку правильное мышление может оказать большое влияние на эффективность овладения процедурами. Это мировоззрение помогает членам команды делиться друг с другом информацией и на основании этих данных самим принимать важные решения по проекту, не полагаясь только на менеджера. Agile-мировоззрение включает открытое планирование, обсуждение дизайна

и совершенствование процессов всей командой. Agile-команда использует методы, при которых все ее участники владеют одинаковой информацией и каждый имеет свой голос при обсуждении применения этих методик.

Реальность agile-методологий для многих команд, не добившихся особого успеха, не оправдывает ожиданий. Причина часто связана с мировоззрением команды, с которым она начинает работу над проектом. Большинство компаний, занимающихся созданием ПО, уже опробовали Agile. Многие достигли успеха, но результаты некоторых нельзя назвать блестящими. Они добились достаточного прогресса в работе над проектами, чтобы оправдать усилия, потраченные на переход к agile-методологиям, но не ощутили ожидаемых изменений. Это и говорит о важности смены мировоззрения всей командой при переходе на Agile.

Но что означает смена мировоззрения? Если вы входите в команду программистов, то каждый день обдумываете, разрабатываете, пишете и сдаете программное обеспечение. Что ваше мировоззрение должно делать со всем этим? Оказывается, методы, которые вы применяете в повседневной работе, во многом зависят от отношения к ним.

Вот пример. Одна из самых распространенных agile-процедур, которые берут на вооружение команды, — это **ежедневные планерки на ходу**, во время которых члены команды рассказывают, над чем работают и с какими проблемами сталкиваются. Такие собрания длятся недолго, потому все участники стоят. Многие команды, внедрившие ежедневные планерки на ходу, добились больших успехов в работе над проектами.

Итак, представьте себе, что менеджер проекта только что узнал об agile-методологии и хочет внедрить в проект ежедневные митинги. Но выясняется, что не все в команде одобряют эту идею. Один из разработчиков недоволен появлением еще одного совещания, его возмущает, что ежедневно придется ходить на встречу, где у него будут выспрашивать о текущей работе.

Что же происходит? Может быть, разработчик ведет себя нерационально? Или менеджер проекта слишком требователен? Почему такая простая и общепринятая процедура порождает конфликт?

И у менеджера проекта, и у разработчика своя — и вполне разумная — позиция. Одна из основных проблем менеджера проекта — затрата массы усилий на планирование проекта. Но когда при создании ПО команда сталкивается с проблемами, она сразу начинает отклоняться от плана. Приходится потрудиться, чтобы оставаться в роли руководителя команды, поэтому ему нужно вносить коррективы в план и помогать сотрудникам справляться с трудностями.

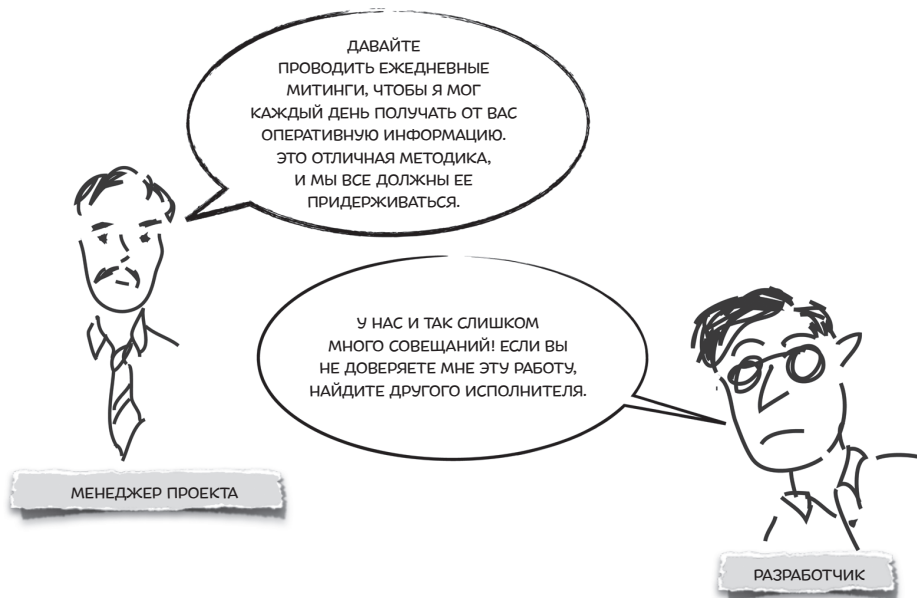


Рис. 1.1. Менеджер проекта, желающий, чтобы команда проводила ежедневные митинги, удивлен, что это нравится не всем

А разработчик недоволен тем, что его по несколько раз в день прерывают, заставляя приходить на совещания, из-за чего ему сложно выполнить работу в срок. Он и так знает, что нужно для написания кода, поэтому не нуждается в чьих-то рассуждениях о планах и изменениях. Его цель — остаться наедине с кодом, и совещание — это последнее, что ему необходимо.

А теперь представьте себе, что менеджер проекта сумел убедить всех — даже несговорчивого разработчика — посещать ежедневные митинги. Как будут выглядеть эти встречи? Менеджер сосредоточится на том, как члены команды отклоняются от его плана, поэтому будет стараться получить информацию о ходе работы каждого сотрудника. А разработчик захочет скорейшего окончания совещания, поэтому, не особенно вслушиваясь в чужие выступления, постарается, дождавшись своей очереди, высказаться покороче, чтобы не затягивать время.

Скажем честно: именно так и проходят многие митинги. И хотя это не оптимальный способ, даже такое ежедневное совещание *принесет результаты*. Менеджер проекта выявит проблемы с выполнением плана, а разработчик получит выгоду в долгосрочной перспективе. Ведь проблемы, которые его *действительно* касаются, лучше решать на ранних этапах. То есть вся

процедура сэкономит команде больше времени и усилий, чем потребуется на ее выполнение. Так что этим стоит заниматься.

Но как быть, если у разработчика и менеджера проекта совсем иное мировоззрение? А каждый член команды будет относиться к ежедневному митингу совершенно по-другому?



Рис. 1.2. Кажется, у обоих есть веские основания для собственного мнения о ежедневных митингах. Как это скажется на проекте?

Например, если менеджер проекта почувствует, что проект планируется всеми членами команды? Тогда он будет выслушивать каждого не для того, чтобы узнать, насколько тот отклонился от плана, а чтобы понять, какие изменения внести в план, над которым работала вся команда. Вместо того чтобы навязывать план и затем оценивать, насколько точно следует ему команда, менеджер будет работать вместе с ее членами, выбирая наилучший подход к проекту. То есть ежедневный митинг — это способ совместной работы, при котором люди убеждаются, что поступают наилучшим образом в каждый момент времени. Поскольку данные о проекте меняются ежедневно, команда использует митинги для принятия максимально эффективных решений. Так как команда встречается каждый день,

изменения, о которых сообщается на совещании, могут быть внедрены немедленно. Ведь теперь не нужно тратить массу времени и усилий, двигаясь в неверном направлении.

А что если разработчик почувствовал, что цель совещания — не только доложить обстановку, но и понять, как продвигается проект, ежедневно вместе искать возможности для оптимизации процесса? Тогда ежедневный митинг *становится важным для него*. Хороший разработчик, как правило, имеет собственное мнение не только о своем коде, но и об общем направлении проекта. Ежедневный митинг — это способ убедиться, что проект реализуется разумно и эффективно. Разработчик понимает, что в долгосрочной перспективе эта процедура принесет пользу его работе, поскольку все, что от него не зависит, тоже выполняется хорошо. Кроме того, он знает: если на совещании придется упомянуть о проблемах планирования, то все прислушаются к его мнению и работа над проектом пойдет еще лучше.



Рис. 1.3. Когда каждый член команды чувствует, что обладает равными правами при планировании проектом и управлении им, ежедневные митинги обретают ценность и высокую эффективность

Иными словами, если члены команды полагают, что ежедневный митинг — это очередная планерка, которую придется вытерпеть, то его все равно стоит проводить. Такое совещание лишь немногим эффективнее традиционных планерок. Но если команда верит: ежедневный митинг — это способ

убедиться, что все работают правильно и для достижения единой цели, что при обсуждении выслушают мнение каждого участника, — то такое собрание становится гораздо эффективнее и приносит настоящее удовлетворение. Разработчик понимает: такое совещание в долгосрочной перспективе помогает ему и всей команде. Менеджер проекта убежден: если в реализации плана принимают участие все сотрудники, то результаты будут выше. Тем, кто разделяет эти взгляды, ежедневные митинги помогают быстрее работать, активнее общаться и качественнее выполнять поставленные задачи.

Это только один из примеров того, как мировоззрение и отношение команды могут повлиять на успешное усвоение agile-методик. Важная цель этой книги — помочь понять, каким образом мировоззрение команды отражается на проектах и вашем отношении к Agile. Изучая Scrum, экстремальное и бережливое программирование, а также Канбан, вы узнаете обе стороны Agile — принципы и методы — и то, как они помогут лучше создавать программное обеспечение.

Кому следует прочитать эту книгу

Можно ли сказать, что одна из приведенных ниже ситуаций возникала у вас и вашей команды?

Вы попробовали agile-методики, но это не помогло. Возможно, вы внедрили ежедневные митинги, команда совещается каждый день, но у вас все равно масса проблем и вы пропускаете дедлайны. Или вы начали писать пользовательские истории и анализировать их с командой и заинтересованными лицами, но разработчики все еще сталкиваются с изменениями в последнюю минуту, добавляя дополнительные функции. А может быть, ваша команда решила полностью перейти на Agile, избрав Scrum или экстремальное программирование, но это выглядит пустой тратой времени, как будто все делают то, что от них требуется, но польза для проектов невелика.

Возможно также, что вы еще не пробовали перейти на Agile, но понимаете: команда столкнулась с серьезными проблемами, а что делать — непонятно. Вы надеетесь, что Agile поможет работать с требовательными пользователями, которые постоянно меняют свое решение. Любое изменение, которого требует заказчик, означает дополнительную работу для вашей команды и ведет к спагетти-коду, из которого так и торчат скотч и скрепки. Поэтому программы становятся все более уязвимыми и сложными для техподдержки. Возможно, ваши проекты — это просто контролируемый хаос; программы пишутся за счет многочасовой работы и личного героизма, так что вы рассматриваете Agile как единственный выход.

Или вы как руководитель обеспокоены тем, что команды, работающие над важными проектами, могут подвести? Возможно, вы слышали об Agile, но плохо представляете, о чем идет речь. Можно ли заставить команду перейти на Agile, или нужно сначала изменить и свое, и командное мировоззрение?

Если какая-то из описанных ситуаций вам знакома и вы хотите улучшить работу команды, то эта книга для вас.

Мы рассказываем, почему agile-методологии были разработаны именно таким образом, с какими проблемами призваны бороться, какие ценности, принципы и идеи они воплощают. Мы объясняем не только «как», но и «почему», то есть помогаем понять принципы, которые применимы к конкретным проблемам развития, характерным для вашей команды, компании и проектов. И мы покажем, как пользоваться этой информацией при выборе методологии и практик.

Еще одна группа людей, для которых написана эта книга, — **agile-коучи**. Команды и компании все чаще полагаются на них, поэтому они должны помочь усвоить agile-методологии и процедуры и изменить мировоззрение членов команды. Если вы agile-коуч, то мы предоставим вам инструменты, чтобы помочь лучше донести идеи до обучаемых и преодолеть некоторые проблемы, с которыми ежедневно сталкиваются те, кто решил перейти на Agile.

Цели, которые мы ставим в этой книге

Мы хотим, чтобы вы:

- усвоили идеи, которыми руководствуются эффективные agile-команды, а также объединяющие их ценности и принципы;
- познакомились с самыми популярными школами — Scrum, экстремальным и бережливым программированием и техникой Канбан — и поняли, как они могут считаться agile-методологиями, несмотря на все их различия;
- научились конкретным agile-методам, которые сможете сразу внедрить в свои проекты. Мы стремимся дать представление о базовых ценностях и принципах, которые понадобятся, чтобы это внедрение было эффективным;
- лучше понимали свою команду и компанию, смогли выбрать тот agile-подход, который соответствует вашему мировоззрению (или максимально близок к нему), а также чтобы начали усваивать новое мышление, которое поможет стать эффективным agile-коллективом.

Каким образом различные agile-методологии и процедуры обеспечивают создание более совершенного программного обеспечения? Почему они дают команде возможность лучше управляться с изменениями? Что значит гибкость? Действительно ли важно использовать карточки для планирования или, например, стоять во время совещаний? Все эти вопросы нередко смущают людей, начинающих свой путь по освоению agile-методологии. Но к концу книги вы сможете ответить на них.

Почти все блоги и статьи, в которых обсуждается гибкая разработка ПО, начинаются с утверждения «Agile — хорошо, водопад — плохо». Но почему Agile — это хорошо, а водопад — нет? Почему они противостоят друг другу? Можно ли работать в команде, которая практикует водопадную модель* (Waterfall), и оставаться гибким? Прочитав книгу до конца, вы найдете ответы и на эти вопросы.

Продвижение Agile в ваше сознание любыми необходимыми средствами

Книга называется «Постигая Agile», потому что мы действительно хотим, чтобы вы постигли Agile. В течение более чем 20 лет мы активно сотрудничали с командами, постоянно создающими ПО для реальных пользователей. Последние десять с лишним лет мы пишем книги о создании ПО (две из них, очень успешные, вышли в издательстве O'Reilly в серии Head First, посвященной управлению проектами и обучению программированию). Нам удалось научиться доносить до сознания читателя сложные технические идеи, не нагоняя на него тоску.

Мы сделали все возможное, чтобы наш материал был максимально интересным и увлекательным... но нам нужна ваша помощь. Вот способы и методы, которые помогут удержать в голове все эти идеи.

ПОВЕСТВОВАНИЯ

Обозначаются значком . Вспомните последнюю техническую книгу, которую вы читали. Можете ли вы воспроизвести все основные темы, которые в ней излагались, и их порядок? Вероятно, нет. А теперь подумайте

* Водопадная модель (англ. waterfall model, иногда называют «каскадная модель») — модель процесса разработки программного обеспечения, при котором команда вначале определяет требования к продукту, планирует проект в целом, разрабатывает программное решение, а затем создает код и тестирует продукт.

о фильме, который смотрели недавно. Вы помните основные элементы сюжета и порядок, в котором они происходили? Наверняка да. Дело в том, что мозг лучше запоминает то, что вызывает у нас эмоциональную реакцию. В книге мы стараемся учитывать это обстоятельство. Мы будем использовать повествования — с участием людей, диалогами и конфликтом, — чтобы показать, как в действительности выглядит переход на Agile. Обычно у этих людей возникают проблемы.

Чего мы хотим от вас. Постарайтесь представить себя в похожей ситуации: это обеспечит эмоциональную связь с данными идеями, поэтому будет проще их запомнить и понять. Отнеситесь к этим рассказам без предубеждения, особенно если вы не большой любитель художественной литературы. В каждом повествовании заключен свой урок, и все они раскрывают основную тему книги.

ИЛЛЮСТРАЦИИ

Люди обучаются по-разному. Некоторым важны визуальные образы, поэтому они легче воспринимают идеи, увидев картинки. Мы хотим снабдить вас максимальным количеством инструментов обучения, поэтому включили в книгу множество иллюстраций. В некоторых случаях мы полностью положились на визуальные метафоры, например используя геометрические формы для передачи различных функций или шестеренки, символизирующие комплексные программы.

Чего мы хотим от вас. Если визуальные образы не имеют для вас большого значения, то некоторые иллюстрации могут показаться избыточными, даже бессмысленными. Это хорошая возможность для обучения: нужно попытаться понять, что человек с визуальным подходом может извлечь из этой иллюстрации. Это поможет лучше понять общую идею.

ИЗБЫТОЧНОСТЬ

В большинстве технических книг существует определенный порядок: идея подается, полностью описывается, а затем автор переходит к следующей. Это эффективный способ передать как можно больше информации, но наш мозг работает иначе. Иногда необходимо не раз и не два взглянуть на идею, прежде чем вы воскликнете «Понятно!». Вот почему мы порой будем возвращаться к одной и той же теме по несколько раз в течение всей книги. **Это намеренная избыточность** — таким способом мы хотим помочь вам поскорее сказать «Понятно!».

Чего мы хотим от вас. Когда вы читаете об одном и том же по несколько раз, возникает искушение спросить: «Разве об этом уже не говорили?» Говорили! И очень хорошо, что вы это заметили. Но есть читатели, которые не обратили на это внимания, да и вы сами наверняка не каждый раз замечаете избыточность. Все это делается для того, чтобы помочь вам учиться.

УПРОЩЕНИЕ (В ПЕРВУЮ ОЧЕРЕДЬ!)

Иногда сложную тему проще понять, если сначала едва коснуться ее и лишь затем погрузиться полностью. В книге мы так и поступаем: сначала вводим упрощенную (но технически верную!) версию идеи, а затем конкретизируем ее. Этот метод работает на двух уровнях. Если вы глубоко понимаете данную идею, то сразу заметите упрощение и эмоционально на него отреагируете, что сохранит вашу заинтересованность. Но если вы незнакомы с идеей, упрощение послужит легким толчком, который подготовит вас к более глубокому описанию.

Чего мы хотим от вас. Если вы отметили чрезмерное упрощение, то не надо дистанцироваться от этого или думать, что мы упустили из виду основную идею, приукрасили действительность или забыли нечто важное. Скорее всего, то, что вас насторожило, будет разъяснено в книге позже. Можете считать упрощенное введение сложной идеи чем-то вроде неформального приветствия: это ободряет незнакомых с идеей читателей, они чувствуют, что находятся на верном пути, а это закладывает основы для более глубокого понимания.

РАЗГОВОРНЫЙ (ПРИВЫЧНЫЙ) СТИЛЬ

На протяжении всей книги мы пользуемся разговорным языком, чтобы сделать излагаемый материал максимально привлекательным. Мы не забываем о юморе и время от времени — ссылаемся на литературные источники, а порой обращаемся непосредственно к вам или к себе, используя личные местоимения. На самом деле это научно обоснованно: когнитивные исследования* показали, что мозг запоминает больше информации, если во время разговора вы испытываете чувства.

Чего мы хотим от вас. Обычно людям нравится разговорный стиль, но не всем. Некоторые терпеть не могут сокращений. Для других разговорный стиль — признак недостаточной авторитетности книги. Мы всё понимаем, но поверьте, вы к этому привыкнете даже раньше, чем думаете.

* Узнать больше о том, как разговорный стиль помогает в обучении, можно в книге Clark, Ruth, Mayer, Richard. *E-Learning and the Science of Instruction*. Wiley, 2011.

КЛЮЧЕВЫЕ МОМЕНТЫ

Обозначаются значком . В каждой главе мы будем резюмировать основные положения, которые были введены. Это поможет убедиться, что вы все усвоили, не пропустили ничего важного. Кроме того, мозг получит краткий отдых после обучения.

Чего мы хотим от вас. Не закрывайте разделы «Ключевые моменты» сразу. Уделите им хотя бы минутку. Вы помните все, что в них описано? Если нет, то не поленитесь вернуться на несколько страниц назад, чтобы освежить память.

ЧАСТО ЗАДАВАЕМЫЕ ВОПРОСЫ

Обозначаются значком . Основную часть времени мы работаем в командах программистов, создавая реальное ПО для реальных пользователей, но при этом немало лет посвятили чтению лекций и проведению презентаций на тему гибкой разработки. В ходе этих мероприятий мы беседовали со многими людьми и можем утверждать, что некоторые вопросы интересуют людей особенно часто.

Чего мы хотим от вас. Прочтите в конце каждой главы раздел «Часто задаваемые вопросы». Есть ли среди них тот, который волнует именно вас? Если да, то удовлетворяет ли вас ответ? Не все ответы подходят к вашему случаю, но постарайтесь и в них найти рациональное зерно. Если все вопросы в разделе неактуальны для вас, то подумайте, с какой целью их могли задать. Так можно по-новому взглянуть на материал (в главе 2 вы узнаете, почему это важно для команды).

ЧТО ВЫ МОЖЕТЕ СДЕЛАТЬ УЖЕ СЕГОДНЯ

Обозначаются значком . Самый эффективный способ чему-то научиться — сделать это! В конце каждой главы мы приводим небольшой раздел, в котором есть список того, что можно сделать прямо сейчас — как самостоятельно, так и вместе с командой.

Чего мы хотим от вас. Разумеется, самое лучшее — это просто начать делать! Но не всем это по душе, а один из главных постулатов книги состоит в том, что попытки внедрять практику в условиях, когда мировоззрение компании не совпадает с ней, могут окончиться плохо. Так что прежде всего подумайте, как отреагирует команда. Это может быть не менее эффективным инструментом обучения, чем само выполнение задания.

ГДЕ ВЫ МОЖЕТЕ УЗНАТЬ БОЛЬШЕ

Обозначаются значком . Исаак Ньютон однажды сказал: «Если я видел дальше других, то потому, что стоял на плечах гигантов». Нам повезло: сейчас существует множество революционных книг по гибкой разработке ПО. После каждой главы мы предлагаем список из нескольких источников, в которых можно найти больше сведений по конкретной теме.

Чего мы хотим от вас. Продолжайте учиться! Наша книга подробно рассказывает о Scrum, XP, Lean и Канбан, но, конечно, мы не можем исследовать все эти идеи детально. Большинство идей, описанных здесь, придумано не нами. К счастью, вы можете учиться и у тех, кому они принадлежат.

ПОДСКАЗКИ

Обозначаются значком . Agile-коуч — это человек, который помогает командам овладеть Agile. Книга написана для тех, кто только обучается Agile, но ее могут использовать и опытные agile-коучи, внедряющие эти идеи в свою команду. Ищите советы для тренеров в конце каждой главы. Они помогут применить идеи и подход, которые используем мы, и адаптировать их к вашей команде.

Чего мы хотим от вас. Даже если вы не коуч, все равно стоит прочитать советы для тренеров. Дело в том, что один из самых эффективных методов обучения — попробовать себя в роли наставника. Если вы узнаете об этих идеях впервые, то подумайте, как можно использовать советы для тренеров, чтобы помочь команде больше узнать об Agile.

Структура книги

Эта книга устроена так, чтобы помочь вам понять agile-методологии, усвоив ценности и принципы эффективной команды разработки, школ, которые воплощают эти ценности, и методик, при помощи которых они реализуются.

Две следующие главы помогут понять ценности и принципы, которые будут способствовать переходу на гибкое мировоззрение. В них приведены способы, благодаря которым можно оценить, готовы ли вы и ваша команда принять agile-методологии, какие именно ее части найдут отклик у коллег и что может оказаться наиболее сложным для внедрения.

— Глава 2 описывает ключевые ценности Agile. Мы расскажем о команде, бьющейся над программным проектом, и объясним, что основной

источник затруднений — это «искаженная перспектива». Мы изложим ценности Agile и при помощи метафоры поможем увидеть, как они дают возможность команде увидеть общую перспективу.

- Глава 3 рассказывает о принципах, в соответствии с которыми agile-команды принимают решения о том, как управлять проектами. Мы поясним, какие цели и идеи лежат в основе этих принципов, проиллюстрировав их практическим примером из программного проекта.

В следующих шести главах говорится о самых популярных школах Agile: Scrum, XP, Lean и Канбан. Вы узнаете, как их применять и внедрить в практику работы вашей команды.

- Глава 4 описывает Scrum, популярный agile-подход, чтобы рассказать о том, как работают самоорганизующиеся команды. Мы дадим несколько советов, как применить технологию Scrum к вашим проектам и обучить команду самоорганизации.
- Глава 5 демонстрирует конкретные процедуры, которые используются в scrum-командах для управления проектами, и объясняет, как эти процедуры помогают команде объединиться и создать качественные программы. Мы покажем, что успех реального перехода на Scrum зависит от того, насколько полно ценности Scrum соответствуют культуре вашей команды и компании, и поясним, что делать, если различия слишком сильны.
- Глава 6 рассказывает об основных методах экстремального программирования, его ценностях и принципах. Вы узнаете, как каждому члену команды прививается мировоззрение, необходимое для улучшения работы над кодом: вместо того чтобы ненавидеть перемены, все сотрудники учатся принимать их с готовностью.
- Глава 7 рассказывает о трех основных методах экстремального программирования и о том, как они помогают избежать серьезных проблем с кодом и проектированием. Вы поймете, что все методы экстремального программирования образуют единую экосистему, которая ведет к созданию лучшего кода — более гибкого, изменяемого и простого в обслуживании.
- Глава 8 знакомит с бережливым программированием и принципами, которые помогут обрести соответствующее мировоззрение. И мы покажем, что методы размышлений, предлагаемые бережливым программированием, могут помочь вашей команде найти излишне принимаемые действия и избавиться от них, а также увидеть общую картину системы, в рамках которой вы разрабатываете ПО.

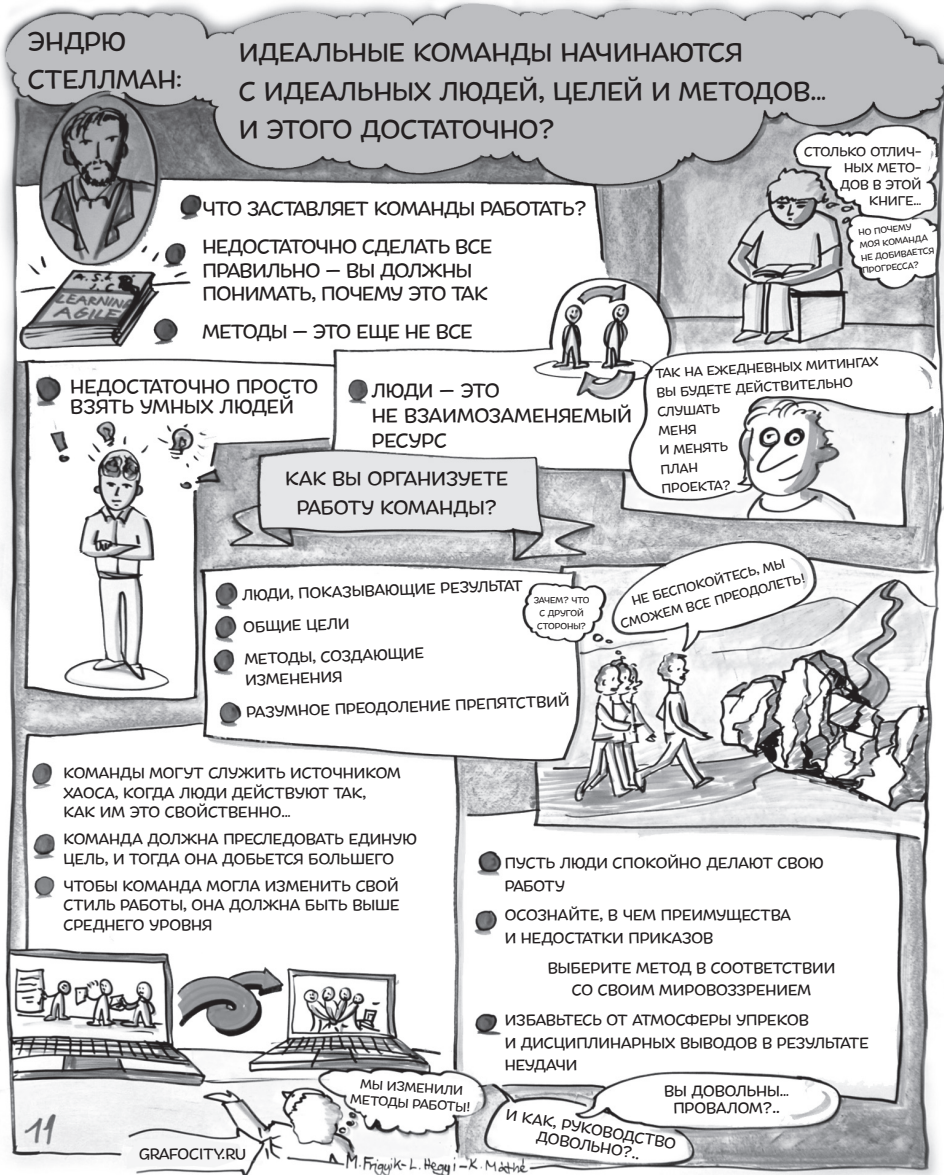


Рис. 1.4. Графическая запись выступления Эндрю Стеллмана на конференции Stretch 2013 в Будапеште (Венгрия).
Выступление было основано на материалах этой главы
Графическая запись: Ката Мате и Марти Фридик,
www.remarker.eu

- Глава 9 рассказывает о Канбане, его принципах и взаимоотношениях с бережливым программированием, а также о методах. Вы узнаете, как концентрация на потоке и теории массового обслуживания поможет вашей команде претворить в жизнь идеалы бережливого программирования. Также вы поймете, как Канбан может создать в команде культуру постоянного совершенствования.

В мире Agile существуют не только мировоззрения, методологии и школы мышления. Компании все чаще полагаются на agile-коучей, способных помочь командам взять Agile на вооружение. Вот почему мы включили в книгу последнюю главу.

- Глава 10 рассказывает о работе agile-коучей: как учатся команды, как коуч помогает изменить мировоззрение, чтобы легче было взять на вооружение agile-методологии и стать более гибкими.

ГЛАВА 2

Понимание ценностей Agile

Мы действуем правильно не потому, что обладаем добродетелью или совершенством; скорее мы приобретаем их потому, что действуем правильно. Мы то, что мы постоянно делаем. Совершенство, таким образом, — это не поступок, а привычка.

Аристотель. Никомахова этика*

Agile как профессиональное движение отличается от существовавших ранее подходов к разработке программного обеспечения тем, что в его основу заложены идеи, ценности и принципы, воплощающие в себе определенный образ мышления. Глядя на мир разработки программного обеспечения сквозь призму этих идей, вы сможете стать более гибким как практик и более ценным как член проектной команды.

Движение Agile революционно. Команды, которые приняли эту технологию, систематически отмечают улучшения (иногда скачкообразные) в умении создавать лучшее программное обеспечение. Те, кто успешно внедрил Agile, создают высококачественные продукты и делают это быстрее, чем раньше.

Благодаря Agile наша отрасль оказалась на переломе своего развития. Agile из аутсайдера превратился в работающий институт. В течение первых лет существования этого метода принявшие его люди активно старались убедить свои компании и коллег, что Agile действительно приносит пользу и им стоит заниматься. В настоящее время практически не осталось сомнений, что agile-методологии — это эффективный способ создания программного

* «Никомахова этика» — одно из трех этических сочинений Аристотеля. Считается, что название эта работа получила, поскольку впервые была издана около 300 года до н. э. Никомахом — сыном Аристотеля. Но есть версия, что книга посвящена отцу Аристотеля, которого тоже звали Никомахом. *Прим. ред.*

обеспечения. В 2008 году было проведено исследование*, которое показало, что более половины всех опрошенных команд, занимающихся разработкой программных продуктов, используют agile-методологии, практики или принципы. С тех пор актуальность Agile только выросла. Agile-команды все чаще задумываются не только над тем, как стать более гибкими, но и как распространить Agile в своих компаниях.

Но так было не всегда. Традиционно при выполнении проектов по разработке программных продуктов компании использовали **водопадный подход**, согласно которому команда вначале определяет требования к продукту, планирует проект в целом, разрабатывает программное решение, а затем создает код и тестирует продукт. Значительная часть программного обеспечения — как грандиозного, так и совсем бестолкового — годами создавалась именно таким образом. Однако на протяжении десятилетий различные команды во всевозможных компаниях сталкивались с одними и теми же проблемами. И некоторые из них заподозрили, что главная причина неудач — сам водопадный подход.

История Agile началась, когда небольшая группа новаторов задумалась о новых способах решения этих проблем. Первым делом они составили список из четырех основных ценностей, общих для успешных команд и проектов (этот документ получил название Manifesto for Agile Software Development, или «Манифест гибкой разработки программного обеспечения»).

- Люди и взаимодействие важнее процессов и инструментов.
- Работающий программный продукт важнее исчерпывающей документации.
- Сотрудничество с заказчиком важнее согласования условий контракта.
- Готовность к изменениям важнее следования первоначальному плану.

В данной главе вы узнаете об этих ценностях — откуда они взялись, что означают и насколько применимы к вашему проекту. Вы проследуете за командой, уставшей от методологии водопада и впервые пытающейся реализовать agile-проект, до тех пор, пока она не поймет, как эти ценности действительно применимы к ней. Читая эту историю, обратите внимание на то, как лучшее понимание ценностей помогает избежать проблем.

* Глобальный онлайн-опрос agile-компаний, проведенный в 2008 году независимой исследовательской компанией Forrester.



**Описание: команда, работающая над проектом
потокowego аудиопроигрывателя**

Дэн — ведущий разработчик и архитектор

Брюс — лидер команды

Джоанна — недавно нанятый менеджер проекта

Том — владелец продукта

Руководитель команды, архитектор и менеджер проекта заходят в бар...

Дэн — ведущий разработчик и архитектор в компании, которая создает игровые автоматы и киоски. Он участвовал в различных проектах, от аркадных игр и пинбола до ПО для банкоматов. Последние несколько лет он работал в команде, возглавляемой Брюсом. Команда занималась выпуском крупнейшего продукта компании, слот-машины Slot-o-matic Weekend Warrior («Боец по выходным»).

Джоанну наняли несколько месяцев назад в качестве менеджера проекта, чтобы возглавить проект создания программного обеспечения для новой линии потоковых аудиоплееров, которые компания хочет вывести на рынок и продавать барам и ресторанам. Девушку переманили из конкурирующей компании, уже имеющей успешный опыт вывода музыкального автомата на рынок. Она отлично ладит с Дэном и Брюсом и воодушевлена работой над новым проектом.

Дэна и Брюса новый проект вдохновляет меньше, чем Джоанну. Как-то раз они зашли выпить после работы, и Брюс с Дэном начали объяснять Джоанне, почему команда придумала для слот-машины имя «Боец по выходным».

Она не очень обрадовалась, узнав, что провальные проекты в этой компании — скорее правило, чем исключение. Последние три проекта, по мнению руководителей компании, были успешно доведены до конца только благодаря чрезвычайно напряженной работе Дэна и Брюса. Более того, им пришлось наступить на горло собственной песне, выбрав кратчайший путь в кодировании. Из-за этого их продолжает мучить совесть. Ведь им пришлось срочно подлатать прототип для одной функции и протолкнуть его в производство, а позднее выяснилось, что появились серьезные проблемы с производительностью, потому что отдельные части этого прототипа оказались непригодны к масштабированию.

Слушая их рассказ, Джоанна поняла, что стало причиной проблем: компания использовала самую неэффективную методику — *водопадный подход*. В рамках этой модели требуется как можно раньше создать полное описание программного обеспечения, которое будет разрабатываться. После того как все пользователи, менеджеры и руководители согласуют точные *требования* к программному продукту, они могут подписать документ (*спецификацию*). Он содержит требования к команде разработчиков, которая создает именно то, что написано. После этого приходят тестировщики и проверяют, соответствует ли программное обеспечение документу. Многие agile-практики называют это «большими требованиями вначале» (big requirements up front, или BRUF).

Джоанна по опыту знала, что теория часто отличается от практики и, хотя отдельные команды имеют эффективную модель водопада, многие все же борются с ней. Она решила, что эта команда может стать одной из них.

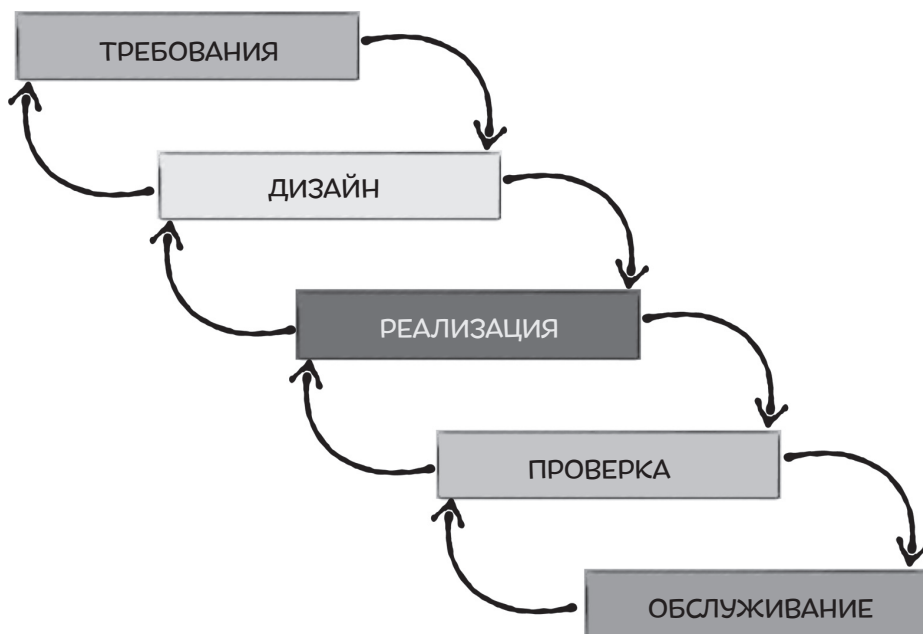


Рис. 2.1. Модель водопадного подхода

За время беседы с Брюсом и Дэном ее уверенность укрепилась. Джоанна не сомневалась, что там было множество спецификаций, переплетенных в скоросшиватели и без дела стоящих на полках по всей компании, собирая пыль. Почему-то все ожидали группу пользователей, менеджеров и руководителей, которые должны создать идеальное техническое задание.

В реальной жизни спецификация менялась настолько радикально, что могла стать ошибочной к тому времени, когда попадала к команде. Кроме того, неточности в ней катастрофически нарастали к концу создания программного продукта. Брюс, Дэн и множество других сотрудников компании знали, что ждать появления совершенной спецификации бессмысленно, но продолжали работать над своими проектами так, как будто это было возможно.

В ходе посиделок Брюс расслабился и рассказал еще об одной проблеме, которая не давала ему покоя: многие команды в компании имели большие проблемы с созданием ПО. Даже если пользователь прислал письменные требования (что случалось редко) и команда, прочитав их, смогла в них разобраться (чего вообще никогда не случалось), то они часто применяли неподходящие инструменты и имели трудности с дизайном и архитектурой программ. Это приводило к тому, что команды Брюса неоднократно создавали программные продукты с множеством ошибок, которые невозможно было исправить.

Обе эти проблемы привели к массе неудавшихся проектов, особенно с учетом тех случаев, когда им приходилось работать по 90 часов в неделю, если код давал сбои. Джоанна объяснила, что главная причина этих неудач — *неспособность водопадного подхода, принятого в компании, справляться с изменениями*. В идеальном мире водопадный подход работал бы прекрасно, потому что на старте проекта известно, что нужно получить в конце. Таким образом, можно было бы записать все в виде аккуратной спецификации и передать команде разработчиков. Но в реальных проектах так не бывает.

Захмелевшие Дэн и Брюс подпали под влияние Джоанны, которая усилила напор. Дэн рассказал, что практически каждый проект, над которым они работали, обрывался на полпути, потому что заказчики неожиданно сообщали, что им нужно что-то другое, отличающееся от первоначального замысла. Затем всем приходилось возвращаться к началу водопада. В соответствии со строгими процессами этой методологии их действия были следующими: создать совершенно новые спецификации, другой дизайн и новый план. Но в реальности этого почти никогда не случалось, крайне редко удавалось урвать время, чтобы переписать весь код заново. Вместо того чтобы следовать методикам водопадного программирования, принималось решение о срочной переделке существующего кода. Это влекло за собой ошибки, потому что ситуация, когда программное обеспечение сначала разрабатывается для одних целей, а затем спешно модифицируется для других, часто приводит к беспорядочному, запутанному коду (особенно когда команда находится под посторонним давлением). Адаптация кода к новым задачам отнимала драгоценное время, так что они завершали проект обходными способами и создавали нестойкий код.

Дэн, Брюс и Джоанна начали понимать, что проблемы их проекта были вызваны *слишком жесткой документацией, плохой коммуникацией и ошибками*, которые привели к тому, что проект не смог идти в ногу с нормальными изменениями.

В конце вечера бармен вызвал для каждого из них такси. Перед тем как уехать, Дэн признался, что он снял камень с души. Джоанна была счастлива, получив более полную информацию о ситуации с проектом, и готова присоединиться... но ее оптимизм поубавился. Она задумалась, сможет ли найти способ, чтобы исправить хотя бы часть проблем.

Серебряной пули не существует

Сегодня мы знаем, что нет идеального способа создания программного обеспечения. Но в прошлом веке многие представители отрасли с этим бы не согласились. Существовало мнение, будто можно найти обладающий высокой эффективностью метод «серебряной пули», позволяющий разом решить все проблемы проекта. Казалось, что разработчики могут создавать программное обеспечение, просто следуя инструкциям или собирая программный продукт как на конвейере.

(По иронии судьбы одна из самых цитируемых работ в области программной инженерии — это эссе Фреда Брукса No Silver Bullet («Серебряной пули не существует», 1986). В нем он убедительно доказывает, что эта задача невыполнима. Что еще нужно, чтобы остановить бесплодные попытки найти загадочную панацею?)

Предлагалось большое количество радикальных способов, претендующих на звание «серебряной пули». Они, как правило, были двух видов: **методология**, дающая командам надежный способ создания программного обеспечения, или технология, которую программисты могли использовать для предотвращения или устранения ошибок. Идея состояла в том, что если компания приняла решение о выборе методологии и технологии, то вся команда должна была следовать им неукоснительно, поскольку только в этом случае можно создавать отличное ПО.

Дэн и Брюс не понаслышке знают, что все это иллюзии, потому что на протяжении многих лет управляли проектами, хватаясь за разные методики и технологии, не получая при этом каких-либо реальных улучшений. Попытки компании найти серебряную пулю для процесса программной разработки, как правило, заканчивались печально для всех участников. Но хуже всех приходилось Брюсу и Дэну, потому что им насильно предлагалось следовать за вечно меняющимися процессами.

Джоанна тоже сталкивалась с этим. На предыдущей работе она регулярно выдвигала множество жестких требований и давала массу распоряжений, чтобы придумать план разработки программного обеспечения. Команды приступали к буквальному выполнению ее плана. Однако они были обречены на создание негодного и бесполезного ПО. Пользователь получал устаревший продукт, не успев им воспользоваться.

Правда, некоторым командам, с которыми работала Джоанна, удавалось выпускать отличное программное обеспечение на основе водопадного процесса (или подобного ему) со сложной в разработке предварительной документацией. Она руководила рядом лучших проектов разработки ПО для медицинских устройств, в которых применялся водопадный подход.

Водопад действительно бывает полезен. Если вы хорошо знаете, что необходимо вам прежде всего, и первым делом создаете именно это, то получаете довольно эффективный способ создания ПО. Проекты создания программного обеспечения для медицинских устройств, которыми руководила Джоанна, — это тот редкий случай, когда требования к программе действительно писались в самом начале работы и в ходе проекта практически не было изменений.

Но для запуска успешного водопадного проекта требуется нечто большее, чем просто стандартные требования, отсутствие которых создает так много проблем. Команды, разрабатывающие прекрасное ПО при помощи водопадного подхода, как правило, имеют несколько общих характеристик. Это:

- хорошая *коммуникация*, потому что успешные команды, уполномоченные компанией вести водопадную разработку, постоянно поддерживают контакт со своими пользователями, менеджерами и руководителями на протяжении всего проекта;
- хорошие *методики*, особенно такие как анализ кода и автоматизированное тестирование, которые направлены на поиск ошибок на раннем этапе. Обычно это называется «предотвращение дефектов» и необходимо команде, чтобы прежде всего понять, как эти ошибки попали в код;
- ящики, полные документации, которые редко открываются, потому что люди в команде понимают, что сам факт написания плана и вопросы, которые возникают во время планирования, — это важнее, чем слепое следование этому плану.

Есть еще одна сторона общей картины. Дэн начал свою карьеру после революционного переворота 1990-х годов в области инструментов разработки ПО и технологий, поэтому он трудился только в командах, которые

использовали объектно-ориентированную разработку для создания лучших образцов программного обеспечения. Системы контроля версий, автоматическое тестирование, лучшие интегрированные среды разработки (integrated development environments, IDEs), которые включают в себя функции для автоматизации программирования, такие как рефакторинг и проектирование классов, и другие революционные средства, помогали Дэну сохранить контроль над кодом. У Брюса опыт взаимодействия с проектами гораздо длительнее, чем у Дэна, он наблюдал разные команды разработчиков, которые часто использовали программные средства для автоматизации рутинных и повторяющихся задач.

Брюс и Дэн знают по опыту собственных проектов, что наиболее успешные люди эффективно используют такие методы, инструменты и идеи. Это позволяет им выделять больше времени на контакты с пользователями и партнерами, а также думать о проблемах, которые приходилось решать вместо того, чтобы трудиться над кодом.

И оказалось, что водопадные проекты эффективны в тех случаях, когда их команды принимают *многие ценности, принципы и практики, которые характерны для agile-проектов*. Проекты, которые выполняются с использованием отдельных гибких методов и практик (а это нельзя считать настоящим следованием ценностям и принципам Agile), в итоге нередко сталкиваются с теми же проблемами, которые преследуют водопадные проекты.

К сожалению, Брюс, Дэн и Джоанна сумеют убедиться в этом лишь на собственном горьком опыте.



Ключевые моменты

Водопадный подход требует от команды полного описания программного обеспечения в начале проекта, а затем точного создания того, что было описано.

Водопадный подход затрудняет возможность реагировать на изменения из-за сосредоточения внимания на документах, а не на сотрудничестве.

Серебряной пули, или практики, которая помогает создавать безупречные проекты, не существует.

Команды, использующие водопадный подход, делают это путем принятия эффективных методов и принципов создания ПО, особенно таких, которые улучшают коммуникацию.

Agile для спасения! (Правильно?)

Вы, наверное, знаете, на что похож водопадный подход, даже если впервые столкнулись с термином «водопад»*. Джоанна, Брюс и Дэн тоже знают. Прежде чем приступить к планированию музыкального автомата, они говорили о том, каким образом водопадные процессы вызывали проблемы у их команд в прошлом.

Над последним проектом вместе с Брюсом и Дэном работал Том, менеджер по работе с клиентами. Он провел много времени в разъездах, помогая заказчикам в торговых галереях, казино, барах и ресторанах устанавливать и использовать разработанное для них ПО. Втроем они потратили несколько недель на создание спецификации для нового игрового автомата.

Том провел в офисе лишь половину из того времени, которое отвели ему Брюс и Дэн, чтобы начать проектирование программного обеспечения и планирование архитектуры. Как только все трое согласовали требования, они назначили встречу с СЕО и топ-менеджерами компании, чтобы ознакомить их с требованиями, внести необходимые изменения и получить разрешение на начало разработки.

К этому моменту Том вернулся к работе с клиентами, переложив ответственность за проект на Брюса и Дэна. Они разбили проект на задачи, разделили их между членами команды, и все начали писать программное обеспечение. Когда команда практически закончила создание ПО, Том собрал группу бизнес-пользователей, менеджеров проектов и руководителей, чтобы продемонстрировать им почти готовую программу для слот-машины «Боец по выходным».

Но все прошло не так гладко, как ожидалось.

Во время демонстрации программы состоялся неприятный разговор с СЕО**. Он сказал: «Программа выглядит замечательно, но разве не предполагалось, что она будет иметь режим видеопокера?» Это было неожиданное заявление. Судя по всему, директор был уверен, что они работают над программным обеспечением, которое может быть развернуто на базе либо игрового автомата, либо оборудования для видеоигры в покер. Состоялся напряженный разговор между топ-менеджерами, советом директоров

* Если вы менеджер проекта и готовились к экзамену PMP, то уже все знаете о водопадном подходе. Сертификация PMP охватывает не только программное обеспечение, но и множество методологий в различных отраслях промышленности. Хорошо иметь полный набор чертежей при строительстве небоскреба или моста, даже если эти документы меняются.

** Chief Executive Officer — главное должностное лицо компании, аналог генерального директора в российской иерархии. *Прим. ред.*

и двумя крупными заказчиками. Жаль, что никто не потрудился передать команде детали разговора.

Конечно, если бы Дэн узнал об этом раньше, было бы легко изменить направление работы над проектом. Но теперь придется вырывать огромные куски из кода, который они уже написали, и заменять их на модернизированные части кода для видеопокера. Они потратили недели на устранение фатальных ошибок, вызванных проблемами интеграции. Дэн провел много бессонных ночей, объясняя Брюсу, что все это было абсолютно предсказуемо, потому что так бывает всегда, когда код создается для одних целей, а потом его спешно переориентируют. Теперь же ему предстоит увязнуть в распутывании спагетти-кода. Будет трудно поддерживать существующий базовый код, кроме того, команда разочарована, потому что это явно ошибочный путь.

Но в сложной ситуации оказались не только Дэн и Брюс. Менеджер проекта, руководивший им, был настолько расстроен случившимся, что покинул компанию. Он доверял тому, как команда характеризовала текущее состояние дел, но они были полностью обесценены изменениями. Команда понятия не имела, что им придется столкнуться с неожиданной сменой оборудования, и это не сделало жизнь руководителя проекта легче. Несмотря на то что ситуация изменилась, срок сдачи остался прежним. К моменту завершения проекта план устарел и стал бесполезен, однако менеджер проекта продолжал нести за него ответственность. После того как руководство раскритиковало его в пух и прах, он смог оправдаться лишь отсутствием в его команде хороших игроков. Затем он покинул компанию, и на его место взяли Джоанну.

Но сильнее всех был расстроен Том, потому что ему единственному пришлось работать с клиентами, когда они столкнулись с проблемами. Самый крупный клиент этого продукта — казино Little Rock в Лас-Вегасе — хотел настроить все свои игровые автоматы так, чтобы они соответствовали тематике игр, установленных в автоматах в городах штата Арканзас. Заказчик просил создать новую функцию, позволяющую менять игры между сменами, не перемещая игровые терминалы. Их инженеры то и дело натыкались на ошибки в программе, разработанной командой Брюса, поэтому Тому и Дэну приходилось неделями вести телефонные переговоры с инженерами, чтобы придумывать заплатки и временные решения. Little Rock не отменила контракта, но их следующий большой заказ перешел к конкуренту. Не секрет, что директора и менеджеры винят за потерю бизнеса проект Slot-o-matic Weekend Warrior.

В конце концов все узнали, что проект пошел не так. И каждый считал, что виноват кто-то другой. Казалось, никто понятия не имеет, как исправить эти повторяющиеся проблемы. И ПО, которое они поставляли, по-прежнему было недостаточно хорошо организованным.

ИСПОЛЬЗОВАНИЕ AGILE ВЛЕЧЕТ ИЗМЕНЕНИЯ

Брюс, Дэн и Джоанна пригласили Тома на обед. После разговора о прошлых проблемных проектах Джоанна предположила, что настало время попробовать agile-методологии.

Как и многие начинающие свое движение к Agile, они приступили к делу с дискуссии о том, что значит для них само это слово. Брюс сослался на сферу agile-разработки: специализированные книги, практики, тренинги, блоги и людей, практикующих Agile. Для Джоанны понятие Agile означало «способность проекта изменяться», основной набор методов, сосредоточенных на достижении цели. Дэн думал, что в рамках Agile не надо писать документацию, а нужно просто погружаться в код. Том понятия не имел, о чем они говорят, но был счастлив узнать массу способов избежать того, что случилось в прошлый раз.

«Стартующие в Agile» обычно начинают знакомиться с гибкими методами, практиками и идеями, и эта команда не была исключением. Дэн присоединился к ресурсу Agile Alliance и начал налаживать связи с другими agile-практиками. Брюс и Джоанна принялись читать блоги и книги о гибкой разработке и управлении проектами. Они познакомились с замечательными идеями и взялись воплощать их в жизнь, чтобы таким путем прогнозировать и устранять проблемы в своих проектах. Каждый из них обнаружил много полезного и сразу же начал пополнять этим свой арсенал.

Дэн уже писал автоматизированные модульные тесты для своих предыдущих проектов, но многие разработчики музыкальных автоматов никогда не делали этого. Он начал сотрудничать с разработчиками, чтобы организовать модульное тестирование и **разработку через тестирование**. Создал **автоматизированный сборочный скрипт** и настроил **сервер сборки** для проверки кода, создания программного обеспечения и проведения тестирования раз в час. И это сработало! Программисты сразу же увидели улучшение в их коде. Каждый день разработчик находил ошибку, которую никогда не выявил бы без автоматических тестов, и было ясно, что удалось избежать недельных отладок и отслеживания неприятных проблем в этой области. Мало того что стало меньше ошибок, облегчилась и процедура изменения кода, который пишут разработчики.

(Ничего страшного, если вы не знаете всех этих методов, включая разработку через тестирование. Вы познакомитесь с ними в нашей книге, мы будем выделять новые практики жирным шрифтом, чтобы их было проще обнаружить.)

Джоанна прошла обучение **Scrum**, и команда решила называть ее scrum-мастер (хотя во время обучения она узнала о большой разнице между понятиями «scrum-мастер» и «руководитель проекта» и вовсе не была

уверена, что роль, которую она играет в проекте, действительно заслуживает названия «scrum-мастер»). Она помогает команде разбивать проект на **итерации**, отслеживая прогресс на **доске задач** и используя **скорость** работы команды и **графики сгорания** (линейные графики, которые позволяют ежедневно отслеживать объем выполняемой работы, «сжигание» ее до нуля, когда работа выполнена полностью), чтобы держать всех в курсе. Впервые команда действительно заинтересовалась тем, чем занят менеджер проекта, и это способствовало продвижению в работе.

Том тоже хотел пройти обучение Agile. Дэн, Брюс и Джоанна начали называть его **владельцем продукта**, и Том принялся писать **пользовательские истории** для команды, чтобы она лучше представляла свою задачу. Он работал с ними ради создания **планов релизов**, основанных на историях, и теперь чувствовал, что напрямую контролирует то, что будет создавать команда.

Но главное — Брюс начал проводить **ежедневные митинги** с Джоанной, Дэном и всеми программистами. Том тоже посещал их. Сначала не все удавалось, но к моменту, когда проект набрал обороты, никто не испытывал дискомфорта и все давали друг другу объективную оценку того, как продвигается их общее дело. Брюс убедил команду проводить совместную **ретроспективу** в конце каждой итерации и был рад, что коллеги действительно пытаются осуществить улучшения, о которых говорили в ходе ретроспективы.

«ЛУЧШЕ-ЧЕМ-НИЧЕГО» КАК РЕЗУЛЬТАТ

Все удавалось. Команда продвинулась, проект становился все лучше и лучше, но... до определенного момента.

За счет «внедрения Agile» все добились улучшений на своем рабочем месте. Дэн и разработчики начали развивать навыки и тщательнее писать код. Джоанна имела точное представление о том, в каком состоянии находится проект в каждый отдельный момент времени. Том гораздо больше общался с командой, и это давало возможность тщательнее контролировать программное обеспечение, которое они создавали, и, следовательно, максимально удовлетворять запросы клиентов. Брюс смог сосредоточиться на улучшении навыков своей команды и коммуникации.

Но действительно ли команда стала гибкой?

Они переняли немало хороших практик. Многие из этих методов были улучшенными версиями прежних, и все вместе они помогли каждому члену команды стать продуктивнее. В этом, несомненно, было продвижение.

Но наряду с тем, что команда становилась счастливее, а проект «Музыкальный автомат» действительно продвигался лучше, чем любой из предыдущих, у них все же остались опасения в отношении того нового, более гибкого мира, обитателями которого они оказались. Дэн, например, считал, что команда определенно создает код лучше, чем раньше, но приходится идти на отступления от технологий, чтобы уложиться в срок.

Джоанна довольна, что контролирует ход выполнения проекта. Но из-за того, что его приходится разбивать на короткие итерации, она чувствует себя некомфортно. Отказавшись от возможности использовать полный график проекта как дорожную карту, она обнаружила, что оказывается во все большей зависимости от команды, которая посвящает ее в текущую ситуацию во время ежедневных митингов. Эти встречи полезны, но на них в основном озвучивается статус каждого члена команды, а Джоанна лишь послушно все записывает, чтобы довести информацию до заинтересованных сторон. Она начала чувствовать себя простым координатором, а не менеджером, управляющим проектом. Теперь она фокусируется на статусе, что затрудняет выявление и устранение препятствий. Команда лучше реагирует на изменения, но это ставит Джоанну в неловкое положение, потому что все ее внимание теперь обращено на реагирование, а не на планирование.

Теперь Том — владелец продукта, и он в восторге от возможности определять, над чем стоит работать команде. Но он разрывается на части, потому что чувствует: команде нужно, чтобы он работал с ней полный день. Он должен принимать участие в ежедневных встречах и постоянно отвечать разработчикам, интересующимся деталями программы, которую они пишут. Иногда они спрашивают о том, чего он не знает, а порой он хочет, чтобы они ответили на эти вопросы сами. Ему и так есть чем заняться, но он чувствует, что остальные члены команды, дойдя до середины пути, готовы переложить всю ответственность за создание столь важного продукта именно на его плечи. А у него нет ответов на все вопросы.

В конце концов, его основная должность — менеджер по работе с клиентами. Ведь музыкальные автоматы не продают сами себя. Как можно вести отчетность и оставаться в курсе потребностей пользователей, если все его время уходит на ответы программистам?

Брюс рад, что команда работает продуктивнее. Но когда он внимательнее всматривается в происходящее, что-то кажется ему неправильным, но он не может понять, что именно. Очевидно, что это является улучшением, учитывая, что большинство его предыдущих проектов были на волосок от провала. По мнению Брюса, использование Agile сделало ситуацию

лучше, уменьшило необходимость личного героизма, сократило количество работы по ночам и в выходные дни. Но он также чувствует, что появились новые проблемы.

Нередко члены команды и особенно ее руководитель испытывают то же самое, что и Брюс: некоторое разочарование после первой попытки применения Agile. Блоги и книги, которые они читали, и то, что они слышали во время обучения, обещали «поразительные результаты» и «гиперпродуктивные команды». Команда ощущает, что проект создания музыкального автомата — это улучшенная версия предыдущих проектов. Но, безусловно, и речи нет о гиперпродуктивности или поразительных результатах.

Сложилось общее мнение, что проект прошел путь от неконструктивной стадии к конструктивной — и это очень хорошо. Получилось то, что мы называем «лучше-чем-ничего». Но действительно ли суть Agile заключается в этом?

Раздробленное видение

Команды сталкиваются с проблемами с тех пор, как начали создавать ПО. Еще в 1960-х годах специалисты открыто говорили о том, что разработка программного обеспечения фактически разрушена. Они использовали термин *кризис программного обеспечения*, который был придуман на конференции для разработчиков НАТО в 1968 году вместе с другим термином — *разработка программного обеспечения**. «Кризис программного обеспечения» говорит о состоянии разработки софта в типичной компании 1970–1980-х годов, когда серьезные (знакомые и сейчас) проблемы были очень распространены, что приводило к провалу множества проектов. Со временем наша индустрия начала понимать основные источники кризиса ПО. Важной вехой стала публикация в 1970 году статьи Уинстона Ройса, инженера компании Lockheed, в которой описывалась крайне популярная, но неэффективная модель разработки ПО. В начале 1980-х этот метод был широко известен как водопадный подход. Потребовалось еще 10–20 лет, прежде чем многие команды, добивавшиеся успеха в прошлом, перестали слепо доверять ему. Многие команды обнаружили, что agile-методы могут решить проблемы, типичные для водопадного подхода, но одновременно выяснилось, что это не так просто, как кажется.

Разработчики используют программные средства каждый день, чтобы построить свой код. Специалист, владеющий несколькими инструментами,

* Питер Наур и Брайан Рэнделл. «Разработка программного обеспечения: доклад о работе конференции под эгидой Научного комитета НАТО», Гармиш-Партенкирхен (Германия), 7–11 октября 1968 года (Брюссель. Отдел по научным вопросам, НАТО, 1969), 231.

всегда более востребован, чем тот, кто знает меньше. Поэтому многие профессионалы, впервые сталкиваясь с Agile, сразу же воспринимают его как совокупность средств, способов и методов. Практически любой разработчик, начавший работу с Agile, в течение нескольких месяцев обновляет свое резюме, чтобы включить в него новую практику. Это первое впечатление от Agile — хороший признак, потому что оно помогает сделать agile-методы привлекательными и вызывает интерес.

Но видя в Agile лишь инструменты, технологии и методы, вы делаете только первый шаг на пути к «переходу к гибким технологиям» — и это создает проблемный побочный эффект. Взгляните на ситуацию с точки зрения Дэна. Как разработчик и архитектор он будет концентрироваться на том, что непосредственно влияет на развитие: методах, которые помогают ему улучшить код путем устранения или предотвращения ошибок, инструментах, позволяющих выполнять сборку быстрее и проще, и практиках, улучшающих способы разработки, проверки и сборки кода.

Джоанна как руководитель проекта прежде всего заботится о затратах на создание ПО и качестве результатов. То есть она будет концентрироваться на инструментах, которые помогут ей понять проект, наладить коммуникации, составить сметы и оценить усилия. Бизнес-пользователь, например Том, как правило, заинтересован, чтобы ПО обслуживало бизнес и было привлекательно для различных методов, помогающих команде правильно понимать то, чего хотят пользователи, поэтому ПО, которое они создают, имеет ценность. А руководитель команды, например Брюс, хочет убедиться, что все в команде движутся в одном направлении, коммуникации налажены и происходит обучение на собственном опыте. Они ищут инструменты, которые помогут в этом.

В частности, один из agile-подходов — **пользовательская история** — это способ выразить конкретную потребность пользователя. Она, как правило, записывается в виде нескольких предложений, часто на карточках, иногда в строго определенном формате, но порой и в свободной форме. Например, одна пользовательская история из проекта «Музыкальный автомат» звучит так: «В качестве постоянного посетителя этого бара я хочу иметь возможность проигрывать новый хит, который был выпущен сегодня».

Каждый человек в команде судит о пользовательской истории по-своему.

— Джоанна, менеджер проекта, которая пытается стать scrum-мастером, рассматривает пользовательскую историю как работу, которую нужно сделать, аккуратно упаковать и приготовить к сборке. Она записала каждую историю на карточку и прикрепила их к доске, чтобы держать все под контролем.

- Дэн, ведущий разработчик и архитектор, видит эту историю как небольшую часть всего функционала, написанную в доступной для понимания форме. Он может разбить историю на задачи, создать учетную карточку для каждой из них и написать свое имя, когда начнет с ними работать. Когда он закончит с разработкой, то переместит их в ту часть доски, где располагаются выполненные задачи.
- Том, владелец продукта, считает пользовательскую историю ценностью, которая будет предоставлена компании, потому что это позволяет ему видеть связь между тем, что создает его команда, и тем, что пользователи будут делать с уже разработанным ПО. Пользовательские истории помогают ему разговаривать с клиентами, чьими счетами он управляет. Именно благодаря им он способен понять, чего ждут клиенты от программного обеспечения для музыкальных автоматов, и гарантирует, что каждая история представляет собой то, что на самом деле нужно пользователю.
- Брюс, руководитель команды, рассматривает пользовательскую историю как конкретную цель, вокруг которой можно организовать команду. Он помогает выстраивать истории в порядке приоритетности и использует прогресс для поддержания мотивации команды.

Добавление пользовательских историй к такой команде, как эта, поможет улучшить способ разработки ПО, потому что каждый из четырех участников увидит, что история полезна лично ему.

Но это может сработать и против команды. В прошлых проектах Дэн имел подробную спецификацию, что оставляло мало места для маневра, теперь же он получил свободу принимать решения. Это хорошая практика, но она способна вызвать некоторые проблемы. Когда Дэн написал код для пользовательской истории «новейший хит», он думал, что нужна функция, позволяющая постоянному посетителю бара проигрывать любую песню, как только она будет загружена на сервер. Но после ее демонстрации Тому в конце итерации возникла дискуссия. Том объяснил: новые песни в музыкальном автомате означают, что владельцы бара должны платить более высокое роялти. Он обсудил с ними все детали, позволяющие постоянным клиентам проигрывать последние хиты максимально часто, но все же не настолько, чтобы это привело к чрезмерным затратам. Дэн был очень расстроен. Это означало, что ему придется полностью переписать большую часть функций. А Тома возмущало, что первый выпуск программы не будет включать эту функцию.

Если бы Дэн понял, что пользовательская история была важна для Тома, чтобы показать, в чем нуждаются потребители, то он, возможно, обсудил

бы с Томом, какое ПО необходимо, прежде чем приступить к программированию. И наоборот, если бы у Тома было время подробнее узнать о том, как Дэн собирается писать программное обеспечение на основе ограниченного знания пользовательской истории, то, возможно, он убедился бы, что разговор необходим в начале итерации. Но общения не было, и поэтому проект столкнулся с теми же проблемами, что и в прошлых водопадных проектах: разработчики делали неверные предположения, писали код, а потом меняли его. Хотя этого можно было избежать, ведь после изменений он становился более хрупким.

Если каждый человек думает только о своей роли, рассчитывая, что конкретный способ описания пользовательской истории поможет ему, и не следит за тем, как остальные члены команды используют данную историю либо другой agile-инструмент, технику или метод, это может привести именно к тем проблемам, с которыми столкнулись Дэн и Том. Мы называем это **раздробленным видением**, потому что у каждого человека свой взгляд на тот или иной agile-инструмент.

Теперь распрощаемся с командой разработчиков потокового музыкального автомата — мы больше не будем вспоминать о ней в этой книге. Смогут ли они преодолеть свои проблемы и закончить сборку программного обеспечения? Продолжив читать эту главу, вы, возможно, обнаружите идеи, которые могли бы им помочь.

КАК РАЗДРОБЛЕННОЕ ВИДЕНИЕ ВЫЗЫВАЕТ ПРОБЛЕМЫ В ПРОЕКТЕ

Когда каждый член проектной команды смотрит на применяемые методы лишь со своей позиции, постоянно возникают старые проблемы. В годы кризиса программного обеспечения разработчики сразу погружаются в ПО, не тратя время на то, чтобы понять потребности пользователей. О новых требованиях они узнают в середине проекта и вынуждены удалять из программы часть кода, чтобы заменить его. Многие agile-методы направлены на формирование у команд четкого понимания потребностей заказчика с самого начала проекта, что помогает избежать больших объемов неэффективной работы. Но когда в команде не налажены коммуникации — например, разработчик создает код и «спихивает» его владельцу, не задумываясь об истинных потребностях клиента, — это может привести к однообразным проблемам, которые придется решать постоянно.

Что касается владельцев продуктов, их радует, что Agile дает возможность решать именно те задачи, которые нужны пользователям. Ведь до этого им приходилось констатировать, что контроль над проектом утерян, и лишь

беспомощно наблюдать, как команда программистов собирает прошлогоднее ПО, потому что последний разговор с пользователями состоялся именно в прошлом году. Но владелец продукта будет по-прежнему разочарован, зная, что пишет пользовательские истории лишь для того, чтобы обнаружить: команда исполняет не совсем то, что он имел в виду. Команда считает, что владелец продукта ждет от нее угадывания его мыслей, а владелец, напротив, полагает, что эти люди хотят узурпировать все его время и постоянно слышать ответы на любые имеющиеся у них вопросы.

Такой же разрыв происходит и с другими ролями в проекте. Менеджеры проектов и лидеры команд счастливы, что разработчики взяли на себя создание структуры и конкретизацию целей. Они отмечают постепенные улучшения, но, к сожалению, не фундаментальные изменения в работе. Последние невозможны, если члены команды работают, соперничая друг с другом. Менеджер проекта, который видит пользовательские истории, приклеенные к доске в качестве прямой замены диаграммы Ганта в файле Microsoft Project, но продолжает по привычке «командовать и контролировать», будет часто реагировать на изменения в проекте, требуя от команды работать сверхурочно, чтобы не отставать от первоначального плана. Руководитель команды может занять жесткую позицию, ограждая сотрудников от дополнительных задач, пересматривая график с точки зрения сроков или сокращения объема работ. И менеджер проекта, и руководитель команды могут улучшать проект, но если бы они рассматривали перспективы с точки зрения друг друга, то могли бы избежать конфликта и при этом достичь хорошего результата.

Другими словами, когда не налажены коммуникации, каждый на своем месте может использовать новые инструменты (пользовательские истории), но придерживается старых установок, что вызывает трения в команде. Новые инструменты лучше, поэтому дела идут успешнее. Но члены команды действительно не чувствуют большой разницы, потому что возвращаются прежние конфликты. И тогда рождается недоумение: неужели это все, что может Agile?

Как показывает опыт, многие команды столкнулись с подобной проблемой при внедрении отдельных практик и получили закономерный результат — «лучше-чем-ничего». Компания VersionOne разрабатывает гибкие программные инструменты и способствует развитию agile-сообщества во многих областях. Но ее главная заслуга — публикуемые ежегодно State of Agile Development Survey. Результаты 2013 года* демонстрируют, что многие команды добились улучшений, применяя agile-методологии.

* Последний отчет о состоянии вы можете найти на сайте stateofagile.versionone.com.

- 88% опрошенных VersionOne в рамках проекта State of Agile Development Survey 2013 заявили, что их компании занимались гибкой разработкой.
- 92% респондентов сообщили о произошедших за год улучшениях во всех областях их деятельности и оцененных в результате проведенного опроса. Ведущие категории — способность управлять изменяющимися приоритетами (92%), повышение производительности (87%), улучшение обзора проекта (86%), улучшение командного духа (86%) и повышение качества программного обеспечения (82%).

Хотя команды agile-проектов продвигаются быстрее и лучше реагируют на изменения, они часто терпят неудачу из-за культурных и мировоззренческих различий между водопадной и гибкой методологиями. Респонденты указали три основные причины провалов agile-проектов: «отсутствие опыта в использовании гибких методов», «философия компании расходится с agile-ценностями» и «внешнее давление со стороны тех, кто придерживается водопадной модели».

Когда новые команды начинают работать по agile-методологиям, они сталкиваются с проблемами. Чаще всего это происходит, потому что они не изменили водопадных способов работы. Пример команды проекта «Музыкальный автомат» показал: недостаточно просто добавить новую практику, чтобы решить проблемы, которые вызывают конфликты и мешают изменениям. Участники проекта «Музыкальный автомат» могли утверждать, что имеют опыт гибкой разработки, но во многих отношениях они оставались водопадной командой, начавшей применять некоторые важные agile-инструменты. (Дэйв Вест из аналитического агентства Forrester Research придумал специальный термин для подобных ситуаций: Water-Scrum-Fall*.) Другими словами, они стали самой эффективной водопадной командой.

ПОЧЕМУ РАЗДРОБЛЕННОЕ ВИДЕНИЕ ВЕДЕТ К РЕЗУЛЬТАТУ «ЛУЧШЕ-ЧЕМ-НИЧЕГО»

То, что создают люди, часто зависит от того, на чем они сосредоточены. Чем больше люди сосредоточены на своих личных целях, а не на целях команды, тем меньше шансов, что они будут иметь реальную ценность для компании.

* Dave West. Water-Scrum-Fall Is The Reality Of Agile For Most Organizations. Forrester, 26 июля 2011 г. <http://bit.ly/water-scrum-fall>.

Это парадоксально для команды, которая пытается работать с использованием Agile. Команды, которые концентрируются только на отдельных методах, в конечном итоге видят улучшение только в тех областях, в которых они и так сильны. Причина в том, что члены команды могут сосредоточиться только на том, что они уже хорошо знают, а для того, чтобы расширить эти знания, необходимо правильно организовать процесс обучения. Просить команду улучшить то, чего они не знают, кажется завышенным требованием.



Рис. 2.2. Отдельные члены команды стремятся добавить agile-практики в те области проекта, где уже есть успехи, поэтому улучшения происходят только в этих местах. Вот почему раздробленное видение приводит к результату «лучше-чем-ничего»

Вот почему команды, которые внедряют отдельные элементы Agile, часто становятся успешнее, чем были до этого. Они применили лучшие методы к тому, что уже умеют делать на отлично, поэтому и добились заметного прогресса. Но они еще не затронули некоторые области проекта, поскольку методы, влияющие на них, пока не привлекли внимания кого-либо из разработчиков, поэтому часть проблем, влияющих на эти области, не была

устранена. А ведь именно эти области зачастую мешают команде быть гиперпродуктивной и достигать высоких результатов.

Как же решить эту проблему?



Ключевые моменты

Налаживание связи помогает команде лучше управлять изменениями.

Планирование в команде важнее, чем чрезмерное документирование плана и слепое следование ему.

Программные проекты были непредсказуемы и имели плохие результаты начиная с 1960-х годов, эта ситуация называлась «кризис программного обеспечения».

Многие команды пытаются «внедрять Agile», после того как им удалось улучшить с его помощью те направления, которые и так были успешны.

Выбор лучших методов приводит к результату «лучше-чем-ничего», потому что команды принципиально не изменили способ общения или работы.

Пользовательские истории — это agile-инструмент, в котором участник команды (часто владелец продукта) описывает на понятном пользователю языке необходимые ему отдельные функции программного обеспечения.

Выборочное применение отдельных элементов Agile в одном проекте — это наиболее распространенный сегодня способ работы с Agile, но такой путь стать гибким нельзя назвать самым эффективным.

Agile-манифест помогает командам видеть цели применения каждой практики

Манифест гибкой разработки программного обеспечения, более известный как Agile-манифест, написан в 2001 году группой из 17 единомышленников, которые собрались на горнолыжном курорте Snowbird Retreat неподалеку от Солт-Лейк-Сити, чтобы придумать решение, способное помочь избежать проблем при разработке программного обеспечения, с которыми

они сталкивались на протяжении всей своей карьеры. После нескольких дней обсуждения был принят основной набор идей и принципов (а также придумано название Agile). Собравшиеся объединили их в один документ, меняющий образ мышления в мире разработки программного обеспечения.

Agile-манифест содержит четыре простые идеи. Приведем полный текст.

Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь непосредственно разработкой и помогая в этом другим. Благодаря проделанной работе мы смогли осознать, что:

Люди и взаимодействие важнее процессов и инструментов.

Работающий программный продукт важнее исчерпывающей документации.

Сотрудничество с заказчиком важнее согласования условий контракта.

Готовность к изменениям важнее следования первоначальному плану.

То есть, не отрицая важности того, что справа, мы все-таки больше ценим то, что слева.

Понимание и эффективная работа с Agile начинается с понимания этих ценностей.

ЛЮДИ И ВЗАИМОДЕЙСТВИЕ ВАЖНЕЕ ПРОЦЕССОВ И ИНСТРУМЕНТОВ

Люди могут выбрать неправильный путь, если слепо следуют процессам. Отличный инструмент иногда помогает сделать неправильную вещь быстрее. Мир ПО полон отличных методов, но не все из них подходят для любого проекта или ситуации. Однако эта универсальность важна, чтобы лучше понимать, как члены команды работают вместе и как каждый человек влияет на всех остальных.

Это особенно полезно для тех, кто нуждается в улучшении работы команды. Вот почему agile-команды ценят людей и взаимодействие больше процессов и инструментов, которых явно недостаточно, чтобы иметь

«правильные» процессы или «лучшие» методы. Если люди, которые должны использовать процесс или инструмент, не примут его, то он окажется отброшен в сторону. Еще хуже, когда члены команды следуют букве процесса, даже если это заканчивается неправильными результатами. Прежде чем реализовать даже самый логичный и осмысленный процесс, необходимо, чтобы люди, работающие с вами, приняли его. Если они не понимают смысла ваших действий, то будут считать, что вы просите о необоснованных изменениях.

Вот почему важно признать, что вы работаете с группой людей, каждый из которых имеет собственные мотивы, идеи и предпочтения.

Есть много agile-практик, которые поддерживают этот принцип, так же как и множество различных способов мышления. Поэтому в книге описаны **ежедневные митинги** и **ретроспективы** (где каждый рассказывает, как прошел день или итерация и какие уроки можно извлечь). И **пользовательские истории** важны в том числе и потому, что они помогают команде вести разговор о том, что означает каждая конкретная история.

РАБОТАЮЩИЙ ПРОГРАММНЫЙ ПРОДУКТ ВАЖНЕЕ ИСЧЕРПЫВАЮЩЕЙ ДОКУМЕНТАЦИИ

Во всем мире существуют целые тома подробной и всеобъемлющей программной документации, стоящие на закрытых полках. Их так много, что трудно предсказать, какой документ пригодится, а какой будет бесконечно пылиться на полке. Из-за этого многие команды и особенно их руководители принимают комплексный подход, при котором каждая мелочь должна быть задокументирована независимо от того, понадобится ли это в будущем.

Agile-команды ценят работающий программный продукт больше исчерпывающей документации. Но в данном случае важно понять смысл слова «работающий». Для agile-практика это такой продукт, который добавляет ценность организации. Например, программный продукт, на котором компания зарабатывает деньги, или ПО, обеспечивающее более эффективную деятельность сотрудников компании. Для проекта это означает, что он должен заработать или сэкономить больше денег, чем стоимость разработки программного продукта. Ценность всегда связана с деньгами, даже если никто не говорит об этом напрямую. И команда должна придавать особенное значение тому факту, что ценность ей придает сборка и поставка работающего ПО. Документация — лишь средство достижения этой цели.

Приоритет работающего ПО над всеобъемлющей документацией не означает, что документы не нужны вовсе. Среди них очень много полезных для команды. Но важно иметь в виду, что документацию и программное обеспечение зачастую пишут одни и те же люди. Документация, помогающая им заранее, прежде чем они соберут ПО, понять проблему, общаться с пользователями и исправлять недостатки, экономит больше времени и усилий, чем нужно на ее создание. Часто также мы имеем дело с документацией такого рода, как каркасная визуализация или диаграммы последовательности, которые программисты вовсе не отказываются писать.

В то же время концентрация на работающем программном обеспечении — это отличный способ убедиться, что команда находится на верном пути. Работа над документацией, которая явно нацелена на создание работающего программного обеспечения, вносит позитивный вклад в проект. На самом деле часто команде может потребоваться новый подход к работе над документацией, что позволит этой документации быть встроенной в саму программу. Например, одна из agile-практик предлагает способ **разработки ПО через тестирование**: программисты строят автоматизированные модульные тесты до создания программного обеспечения, для проверки которого они предназначены. Эти тесты существуют в виде кода, хранящегося вместе с остальным кодом ПО. Но он также служит в качестве документации, потому что дает разработчикам сведения о том, что код должен делать и каким должно быть ожидаемое поведение отдельных элементов программного обеспечения.

СОТРУДНИЧЕСТВО С ЗАКАЗЧИКОМ ВАЖНЕЕ СОГЛАСОВАНИЯ УСЛОВИЙ КОНТРАКТА

Многие, читая «условия контракта», полагают, что они нужны лишь консультантам и подрядчикам, работающим в рамках контракта. На самом деле это касается многих команд, работающих в одной компании. Когда программисты, тестировщики, владельцы бизнеса и менеджеры проектов работают в разных командах и не сотрудничают в целях реализации единой рабочей программы, они часто ведут себя так, будто работают по разным контрактам. В большинстве компаний они явно будут обсуждать SLAs (service-level agreements — соглашения об уровне обслуживания) между командами программирования, тестерами и разработчиками, а также между командами и их пользователями.

Это, конечно, снизит риск конфликтов с руководством (потому что в подобном случае легче обвинить другую команду за непоставку ПО), но не сможет достичь главной цели — получить работающее программное

обеспечение, необходимое пользователям. Разработчик, занимающий круговую оборону, имеет меньше возможностей для поиска новых путей сотрудничества и инноваций с пользователями ПО.

Один из способов, который могут взять на вооружение agile-команды, — поставить владельца программного продукта в центр внимания, сделать его главным членом команды. Он может не заниматься активной разработкой кода, но будет присутствовать на планерках, вносить идеи и, главное, чувствовать свое право собственности на конечный продукт. Владельцы продукта часто воспринимают пользовательские истории как способ взаимодействия с остальными членами команды.

ГОТОВНОСТЬ К ИЗМЕНЕНИЯМ ВАЖНЕЕ СЛЕДОВАНИЯ ПЕРВОНАЧАЛЬНОМУ ПЛАНУ

Существует старое правило управления проектами, которое гласит: «план работы — рабочий план». Если вы работаете по неправильному плану, то создадите неправильный продукт. Именно поэтому командам нужно постоянно следить за изменениями и быть уверенными, что они четко реагируют на них, если эти изменения нужны пользователям или процессу создания ПО. Когда ситуация меняется, проекту нужен новый план.

Сопротивление переменам — не редкость среди тех, кто создает план, потому что изменение плана требует дополнительных усилий. Возможно, было вложено много усилий в разбивку проекта на пакеты работ и оценку каждого этапа. Изменение в этом случае может потребовать от менеджера проекта переделки всей работы. И если для него следование первоначальному плану превышает готовности меняться, то ему придется и дальше проявлять упорство. Это делает работу над проектом более гладкой, но если изменения действительно необходимы, то это будет гораздо труднее сделать позднее, когда работа над кодом будет в самом разгаре.

Доска задач — хороший метод, помогающий команде правильно реагировать на изменения. Каждый элемент работы (как правило, пользовательская история) написан на карточке и прикреплен к доске — вроде той, которую Джоанна использовала для проекта «Музыкальный автомат». Все задачи обычно распределяют по трем столбцам, размещая их в том порядке, который показывает состояние каждой из них. Доска задач может управляться при помощи компьютерной программы. Но многие команды считают наиболее эффективным использование настенной доски, потому что, стоя перед ней, можно дискутировать и перемещать истории. Такая форма общения намного продуктивнее простого разговора. Доска создана

так, что любой может изменить порядок задач, и даже рекомендуется это делать. Если происходят изменения, то любой может добавить учетные карточки на доску задач, а не удалять каждое изменение при помощи единого центра управления проектом. Это помогает держать всех в курсе дела, поэтому план не устаревает.

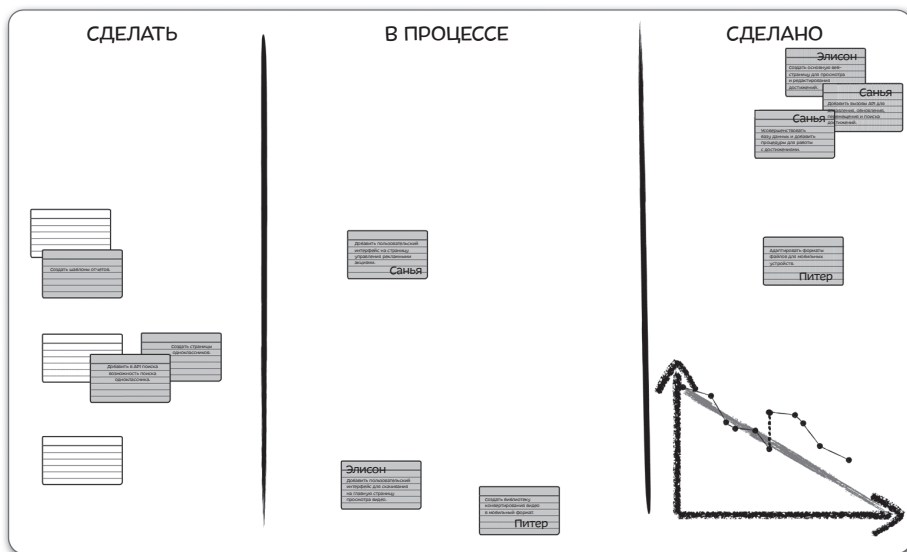


Рис. 2.3. Agile-команды часто используют доски задач, чтобы размещать на них задачи и отслеживать прогресс. Они будут писать задачи или пользовательские истории на карточках и перемещать их по доске, отмечая прогресс. Многие команды также рисуют диаграммы на своих досках, чтобы демонстрировать прогресс

ПРИНЦИПЫ ПРЕВЫШЕ МЕТОДОВ

Наша команда музыкального автомата получила хорошие результаты, потому что воспользовалась некоторыми прекрасными методами и благодаря им смогла улучшить проект. Но из-за раздробленного видения члены команды не получили полной отдачи от совместной работы над усовершенствованием способа создания ПО. Существует agile-мышление, которое выходит за рамки практик, и команды, ищущие свой путь к ключевым идеям Agile, найдут лучший способ сотрудничества и взаимодействия.

Другими словами, команда, использующая agile-практики, чтобы построить рабочее программное обеспечение, и находящая ценность работы

с клиентами во взаимодействии и сотрудничестве с ними, в ответ на собственные изменения *получит больше от своих проектов*, чем те, кто просто применяет лучшие методы планирования, программирования и ведения документации.

Джим Хайсмит удачно обобщил эту идею в своей книге Agile Project Management: Creating Innovative Projects:

Без конкретных практик принципы бесплодны, но без принципов в практиках нет жизни, нет характера, нет сердца. Великие продукты возникают у великих команд — принципиальных, которые имеют характер, у кого есть сердце, упорство и мужество*.

Так как команды выходят за рамки простого принятия методов и становятся «принципиальными», могут ли они создавать отличные продукты?



Ключевые моменты

Agile-манифест содержит общие ценности и идеи, которые делают команды эффективными.

«Люди и взаимодействие важнее процессов и инструментов» означает, что команда должна сосредоточиться на людях и прежде всего на том, как они общаются, а затем уже на инструментах и методах, которые они используют.

«Работающий программный продукт важнее исчерпывающей документации» означает, что поставка программного обеспечения, которое делает именно то, что от него нужно пользователю, значительно ценнее, чем поставка спецификации, описывающей это ПО.

«Работающий программный продукт» означает такое ПО, которое обеспечивает ценность компании.

«Сотрудничество с заказчиком важнее согласования условий контракта» означает уважительное отношение ко всем, как будто они в одной команде.

* Jim Highsmith. Agile Project Management: Creating Innovative Projects. 2nd Edition (Upper Saddle River, NJ: Pearson Education, 2009).

Многие эффективные agile-команды рассматривают владельца продукта в качестве члена проектной группы, с которым нужно сотрудничать, а не только вести переговоры как с клиентом или заказчиком.

«Готовность к изменениям важнее следования первоначальному плану» означает признание того, что планы становятся неточными, и поэтому главное — поставить программное обеспечение, а не только разработать план работы.

Доска задач — это инструмент agile-планирования, в котором пользовательские истории крепятся к доске и делятся на столбцы в зависимости от своего статуса в текущем проекте или итерации.

Понимание слона

Лисса Адкинс в своей книге Coaching Agile Teams* объясняет, как метафора может стать ценным инструментом для понимания концепции.

Метафора — это мощная вещь. Профессиональные коучи знают об этом уже давно. В самом деле, «метафора» — это основной навык, которому учат на профессиональных коуч-курсах... Коучи задают вопросы, помогающие клиентам создавать свою собственную метафору, которая одновременно должна быть глубокой и яркой. Клиенты используют метафору, чтобы прокладывать свой путь через события, происходящие в их жизни**.

Существует полезная метафора, которая поможет получить представление о том, что значит сломанная перспектива и почему она ведет команду по наименее эффективному пути. Это история о слепых и слоне.

Чтобы определить, как выглядит слон, шестерых слепых попросили ощупать разные части тела слона и описать свои ощущения. Слепец, ощупывающий ногу, сказал, что слон похож на столб. Тот, кто ощупывал хвост, утверждал, что слон похож

* Русский перевод книги готовится к выходу в издательстве «Манн, Иванов и Фербер».
Прим. ред.

** Lissa Adkins. Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Transitions (Boston: Addison-Wesley, 2010).

на веревку. Ощупывавший хобот предположил, что слон похож на дерево. А тот, кто трогал ухо, был уверен, что слон шершавый, как веер. Слепец, который трогал живот, сказал, что слон похож на стену. Трогавший бивень утверждал, что слон похож на твердую трубу.

Король объяснил им: «Все вы правы. Причина, по которой ваши мнения не совпадают, в том, что каждый ощупывал разные части слона. На самом деле слон имеет все свойства, которые вы описали»*.

Команды, добивающиеся от гибких методологий результата «лучше-чем-ничего», зачастую способны получить достаточно хорошее программное обеспечение еще до начала применения Agile. Они надеются, что Agile поможет им сделать проект еще лучше. Суть в том, что до начала внедрения agile-методов команды уже испытывают проблемы (хотя это еще нельзя назвать серьезным кризисом программного обеспечения), которые причиняют вред проектам не сразу, но вызывают трения в команде. Это и есть «раздробленное видение»: разработчики думают только о разработке, проектные менеджеры — лишь об управлении проектом, они пишут код и не стараются разрушить стену непонимания, отделяющую их от клиента, который думает только о бизнесе.

Все заняты мыслями о собственной работе над проектом настолько сильно, что используют такую фразу, как «перебросить соседу через забор», которая явно разделяет команду и уничтожает сотрудничество. Если каждый думает исключительно о собственной задаче и мало коммуницирует с другими членами команды, то все работают по отдельности, а не как единое целое.

В этой ситуации очень подходит история «Слепые и слон». В раздробленном agile-видении человек использует только те методы, которые влияют на его работу (так же как каждый из слепых ощущает только одну часть слона). Например, разработчики концентрируются на разработке через тестирование, рефакторинг и автоматической сборке. Руководители проекта — на доске задач, скорости реализации проектов и выполнении графика. Бизнес-пользователи применяют планирование выпуска и пользовательские истории, чтобы получить более точное представление о том, что делает команда. Руководители используют ежедневные встречи и ретроспективы для управления и улучшения команды. Каждый хочет чем-то

* Страница из англоязычного варианта «Википедии» (Blind Men and the Elephant, проверено 25 июня 2014 года).

отличиться в проекте и видит несколько методов, позволяющих делать что-то конкретное, чтобы помочь ему в работе. (Мы рассмотрим каждый из этих методов чуть позже, так что не беспокойтесь, если вы еще не знакомы с ними.)

Внедряя эти методы по отдельности, вы, безусловно, улучшите положение вещей, потому что agile-практики действительно очень полезны. Проблема в том, что все — разработчики, менеджеры проектов, бизнес-пользователи и лидеры команды — видят проект с разных точек зрения и концентрируются только на тех методах, с которыми непосредственно связаны. Существует парадоксальный эффект («Именно я был прав!»), когда каждый человек видит лишь ту часть agile-методологий, которая влияет на его непосредственную задачу, и приходит к выводу, что Agile нужен для того, чтобы остальные придерживались его точки зрения.

Таким образом, хотя agile-«слон» и состоит из множества потрясающих практик, все же он существенно больше, чем простая сумма всех частей. И если вы отмечаете лишь отдельные методы — особенно те, которые влияют исключительно на вашу работу в проекте, — то будете видеть только небольшой кусок Agile. «Слон» Agile состоит из повседневных практик, но это гораздо больше, чем просто agile-практики.



Рис. 2.4. Agile-«слон» больше, чем сумма гибких практик

Команда, члены которой видят только практики и не думают о принципах, упускает из виду важные моменты взаимодействия между людьми. Их видение останется раздробленным, члены команды будут действовать по отдельности, а не как эффективно функционирующее целое. Они, безусловно, продолжают выполнять свою часть работы, но без усиления командного взаимодействия и сотрудничества, которые делают agile-методологии действительно мощными.

Это то, из чего состоит Agile. Вспомним еще раз первый постулат из Agile-манифеста.

Люди и взаимодействие важнее, чем процессы и инструменты.

Процессы, методологии и инструменты по-прежнему значимы (поэтому манифест заканчивается словами «...не отрицая важности того, что справа, мы все-таки больше ценим то, что слева»). Но еще важнее люди и их взаимодействие между собой. Именно эти ценности (наряду с 12 принципами, о которых вы узнаете в главе 3) демонстрируют нам, как практики работают вместе и служат руководством для команд, желающих принять их.

МЕТОДОЛОГИИ ПОМОГАЮТ ВАМ ПОЛУЧИТЬ ВСЕ ЗДЕСЬ И СЕЙЧАС

Существует большой разрыв между пониманием ценностей Agile-манифеста (и стоящих за ними принципов) и фактическим изменением способа создания программного обеспечения вашей командой. К счастью, есть еще один важный аспект гибкой разработки, помогающий уменьшить этот разрыв, — **agile-методологии**, которые предназначены помочь командам принять Agile и улучшить свои проекты.

Agile-методологии ценны, потому что помогают увидеть методы в контексте. Они особенно полезны для команд, незнакомых с гибкими методами. Каждая методология на протяжении многих лет разрабатывалась и совершенствовалась agile-экспертами, ориентированными на различные части «слона». Принятие методологии означает, что вы будете следовать надежным решениям, которые нужно соблюдать от начала и до конца проекта без судебных разбирательств и ошибок, ведущих к раздробленному видению.

Agile — это набор методов в сочетании с идеями, советами, а зачастую и совокупностью знаний и богатого опыта agile-практиков. Гибкие методологии будут подчеркивать разницу ролей и обязанностей для каждого участника проекта и рекомендовать определенные методы для каждого из них на различных этапах.

Результаты опроса о гибкой разработке, проведенного в 2013 году компанией VersionOne, содержат список самых популярных методик. Первое место занимает **Scrum**, затем идет сочетание Scrum и **XP**. Респонденты также сообщили об использовании Lean и Канбана. Это не гибкие методологии (о них вы узнаете в главах 8 и 9), но они по-прежнему составляют основу Agile.

Алистер Коберн так описывает Scrum в своей книге «Быстрая разработка программного обеспечения»:

Scrum можно сформулировать (но не применять) очень просто:

- Команда и спонсоры проекта создают упорядоченный список всего, что нужно сделать. Это могут быть задачи или функции. Перечень обозначается как невыполненный бэклог продукта.
- Каждый месяц команда вытягивает верхнюю часть списка, которую она оценивает как часть работы на месяц. Расширяет и детализирует перечень задач и называет его бэклогом спринта. Команда обещает спонсорам продемонстрировать или предоставить результаты работы в конце месяца.
- Каждый день члены команды собираются вместе на пять–десять минут, чтобы сообщить друг другу о состоянии дел и любых препятствиях, которые тормозят их. Это называется ежедневным митингом.
- Кто-то один назначается scrum-мастером. Задача этого человека — самому или с чужой помощью устранить любые проблемы, о которых говорится на ежедневных совещаниях*.

Для многих команд начало перехода к Agile означает применение конкретных методов (которые мы выделили жирным шрифтом и объясним более подробно в главе 4):

- Владелец продукта создает и поддерживает **бэклог продукта** (невыполненные работы по продукту), перечень требований для программного обеспечения.
- Команда выполняет сгруппированные по времени месячные **спринты** (короткие фиксированные итерации в работе над проектом),

* Коберн А. Быстрая разработка программного обеспечения. М. : Лори, 2013.

в рамках которых они готовят месячный объем требований продуктового бэклога по созданию, тестированию и демонстрации продукта. Требования для текущего спринта называют **бэклогом спринта** (списки невыполненных работ спринта). (Некоторые scrum-команды используют спринты, продолжительность которых составляет две или четыре недели.)

- Команда встречается на **ежедневных митингах**, во время которых каждый рассказывает о работе, которую он делал накануне, планах на сегодня и обо всех возникших проблемах.
- **Scrum-мастер** выступает в качестве лидера и коуча, он поддерживает команду в рамках проекта.

Но внедрение Scrum требует большего, чем простое принятие этих серьезных практик. Каждый из перечисленных методов в принципе может быть использован таким образом, что не будет отражать ценности Agile. Ежедневные митинги, например, очень полезны, если команда использует их, чтобы сотрудничать и двигать проект вперед. Но эти же совещания могут попросту использоваться руководителем проекта для информирования команды об индивидуальных заданиях и получения от каждого данных о текущем состоянии дел. Также каждый разработчик не упустит случая, чтобы на встрече сообщить менеджеру проекта: «Вот несколько препятствий на моем пути — давай, разберись с ними». Если каждый цепляется за свою роль, а о других думает примерно так: «Занимайся своими проблемами сам!», то он начинает воспринимать любую новую проблему как чужую. Собрание превращается в говорильню, а не в место для сотрудничества. Команда, попадающая в эту ловушку, может внедрять scrum-методы, но не использовать их.

(В дальнейшем вы больше узнаете о том, как работает Scrum и его методы.)

Вторая методология — **экстремальное программирование** (или XP). В книге Джеймса Шора и Шейна Уордена The Art of Agile Development так описывается XP: «Используя параллельные фазы, команда XP производит развертывание программного обеспечения каждую неделю. В каждой итерации команда анализирует, проектирует, кодирует, тестирует и разворачивает подмножество функций». (Многие XP-команды используют итерации, продолжающиеся одну неделю, а некоторые — две недели или месяц. Кроме того, XP может быть адаптирован для использования при различных продолжительностях итераций. Дальше в нашей книге вы узнаете больше об адаптации гибких методологий.) XP устанавливает конкретные методы разработки, направленные на улучшение сотрудничества с пользователями, планирования и тестирования. Но XP выходит за рамки, применяя эти

методы, чтобы помочь команде собирать простые, гибкие конструкции программного обеспечения, которые команда может поддерживать и расширять.

Scrum и XP имеют много общего, включая тот факт, что они **итерационные**. Это означает, что проект делится на итерации, в которых команда выполняет все активности полного проекта, необходимые, чтобы произвести развертывание программного обеспечения в конце каждой итерации. Некоторые XP-команды используют итерации, длящиеся неделю, а scrum-команды — длиною в месяц. Установка ограничений на продолжительность итераций называется **таймбоксинг (timeboxing)**, и это помогает пользователям узнавать, когда они могут ожидать появления дополнительных функций у ПО.

Многие команды считают, что внедрение методологий — особенно Scrum и XP — плодотворнее, чем простое использование индивидуальных практик. Хотя последнее позволяет каждому участнику команды выбирать методы, характерные именно для его работы, внедрение единой методологии собирает всех членов команды вместе, чтобы выяснить, как выбранные по отдельности практики будут укладываться в рамки одной методологии. Чтобы сделать это, нужно изменить способ мышления. Методологии строятся вокруг agile-ценностей и принципов, поэтому перемены, как правило, касаются сотрудничества и взаимодействия, разработки программного обеспечения и реагирования на изменения. Переход облегчают книги и знания, собранные другими agile-практиками, а также поддержка со стороны уже существующих сообществ, формирующихся вокруг этих методологий.

Lean — это не методология, а скорее образ мышления. Он имеет собственный набор ценностей и инструментов мышления, помогающих принять его. Lean так же важен для мира Agile, как XP и Scrum, и вы сможете многое узнать о том, что значит быть гибким, если разглядите общее между ними. **Канбан** — это agile-метод, помогающий командам улучшать сборку программного обеспечения. Он строится на ценностях Lean и включает собственные методы, помогающие команде лучше работать и развиваться.

Если приглядеться повнимательнее, то методы и приоритеты в XP отличаются от методов и приоритетов в Scrum. Lean и Канбан имеют различный подход к выбору практик и основных направлений. Почему эти непохожие друг на друга подходы к Agile, имеющие абсолютно разную направленность и практики, все еще считаются гибкими? Дело в том, что все гибкие методологии основываются на одних и тех же принципах, опираются на членов команды и на совместную работу над каждым аспектом проекта. Ценности и принципы Agile-манифеста — это то, что объединяет все методологии.

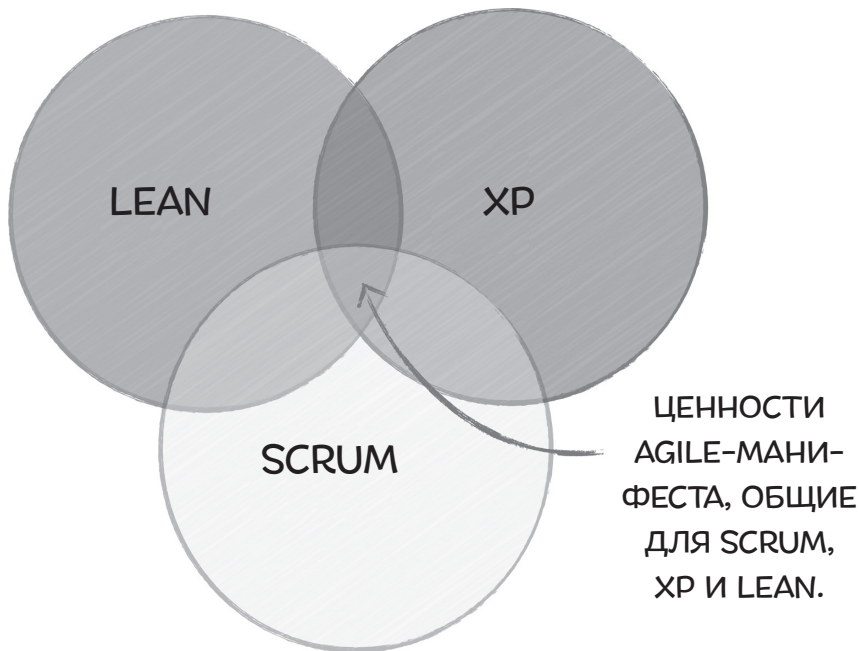


Рис. 2.5. Scrum, XP и Lean все в своей основе имеют agile-ценности и разделяют некоторые ценности, идеи и методы друг с другом

С чего начинать при работе с новой методологией

Приняв решение о цели внедрения методологии, каждый член команды начинает обсуждать с остальными методы, идеи и перспективы. Это противоположность раздробленного видения. Глядя на ситуацию в целом, команды начинают понимать, как индивидуальные практики взаимодействуют друг с другом. Вот где Брюс, Дэн, Джоанна и Том хотели бы быть! Но они не знают, как туда попасть.

Когда команда впервые опробовала новые практики, она все еще не осознает, как они будут соотноситься с хорошо знакомыми ей методами. Понимание придет тогда, когда будет накоплен опыт работы с методологией. Такой способ работает, потому что Agile представляет собой комплексную систему, включающую успешно взаимодействующие практики, которую команды используют, чтобы стать продуктивнее. Внедрение полного набора практик поможет команде получить основу для изучения этих взаимодействий.

Но внедрение целой методологии — более сложный процесс, чем выбор практик, которые вписываются в текущий способ работы команды. Если

команде удастся внедрить всю методологию сразу, то она будет иметь больше шансов получить максимальную отдачу от agile-усилий. Отчасти это связано с тем, что помимо уже знакомых методов каждый внедряет идеи, которые не представляются необходимыми в первую очередь.

Как вы помните, команда разработчиков музыкального автомата столкнулась с проблемами, потому что Брюс, Дэн, Джоанна и Том подошли к agile-практикам независимо друг от друга. Чтобы получить наибольшую отдачу от Agile, они должны были прежде всего обсудить каждый из этих методов с точки зрения их необходимости для проекта в целом и членов команды в частности. Но дело в том, что они не знают, как начать такое обсуждение. Как и многие команды, они сталкиваются с дилеммой: если они уже знают, как именно использовать agile-практики для проекта и как работать вместе, то незачем это обсуждать. Но поскольку они этого еще не знают, то проводить такого рода дискуссии на ранних стадиях трудно.

Решение проблемы — в 12 принципах, которые тесно связаны с ценностями Agile-манифеста. Вы узнаете о них в главе 3.



Ключевые моменты

Команда, которая фокусируется только на отдельных практиках, может упустить из виду более глобальные цели, такие как улучшение коммуникации и реагирование на изменения.

Agile-методологии — это набор практик в сочетании с идеями, советами и опытом сообщества специалистов-практиков.

Такие agile-методологии, как Scrum, XP и Lean, включают в себя большой набор практик, но они также акцентируют внимание на идеях, которые помогают удерживать внимание команды на достижении этих целей.

Agile-коучи часто используют метафоры в качестве инструмента, чтобы помочь своим командам учиться.



Часто задаваемые вопросы

Если в Agile-манифесте утверждается, что исчерпывающая документация не нужна, означает ли это, что мы не должны ничего документировать?

Это очень распространенный вопрос. Прочтите еще раз то, что написано в этом пункте манифеста:

Мы ценим <...> работающий программный продукт выше исчерпывающей документации.

Это не значит, что мы как практики agile-методологий не ценим исчерпывающую документацию. И мы, конечно, не думаем, что вы не должны писать никаких документов! Существует много полезной документации, которая не является исчерпывающей.

Это означает, что передача рабочего ПО в руки пользователей — это лучший способ узнать, насколько хорошо мы как команда добиваемся улучшений.

Но в наших проектах есть место и для записей. Мы будем документировать наш код, используя комментарии к коду (например, чтобы объяснить, почему мы приняли такое решение, или не пишем код другим способом, или используем иной алгоритм). Далее вы узнаете о документе под названием «пользовательская история». Он обычно написан на карточке и помогает вам, команде, пользователям и другим заинтересованным сторонам работать вместе, чтобы выяснить, что именно вы будете строить. Есть много других видов документации, в чем-то более подробных, чем те, которыми пользуются agile-команды.

Вы уверены? Я точно знаю, что Agile — это значит ничего не писать и не планировать, а сразу переходить к программированию. Разве это не эффективнее?

Один из самых распространенных мифов о гибкой разработке программного обеспечения заключается в том, что agile-команды ничего не планируют. Но на самом деле они проводят гораздо более тщательную работу по планированию, чем многие традиционные проектные команды. Однако разработчикам, пришедшим в Agile недавно, может показаться, что планирования практически нет, потому что в нем участвует вся команда — и никто не стонет (а ведь жалобы — это типичная реакция программистов в ответ на приглашение принять участие в планерке).

Scrum-команды, например, посвящают обычно целый рабочий день планированию тридцатидневной итерации. Кроме того, они проводят ежедневные совещания (обычно длящиеся 15 минут), на которых вместе рассматривают план. Для пяти человек в команде это составляет 40 человеко-часов планирования в начале итерации и еще столько же в течение ближайших 30 дней. Это гораздо больше, чем многие традиционные

команды делают за 30 дней разработки программного обеспечения. Неудивительно, что scrum-команды выполняют такую работу точно в срок! При этом члены команды вовсе не считают, что планирование — это «скучно». Дело в том, что они вовлечены в процесс, заботятся о результате и чувствуют, что усилия, потраченные на планирование проекта, необходимы, чтобы остальные итерации протекали успешно.

(Вы подробнее узнаете о механизмах планирования scrum-проекта в главе 4.)

Но разработчику, видящему это со стороны, может показаться, что все это напоминает погружение прямо в проект без планирования. Если команда занята планированием всего один день в начале 30-дневной итерации, то, значит, на следующий день она уже может начать программирование (если для них это наиболее важно в данный момент). Поэтому нередко кажется, будто они практически не занимаются планированием, хотя на самом деле их усилия, выраженные в сумме потраченных на это часов, существенны.

Правда ли, что Agile подходит только очень опытным разработчикам, которые хорошо справляются с планированием?

Нет, Agile подходит для любого уровня мастерства. Планирование — это навык, а единственный способ улучшить его — практика. Иногда (можно утверждать, что довольно часто) даже опытным разработчикам приходится пересматривать свою оценку. Мы читали много историй о реально существующих командах, в которых младшие разработчики проделывали грандиозную работу по внедрению Agile и это помогало создавать программное обеспечение, выходящее далеко за пределы ожиданий компании.

Однако есть один нюанс: младшие разработчики в эффективных agile-командах недолго остаются на своих невысоких позициях. Может быть, это одна из причин, почему люди думают, что Agile подходит лишь опытным специалистам.

Если все члены команды (тестировщики, бизнес-аналитики, UX-дизайнеры, руководители проектов и т. д.) не используют agile-методологию, то могут ли я заниматься гибкой разработкой самостоятельно?

Да. Но, вероятно, это будет не очень эффективно. Когда люди говорят о введении agile-методологий только для разработчиков, значит, они собираются применять только отдельные ее методы. Разработчики получают импульс для улучшения собственной продуктивности, поэтому польза все равно будет (результат «лучше-чем-ничего»). Но команда

не изменится и не поменяет взглядов на ведение проектов, а это серьезно ограничивает позитивное влияние гибкого мышления на производительность. Способ работы над проектом, при котором команда работает по сценарию Water-Scrum-Fall (гибридная модель создания ПО), оставляет у ее членов чувство неудовлетворенности agile-методологиями.

Если я не использую Scrum, XP, Lean или Канбан, то значит ли это, что моя команда не гибкая?

Конечно, нет. Существует много гибких методологий — мы сосредоточились на нескольких и используем их, чтобы раскрыть перед вами идеи Agile. К тому же одна из целей этой книги — помочь ответить на вопрос «Что такое Agile?». В дальнейшем мы расскажем вам о ценностях и практиках различных методологий. С их помощью вы поймете, что значит быть гибким, а также узнаете о том, как такие, казалось бы, разные методы могут быть гибкими.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой):

- Запишите все методы, которые вы используете при создании программного обеспечения. Это может быть описание спецификации, проверка кода в системе контроля версий, использование диаграммы Ганта для документирования плана проекта или ежедневные митинги.
- Попросите кого-нибудь из вашей команды также составить перечень методов, которые вы используете. Сравните списки. Какие методы есть только в одном из них? Обсудите их. Можете ли вы в будущем найти разницу между двумя списками?



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы узнаете больше об agile-ценностях и принципах гибкой разработки программного обеспечения из книги Алистера Коберна «Быстрая разработка программного обеспечения» (М. : Лори, 2013).
- Вы можете узнать подробнее о том, как принципы соотносятся с практиками, в книге Джима Хайсмита Agile Project Management: Creating Innovative Projects (Addison-Wesley, 2009).
- Узнайте больше об agile-коучинге из книги Лиссы Адкинс Coaching Agile Teams (Addison-Wesley, 2010).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- При работе с новой командой поговорите индивидуально с каждым из ее членов и постарайтесь понять, как видение зависит от его роли.
- Расспросите всех участников группы о ценностях Agile-манифеста: что они думают о них, какие считают важными, полагают ли, что эти ценности касаются всех.
- Команды часто испытывают по отношению к результату своей работы чувство «лучше-чем-ничего», но не могут описать это словами. Попросите членов команды привести примеры из своей практики, которые они считают никчемными или требующими значительных усилий, но не приносящими особой выгоды.
- Начните разговор об отдельных ценностях или методах Agile-манифеста — например, если команда рассказывает о «договоре», который они заключают со своими пользователями, то используйте его в качестве отправной точки для разговора об условиях этого договора вместо обсуждения обычного взаимодействия с клиентами. Помогите им понять, какой они делают выбор.

ГЛАВА 3

Agile-принципы

Если бы я спросил людей, чего они хотят, они бы сказали, что хотят более быстрых лошадей.

Генри Форд*

Нет единого рецепта создания превосходного программного обеспечения. Agile-команды признают это. Но есть идеи и правила, которые помогают делать верный выбор, избегать проблем или справляться с ними, когда они неизбежны.

Мы уже познакомились с четырьмя ценностями, изложенными в Agile-манифесте. Помимо них существует 12 принципов, которые должен использовать каждый agile-практик — член команды разработки программного обеспечения. Когда 17 авторов Agile-манифеста встретились в Snowbird, они зафиксировали в этом документе четыре главные, по их мнению, ценности. Но чтобы придумать 12 дополнительных принципов, им потребовалось больше времени. Алистер Коберн, подписавший манифест, вспоминал**:

Группа из 17 человек быстро согласилась с выбором основных ценностей. Разработка же следующего уровня утверждаемых положений потребовала больше времени, чем то, которым мы располагали. Принципы, включенные в этот список, составляют набор текущих рабочих моментов.

Эти положения должны эволюционировать по мере того, как нам становится понятно, как именно люди воспринимают наши слова, а мы сами придумываем более точные выражения.

* Существуют некоторые разногласия по поводу того, действительно ли Генри Форд так говорил, но почти все согласны с тем, что ему бы понравилась эта мысль.

** См. в уже упоминавшейся книге Алистера Коберна «Быстрая разработка программного обеспечения».

Я буду удивлен, если эта версия не устареет вскоре после публикации книги. Ознакомиться с обновленным вариантом можно на сайте Agile Alliance.

Алистер был прав: манера подачи материала на сайте по языку и сути отличается от стиля этой книги. Язык всегда развивается, но идеи и принципы остаются неизменными.

В этой главе вы узнаете о 12 принципах гибкой разработки программного обеспечения: что они собой представляют, для чего нужны и как влияют на проект. На практическом примере мы покажем, как эти принципы применяются в жизни. Чтобы было легче учиться, разделим принципы на четыре части: поставка, коммуникация, выполнение и совершенствование, потому что они являются последовательными этапами. И хотя применение одного из них — эффективный способ узнать об остальных, каждый из этих принципов является самостоятельным.

12 принципов гибкой разработки программного обеспечения

1. Наш наивысший приоритет — это удовлетворение заказчика при помощи частых и непрерывных поставок ценного для него программного обеспечения.
2. Мы принимаем изменения в требованиях даже на поздних этапах реализации проекта. Agile-процессы позволяют использовать изменения для повышения конкурентоспособности продукта.
3. Мы стремимся поставлять полностью рабочее программное обеспечение каждые несколько недель, в крайнем случае — каждые несколько месяцев. Чем чаще, тем лучше.
4. Наиболее эффективный и действенный способ передачи информации — это встреча членов команды разработки ПО.
5. Представители бизнеса и команда разработки должны работать над проектом вместе.
6. Проекты строятся вокруг мотивированных людей. Создайте для них подходящую окружающую среду, снабдите всем необходимым и доверьте сделать свою работу.
7. Рабочее программное обеспечение — это главная мера прогресса проекта.

8. Гибкие процессы способствуют непрерывному развитию. Спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп работы в течение неопределенного срока.
9. Постоянное внимание к техническому совершенству и качественной архитектуре способствует гибкости.
10. Простота — это искусство не делать лишней работы.
11. Лучшая архитектура, требования и дизайн создаются в самоорганизующихся командах.
12. Команда постоянно ищет способы стать более эффективной путем настройки и коррекции своих действий*.

Клиент всегда прав, не так ли?

Вернитесь к началу главы и перечитайте цитату. О чем говорит Генри Форд? О том, что людям надо предлагать то, что им действительно нужно, а вовсе не то, что они просят. У клиента есть потребность, и если вы создаете ПО, чтобы ее удовлетворить, то должны понять, способен ли он донести ее до вас. Как вы работаете с клиентом, который в начале проекта еще не знает, что ему нужна машина, а не очень быстрые лошади?

Это мотивация, выходящая за рамки 12 принципов: необходимо собрать команду, способную создать такое программное обеспечение, в котором пользователь действительно нуждается. Принципы зависят от идеи, которую мы закладываем при разработке проекта с целью получения некой ценности. Но понятие «ценность» сложное, потому что каждый понимает ценность программного обеспечения по-своему: люди ставят перед ПО совершенно разные задачи.

У вас в руках хороший пример воплощения этой идеи. Если вы читаете эту книгу на портативном устройстве для чтения электронной литературы, то значит, вы используете программное обеспечение, написанное для отображения книг на этом устройстве. Потратьте немного времени, чтобы подумать обо всех стейкхолдерах (заинтересованных лицах) проекта создания программного обеспечения для электронной книги.

- Как читатель вы хотите, чтобы программное обеспечение сделало книгу легкой для чтения. Вы заботитесь о том, чтобы можно было легко переворачивать страницы, выделять разделы, делать заметки, искать нужный текст и находить страницу, на которой остановились.

* Источник: agilemanifesto.org/principles.html (по состоянию на июнь 2014 г.).

- Как авторы мы очень заботимся о том, чтобы слова, которые мы пишем, отображались корректно, пункты маркированного списка с отступом располагались так, чтобы вам было удобно его читать, а также чтобы вам было легко перемещать взгляд между основным текстом и сносками, а общее впечатление позволило получить удовольствие и знания от прочитанного.
- Нашего издателя O'Reilly волнует легкий способ дистрибуции этой книги и простота сбора позитивных читательских отзывов, а также продажи других книг.
- Продавец или розничный торговец, продавший вам эту книгу, хочет иметь возможность легко просматривать и покупать другую литературу, пригодную для торговли, быстро и легко скачивать ее для своих читателей.

Вы, наверное, сможете назвать еще больше стейкхолдеров и целей, важных для них. Каждая из целей представляет собою ценность для заинтересованной стороны.

Самые первые электронные книги не предлагали всех этих ценностей. Потребовалось много времени, прежде чем появилось программное обеспечение, способное удовлетворить эти требования читателей. И можно сказать с уверенностью, что ПО будет все лучше и лучше, потому что команды, работающие над его созданием, открывают для себя новые способы, позволяющие придавать ему новую ценность.

Легко понять ценность электронной книги задним числом. Гораздо труднее осознать ее на старте проекта. Давайте проведем мысленный эксперимент, чтобы исследовать это. Рассмотрим такой вопрос: как гипотетический читатель может узнать, что разработка программного обеспечения проводилась с использованием водопадного подхода?

«ДЕЛАЙ ТАК, КАК Я ГОВОРЮ, А НЕ ТАК, КАК ГОВОРИЛ»

Представьте, что вы в команде разработки самой первой портативной электронной книги. Техническая команда создала прототип устройства, которое имеет USB-порт для загрузки электронных книг и маленькую клавиатуру, позволяющую взаимодействовать с ним. Перед вашей командой стоит задача решить, какое программное обеспечение вы будете поставлять для этой электронной книги.

К сожалению, ваша компания имеет долгую историю создания ПО с использованием неэффективной, требующей подробной спецификации

водопадной методологии. Поэтому первым делом ваш менеджер проекта организует большую встречу с участием всех, кого сможет найти. Ваша команда проводит несколько недель в переговорной комнате, встречаясь с топ-менеджерами компании, представителями издательства, которое хочет издавать электронные книги, ведущим менеджером по продажам из онлайн-магазина, желающего их продавать, и другими стейкхолдерами, которых нашел менеджер проекта и смог привести на встречу.

После нескольких дней интенсивных дискуссий ваши бизнес-аналитики смогли собрать воедино большую спецификацию с требованиями различных заинтересованных сторон, участвовавших в опросе. Была проделана тяжелая работа, но теперь у вас есть спецификации и все этому рады. Определен обширный пользовательский функционал, позволяющий предположить, что это будет самое передовое портативное устройство из всех доступных читателю. Появились функции сбора маркетинговой статистики для издателей, способные охватить все интернет-магазины и облегчающие процесс покупки книг. Есть инновационные функции для авторов с возможностью просматривать и редактировать свои работы, упрощая процесс публикации. Иными словами, предполагается поистине революционное программное обеспечение. После проведения всех подсчетов команда приходит к 15-месячному графику работ. Это кажется большим сроком, но все полны энтузиазма и уверены, что смогут довести дело до конца.

Перенесемся на полтора года вперед. Команда разработчиков электронной книги трудилась невероятно упорно, в том числе по ночам и в выходные дни, в результате чего распалось несколько семей. Такое колоссальное напряжение позволило выполнить проект точно по плану, практически день в день. (Это кажется невероятным! Но ради нашего воображаемого эксперимента попробуйте поверить, что так и произошло.) Каждое требование в спецификации реализовано, протестировано и исполнено. Команда очень гордится собой, и все заинтересованные стороны согласны, что получили именно то, что просили.

Итак, продукт выходит на рынок... и его ждет провал. Никто не покупает книгу, стейкхолдеры недовольны. Что же произошло?

Оказывается, ПО, актуальное всего полтора года назад, сегодня никому не нужно. За это время в отрасли произошли изменения, появились новые стандарты и отраслевой формат для электронных книг, которые не вошли в созданную ранее спецификацию. И теперь никто из интернет-продавцов не хочет размещать на своих ресурсах нестандартный формат. Команда создала отличный шаблон для интернет-магазина, но его интерфейс сильно проигрывает шаблонам, которые используют розничные продавцы в настоящее время. Поэтому он никого не привлекает.

Труд и усилия вашей команды по созданию специальной функции предварительного просмотра для авторов потрачены впустую, потому что эта функция уступает опции, разработанной конкурентами и позволяющей поддерживать документы MS Word и отправлять их по почте.

Какая неприятность! Спецификация, разработанная вашей командой в самом начале, имела большое значение для клиентов и была важна для всех как внутри компании, так и за ее пределами. Но теперь ПО, задуманное полтора года назад, потеряло свою ценность. Необходимость внесения некоторых изменений можно было обнаружить на ранних этапах проекта, но многое требовалось поменять с самого начала. Необходимая при водопадном подходе «подробная спецификация перед началом работы» лишает команду гибкости и не позволяет реагировать на изменения.

Так как же все-таки суметь удовлетворить потребности стейкхолдеров и запустить проект создания работающего программного обеспечения?

Реализация проекта

Agile-команды признают, что наиболее важная задача для них — предоставление работающего программного обеспечения для своих клиентов. Вы уже знаете из главы 2, как они делают это: работая вместе как команда, сотрудничая со своими клиентами и реагируя на изменения. Но как это воплотить в повседневной работе команды?

Чтобы что-то изменить, прежде всего необходима частая поставка ценностей, восприятие каждого изменения в качестве положительного стимула для проекта, частая поставка ПО, совместная работа команды и клиента. Программное обеспечение, которое разрабатывает команда, может быть не таким, каким планировалось изначально, *но это хорошо* — потому что в итоге она создаст такое ПО, в котором клиенты больше нуждаются.

ПРИНЦИП № 1. НАИВЫСШИМ ПРИОРИТЕТОМ ДЛЯ НАС ЯВЛЯЕТСЯ УДОВЛЕТВОРЕНИЕ ПОТРЕБНОСТЕЙ ЗАКАЗЧИКА БЛАГОДАРЯ РЕГУЛЯРНОЙ И РАННЕЙ ПОСТАВКЕ ЦЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Первый принцип включает в себя три важные идеи: ранний выпуск программного продукта, непрерывная реализация ценности и удовлетворение клиента. Чтобы понять суть этого принципа, нужно знать, как эти три идеи работают вместе.

Проектные группы существуют в реальном мире, в котором не бывает ничего совершенного. Даже блестяще работающей команде при сборе и записи требований к проекту будет чего-нибудь не хватать, потому что невозможно полностью и точно отразить требования для каждой системы. Это не означает, что команды не должны стараться, а agile-методологии основываются на беспрецедентных методах коммуникации и фиксации требований. Но дело в том, что, пока клиенты не получают в руки работающее ПО, им трудно представить, как именно оно будет функционировать.

Так что если вы хотите получать от клиента реальную, обоснованную обратную связь после просмотра рабочего программного продукта, то лучше всего делать это через **раннюю поставку**: «отгрузить» заказчику первую рабочую версию программного обеспечения как можно раньше. Даже если вы реализовали только одну рабочую функцию, которую клиент может использовать, — это уже победа для команды. Потому что теперь потребитель способен предоставить вам обратную связь, и она поможет двигать проект в нужном направлении. Это также победа для заказчиков, потому что, имея эту рабочую программу, они могут реализовать то, о чем раньше только мечтали. Можно сказать, что команда создала реальную ценность, поскольку ПО работает и клиенты могут его использовать. Пусть это лишь небольшая часть того, в чем нуждается потребитель, но все-таки это «лучше-чем-ничего», особенно если альтернатива этому — расстроенный пользователь, который должен долго ждать, прежде чем получит программный продукт.

Недостаток ранней поставки в том, что программное обеспечение далеко от совершенства. Это представляет сложность для некоторых клиентов и заинтересованных сторон: если одни пользователи привыкли к мысли, что увидят программное обеспечение рано, то другим свыкнуться с этой мыслью труднее. Многие люди очень переживают, когда их софт не идеален. В компаниях (особенно крупных, где затрачиваются годы на работу с командами, разрабатывающими программное обеспечение) эти команды должны очень тщательно оговаривать условия поставки ПО заинтересованным сторонам. При отсутствии партнерских отношений между заказчиком ПО и командой-разработчиком поставляемый неполный программный продукт может быть оценен строго и даже привести пользователя в ужас, если действительно не соответствует ожиданиям.

Основные agile-ценности отвечают на этот вопрос: сотрудничество с заказчиком важнее согласования условий контракта. Команда, жестко привязанная к спецификации, не может вносить изменения в программное обеспечение в процессе его создания. Работая в таких условиях, требуется запустить новый процесс тотального управления изменениями, который потребует новых переговоров по контракту с заказчиком. Зато команда,

действительно сотрудничающая с клиентами, имеет возможность внести все необходимые изменения в ходе работ. Вот что означает **непрерывная поставка**.

Именно поэтому гибкие методологии, как правило, итеративны. Agile-команды планируют итерации проекта путем выбора показателей и требований, которые обеспечивают максимальную отдачу. Единственный способ, при помощи которого команда может выяснить, какие показатели реализуют эту ценность, — сотрудничество с клиентом и встраивание обратной связи от предыдущей итерации. Это позволяет команде удовлетворить запросы потребителя в краткосрочной перспективе, демонстрируя ценность продукта на раннем этапе сотрудничества, а в долгосрочной перспективе предоставить ему готовый продукт, обладающий всеми возможными ценными качествами.

ПРИНЦИП № 2. ИЗМЕНЕНИЕ ТРЕБОВАНИЙ ПРИВЕТСТВУЕТСЯ ДАЖЕ НА ПОЗДНИХ СТАДИЯХ РАЗРАБОТКИ. AGILE-ПРОЦЕССЫ ПОЗВОЛЯЮТ ИСПОЛЬЗОВАТЬ ИЗМЕНЕНИЯ ДЛЯ ОБЕСПЕЧЕНИЯ ЗАКАЗЧИКУ КОНКУРЕНТНОГО ПРЕИМУЩЕСТВА

Многие успешные agile-практики, впервые сталкиваясь с этим принципом, погружаются в проблемы. Легко рассуждать о готовности к изменениям. Но в разгар проекта, когда команда работает над изменениями, требующими больших трудозатрат, эмоции бывают на пике, особенно у разработчика, который знает, что руководство не изменит сроков независимо от того, каких усилий требуют эти изменения. Очень сложно согласиться с этим, особенно если в ходу критика за нарушение сроков. Но это бывает и полезно, потому что, приветствуя изменения в требованиях, можно овладеть одним из самых мощных agile-инструментов.

Почему изменения в проекте несут эмоциональную окраску? Понимание этого вопроса — ключ к данному принципу. Вспомните, когда в последний раз вы, работая над проектом, узнали, что нужно внести изменения в уже созданное. Что вы чувствовали? До этого момента казалось, что дела идут хорошо. Вы, наверное, многое успели обдумать — например, как структурировать работу, что создавать и что обещать клиентам. И вдруг кто-то, не участвующий в проекте, заявляет, что вы ошибались — и планирование, и работа были сделаны неправильно.

Такое трудно принять, особенно если вы делали работу именно для этого человека. Почти всеми инженерами-программистами движет чувство *гордыни великого мастера*: они хотят поставлять продукты, которые могут

поддерживать и которые удовлетворяют потребности пользователей. Изменения в проекте угрожают этому, потому что ставят под сомнение тот путь, который они выбрали, и те допущения, которые они сделали.

Очень часто вам сообщают, что нужно изменить курс, после того как вы его выбрали.

Если этот человек уже объяснил, что именно нужно делать, и вы наполовину выполнили его требования, то вас очень огорчит его неожиданное заявление: «Я подумал вот о чем — а не можем ли мы создать что-то совсем другое?» Он не уважает ваш труд. Теперь вам придется внести изменения в уже сделанное. Трудно при этом не испытать возмущения! И хуже всего приходится тому, кто несет ответственность за результат и не соблюдает сроки.

Почти каждый профессиональный разработчик хотя бы раз попадал в подобную ситуацию. Поэтому понятно, что мы не можем заставить себя приветствовать изменяющиеся требования.

Первый шаг навстречу меняющимся требованиям — попытаться посмотреть на ситуацию с точки зрения клиента. Это не всегда легко, но поучительно. Как вы думаете, заказчик, направивший вас по ложному пути, хотел этого? Представляете, что творилось у него в душе, когда он понял, что несколько месяцев назад распорядился создать неправильное ПО и из-за этого вы потратили массу времени на выполнение ненужной работы? Он приходит к вам с просьбой изменить это. Кроме того, он признает свою ошибку, которая, к сожалению, будет стоить вам дополнительной работы. Это нелегко сделать. Неудивительно, что часто клиенты подолгу не решаются прийти к команде и попросить об изменениях! Им стыдно и к тому же не хочется приносить плохие новости. Все сроки будут сорваны, ну и пусть! Если у заказчика есть потребности, а компания готова тратить деньги на разработку ПО, удовлетворяющего их, то в случае их неудовлетворения проект не будет иметь ценности. И это вина заказчика, потому что именно он ошибся в начале проекта.

Другими словами, двум людям нужно сделать невозможное. Вы должны прочесть мысли заказчика. А он — предсказать будущее. Когда вы смотрите на вещи с таких позиций, вам намного проще принимать изменения. Но если вы хотите противостоять изменению проекта и упорно придерживаться изначального плана, то выход тоже есть: просто убедитесь, что все члены вашей команды ясновидящие.

Так что же такое «приветствовать изменения»? Это значит:

— Никто не попадет в «беду», когда есть изменение. Мы и руководство компании признаем, что мы люди и можем ошибаться. Гораздо

разумнее позволить нам допускать оплошности, а затем исправлять их, чем ждать, что мы сделаем все идеально с первого раза.

- Мы все в одной лодке. Каждый член команды, в том числе клиенты, с которыми мы сотрудничаем, имеет требования к проекту и меняет их. Если требования неправильные, то это наша общая с клиентом ошибка, поэтому нет смысла искать виноватого.
- Мы принимаем решение об изменениях быстро, не дожидаясь, когда будет слишком поздно. Ошибаться, конечно, плохо. Но мы все признаем ошибку, поэтому делаем все, что в наших силах, чтобы исправить ее как можно раньше. То есть стараемся максимально ограничить ущерб.
- Мы перестаем думать об изменениях как об ошибках. На основании имевшейся на тот момент информации мы делали все возможное, и только теперь стало ясно, что мы были неправы. Благодаря принятым решениям мы увидели перспективу и признали необходимость в изменениях, которые нужно сделать сегодня.
- Мы учимся на изменениях. Это наиболее эффективный способ командного роста и строительства лучшего ПО.

ПРИНЦИП № 3. РАБОТАЮЩИЙ ПРОДУКТ СЛЕДУЕТ ВЫПУСКАТЬ КАК МОЖНО ЧАЩЕ, С ПЕРИОДИЧНОСТЬЮ ОТ ПАРЫ НЕДЕЛЬ ДО ПАРЫ МЕСЯЦЕВ

Возможно, вы подумали, что идея приветствовать изменения требований интересна и может помочь проектам или, наоборот, что она ужасна. Это довольно распространенная реакция. Многие из тех, кто работает в команде разработчиков программного обеспечения (особенно традиционные руководители проектов), чувствуют себя неуютно, впервые узнав об этом.

Эти менеджеры ежедневно сталкиваются с изменениями, но гибкие методологии предполагают совершенно иное решение. В agile-практиках традиционное отношение к изменениям проекта считается **командно-административным** подходом.

Термин «командно-административный» заимствован из военной терминологии. В уже упоминавшейся книге «Идеальные команды. Вдохновляющие и предостерегающие рассказы ветеранов тимлидинга», выпущенной в 2010 году, есть интервью Нила Сигела, главного инженера компании Northrop Grumman, который дал определение этому термину:

Эндрю. Я не знаком с военными системами — что такое командно-административная система?

Нил. Это информационная система военных командиров. Она позволяет им общаться друг с другом и поддерживать ситуационную осведомленность: знать расположение войск и их статус. Благодаря этому командир понимает, что происходит. Раньше сражения проходили на небольших площадях и командир мог стоять на холме с биноклем и наблюдать за происходящим. Но примерно к 1900 году поля сражений стали настолько большими, что вы уже не можете стоять на холме, как это делал Наполеон. Вы должны начать использовать технологию, чтобы «видеть» все поле боя. И системы, которые делают это, называются командно-административными.

Командно-административное управление проектом похоже на военную систему.

- Слово «командное» имеет отношение к тому, как руководитель проекта распределяет работу в команде. Команда может не отчитываться перед руководителем, но он контролирует выполнение всех своих распоряжений. Он делит работу на части, создает график выполнения задач, распределяет их и ресурсы внутри команды.
- Слово «административное» описывает способ, при помощи которого руководитель управляет изменениями. Каждый проект сталкивается с изменениями: работа занимает больше времени, чем ожидалось, люди болеют или покидают проект, оборудование выходит из строя и т. д. Руководители проектов постоянно следят за этими изменениями и управляют проектом, оценивая каждое изменение, обновляя планы, включая изменения в графики работ, производя назначения в команде, а также управляя ожиданиями всех заинтересованных сторон, чтобы исключить неожиданности.

Причина, по которой традиционный менеджер проекта испытывает неудобства, впервые столкнувшись с изменениями, в том, что те же самые проблемы будут происходить и в agile-проектах, и команда должна быть в состоянии отвечать на них. Простое принятие изменений выглядит как гарантированный способ хаотизировать проект. Если agile-команды не используют командно-административный подход, то как они умудряются справляться со всеми этими изменениями, одновременно решая и все те проблемы, с которыми проектные команды сталкиваются ежедневно?

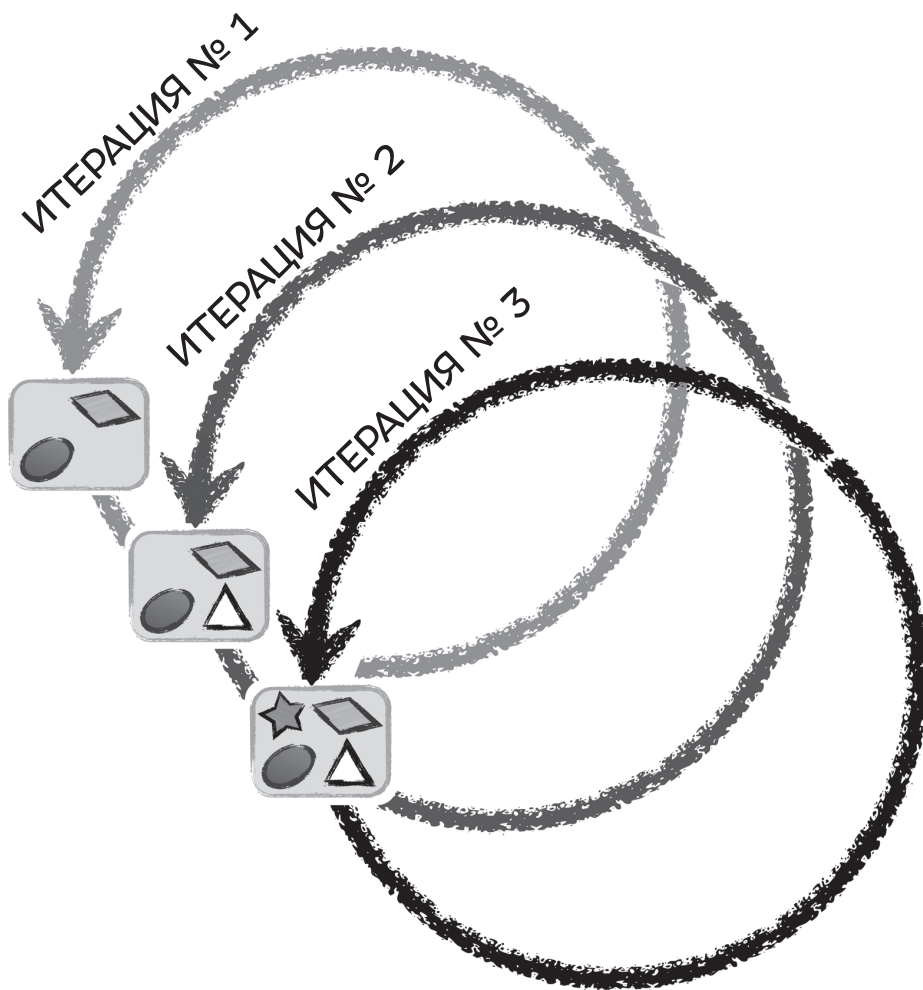


Рис. 3.1. Команда использует итерации для частой доставки работающего программного обеспечения, добавления новых функций с каждым релизом

Ключ к одобрению изменений без внесения хаоса заключается в частой поставке рабочей программы. Команда использует итерации, чтобы разбить проект на части с регулярными сроками сдачи. Во время итераций команда предоставляет рабочее ПО. По окончании каждой итерации команда проводит демонстрацию, показывая клиенту созданный продукт, а также предыдущие варианты, чтобы посмотреть, какие уроки можно извлечь из данной ситуации. Затем начинают сеанс планирования, чтобы выяснить, что они будут создавать в следующей итерации. Предсказуемый

график и постоянные точки контроля помогают команде отследить эти изменения на ранних стадиях и создают атмосферу, в которой не принято искать виноватого, когда каждый может обсудить изменения и придумать стратегию, чтобы включить ее в проект.

В этот момент Agile становится привлекательной для традиционного командно-административного менеджера проекта. Такой менеджер хочет контролировать срок. Установка ограничений на продолжительность итераций дает ему эту возможность. Кроме того, решается также одна из главных задач менеджера — работа с изменениями, которые возникают в самом конце проекта. Один из наиболее сложных моментов в работе традиционного менеджера — мониторинг изменений. Ежедневные отчеты и ретроспективы итераций позволяют руководителю проекта заручиться поддержкой всех членов команды, ведь они теперь глаза и уши руководителя, помогающие обнаружить необходимость изменений, прежде чем они станут причиной более серьезных проблем в проекте.

Роль менеджера проекта в том, чтобы инициировать замену командно-административной системы управления, при которой ему необходимо ежедневно давать задания членам команды и постоянно корректировать план работ, чтобы направить их в нужное русло. Теперь менеджер взаимодействует с командой лишь для того, чтобы убедиться: каждый специалист видит единую картину и трудится над общими целями. Проще всего это выполнить, когда команда работает в режиме коротких итераций, позволяющих поставлять рабочее программное обеспечение. Это ставит перед каждым участником конкретные цели и дает ему исчерпывающее представление о том, над чем работает команда, а также чувство ответственности не только за свою работу, но и за общий результат по окончании итерации.

ЛУЧШАЯ ПОСТАВКА ДЛЯ ПРОЕКТА «ЭЛЕКТРОННАЯ КНИГА»

Как эти принципы могут помочь нашему проекту «Электронная книга»? Вспомним о проблемах, с которыми столкнулась команда: их проект провалился, потому что отсутствовали важные функции, которые были у конкурентов (поддержка отраслевого стандартного формата для электронных книг и предоставление пользователям доступа к почте на этом устройстве), а функции оказались неактуальными (шаблон для создания интернет-магазина).

Давайте снова запустим проект, но на этот раз менеджер разделит работу команды и стейкхолдеров на месячные спринты. Ситуация сразу стала иной.

— После третьего спринта один из разработчиков сообщил, что был утвержден новый формат отраслевого стандарта для электронных книг.

Команда решила реализовать библиотеку, которую согласилась разрабатывать на четвертом спринте, а на пятом включила ее в пользовательский интерфейс читателя.

- После десяти месяцев разработки они произвели сборку программы, которую можно было загружать в прототип электронного устройства и раздавать бета-версии первым пользователям. Руководитель проекта, поговорив с ними, обнаружил, что они действительно хотели бы иметь доступ к документам Microsoft Word и газетным статьям в своих электронных книгах. Команда выделила часть следующего спринта на разработку почтового интегратора для электронной книги, чтобы пользователи могли рассылать статьи со своих устройств.
- Год активного участия стейкхолдеров в проекте позволил команде узнать, что шаблон для интернет-магазина не нужен, потому что розничные продавцы используют стандартный формат электронной книги. К счастью, на разработку этой функции ушло не так много времени, потому что нужно было заниматься более важными вещами.

Поскольку команда всегда поставляла рабочее программное обеспечение после каждого спринта, удаление функции из списка невыполненных работ означало, что они могли завершить работу раньше! Партнер-издатель был уже с книгами, потому что все его старшие руководители получили ранние версии ПО и прототипы оборудования для его использования. Это поддержало их вовлеченность и простимулировало подготовку к получению книги к моменту выхода первой версии продукта.

Используя непрерывные релизы, приветствуя изменения и предоставляя рабочее программное обеспечение после каждой итерации, команда проекта «Электронная книга» сумела поставить успешный продукт гораздо раньше. В отличие от неэффективного водопадного подхода, в котором утвержденные в начале проекта требования отгородили команду от клиентов, agile-методология позволила команде все время взаимодействовать с клиентами. Это дало возможность реагировать на изменения и создать лучший продукт.

Но не все идеально в команде проекта «Электронная книга» — она еще далека от совершенства. Разработчики используют итерации для поставки рабочего программного обеспечения, но они завалены документацией. Все действительно счастливы, что не застряли над созданием ПО, которое не будет продаваться. Но каждый раз, когда команда находит хорошие изменения, которые нужно внести в проект, половина ее участников застревает, возвращаясь к спецификации, и обновляет ее, чтобы их планы стали актуальными и соответствовали выбранному курсу. Кажется, что они тратят столько же усилий на обновление документации, сколько и на написание кода.

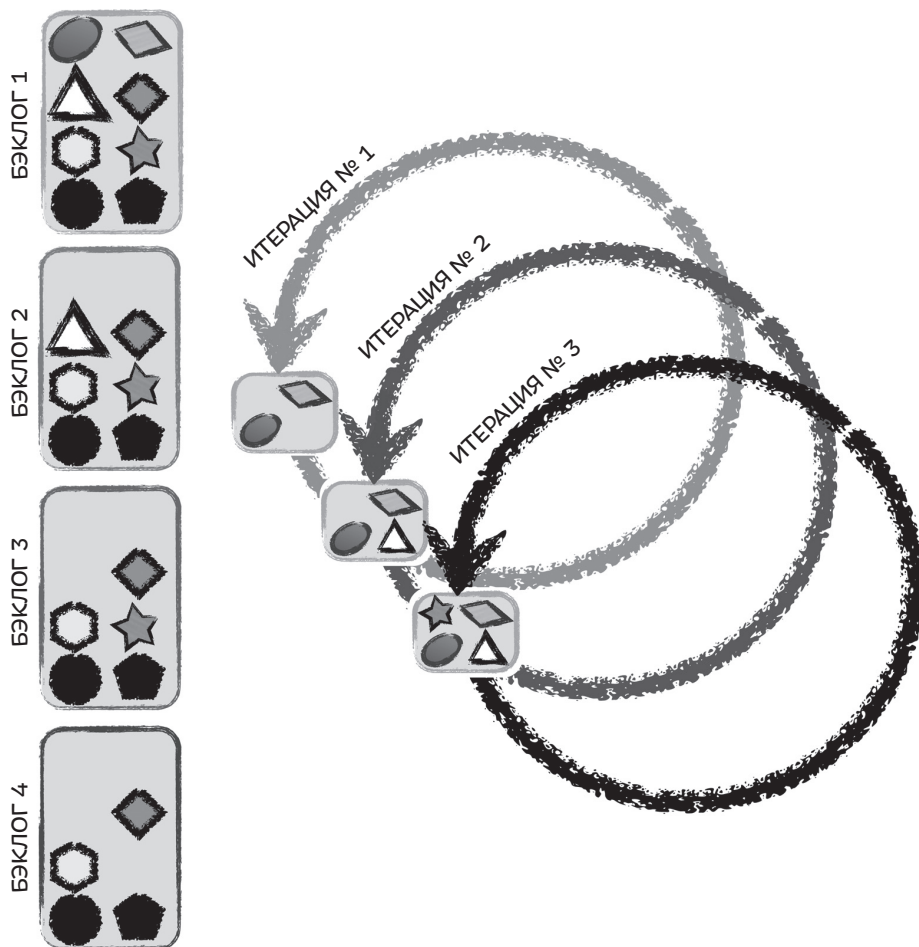


Рис. 3.2. В начале каждой итерации команда выбирает функции для реализации из списка невыполненных работ

В команде уже обсуждались способы сокращения объема работ, необходимых для сохранения этих документов. Состоялись обстоятельные дискуссии об «уровне детализации» документации. Но всякий раз при попытке что-либо убрать кто-нибудь приводит «вескую» причину, почему это не стоит делать. Ведь если не создать описание этой конкретной функции, требования, дизайна или примера, то кто-нибудь неправильно поймет документ. А если он будет неправильно реализован, то появится повод предъявить претензии. Со стороны кажется, будто каждый фрагмент документации необходим, потому что в противном случае над командой нависнет опасность создать плохое программное обеспечение.

Есть ли способ избежать этой опасности без вреда для проекта? Существует ли «правильный» уровень степени документированности проекта?



Ключевые моменты

12 принципов гибкой разработки, которые сопровождают Agile-манифест, укажут agile-практикам направление, в котором надо работать, и дадут понимание методов и методологий.

Agile-команды удовлетворяют своих клиентов, получая обратную связь на ранних этапах проекта, и непрерывно поставляют программное обеспечение, чтобы сохранить эту связь актуальной (принцип № 1).

Agile-команды одобряют изменения, рассматривая их как положительные и полезные события для проекта (принцип № 2).

Используя ограниченную продолжительность итераций при частой поставке работающего программного обеспечения, agile-команды постоянно корректируют проект, поэтому он обеспечивает наибольшую ценность для потребителя (принцип № 3).

Общение и совместная работа

Команды-разработчики задавали вопрос, сколько документации необходимо на протяжении всего времени разработки программного обеспечения. Многие команды годами искали методологию «серебряной пули» для улучшения процессов, программирования и решения проблем с поставкой и подобный волшебный шаблон для создания документации, позволяющий записывать все необходимое для создания программного обеспечения сегодня и сохранения его в будущем.

Традиционное представление о программной документации выглядит примерно так: нужна система документооборота, в которой каждый может разместить имеющиеся у него сведения, а она должна связать эти сведения с остальной информацией. Если у нас появится возможность ее отслеживать, то мы сможем установить связи между всеми данными, что даст почти идеальный обзор того, что мы создаем, как будем тестировать, разворачивать и поддерживать ПО. Идея заключается в том, что разработчики могут отследить каждый элемент дизайна по сравнению с изначальными требованиями, а тестировщики — результаты каждого теста относительно первоначальных элементов конструкции, требований и области применения. Всякий раз, когда команда должна изменить, скажем, часть конструкции,

она имеет возможность точно видеть, какой код, требования, область и тестовые примеры будут затронуты, поэтому не нужно тратить массу времени на выполнение анализа. Программисты называют это *импакт-анализом* (анализом воздействия) и часто стараются поддерживать в актуальном состоянии подробную таблицу всех обнаруженных неисправностей в каждой области, требованиях, дизайне и тестах.

Итак, вернемся к вопросу о том, сколько документов должны создавать команды. Ответ для agile-команды — ровно столько, сколько нужно, чтобы создать проект. Конкретное количество зависит от команды, налаженности коммуникации и того, какие проблемы она решает. Что представляет собой необходимая программистам документация? Это фиксация всех решений, принятых на собраниях с учетом большого количества примечаний, создание перекрестных ссылок на документы, описывающих каждый аспект каждого источника данных или хранилища, сложная матрица подробных функций, обязанностей и правил, которыми каждый должен руководствоваться. Образец должен быть сделан для всех видов документации. Если все это не помогает создавать программное обеспечение или не требуется по другим причинам (например, для регламентирующего органа, инвестора, менеджера высшего звена или других стейкхолдеров), то agile-команда может не создавать такую документацию. Но та команда, которая считает, что функциональная документация *действительно* поможет, может сделать ее и при этом остаться гибкой.

Все зависит от того, что больше подходит данной команде, и это одна из свобод, которую предоставляет Agile. Развернутая документация, матрица трасировок, импакт-анализ — вся эта дополнительная работа нужна в основном для того, чтобы полностью проанализировать влияние каких-либо изменений. Agile-методологии имеют другой (часто гораздо более эффективный) подход к управлению изменениями в проекте. Agile-практики признают, что традиционный подход к документации соответствует их целям. В традиционном водопадном проекте весь смысл исчерпывающей документации заключается в том, чтобы внести в проект наиболее удачные изменения.

Как ни странно, именно исчерпывающая документация чаще всего становится преградой на пути управления изменениями. Мечтать о совершенной документации и отслеживании изменений в ней — значит иметь такую систему, в которой команда может автоматически фиксировать все производимые в продукте изменения, поэтому команда может точно определить, что должно быть исправлено.

Для большинства команд, к сожалению, реальность управления изменениями при помощи исчерпывающей документации не столь привлекательна. Им придется потратить огромное количество времени в начале проекта,

пытаясь предсказать будущее и безупречно записать его. В ходе проекта они должны сохранять документы, которые написали, и отслеживать любые новые события. Если есть изменения в понимании сути продукта, который они создают, то им придется вернуться и внести изменения во все документы, имеющие к этому отношение. Со временем это может привести к беспорядку в устаревших документах, а также потребует значительных усилий для их создания и поддержания.

Из-за того, что исчерпывающая документация неидеальна, это приводит к ненужным изменениям и затраченным впустую усилиям. Любой фрагмент документации написан с точки зрения человека, имеющего определенную роль: бизнес-аналитики делают это с одних позиций, архитекторы, создающие дизайн, — с других, а тестировщики (QA-инженеры), строящие планы тестирования, — с третьих. Когда нужно связать это воедино с матрицей отслеживания, они обнаруживают точно такие же несоответствия, которые вы ожидаете от внедрения раздробленного видения. Создание документации становится самоцелью и требует все нарастающих предварительных усилий. И наконец, когда понадобится изменение, всю работу по согласованию их точек зрения и объединению документов в обширный комплекс придется переделывать. Это ведет к переписыванию командой документации, формированию заново матрицы трассируемости и улаживанию конфликтов. Хотя на самом деле требуется написать код, протестировать ПО и внедрить изменения.

Должен быть более эффективный способ разработки программного обеспечения.

ПРИНЦИП № 4. НЕПОСРЕДСТВЕННОЕ ОБЩЕНИЕ ЯВЛЯЕТСЯ НАИБОЛЕЕ ПРАКТИЧНЫМ И ЭФФЕКТИВНЫМ СПОСОБОМ ОБМЕНА ИНФОРМАЦИЕЙ КАК С САМОЙ КОМАНДОЙ, ТАК И ВНУТРИ КОМАНДЫ

Agile-практики признают, что документация — это просто одна из форм общения*. Когда я пишу документ и передаю его вам, *цель не в написании документации*. Моя задача состоит в том, чтобы убедиться: мои мысли почти полностью соответствуют вашим. Во многих случаях документиро-

* Традиционные руководители проектов также признают, что документация — это просто одна из форм общения. Типичные менеджеры изучают разницу между формальным и неформальным способами общения, а также разницу между письменной и устной коммуникацией. Они также изучают невербальные сигналы в общении и узнают, что общение — наиболее эффективное средство генерации идей. Все эти знания фактически проверяются на экзамене по RMP!

вание — отличное средство для достижения этой цели, но не единственное из имеющихся у нас.



Рис. 3.3. Когда члены команды не общаются, они могут о чем-то договариваться, но все-таки цели их работы разные. Исчерпывающая документация может ухудшить ситуацию и легко привести к ее неоднозначному пониманию

Непосредственное общение — это почти всегда более предпочтительное средство для обмена идеями внутри команды разработчиков, чем документация. Известно, что совместное обсуждение проблемы — самый эффективный способ понять новую идею. У нас гораздо больше шансов запомнить идеи, высказанные во время разговора, чем почерпнутые из книги или электронного носителя. Вот почему практики agile-общения в основном сосредоточены на людях, непосредственно общающихся друг с другом и использующих документацию только в тех случаях, когда позднее необходимо детализировать сложную информацию.

К счастью, это несложно для команд, которые используют исчерпывающую документацию для более эффективной личной коммуникации своих членов. Потому что большинство из них не стремятся достичь «идеала» в детализации информации и трассируемости. Разработчики программного

обеспечения, как известно, практичны. Когда они видят весь объем работ, необходимых для создания исчерпывающей информации, они идут по пути разговоров друг с другом. Это единственный надежный способ эффективного построения ПО. Признав, что исчерпывающая документация не всегда уместна, они перестанут чувствовать вину за то, что не совершили невозможного — не создали идеальной, подробной документации. Ведь если бы они это сделали, польза для их проектов была бы невелика!

У команды есть более эффективный способ донести важные для проекта идеи — нужно сделать так, чтобы все думали одинаково. Таким образом, каждый может участвовать в принятии предложенного решения. Когда группа людей смотрит на ситуацию с одинаковых позиций и откровенно обсуждает как ее положительные, так и отрицательные стороны, она в итоге придет к единой точке зрения. Поэтому в процессе изменений, требующих переосмысления, члены команды уже не будут затрачивать массу усилий, чтобы что-то объяснить друг другу.

Конечная цель, ради которой команда коммуницирует, — это создание чувства общности, так как это подразумевает знание, потому что неэффективно снова и снова объяснять одно и то же. Без чувства общности люди, исполняющие разные роли, будут вынуждены работать гораздо больше, чтобы их видение совпадало. Чем выше в команде чувство общности, чем ближе по взглядам ее члены, тем чаще каждый будет самостоятельно приходить к аналогичному ответу, когда начнет сталкиваться с возникавшим ранее вопросом. Это дает стабильную почву для управления изменениями, потому что конфликты остаются в прошлом и можно с уверенностью начать работу над кодом, причем не придется отвлекаться на управление документацией.

ПРИНЦИП № 5. НА ПРОТЯЖЕНИИ ВСЕГО ПРОЕКТА РАЗРАБОТЧИКИ И ПРЕДСТАВИТЕЛИ БИЗНЕСА ДОЛЖНЫ ЕЖЕДНЕВНО РАБОТАТЬ ВМЕСТЕ

Agile-команды иногда забывают, что бизнесмены завалены работой. Это приводит к естественной потере связи между командами-разработчиками и предпринимателями, для которых они создают программное обеспечение.

Для создания хорошего программного продукта команда должна часто встречаться с предпринимателями, потому что ей необходимо получить доступ к знаниям, нужным для решения проблем при помощи ПО. Предприниматели получили знания, решая те же проблемы без программного обеспечения. Поэтому для большинства бизнесменов, которые нуждаются в услугах программистов, разработка ПО — это лишь небольшая часть их

забот. Чаще всего в их планы входит всего несколько личных встреч с программистами. Разработчики встречаются с заказчиками пару раз — чтобы выяснить, какое именно программное обеспечение требуется, а затем спустя короткое время, чтобы представить идеальное рабочее ПО.

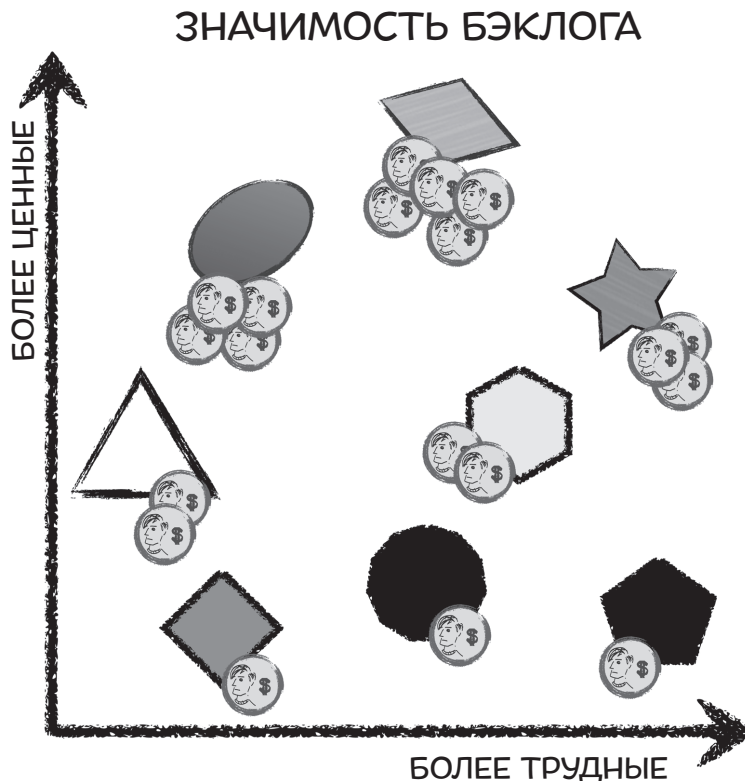


Рис. 3.4. Некоторые функции более значимы для компании, чем остальные. Команде нужно сбалансировать ценности (сколько стоит функция) по отношению к затратам (сколько потребуется работы для ее создания) при определении, какие из них следует включить в каждую итерацию

Команды между тем хотят иметь как можно больше контактов с предпринимателями. Программисты должны узнать о бизнес-задаче, которую нужно решить, как можно больше. Они делают это, разговаривая с бизнесменами, наблюдая за их работой и разглядывая произведенный ими продукт. Нуждающийся в информации программист стремится максимально завладеть вниманием заказчика. Потому что чем дольше он ждет ответов на свои вопросы, тем медленнее движется проект. Но предприниматели

не спешат тратить все свое время на команду разработчиков, потому что им нужно заниматься своими делами. Итак, кто же победит?

Agile-команды знают, как добиться того, чтобы обе стороны остались в выигрыше. Они начинают со взаимного понимания того, что команда поставляет компании ценное программное обеспечение. Готовое программное обеспечение стоит денег. Если ценность ПО превышает издержки на его создание, то компании имеет смысл инвестировать в свое развитие.

Хороший проект должен быть достаточно ценным, чтобы предприниматели убедились: затраченные усилия необходимы для дальнейшего хода проекта.

Когда бизнесмены и разработчики трудятся над созданием ПО как одна команда, в долгосрочной перспективе наиболее эффективно ежедневное сотрудничество на протяжении всего проекта. Альтернатива — это ждать окончания проекта, когда станут видны результаты работы команды, и дать обратную связь. Но внесение изменений на этом этапе стоит гораздо дороже. Каждый из участников проекта сэкономит немало времени, если отследит изменения как можно раньше. Ежедневная командная работа на протяжении всего проекта экономит личное время каждого из его участников.

Именно поэтому команды, поставляющие рабочее программное обеспечение, должны отдавать приоритет наиболее значимым функциям — чтобы предприниматели могли как можно раньше получать ценное ПО. Эта часть сделки и объясняет, почему хорошие agile-команды считают предпринимателей частью команды наравне с программистами. В этом различие между гибкой и традиционной командами.

Традиционная команда рассматривает бизнес-пользователей как сторону, с которой нужно договариваться. Agile-команда сотрудничает с клиентом (как правило, это владелец продукта) как с равным в выборе пути осуществления проекта. (Вспомните одну из основных agile-ценностей, гласящую, что сотрудничество с клиентом главнее переговоров по контрактам!)

Хороший владелец продукта поможет уменьшить количество времени, которое бизнесмены проводят с командой. Им все равно придется встречаться ежедневно, но владелец продукта сосредоточит свое внимание на понимании ценности программного обеспечения и бизнес-задачи, которую нужно решить. Таким образом, команда будет использовать личную встречу с предпринимателями для проверки информации, которую она уже узнала от владельца продукта.

ПРИНЦИП № 6. НАД ПРОЕКТОМ ДОЛЖНЫ РАБОТАТЬ МОТИВИРОВАННЫЕ ПРОФЕССИОНАЛЫ. ЧТОБЫ РАБОТА БЫЛА СДЕЛАНА, СОЗДАЙТЕ УСЛОВИЯ, ОБЕСПЕЧЬТЕ ПОДДЕРЖКУ И ПОЛНОСТЬЮ ДОВЕРЬТЕСЬ ИМ

Проекты работают лучше, когда каждый в компании признает, что команда создает значимое программное обеспечение, и ее члены, в том числе владелец продукта, понимают, что придает продукту значимость и делает его ценным для компании.

В то же время проекты могут разрушаться в среде, где не ценят ПО или где люди не вознаграждены по заслугам за разработку программного продукта. Нередко компании внедряют проверку производительности и систему компенсации, которые мешают командам двигаться по эффективному, гибкому пути создания ПО, а это может помешать проекту. Вот некоторые распространенные антистимулы, которые работают против agile-команд:

- программистам предоставляется недостаточно подробный анализ эффективности: при анализе кода регулярно повторяются ошибки и им присваивается статус «чистый» код (в результате программисты перестают искать ошибки в анализе кода);
- тестировщиков награждают за количество ошибок, которые они обнаруживают (тем самым поощряются придирки и плохая отчетность, а также нарушается сотрудничество с программистами, потому что это создает антагонистические отношения между ними);
- оценки эффективности бизнес-аналитиков базируются на количестве производимой документации (а не объеме знаний, которыми они поделились с командой).

Производительность каждого специалиста должна основываться на достижениях команды, а не на тех ролях, которые они играют в проекте. Это не означает, что программист, который плохо выполняет свои обязанности или отрицательно влияет на команду, не должен отвечать за это перед руководителем. Людей нужно оценивать по их вкладу в достижение общих целей команды. Но их *определенно* не должен обескураживать выход за рамки определенной роли. Хорошие условия работы будут стимулировать программиста, который распознает часть бизнес-задачи и решит ее, или тестировщика, увидевшего недочеты кода либо архитектуры и устранившего их. Рабочая среда дает всем участникам группы необходимую поддержку и делает проект более успешным.

Исчерпывающая документация и матрицы трассируемости — это особенно коварные источники проблем для командной среды и поддержки внутри команды. Вместо обстановки доверия они поощряют СУА-среду (cover your ass, что можно перевести как «каждый спасает свою шкуру»), когда команда движется в сторону «переговоров по контракту», а не сотрудничества с клиентом.

Тестировщики в СУА-среде прежде всего стремятся убедиться в том, что каждое условие протестировано, а не в улучшении качества ПО. Разработчики соблюдают требования буквально, не затрудняя себя мыслями о том, действительно ли они создают ценный для клиента продукт. Потому что иначе, работая в СУА-среде и пытаясь сделать то, что нужно клиенту, они рискуют получить выволочку за то, что не следуют спецификации. Бизнес-аналитики и владельцы продукта в СУА-среде заботятся прежде всего о том, чтобы сфера охвата пользователей и требования выстраивались в идеальную линию. Поэтому они нередко пренебрегают документными требованиями, которые не вполне совпадают с существующей документацией, независимо от их значимости.

Члены команды программного обеспечения должны бороться каждый за себя там, где негативно относятся к изменениям. Команде, которая использует исчерпывающую документацию, легко понять, что изменения — это плохо: они требуют слишком много работы, чтобы пересмотреть область охвата, обновить спецификации, изменить дизайн, произвести ремонт матрицы трассируемости и т. д. Это приводит к расколу среды, потому что менеджеры в такой компании обычно пытаются найти «виноватого», на которого можно возложить ответственность за все дополнительные работы, вызванные изменениями. Члены команды все чаще обращаются к «оборонительной документации», чтобы защитить себя от незаслуженных обвинений. Это заставляет их применять принцип «каждый за себя»: чтобы избежать критики и наказания, они ссылаются на документацию, которой придерживались.

Альтернатива СУА-среде — доверие. Компания, разрабатывающая только документацию, необходимую для проекта, создает среду, в которой команде доверено делать именно то, что надо, когда происходят изменения. В agile-команде с отношением «мы все вместе» в случае провала проекта все разделяют вину и никто не «прикрывает свой зад». Им проще внедрять изменения, потому что не нужно придерживаться всей этой ненужной документации. Вместо этого они могут работать лицом к лицу, решать реальные проблемы и записывать только необходимое. И они могут делать это, зная, что компания верит, что они сделают все правильно — даже если проект займет больше времени.

УЛУЧШЕНИЕ КОММУНИКАЦИИ В КОМАНДЕ ПРОЕКТА «ЭЛЕКТРОННАЯ КНИГА»

Наш проект «Электронная книга» определенно выиграл от улучшения коммуникации. Помните интенсивные встречи, на которых бизнес-аналитики совместными усилиями тщательно выводят обширный список исчерпывающих требований? Эти требования начинались вовсе не как циничная СУА-схема. Каждый в команде искренне полагал, что если он будет последовательно отстаивать свою точку зрения по поводу полноты спецификации, то это поможет создать более успешный продукт.

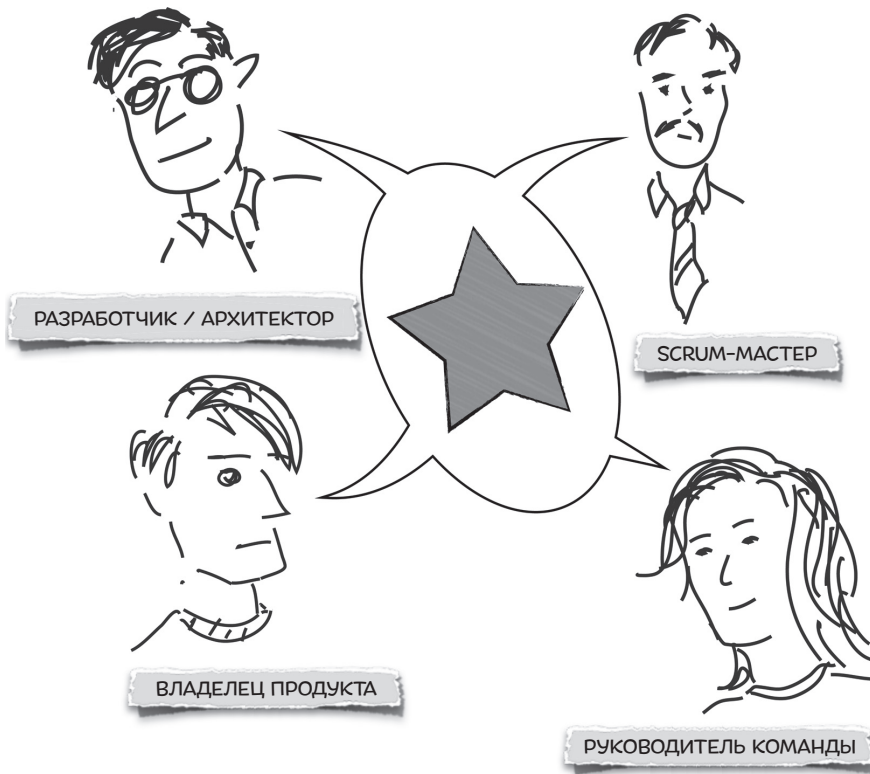


Рис. 3.5. Когда команда максимально полагается на личное общение и использует только минимальный объем документации, необходимой для создания проекта, это упрощает синхронизацию действий каждого

Из-за того, что они потратили массу времени, чтобы в перспективе получить результат, они держали оружие наготове и защищали оригинальную спецификацию, даже когда оказалось, что конечный продукт нежизнеспособен.

И если бы можно было предсказать, в чем будет нуждаться рынок через два года, то все бы превосходно сработало! Очень жаль, что не получилось, но по крайней мере никто не потерял работу, потому что можно было сослаться на в точности реализованную спецификацию.

Что было бы, если бы сразу применялась более эффективная коммуникация? Как бы изменился продукт, если бы вместо развернутых требований команда с самого начала написала минимальное количество документов, необходимых для работы?

Они должны были доверять друг другу, чтобы принимать правильные решения. Это бы помогло при работе над форматом электронной книги, потому что отсутствовала бы привязка к устаревшему формату, определенному в начале проекта, и они могли бы принять новый. А еще лучше перед началом работы над шаблоном интернет-магазина сразу понять, что это плохая идея, и отказаться от нее. Но это невозможно, потому что шаблон — это часть спецификации, и команда подвешена на ней, как рыба на крючке. Лучшая коммуникация позволила бы сохранить проект в актуальном состоянии и помогла в создании более ценного продукта.

Давайте представим, что так и случилось с проектом «Электронная книга». Благодаря возможности самостоятельно писать документацию наша команда стала счастливее, потому что сократилась нагрузка. Участники группы полагали, что смогут сэкономить время за счет повышения качества общения и отказа от ненужной документации. Но что если, несмотря ни на что, проект по-прежнему отстает от графика?

Почему-то эта экономия времени никогда не реализуется на практике. Похоже, что члены команды, пытаясь включить в ограниченные по времени итерации все те функции, которые захотел владелец продукта, потратили больше ночей и выходных, чем когда-либо. Выходит, что, став более гибкими, они вынуждены работать еще интенсивнее! Какое же это улучшение? Можно ли как-нибудь исправить эту ситуацию, прежде чем команда навсегда разочаруется в Agile?



Ключевые моменты

Слишком подробная документация повышает риск неоднозначности, недопонимания и расхождений во взглядах между членами команды.

Наиболее эффективное общение между членами agile-команды происходит тогда, когда они сосредоточены на разговоре лицом

к лицу и опираются на минимальное количество документации, необходимой для реализации проекта (принцип № 4).

Для достижения наибольшей ценности программного продукта разработчики ежедневно общаются с бизнес-пользователями (принцип № 5).

Каждый член agile-команды чувствует свою ответственность за проект и отвечает за успех (принцип № 6).

Выполнение проекта — перемещение по проекту

Эффективное общение и доверие между членами команды — это отличное начало. После того как все наладили отношения и узнали, как им настроиться на проект, можно задуматься о главной проблеме — ежедневной работе. Как agile-команда продолжает трудиться над проектом?

ПРИНЦИП № 7. РАБОТАЮЩИЙ ПРОДУКТ — ОСНОВНОЙ ПОКАЗАТЕЛЬ ПРОГРЕССА

Хорошая работа команды определяется тем, что все участники — члены команды, менеджеры, стейкхолдеры и клиенты — в любой момент знают, как продвигается работа над проектом. Но как вы сообщаете о его статусе? Эта проблема гораздо сложнее, чем может показаться на первый взгляд.

Типичный командно-административный менеджер старается сохранить верное направление проекта и подробно информировать всех о содержании планов и отчетов о состоянии дел. Но из отчета порой трудно понять реальный статус проекта, потому что это несовершенный инструмент коммуникации. Зачастую разные люди, читая один и тот же отчет, получают абсолютно несовпадающие впечатления о статусе проекта. Менеджеры проекта порой используют отчеты о статусе для своих целей. Почти каждый менеджер проекта, оказавшись в сложной ситуации, старается не вносить в документ какие-нибудь неудобные для него сведения, предоставленные менеджером или руководителем команды. И чаще всего это именно та информация, которая необходима для принятия решения. Так как же можно говорить о прогрессе, если отчеты недостаточно полны?

Ответ кроется в самом программном продукте. Вы можете мгновенно увидеть реально работающее программное обеспечение, оно перед вами, вы «получаете» его. Вы сразу видите, что этот продукт делает (или не делает). Если менеджер обещал предоставить то, чего данное ПО не в состоянии

выполнить, то может возникнуть неловкая ситуация. Но скрыть это невозможно, потому что программа говорит сама за себя.

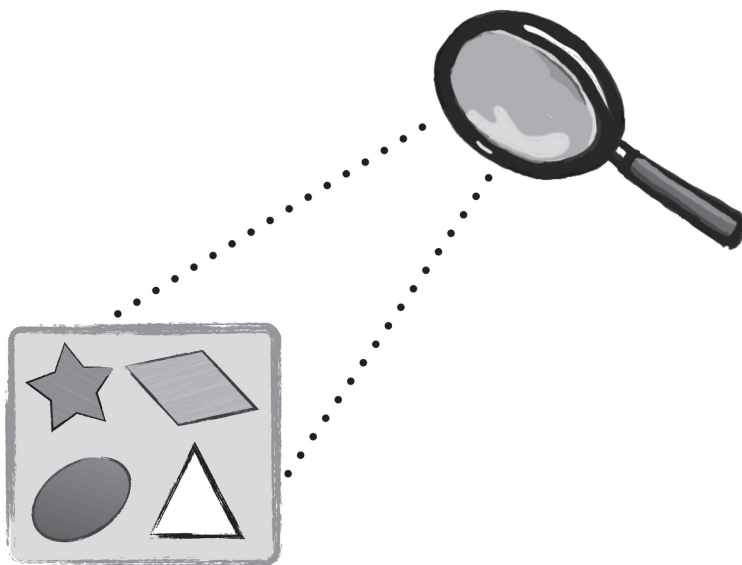


Рис. 3.6. Работающее программное обеспечение лучше, чем последние отчеты о ходе работ, предоставляющие обновленную информацию о статусе проекта. Потому что это наиболее эффективный способ сообщить, чего достигла команда

Это одна из причин, почему agile-команды используют итеративную разработку. Предоставляя работающие программные продукты по окончании каждой итерации и демонстрируя, что именно сделала команда, они держат всех в курсе того, на каком этапе находится проект. Поэтому практически невозможно неправильно истолковать ситуацию.

ПРИНЦИП № 8. ИНВЕСТОРЫ, РАЗРАБОТЧИКИ И ПОЛЬЗОВАТЕЛИ ДОЛЖНЫ ИМЕТЬ ВОЗМОЖНОСТЬ ПОДДЕРЖИВАТЬ ПОСТОЯННЫЙ РИТМ БЕСКОНЕЧНО. AGILE ПОМОГАЕТ НАЛАДИТЬ ТАКОЙ УСТОЙЧИВЫЙ ПРОЦЕСС РАЗРАБОТКИ

Команда проекта «Электронная книга» — далеко не первая из тех, кто напряженно работает, чтобы уложиться в нереальный срок. Жесткие рамки сдачи проекта — основной инструмент в командно-административном наборе. Всякий раз, когда приближается дедлайн, все первым делом думают о работе по ночам и в выходные дни. Невыполнимый срок — это коварный

инструмент, заставляющий команду работать на пределе и сокращающий еженедельные сроки.

Кроме того, в долгосрочной перспективе это не дает желаемого результата. Хорошо известно: команда может работать в напряженном режиме в течение нескольких недель, но после этого ее производительность резко снижается. Это объяснимо: люди переутомились, демотивированы, начинает сказываться усталость. Все незавершенные и повседневные дела, от которых они старались уйти, так или иначе возвращаются и начинают оказывать давление. В самом деле, команды, много работающие сверхурочно, фактически выполняют меньше работы, чем те, кто трудится согласно нормативам. Кроме того, результаты сверхурочных работ, как правило, более низкого качества.

Именно поэтому agile-команды стремятся к сохранению устойчивого темпа. Они планируют выполнение задания, которое действительно можно сделать за выделенное для него время. Итеративная разработка позволяет добиваться этого. Намного проще оценить, сколько программных продуктов можно разработать в течение двух, четырех или шести недель, чем за год-полтора. Давая реальные обещания, команда создает среду, в которой работа по ночам — это исключение из правил*.

ПРИНЦИП № 9. ПОСТОЯННОЕ ВНИМАНИЕ К ТЕХНИЧЕСКОМУ СОВЕРШЕНСТВУ И КАЧЕСТВУ ПРОЕКТИРОВАНИЯ ПОВЫШАЕТ ГИБКОСТЬ ПРОЕКТА

Недостаточно подробная оценка проделанной работы — это причина не только сверхурочных работ. Большинство разработчиков признают: ощущение, что ты тонешь, приходит с осознанием того, что, казалось бы, пустяковое изменение в коде оборачивается кошмаром для разработки. Затем подряд следуют три рабочих выходных, потраченных на исправление ошибок и латание кода.

В долгосрочной перспективе намного надежнее избегать текущих ошибок, чем исправлять их потом. И проще поддерживать отлично спроектированный код, потому что его легко расширить.

* Это хороший пример упрощения, о котором говорилось в главе 1. Правда, сейчас мы будем говорить о том, что «устойчивый темп» означает следующее: команда имеет достаточно времени для создания ПО, поэтому ей не нужно работать по ночам и в выходные. Далее мы подробно расскажем о важности устойчивых темпов для рабочей среды в команде и для культуры компании в целом, а также о том, что она оказывает влияние на качество производимого ПО.

Последние два десятилетия были революционными в разработке программного обеспечения. Объектно-ориентированное проектирование и анализ, паттерны проектирования, независимая и сервис-ориентированная архитектура и другие инновации дали разработчикам шаблоны и инструменты для технического совершенствования в каждом проекте.

Но это не значит, что agile-команды тратят массу времени на создание крупномасштабных конструкций в начале каждого программного проекта. Agile-разработчики приобретают большое количество навыков, помогающих им создавать хорошо спроектированный код. Они находятся в постоянном поиске дизайна и ошибок в коде и немедленно их исправляют. Если во время работы над проектом уделять немного больше внимания написанию надежного кода и исправлению ошибок, то можно создать надежную основу кода, которую можно легко поддерживать в будущем.

ЛУЧШАЯ РАБОЧАЯ СРЕДА ДЛЯ КОМАНДЫ ПРОЕКТА «ЭЛЕКТРОННАЯ КНИГА»

Команда проекта «Электронная книга» и их домочадцы наверняка бы оценили стабильные темпы работы. Но и сам проект должен получиться лучше. С самого первого дня команда была обречена работать сверхурочно, потому что просто не имела средств для создания реалистичного плана, который оставался бы точным полтора года спустя.

Еще хуже то, что в самом начале проекта команда заложила дизайн программного обеспечения и архитектуру для поддержания очень подробной спецификации. В итоге получился крайне сложный код, который трудно расширить. Это привело к большому количеству изменений в коде и к такому же числу «заплаток», которые запутывают код. Если бы команда придерживалась итеративного подхода и поставляла рабочее ПО на протяжении всего проекта, то могла бы планировать каждую итерацию для сохранения стабильного темпа работы. Упрощенный подход just-in-time («точно вовремя») к архитектуре позволил бы создать более гибкий и расширяемый дизайн.

Давайте представим, что наша команда проекта «Электронная книга» применила эти принципы и сейчас они похожи на хорошо отлаженную машину по производству программного обеспечения. Они регулярно воспроизводят и поставляют работающее программное обеспечение и постоянно корректируют работу, чтобы быть уверенными в умении создавать ценное ПО. Члены команды хорошо коммуницируют, документируют только то, что им действительно нужно, используют методы проектирования

и строительства, позволяющие создавать обслуживаемый код. И все это — без необходимости работать сверхурочно. Наша команда стала гибкой!

Но грозовые тучи уже сгущаются над следующим проектом. Новый менеджер просто разослал приглашения на общую встречу всем, кого смог найти. Участники приняли приглашения и начали бронировать номера, аргументы в пользу необходимых требований, которые нужно было задокументировать, витают в воздухе... и все в новоиспеченной agile-команде почувствовали, как у них засосало под ложечкой.

Команда понимает, что происходит. Первые из многочисленных спецификаций, планов и диаграмм Ганта только начинают циркулировать. Как убедиться, что следующий проект не попадет в ловушку, из которой они с таким трудом выбирались?



Ключевые моменты

Наиболее эффективный способ коммуникации в ходе реализации проекта — поставка рабочего программного обеспечения и передача его в руки пользователя (принцип № 7).

Наиболее продуктивно команды работают в устойчивом темпе, без ненужного героизма и сверхурочных (принцип № 8).

Хорошо разработанное и реализованное программное обеспечение намного быстрее поставляется клиенту, потому что в него проще вносить изменения (принцип № 9).

Постоянное совершенствование проекта и команды

Один из основных принципов проектирования не только программного обеспечения, но и во всем инженерном деле — это KISS* («чем проще, тем лучше»). Agile-команда живет по этому принципу, когда планирует проект, разрабатывает программное обеспечение и движется вперед.

ПРИНЦИП № 10. ПРОСТОТА — ИСКУССТВО МИНИМИЗАЦИИ ЛИШНЕЙ РАБОТЫ — КРАЙНЕ НЕОБХОДИМА

Часто бывает довольно сложно добавить код в существующий проект, особенно если дописываемый код зависит от уже существующего. Зависимости

* Keep it simple, stupid.

между системами, объектами, услугами и прочим делают код более сложным и трудно изменяемым: зависимости повышают вероятность того, что одно изменение может привести к каскадным изменениям в другой части системы, которые, в свою очередь, вызовут изменения в третьей, создавая эффект домино с возрастающей сложностью для каждого изменения. При помощи итераций и создания минимальной документации в начале проекта команда может избежать поставки ненужного программного обеспечения.

Однако многие разработчики испытывают неловкость, когда впервые слышат фразы вроде «использование итеративной разработки» и «сделать минимальный объем планирования, необходимого для запуска проекта». Они чувствуют, что преждевременно писать код, пока не рассмотрены и не записаны разные варианты конструктивных и архитектурных решений. Иначе написанный сегодня код придется удалять завтра в процессе изменений в дизайне.

Это понятная реакция, ведь для многих проектов вне сферы разработки программного обеспечения такой подход не имеет смысла. Нередко программисты, только что вступившие в сферу agile-разработки, выдвигают возражения такого рода: «Если подрядчик ремонтирует мой дом, то первым делом я хочу видеть подробные чертежи. Я не допущу, чтобы, второпях поговорив со мной, он начал ломать стены».

Этот аргумент имеет смысл в ситуации с ремонтом дома. Дело в том, что в этом случае нет ничего более разрушительного, чем взять кувалду и сломать стену. Но программные проекты отличаются от домов и прочих физических объектов. Удаление кода не несет глобальных разрушений, потому что вы легко можете восстановить его из системы управления версиями. Самая разрушительная вещь для вашего проекта — это создать новый код, а затем другой, зависимый от первого, и так далее, как выстраивают костяшки домино. Получится до боли знакомый эффект каскадных изменений... И в итоге вы останетесь с неремонтопригодным, запутанным спагетти-кодом.

Максимальное увеличение объема работы не поможет избежать этой путаницы. Лучше создавать систему без большого количества зависимостей и ненужного кода. Наиболее эффективный способ создания максимально полезного и ценного программного обеспечения — это работа с клиентами и заинтересованными сторонами. Если функция не является ценной, то в долгосрочной перспективе для компании дешевле не создавать ее вовсе. Потому что расходы на содержание дополнительного кода выше, чем стоимость самой разработки. Когда команды пишут код, они могут хранить свои несложные проекты программного обеспечения, создавая дизайн ПО

на базе небольших автономных единиц (например, классов, модулей, сервисов и т. д.), которые делают только одну вещь — помогают избежать эффекта домино*.

ПРИНЦИП № 11. САМЫЕ ЛУЧШИЕ ТРЕБОВАНИЯ, АРХИТЕКТУРНЫЕ И ТЕХНИЧЕСКИЕ РЕШЕНИЯ РОЖДАЮТСЯ У САМООРГАНИЗУЮЩИХСЯ КОМАНД

Слишком сложные конструкции характерны для команд, которые уделяют чересчур много внимания планированию. Неудивительно, если задуматься об этом. Вернемся к главе 2 и рассмотрим картину водопадного процесса. В водопадной методологии выделяют целые этапы, посвященные разработке требований, дизайну и архитектуре. Наиболее тщательно выполненные работы на этапе дизайна и архитектуры предполагают создание самой удивительной структуры, которую команда только может придумать. Работающие таким образом интуитивно воспринимают небольшое количество требований и простой дизайн как недоработанный проект. Стоит ли удивляться, что они возвращаются с большим списком требований и сложной архитектурой? Конечно, нет, потому что это именно то, что они просят сделать. Ведь им известно: в водопадном подходе каждый этап должен быть подкреплён соответствующей документацией.

Но **самоорганизующаяся команда** не имеет явных требований или фаз разработки. Ее члены совместными усилиями планируют проект (вместо того чтобы полагаться на одного человека, «владельца» плана) и регулярно его пересматривают. Такие команды, как правило, разбивают проект на пользовательские истории или другие мелкие части и начинают работать с наиболее ценными для компании частями. Только после этого подходит очередь подробных требований, дизайна и архитектуры.

Это делает работу традиционного архитектора программного обеспечения сложнее, но и приятнее. Обычно программный архитектор сидит в кабинете с закрытой дверью и абстрактно рассуждает о проблемах, которые нужно решать. Конечно, бывают исключения, но для них типично отдаляться от повседневной работы остальной группы.

В agile-команде каждый несет свою долю ответственности за архитектуру. Старший архитектор программного обеспечения или дизайнер, как

* Еще одно упрощение. Позднее мы подробнее поговорим о том, как команды могут строить прекрасный код без предварительного создания больших конструкций, какое влияние они оказывают на проекты, и о том, как они могут влиять на работу, пока изменения еще не охватили весь проект.

и прежде, играет важную роль, но при этом не может работать в изоляции. В самом деле, когда команда создает программное обеспечение по частям, начиная с наиболее ценных частей, работа архитектора становится сложнее (но часто и интереснее). Вместо формирования в начале проекта одной большой конструкции, охватывающей все требования, agile-архитекторы используют **инкрементальный дизайн**, включающий в себя методы разработки, позволяющие не только создавать полную систему, но и легко вносить в нее изменения.

ПРИНЦИП № 12. КОМАНДА ДОЛЖНА СИСТЕМАТИЧЕСКИ АНАЛИЗИРОВАТЬ ВОЗМОЖНЫЕ СПОСОБЫ УЛУЧШЕНИЯ ЭФФЕКТИВНОСТИ И СООТВЕТСТВЕННО КОРРЕКТИРОВАТЬ СТИЛЬ СВОЕЙ РАБОТЫ

Команда не может быть гибкой, если не совершенствует способы создания программного обеспечения. Agile-команды постоянно занимаются контролем и адаптацией — они следят за тем, как работают их проекты, и используют эти знания для улучшений в будущем. И делают это не только в конце проекта — на ежедневных встречах члены команды ищут пути изменения и меняют текущую работу, если в этом есть необходимость*. Вам не должно быть неловко признаваться себе и коллегам в том, что получается, а что нет. Это особенно важно для тех, кто только начинает заниматься гибкой разработкой. Единственный способ стать более квалифицированным специалистом — это постоянно оценивать уже сделанное, чтобы понять, насколько полезно ваше участие в команде, а также придумать план еще лучше.

Почти все согласны, что это полезное и нужное дело. Хотя анализ сделанного — это то, что многие команды планируют, но не все выполняют. Действительно, поначалу это не очень удобно, ведь нужно отслеживать ошибки других членов команды, а затем публично указать на них. Не все к этому готовы. Но со временем атмосфера в группе меняется, и людям все проще говорить на эти темы. Все начинают понимать, что это наиболее конструктивный путь.

Еще одна причина, по которой команды не анализируют случившееся, — в недостатке времени. Бывает, что оно запланировано, но желание думать о сделанном пропадает, как только появляется необходимость дать старт

* Еще одно упрощение. Пока мы просто выдвинем идею, что agile-команды занимаются проверкой сделанного и адаптацией. В главе 4 вы подробнее узнаете об этом на примере scrum-команды и о том, как это связано с самоорганизацией команд.

новому проекту. Команды, начинающие новый проект с резервирования времени для общей встречи в конце каждой итерации и проекта в целом, с *большой вероятностью* проведут обзор сделанного, оценят его и придумают план улучшения. Это помогает учиться на собственном опыте и становиться эффективнее.



Ключевые моменты

Гибкие команды создают максимально простые решения, избегая ненужных функций или чрезмерно сложного программного обеспечения (принцип № 10).

Самоорганизующиеся команды разделяют ответственность за все аспекты проекта, начиная с замысла создания продукта и заканчивая его реализацией (принцип № 11).

Выделяя время на то, чтобы обсудить уроки, которые они получили после каждой итерации и в конце проекта, agile-команды постоянно улучшают свои компетенции в области разработки программного обеспечения (принцип № 12).

Agile-проект: объединение всех принципов

Гибкая разработка — уникальное явление в истории программной инженерии. Она отличается от метода «серебряная пуля», который на протяжении многих лет обещал решить все проблемы ПО за счет использования магических практик в сочетании с блестящими программными средствами, а порой и больших счетов из консалтинговых компаний.

Отличие между командами, работающими под девизом «лучше-чем-ни-чего», и теми, которые получают намного больше выгод от agile-методологий, в следующем: последние признают, что практики — это не просто «меню» для выбора. Ключ к совместному использованию этих практик — понимание, что результат, который команда привносит в проект, проистекает из восприятия agile-ценностей и принципов.

Agile-методологии начинаются с ценностей и принципов. Команда, планирующая стать гибкой, должна ясно понимать, как она создает программное обеспечение, каково взаимодействие внутри группы и с остальными сотрудниками компании. Понимая, что принципы первостепенны, а методология вторична, отдавая себе отчет в том, что именно от методологии зависит результат работы, его качество и возможность совершенствования в процессе деятельности, agile-команда реалистично оценивает процесс

поиска более эффективных способов выполнения проектов. Это дает ей реальный способ повысить гибкость и позволяет строить и улучшать программное обеспечение.



Часто задаваемые вопросы

Я суперразработчик, и мне нужно, чтобы все убрались с моего пути, потому что я могу создать великое программное обеспечение! Почему я должен думать о таких вещах, как доски задач и графики сделанной работы?

Даже самые великие разработчики сталкиваются с ситуацией, когда, создав большую часть кода, приходится кромсать его и исправлять, потому что заказчик, ничего не понимающий в ПО, в самый последний момент просит изменить код. Те, кто действительно заботится о мастерстве программирования, испытывают острое чувство досады, если приходится идти на ненужные технические компромиссы, вместо того чтобы создавать код по всем правилам начиная с первого дня, только потому, что некоторые далекие от разработки люди не позаботились подумать обо всем необходимом до того, как проект оказался на полпути до окончания.

Вот почему многих разработчиков привлекают гибкие команды. Да, вы должны будете заботиться о планировании и привыкать к использованию таких инструментов, как доска задач и график выполнения. Agile-методологии основаны на практиках, выбранных именно потому, что они просты и сокращены до минимума, необходимого для эффективного планирования и выполнения проекта. Участвуя в планировании проекта (и на самом деле заботясь о нем!), вы получаете явную выгоду: удастся избежать чувства досады, вызванного срочными изменениями, задавая трудные вопросы в тот момент, когда проект находится на стадии планирования. Это возможно только потому, что эффективные гибкие команды общаются со своими пользователями с самого начала проекта. Много хороших разработчиков начали свое участие в планировании, в первый раз задав одному из своих пользователей трудный вопрос, который в противном случае встал бы позднее. Возможность избежать вечной проблемы, когда в последнюю минуту приходится переделывать код, оказывается отличным стимулом для внедрения Agile.

Agile-планирование — это еще и способ общения с остальными членами команды, который действительно может помочь совершенствоваться великим разработчикам. Такие люди всегда готовы учиться, а в командно-административных системах они изолированы от коллег по команде, поэтому им приходится заниматься самообразованием. Действительно, самоорганизующаяся команда очень много общается. Но это не похоже на бесконечные разговоры — встречи, которые так ненавидят разработчики, вынужденные их посещать. Вместо этого члены команды сами решают, что обсудить, чтобы правильно сделать проект. Это приводит не только к улучшению проектов. Если разработчик, сидящий на встрече рядом с вами, обладает полезными для вас знаниями, то и у вас есть шанс получить их. Например, если ваш сосед применял новый шаблон проектирования для своей работы неизвестным вам способом, то к концу проекта вы узнаете, действительно ли это хороший метод. Узнав о любой полезной идее, вы сможете добавить ее в свою панель инструментов. Такое обучение происходит автоматически, без дополнительных усилий с вашей стороны, просто потому, что члены команды активно общаются. В этом одна из причин, почему разработчикам, использующим agile-методологии, часто становится легче работать с технической точки зрения и они чувствуют, что постоянно совершенствуются в программировании.

Я менеджер проекта, и мне до сих пор непонятно, как я могу вписаться в agile-команду. Какова моя роль во всем этом?

Если вы менеджер проекта, вполне вероятно, что вам отводится одна из трех традиционных ролей управления проектами:

- менеджер, который получает сметы, строит графики проектных работ и руководит повседневной работой команды;
- эксперт по продукции в роли бизнес-аналитика, который определяет требования, направляет их команде и гарантирует, что она создаст необходимое программное обеспечение;
- куратор, который работает с топ-менеджерами и руководством вашей компании, информирует их о том, как окупаются их инвестиции в проект.

В главе 4 вы узнаете о Scrum — наиболее распространенной гибкой методологии — и о ролях, существующих в scrum-командах. Если вы менеджер, не боящийся грязной работы, и готовы вдаваться в детали вместе с командой, то вам, вероятно, подойдет роль scrum-мастера. Scrum-мастер помогает команде распланировать работу, разбить

проект на части, чтобы можно было начать разрабатывать программное обеспечение. Если же ваша задача — понять, в чем нуждается компания-заказчик, и сообщить об этом команде разработчиков, то вы, скорее всего, владелец продукта. В этом случае вам предстоит руководить бэклогом, решать, какие функции должны входить в каждую итерацию, и отвечать на вопросы команды на протяжении всего проекта, чтобы ее участники ориентировались в происходящем и создавали нужное программное обеспечение.

Если вы менеджер проекта, исполняющий контролирующую роль, то, скорее всего, вы не станете членом гибкой команды, и это нормально. Но у вас будет одна из самых важных функций — отстаивать agile-методологии, подталкивать команду и менеджеров применять гибкие методы и продвигать agile-ценности. Вы узнаете, что разделение бэклога функций на итерации наряду с их подробным описанием позволяет правильно оценить уровень детализации, необходимый для общения с управляющими и руководителями высшего звена. И чем лучше проектные команды чувствуют свой прогресс и то, насколько хорошо они работают на достижение целей проекта, тем лучше они могут делать свою работу, давая вам реалистичное понимание этого прогресса. Но чтобы делать это эффективно, нужно хорошо понимать, как работают agile-команды, говорить с ними на одном языке, перерабатывать информацию, которую они дают, и облекать ее в форму, понятную вашим управляющим.

Но если вся команда участвует в планировании, то значит, никто ни за что не отвечает? Это непрактично. Как принимаются решения?

Все зависит от того, что это за решения. Если вы имеете в виду разрешение конфликтов, то это должно происходить именно так, как практикуется в вашей команде сегодня. Кто главный в вашей команде сейчас или был главным в последней проектной команде, в которой вы работали? Кто разрешал споры между членами команды, рецензировал работу? Существует множество способов создания иерархии в компании, и agile-команда должна уметь работать в любой иерархической структуре. Однако agile-команды, как правило, лучше решают конфликты самостоятельно, потому что их члены сосредоточены на общении друг с другом и приходят к согласию гораздо эффективнее, чем другие команды.

Но если вы спрашиваете о том, кто отвечает за выбор параметров, которые будут включены в программное обеспечение, или как они будут определяться, то обычно эти вопросы решают обладатели определенных ролей в гибкой команде. В scrum-команде владелец продукта

имеет право решать, какие характеристики войдут в ПО. Однако только команда выбирает, какие характеристики могут вписываться в итерации, опираясь при этом на реальную информацию. Мы расскажем об этом подробнее в главе 4. Однако план принадлежит всей команде, потому что она самоорганизующаяся.

Но это вовсе не означает, что нет руководителя. Конечно, он есть. Если вы только собираетесь стать гибкой командой, то вашим руководителем, вероятнее всего, будет тот же человек, что и сегодня. Но он должен верить в agile-методологии настолько, чтобы дать команде полномочия принимать решения, связанные с проектом, и будет их отстаивать, не пытаясь заниматься микроменеджментом или менять требования задним числом. Это единственный способ, который может работать в реальном мире.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой).

- Если вы находитесь в процессе создания проекта, то, прежде чем начать писать код, не пожалейте 15 минут и обсудите вместе с командой, какие функции вы будете разрабатывать. Можете ли вы вспомнить примеры, когда два человека имеют разные представления о том, что собираются создавать?
- Составьте список функций, над которыми вы работаете. Попробуйте упорядочить их по значимости и сложности.
- Составьте список всех документов, которые вы и ваша команда создали или использовали. Есть ли среди них то, что команда не использует для построения кода?
- В следующий раз, когда вы будете работать допоздна, задумайтесь, что вас заставило задержаться. Можно ли было как-то предотвратить это? Возможно, установленные сроки слишком жесткие? Или в последнюю минуту появилась дополнительная работа?

Признать наличие проблемы и выделить время на ее осознание — вот первый шаг к исправлению ситуации.



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Узнать больше о ценностях и принципах Agile-манифеста и о том, как он был создан, можно в книге Алистера Коберна «Быстрая разработка программного обеспечения» (М. : Лори, 2013).
- Узнать подробнее о значимости итерации и других аспектов agile-управления проектами можно в книге Джима Хайсмита *Agile Project Management: Creating Innovative Projects* (Addison-Wesley, 2009).
- Узнать больше о сложных задачах, с которыми сталкиваются команды, желающие стать гибкими, и о том, как их преодолеть, можно в книге Майка Кона «Scrum. Гибкая разработка ПО» (М. : Вильямс, 2016).
- Узнать больше о том, как оставить в прошлом командно-административное мышление, можно, прочитав книгу Лиссы Адкинс *Coaching Agile Teams* (Addison-Wesley, 2010). В настоящее время эта книга готовится к изданию.



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Помогите команде понять, что сверхурочная работа — это причина сокращения объема созданного кода, более того — она ухудшает его качество.
- Поговорите с каждым членом команды о том, какой работой он занимается. Что им движет? Что расстраивает? Чем он руководствуется при принятии решений?
- Попросите каждого участника группы выбрать три agile-принципа, оказывающих на него наибольшее влияние (как негативное, так и позитивное). Люди будут удивлены, что их коллеги выбрали разные принципы. Это поможет найти точки соприкосновения между всеми участниками команды.
- Используйте принципы, общие для всех членов команды, как отправную точку, чтобы выяснить, какие методы наиболее соответствуют мировоззрению команды.

ГЛАВА 4

Scrum

и самоорганизующиеся команды

Великие принципы, не рождающие действия, являются всего лишь эфемерной субстанцией. И наоборот, конкретные практики при отсутствии руководящих принципов часто используются неадекватно.

Джим Хайсмит*

Лозунг настольной игры «Отелло» — «Минута на обучение, вся жизнь на совершенствование». Это очень подходит команде, которая учится Scrum. Базовые практики и ценности Scrum просты и легки в применении. Но понять, как они превратятся в результат — лучшее программное обеспечение, — может оказаться непросто.

Правила Scrum просты и легки для понимания, что позволяет многим командам, применяющим agile-методологии, использовать его в качестве отправной точки. Вот основная схема scrum-проекта.

- В scrum-проекте существует три основные роли: **владелец продукта, scrum-мастер и член команды**. (Мы придаем особую важность понятиям «владелец продукта» и «scrum-мастер», когда речь идет о scrum-ролях.)
- Владелец продукта работает с остальной частью команды, чтобы поддерживать и определять приоритеты функций и требований продуктового бэклога, которые необходимо реализовать.

* Jim Highsmith. Agile Project Management: Creating Innovative Products (Upper Saddle River, NJ: Pearson Education, 2009).

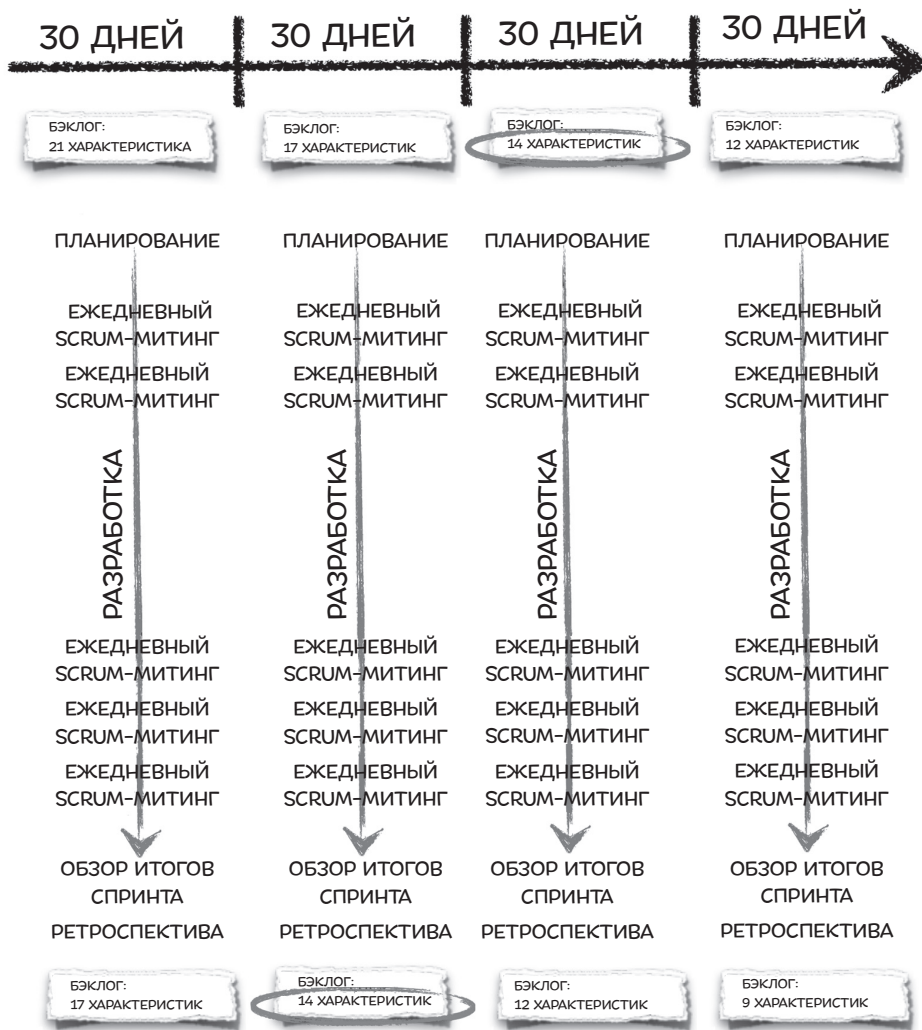


Рис. 4.1. Базовая схема Scrum

- Программное обеспечение строится с использованием ограниченных по времени итераций, называемых **спринтами**. В начале каждой такой итерации команда выполняет **планирование спринта**, чтобы определить, какие функции из бэклога они будут реализовывать. Это называется **бэклог спринта**. На протяжении спринта команда работает над созданием всех тех функций, которые в него вошли.
- Ежедневно все члены команды участвуют в короткой встрече (**Daily Scrum** — **ежедневный scrum-митинг**), чтобы рассказать друг другу

о достижениях и обсудить то, что препятствует дальнейшей работе. Каждый человек отвечает на три вопроса: что я сделал с момента последнего ежедневного совещания? Что буду делать вплоть до следующего ежедневного совещания? Какие препятствия есть на моем пути?

- Scrum-мастер поддерживает правильное направление работы над проектом, устраняет препятствия на пути команды и помогает ей, если есть просьбы о помощи. В конце спринта работающее ПО показывают владельцу продукта и стейкхолдерам проекта. Команда проводит одновременно **обзор итогов спринта** и **ретроспективный обзор**, чтобы выявить ошибки. Таким образом удастся улучшить сам процесс спринта и качество создания программного продукта в будущем.

Чтобы стать успешной, scrum-команде нужно делать больше, чем просто придерживаться базовой схемы Scrum. Эффективная scrum-команда должна быть **самоорганизующейся**. В это понятие мы вкладываем тот же смысл, что и Кен Швабер в своей книге *Agile Project Management with Scrum* (обратите внимание на слова, которые мы выделили курсивом).

Чтобы работать по Scrum, команда должна глубоко и на интуитивном уровне понимать *коллективную ответственность и самоорганизацию*. Scrum-теорию, методы и правила легко понять. Но пока группа, состоящая из отдельных людей, не взяла на себя коллективное обязательство за создание чего-то материального в течение ограниченного промежутка времени, *эти индивидуумы, пожалуй, не овладеют Scrum*. Когда же члены команды перестают действовать как многие и принимают на себя обязательства за общие цели, команда становится способной к самоорганизации, может быстро пройти через сложности и на практике реализовать планы.

Цель этой главы — помочь вам «заполучить Scrum», опираясь на идеи из глав 2 и 3, то есть научить вас практикам и моделям этой методологии. Мы будем использовать эти практики, чтобы продемонстрировать идеи, стоящие за принципами коллективной ответственности и самоорганизации.

Правила Scrum

В книге Кена Швабера *Agile Project Management with Scrum* изложены правила Scrum, описывающие основной шаблон scrum-проекта. Вы можете бесплатно скачать правила Scrum на сайте www.scrum.org в формате PDF — это электронная книга *The Scrum Guide*, написанная Кеном Швабером

и Джеффом Сазерлендом. Эти люди создали Scrum и помогли распространить его влияние на остальные направления программирования. Правила должны быть вам хорошо знакомы, потому что многие из них описаны в главах 2 и 3. Вот каким правилам следует типичный scrum-проект.

- Каждый спринт начинается с планирования, в котором участвуют scrum-мастер, владелец продукта и остальные члены команды. Встреча разделена на две части, по четыре часа каждая. Владелец продукта заранее выделяет приоритетные направления продуктового бэклога, который состоит из набора позиций, выбранных для разработки потребителями и заинтересованными сторонами. В первой части встречи владелец с командой выбирает элементы бэклога, которые будут поставлены в конце спринта, исходя из их значимости и оценки командой объема работ. Команда обязуется провести презентацию работающего программного обеспечения, включающего все эти элементы бэклога в конце спринта. На это тратится половина отведенного для планирования времени (для 30-дневного спринта это четыре часа, а для более коротких время пропорционально сокращается). В результате команда фиксирует элементы бэклога, одобренные командой на этапе планирования, и использует их в качестве бэклога спринта. Во второй части встречи члены команды (при помощи владельца продукта) знакомятся с индивидуальными заданиями, которые они будут выполнять для фактической реализации этих элементов.

Длина этой части встречи также зависит от длины спринта (но часто занимает меньше времени, чем другая). По окончании планирования выбранные участниками элементы становятся бэклогом спринта.

- Команда проводит scrum-митинг каждый день. Все члены команды (включая scrum-мастера и владельца продукта) обязаны* в них участвовать. Заинтересованные лица также могут присутствовать (но только в качестве наблюдателей). Встреча длится не больше 15 минут, поэтому все члены команды должны приходить вовремя. Каждый из них отвечает на три вопроса: что я сделал с момента прошлой встречи? Что буду делать с сегодняшнего дня и до следующей встречи? Какие препятствия есть на моем пути? Все члены команды должны быть кратки-

* Не кажется ли вашей команде, что ежедневные встречи — это нереально? Может быть, вы так думаете, потому что люди распределены между командами или имеются другие обязательства? Вы уже начали искать альтернативные варианты, например заменить ежедневные scrum-митинги на онлайн-собрания или вики-страницу? Это может быть показателем того, что вам необходимо изменить свой образ мыслей, чтобы получить наилучшие результаты от Scrum.

ми. Если ответ требует обсуждения, то ответственные за составление графика немедленно организуют дополнительную встречу.

- Продолжительность любого спринта определяется на совместной встрече. Многие команды планируют спринт на 30 календарных дней, но возможны варианты — некоторые команды выбирают двухнедельные периоды. Во время спринта команда работает над задачами, вошедшими в бэклог спринта. Разработчики могут получить помощь от участников проекта, не входящих в команду, но они не имеют права указывать команде, как надо работать, и должны доверять ей. Если любой член команды в середине спринта обнаружит, что требуется дополнительное время или дополнительные элементы бэклога, то, как только станет ясно, что спринт в опасности, нужно уведомить об этом владельца продукта. Владелец продукта — это член команды, который работает с пользователями и стейкхолдерами и оповещает их о ходе проекта. Он использует полученную информацию для коррекции спринта, чтобы он соответствовал реальным возможностям команды. Если же команда считает, что закончит работу до окончания спринта, то она может добавить в бэклог дополнительные позиции. Команда должна актуализировать состояние бэклога спринта и предоставлять к нему доступ остальным членам проектной группы. В экстренной ситуации владелец продукта может завершить спринт раньше и начать планирование нового спринта. Так бывает, если команда понимает, что не может предоставить работающее ПО (например, возникают серьезные технологические, организационные или кадровые проблемы). Но каждый должен знать: прекращение спринта случается редко и имеет крайне негативные последствия для производительности, а также с точки зрения доверия со стороны пользователей и стейкхолдеров.
- В конце спринта команда проводит его обзор, где демонстрирует работающее программное обеспечение для пользователей и заинтересованных сторон. Демоверсии могут содержать только те элементы бэклога, которые фактически были *сделаны** (то есть команда закончила все работы по созданию этих пунктов и владелец продукта принимает их как готовый продукт). Команда может представлять только функционирующее ПО, а не промежуточные элементы (схемы архитекту-

* То, что фактически сделанная работа — та, которая действительно выполнена полностью, и нет ничего, что нужно доделывать, интуитивно понятно. Но существует много нюансов. Это еще один пример упрощения, и мы будем возвращаться к этой концепции несколько раз на протяжении всей книги. Будьте внимательны, следите за ходом мысли! Теперь, когда вы видели некоторые из сделанных упрощений, мы не будем помечать новые упрощения сносками. Но мы все еще будем использовать этот прием в книге.

ры, базы данных, функциональные спецификации и т. д.). Заинтересованные стороны имеют право задавать вопросы, на которые команда должна ответить. По окончании демонстрации стейкхолдеров просят высказать свое мнение. Если необходимо внести изменения, то это учитывается при планировании следующего спринта. Владелец продукта может добавить изменения в продуктовый бэклог, и если они должны быть выполнены немедленно, то их в итоге внесут в следующий бэклог спринта.

- По окончании спринта команда проводит ретроспективу (чтобы понять, как улучшить свою работу), где присутствует и scrum-мастер, и, при необходимости, владелец продукта. Каждый участник отвечает на два вопроса: что было сделано хорошо во время спринта? Что можно улучшить? Scrum-мастер отмечает любые улучшения, а также отдельные пункты (монтаж нового сервера сборки, внедрение нового метода программирования или перепланировка офиса и т. д.) и добавляет их в список продуктового бэклога как нефункциональные элементы.

Вот и все! Ничего сложного.

Но если это так просто, то почему все мы не используем Scrum?

Почему же такой значительной части нашей команды, работающей со Scrum и соблюдающей все правила, кажется, что получаемый ими результат — это всего лишь немногим лучше, чем ничего? Чего не хватает?



Описание: команда, разрабатывающая приложение для мобильного телефона в небольшой компании

Роджер — руководитель команды, пытающийся использовать Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт I. Я могу использовать Scrum?

Hover Puppy Software — небольшая компания, создающая сайты и приложения для мобильных телефонов. Она пользуется популярностью. Шесть месяцев назад выпущенное ими приложение для мобильного телефона принесло огромную выручку, и CEO решил вложить деньги в запуск нового

сайта Lolleaderz.com, который позволяет пользователям создавать рейтинги любимых видеороликов.

В начале проекта его руководитель Роджер хотел использовать Agile. Он был по-настоящему счастлив, когда узнал, что команда рада такой перспективе. Разработчики мобильных телефонов в другой компании уже использовали Scrum для создания очень популярного приложения, и идея получила там широкое распространение. Один из членов команды стал называть Роджера scrum-мастером, и эта роль закрепились за ним.

Первым делом Роджер начал искать владельца продукта, но нашел его не сразу. К счастью, Hover Purrry — небольшая компания, где сотрудники называют директора по имени. Роджер просто подошел к нему и объяснил ситуацию, описывая владельца продукта как «короля заинтересованных сторон» проекта. Примерно в то же время один из менеджеров, Ави, только что закончил проект. CEO познакомил его с Роджером, объяснил, что полностью поддерживает Scrum, и оставил их вдвоем — выяснять, как лучше организовать работу над проектом.

Поначалу все шло хорошо. Роджер установил продолжительность каждого спринта в один месяц, а Ави придумал бэклог задач для разработки. Роджер организовал ежедневные scrum-митинги, Ави занес их в свой календарь, чтобы видеть их каждый день. Первый спринт прошел отлично: все двигались вперед вместе, и команда добилась определенных успехов. В конце спринта команда предоставила Ави демоверсию сайта с несколькими функциями, которые они вместе планировали. Казалось, что эксперимент со Scrum удался.

В течение следующих нескольких недель в проекте начали появляться трещины, но он все еще держался на плаву. Один менеджер по обслуживанию клиентов настраивал показ рекламы нового блокбастера своего клиента, кинокомпании, чтобы он происходил при каждом наведении мышки на сайт Hover Purrry. Команда пообещала Ави, что предоставит ему демоверсию этой функции в конце спринта. Но на раннем этапе разработки возникли технические проблемы, и пришлось перенести эту задачу на следующий спринт. Роджер объяснил, что scrum-команды всегда поставляют работающее ПО, а если создание функции еще не закончено, то задача переносится на следующий спринт. Но они не были уверены, что именно так должен работать Scrum.

Работоспособность команды снижалась с каждым спринтом. К концу третьего спринта Ави почувствовал, что в основном занимается командой, поэтому на клиентов у него остается совсем мало времени. В начале проекта ему казалось, что он контролирует работу команды. Теперь же он начал

понимать, что поторопился, согласившись стать владельцем продукта в проекте Lolleaderz.com. Роджер расстроился, услышав жалобы Ави, что команде тяжело работать с другими аккаунт-менеджерами.

Но хуже всего было то, что команде на самом деле надоела вся эта scrum-возня. Прежде чем Роджер и Ави получили возможность разобраться в этой проблеме, случилась еще одна неприятность. Три разработчика высказали Роджеру свое недовольство, что им приходится ходить на ежедневные scrum-митинги. Один из них сказал: «У меня и так много работы, а эти совещания просто пожирают мое время. Почему я должен сидеть и смотреть, как вы раздаете всем задания на ближайшие дни? Разве нельзя отправить их нам по электронной почте?» Роджер пробормотал, что таковы правила Scrum, поэтому разработчики должны продолжать ходить на собрания. Такой невнятный ответ никого не удовлетворил, и Роджер уже начал задумываться, не стоит ли прислушаться к мнению разработчиков.

Крупный скандал произошел, когда Роджер планировал четвертый спринт. Ави настаивал на добавлении функции, позволяющей пользователям голосовать за понравившееся видео. Он считал, что если не сделать этого, то они начнут терять рекламодателей. Но разработка базы данных для этой функции была трудоемкой. Чтобы внести основные изменения в модель данных, требовалось привлечь высококвалифицированного администратора баз данных (DBA), а это означало, что у него не останется времени для написания хранимых процедур. Создалось впечатление, что на разработку этой функции потребуется еще неделя, но переносить ее на следующий спринт не представлялось возможным.

Миновало уже шесть месяцев, пять спринтов были пройдены. Роджер чувствовал, как Ави усложняет требования для команды разработчиков. Ави был расстроен, что не удастся разработать сайт, который он мог бы продать своим клиентам. Команда предполагала закончить страницу видеотегов и социальных медиа двумя спринтами ранее, но до сих пор не сделала этого. На последней встрече аккаунт-менеджеров Ави обвинил команду в отставании от сроков. Роджер понял: проект находится под угрозой закрытия.

То, что начиналось так хорошо, превратилось в проект-монстр. Роджер чувствовал, что над ним нависла угроза, но не знал, как все исправить. Он перечитал книги и пересмотрел сайты о Scrum и пришел к выводу, что делал все правильно — по крайней мере, на бумаге. Проводились спринты, ежедневные scrum-митинги и ретроспективы. Он работал с владельцем продукта над приоритетами бэклога, для каждого спринта выбирал наиболее ценные функции, работал с командой над оценкой трудозатрат и назначал им разработчиков.

Обращаясь к команде, Роджер в отчаянии сказал: «Меня просто затащили в кабинет к генеральному директору. Он недоволен отсутствием результатов, так же как и Ави. Послушайте, я буду защищать вас и возьму ответственность за случившееся на себя. Но для этого нужно поработать над нашими расчетами, потому что они совершенно неверные. И, кроме того, мне действительно необходимо, чтобы вы поработали по выходным и выправили ситуацию». За последние два месяца он просил об этом свою команду уже в третий раз. Этот проект так же, как и предыдущие, выбился из расписания.

Но что пошло не так? Можете ли вы определить, в чем проблема? Что бы вы сделали, если бы вас назначили scrum-мастером этого проекта? Вспомните об agile-ценностях и принципах. Есть ли способ их применения, который бы помог проекту?

Каждый член scrum-команды — владелец проекта

У каждого scrum-проекта есть владелец продукта, scrum-мастер и команда. Но не всякий проект можно назвать эффективным. Два владельца продукта, которые придерживаются разных методологий (Scrum и Waterfall), будут действовать по-разному. Scrum-мастер не делает того же, что командно-административный менеджер проекта или технический руководитель команды. Вот когда scrum-мастер, владелец продукта и команда начинают работать вместе, а не врозь, — это начинает походить на scrum-проект.

SCRUM-МАСТЕР УПРАВЛЯЕТ ПРОЦЕССОМ ПРИНЯТИЯ КОМАНДНЫХ РЕШЕНИЙ

То, как scrum-мастер делает свою работу, сильно отличается от командно-административной модели управления.

В командно-административном проекте его руководитель — владелец и хранитель расписания и плана работы. Он беседует со стейкхолдерами, получает требования, меняет план работы, получает от команды расчеты, распределяет задачи и строит график. Именно такой подход Роджер использовал в проекте Lolleaderz.com — он получил требования от Ави, смету от команды и распределил задачи между ее членами.

Участники командно-административной команды всеми силами спасают свою шкуру и не берут на себя ответственность, если в проекте возникают проблемы, вызванные планами каких-то других работ. Любому, кто имеет исключительное право собственности на график и план работы, остальные

члены группы и владелец продукта с удовольствием позволяют принимать решения самостоятельно.

Это одна из причин, почему Scrum не отводит отдельной роли для человека, который бы единолично был владельцем плана. Если за точность выполнения плана в команде отвечает только тот, кто исполняет роль его владельца, то все остальные, столкнувшись с неприятностями, могут отмахнуться от них, сообщив: «Это проблемы того парня». Практически в каждом проекте рано или поздно возникают трудности, потому что все они обусловлены проблемами планирования. Поэтому scrum-мастер не имеет собственного плана. Он помогает команде создать его и, что еще важнее, правильно использовать Scrum и его практики. Он дает возможность каждому члену команды участвовать в планировании, а практики и ценности Scrum помогают им ощутить свою причастность.

ВЛАДЕЛЕЦ ПРОДУКТА ПОМОГАЕТ КОМАНДЕ ПОНЯТЬ ЦЕННОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Представьте себе такой разговор между CEO и scrum-мастером. CEO спрашивает, что он получит в следующем году, инвестируя 2 миллиона долларов в программное обеспечение. Scrum-мастер отвечает: «Пока точно не знаю, предстоят ежемесячные обновления ПО, а в конце проекта получим программное обеспечение стоимостью не менее 2 миллионов долларов». Ни один здравомыслящий CEO не одобрит такой проект. Почему?

Причина в том, что он не основан на реальных **обязательствах**. Обязательства — это обещание сделать что-либо к определенному сроку. Настоящее обязательство тесно связано с дополнительной ответственностью, что подразумевает возможность информировать членов команды и искать решения, если обещание невозможно выполнить.

Большинство из нас присутствовали на собраниях, где члены команды утверждали, что не «подписывались» под сроками выполнения работ, поэтому не несут за них никакой ответственности. В этом вина некоторых неопытных руководителей. Они ошибочно полагают, что, как только задача записана в план, команда получает непреодолимое стремление ее выполнять.

Обязывают не планы, а наши обещания. План — это просто удобный способ их фиксации. Обязательства даются людьми и документируются в планах проекта. А это не просто бумажка. Это зафиксированная информация, и все держат ее в голове.

В scrum-команде владелец продукта — это человек, который взял на себя выполнение обязательства. Он должен пообещать конкретные результаты, которые будут достигнуты в конце проекта. Владелец продукта — ключевое звено в реализации бизнес-цели, ради которой затевался проект.

Чем эффективнее на встрече с командой он сможет передать эти задачи и позволить ей осознать обязательства, тем лучше будет выполняться проект. Когда проект сталкивается с неизбежными трудностями (техническими проблемами, изменениями в бизнесе, увольнением сотрудников и т. д.), владелец продукта должен найти способ сохранить взаимопонимание в команде и поддерживать в ней чувство ответственности. Ежедневно он принимает решения, основанные на бизнес-изменениях, и встречается с командой, чтобы узнать, правильно ли она понимает бэклог и трансформировавшиеся задачи проекта.

Владелец продукта не ждет безучастно окончания спринта. Он обязан ориентироваться в происходящем. Его дело — расставлять приоритеты в бэклоге, чтобы быть голосом заказчика в команде разработки, помогать выявлять наиболее важные и ценные истории и задачи. Он должен быть уверен, что каждый член команды правильно понимает выражение *фактически сделано* в контексте бэклога задач. В процессе планирования спринта вся команда решает, какие функции из продуктового бэклога нужно переместить в спринт бэклога. Выбор делается на основании их значимости и расчетов трудозатрат на их выполнение. На этом его работа не заканчивается, затем владелец продукта руководит этим процессом.

Ежедневно владелец продукта принимает активное участие в проекте. Как и все успешные agile-команды, scrum-команды во многом зависят от личного общения, которое помогает правильно понимать, что они создают. Планирование спринта, проведенное в начале, дает каждому члену команды достаточно информации, чтобы начать работу, но владелец продукта не успевает сообщить всей команде подробности. Поэтому во время спринта владелец продукта каждый день занят. Он подробно отвечает на многочисленные вопросы членов команды, дает конкретные рекомендации, как вести разработку, и рассказывает, каким образом пользователи будут применять созданные функции, а также решает массу других вопросов, касающихся работы продукта.

Владелец продукта имеет право принимать такие решения. (Если он не занимается этим, значит, занимает чужое место. Scrum зависит от способности владельца продукта принимать решения от имени заказчика, в том числе выполненную работу.) Но он не располагает исчерпывающими сведениями. Обычно в проекте принимает участие много пользователей

и заинтересованных сторон, владеющих ценной информацией, поэтому владелец продукта тратит много времени на общение с ними, чтобы получить ответы на вопросы разработчиков. Кроме того, он использует такое общение, чтобы отслеживать происходящие изменения и поддерживать актуальность продуктового бэклога, где отражаются последние изменения, в которых нуждается компания. Таким образом, если есть изменения в относительной ценности различных элементов продуктового бэклога, то владелец может изменить их очередность, подготавливая тем самым команду к следующей сессии по планированию спринта.

КАЖДЫЙ ВЫСТУПАЕТ В РОЛИ ВЛАДЕЛЬЦА ПРОЕКТА

В scrum-командах любят рассказывать басню про свинью и курицу, чтобы разобраться в том, как работает принятие на себя обязательств.

Свинья и курица идут по дороге.

Курица говорит свинье: «Давай откроем ресторан!»

Свинья отвечает: «Хм, можно. А как мы его назовем?»

Курица отвечает: «Почему бы не назвать его “Яичница с беконом”?»

Свинья, немного подумав, говорит: «Нет, спасибо, ведь тогда мне придется посвятить себя проекту полностью, а ты будешь вовлечена лишь частично»*.

Так и в scrum-проекте: кто-то вовлечен частично, как курица, а кто-то полностью, как свинья. Чем это отличается от неэффективного водопадного подхода? Все сводится к тому, как действуют члены команды, руководитель проекта и владелец продукта**.

В scrum-командах часто говорят о ролях «свиней» и «куриц». Это упрощение введено для того, чтобы легче было определить человеку ту роль, которую он играет в проекте. Вспомните проекты, в которых вы участвовали: всегда ли вы действительно связывали собственные успехи или неудачи с успехом проекта?

* Из англоязычного варианта «Википедии» (The Chicken and the Pig, проверено 26 июля 2014 года).

** Разговор о «свиньях» и «курицах» выглядит странно, но scrum-команды действительно делают это. На самом деле некоторые старые версии scrum-руководства включают раздел, посвященный «свиньям» и «курицам»!



Рис. 4.2. В истории про свинью и курицу роль первой значительно превышает вклад второй в готовый завтрак

Дело в том, что большинство людей считают себя «курами». Они готовы внести свой вклад в проект, но не хотят рисковать, ведь проверки, повышения, достижение будущих карьерных целей, сохранение рабочего места — все зависит от успеха проекта. Также легко забыть, что люди работают за деньги. Почему люди выполняют свою работу? Что на самом деле их мотивирует? Это не обязательно успех в текущем проекте.

Всех членов *эффективной* agile-команды можно считать «свиньями». Они искренне считают, что их личный успех зависит от успешности проекта. Но даже очень опытная scrum-команда легко возвращается к менталитету «курицы».

Бывало так, что вы как разработчик выбирали технологию для проекта лишь потому, что хотели изучить ее? Или в качестве руководителя проекта

предпочитали agile-методологии, потому что они делают вас более востребованным? Скорее всего, да. Мы все хотя бы раз прошли через это. Каждый целеустремленный человек поступал подобным образом, и это важно признать. Но есть одно обстоятельство, предусмотренное правилами Scrum: выполнение задач спринта важнее всех прочих *профессиональных* целей*, которые у вас есть. Иными словами, пока ты в scrum-команде — оставайся «свиньей».

Когда все члены команды — «свиньи», значит, каждый взял на себя обязательства и будет делать все, что необходимо для их выполнения.

Вот пример, помогающий понять, что значит брать на себя обязательства. Допустим, вы CEO компании и у вас есть обязательства перед проектом. И выясняется, что члены проектной команды устали и нуждаются в кофе. Если все заняты и не могут прерваться прямо сейчас, то несмотря на занимаемую должность вы пойдете за кофе для команды, потому что действительно имеете обязательства и убеждены: именно на это нужно потратить свое время.

В то же время в каждом проекте программного обеспечения должны быть и «куры». Например, каждый пользователь — это потенциальная «курица». Сколько раз вы были разочарованы возможностями браузера, текстовым редактором или почтовым клиентом? Вы когда-нибудь оставляли свои отзывы на сайте или отправляли их по электронной почте команде поддержки? Это один из способов превратить себя в «курицу». Раз команда прислушивается к вашему мнению и решает проблему, значит, вы помогли повысить ценность продукта. И чем активнее «курица», тем более высокую ценность она может принести.

Если в scrum-проекте вы играете роль «курицы», то ваше мнение важно. Вы заинтересованы в результатах, и команде интересны ваши мысли. Но ваша работа напрямую не связана с проектом, у вас другие цели (например, продажа продукта, его поддержка или управление компанией). Все это имеет значение, но не связано с разработкой конкретных функций.

Вот почему scrum-команды — и особенно владелец продукта — налаживают отношения со своими «курами». Один из самых эффективных способов, помогающий в этом, — обеспечить пользователей новыми версиями работающего ПО на регулярной основе и с предсказуемым графиком. Это поддерживает вовлеченность «кур» и помогает им видеть свое влияние на проект.

* Уточню: настоящая «свинья» заботится об успехе проекта больше, чем о чем бы то ни было в своей *профессиональной* жизни. Есть немало вещей в *личной* жизни — например, семья, — о которых следует заботиться больше, чем обо всем прочем. Если это не так, то у команды проблемы с образом мыслей и он будет мешать устойчивому темпу работ.

КАК ВЛАДЕЛЬЦЫ ПРОДУКТА, SCRUM-МАСТЕРА И ЧЛЕНЫ КОМАНДЫ МОГУТ ЛУЧШЕ ИГРАТЬ РОЛЬ «СВИНЕЙ»

Когда член команды бесстрастно выполняет задачи, возложенные на него руководителем проекта, не беспокоясь о возникновении каких-либо проблем, он ведет себя как «курица». Он не чувствует своей ответственности за этот проект, впрочем, как и ответственности других членов команды. Но если этот член команды искренне заинтересован в создании правильного плана и наиболее ценного программного обеспечения для клиента и своей компании, то он ведет себя как «свинья».

К сожалению, многие компании ожидают от членов команды действий, характерных для «кур», а не для «свиней». Программисты нередко обнаруживают, что их попытки принять участие в планировании заканчиваются ничем, потому что планирование и принятие решений — это привилегия менеджеров. («Уж не возомнил ли ты себя менеджером? Отправляйся на свое рабочее место, жалкий программист!») Если такое видение характерно для команды, то ей трудно эффективно применять Scrum.

Scrum-мастеру непросто поощрять команду «кур» в их стремлении стать владельцами или «хранителями» плана. Командно-административному менеджеру, стремящемуся стать scrum-мастером, трудно ломать стереотипы. Это не помогает успешным менеджерам проектов, привыкшим единолично разрабатывать план. А высшему руководству гораздо удобнее, когда за все отвечает один человек. (Некоторые любят рассуждать о нем как о «единственном, кому должны свернуть шею».) Он чувствует свою значимость, потому что создал порядок из хаоса.

Планирование необходимо. Но им нужно заниматься вместе с командой. Когда scrum-мастер разбивает работу на спринты, самостоятельно получает оценки у команды, распределяет между ее членами задачи и проверяет их выполнение, он ведет себя как «хранитель» плана. И одновременно поощряет «кур» вместо «свиней».

Scrum-мастеру приходится устранять одно из основных препятствий на пути внедрения Scrum в команде — менталитет «курицы». Часто команда голосует за командно-административного менеджера проекта, потому что он берет на себя всю ответственность. Но гораздо важнее сделать проект быстро и хорошо, потому что команды разработчиков, применяющие близорукий подход, создают плохо спроектированные системы. Когда команда выбирает кратчайший путь и срезает углы, она может получить продукт, который на первый взгляд выглядит работоспособным, но на самом деле очень хрупок и труден для поддержания.

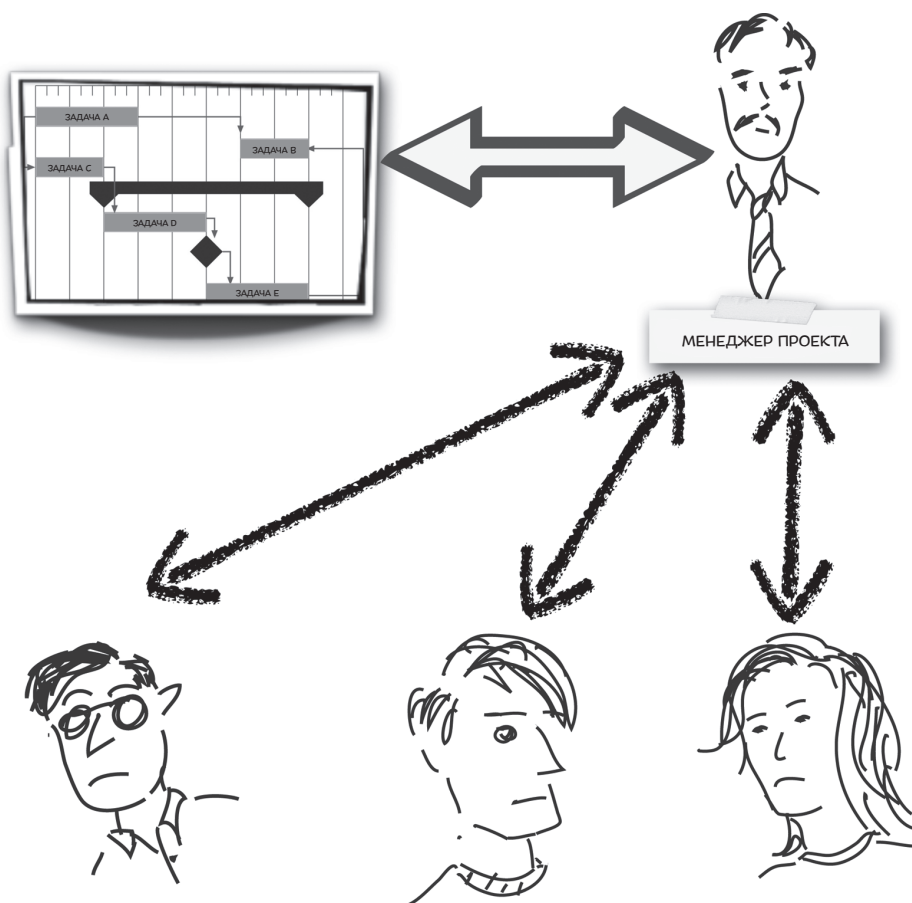


Рис. 4.3. Когда командно-административный менеджер проекта выступает в качестве «хранителя» плана, он вынуждает команду быть «курицами»

Scrum-мастер может стимулировать членов команды стать «свиньями» в рассмотрении оценок как еще не раскрытых фактов, а не обязательств, выбитых из команды. Оценка длительности выполнения задачи — это уже факт, потому что команда будет тратить совершенно конкретное время на выполнение работы. Точные расчеты — это реальная попытка спрогнозировать будущее, а не многообещающие предположения, которые используют, чтобы успокоить менеджера или поскорее закончить совещание.

Хороший план — это как история, которая еще не написана. В конце спринта команда сможет оглянуться назад и точно указать, что именно сделано и сколько времени потрачено на каждую задачу. Спринт окажется пройден, и независимо от того, что произошло во время этого спринта, будет

получена информация — зафиксированные факты. Если работа выполнена хорошо, то изначальный план спринта будет предельно близок к конечным результатам. Чем больше совпадений, тем точнее был план.

Это настоящее изменение в мышлении многих команд. Члены команды, использующие разработку плана как способ постановки оптимистических целей, часто не могут их достичь. А ведь одного чрезмерно оптимистичного члена или менеджера команды достаточно, чтобы разрушить планы каждого (в том числе и на следующие четыре выходных), даже если он думает, что таким образом помогает. Но если команда рассматривает план как наиболее реалистичный вариант спрогнозировать события в течение ближайших 30 дней, то у них гораздо меньше шансов «взвалить на себя больше, чем могут унести», есть возможность достичь цели вовремя, не срезая углов и не создавая хрупкий код.

Для многих менеджеров проекта это совершенно новый способ планирования. Традиционные руководители проектов воспринимают план как способ мотивировать команду и фиксировать сроки выполнения работ. Они убеждены, что план заставляет команду работать, а без него сотрудники будут бездельничать. Менеджер проекта, требующий от команды чересчур оптимистичных оценок объема работ и «утверждающий» их в начале спринта, будет чувствовать себя комфортно, используя этот план для дальнейшего третирования команды. Вот почему такое отношение к «плану работ, рабочему плану» может привести к разладу между руководителем проекта и командой.

Вместо того чтобы требовать оценки с каждого сотрудника, а затем призывать к ответственности, в эффективной scrum-команде scrum-мастер организует командную работу над каждой оценкой. Чтобы достичь максимальных результатов, scrum-мастер и команда работают вместе и могут предвидеть дальнейшие события. Совместное планирование помогает команде удерживать в памяти точную картину на протяжении всего проекта, дает ей возможность делать работу лучше и в итоге способствует поставке наиболее ценного программного обеспечения.

Члены команды чувствуют себя полностью вовлеченными в проект, когда участвуют в распределении задач во время планирования спринта, а не в ходе выполнения задания. Если сотрудники действительно испытывают чувство ответственности, *то в начале спринта нет необходимости распределять задания между ними*. В эффективных scrum-командах персонал сам распределяет задачи на основании имеющихся навыков и опыта. Это один из ключей к пониманию того, как работают самоорганизующиеся команды.

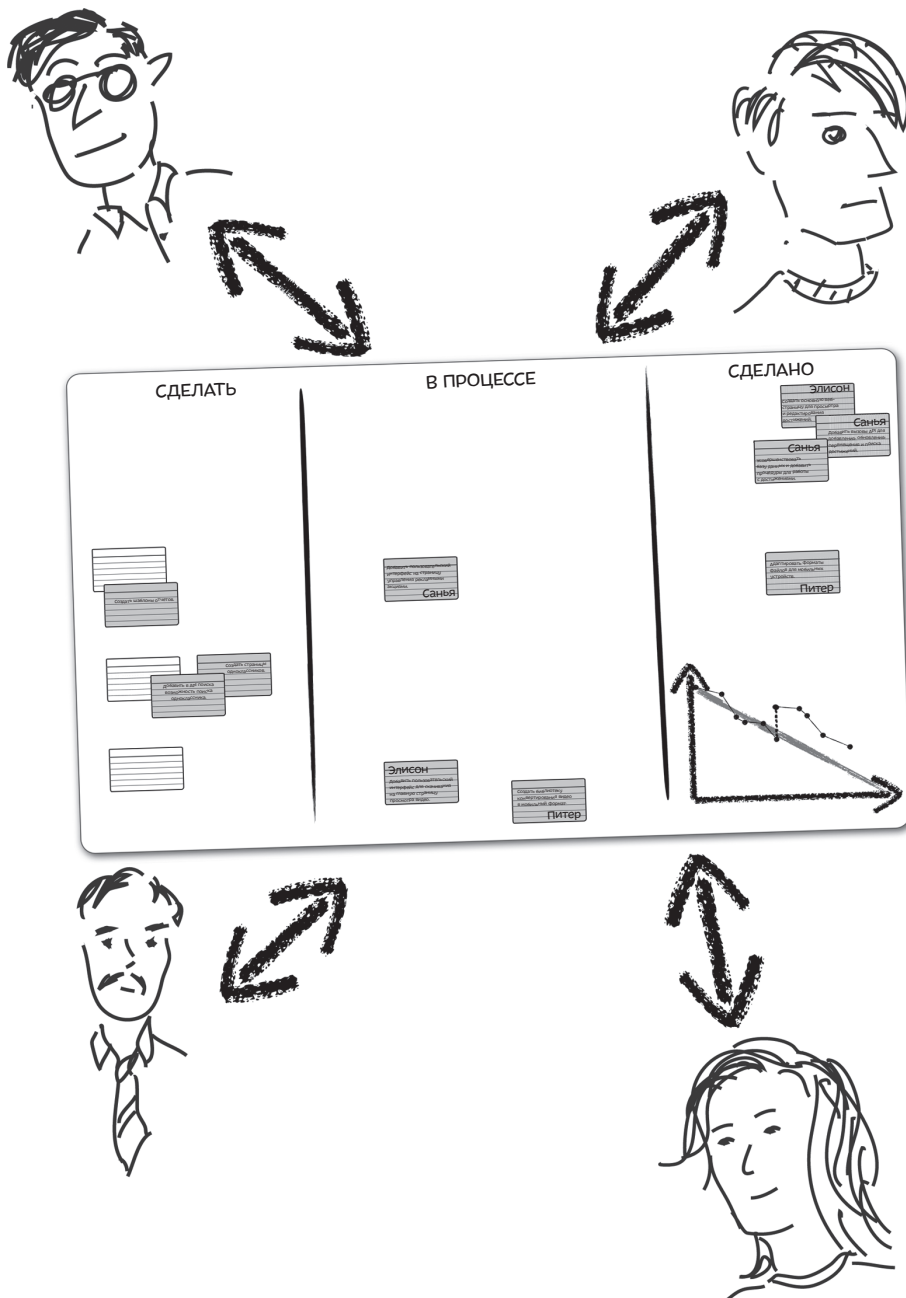


Рис. 4.4. Самоорганизующиеся команды относятся к оценке работ и планам как к неким фактам, которые могут быть выявлены, а не как к обязательствам, полученным под давлением

В успешных scrum-командах сотрудники не ограничиваются простым выполнением задач. Каждый из них — настоящий энтузиаст своего дела и старается поставлять пользователям и стейкхолдерам наиболее ценное программное обеспечение. Когда все члены команды разделяют это чувство и обязуются поставлять работающее ПО в конце каждого спринта, мы говорим о **коллективной ответственности**. Не каждый в отдельности берет на себя ответственность за задачи на микроуровне, а команда в целом обязуется поставлять ценные функции в бэклог. Это *дает команде свободу и возможность регулировать работу* по мере обнаружения в проекте новых фактов. Например, если в середине проекта Роджер и разработчики видят, что должны внести изменения в уже сделанную работу, то Ави доверит им делать это до тех пор, пока весь бэклог спринта не будет выполнен. И когда выяснится, что они ошибались (бывает и такое!) и не успевают выполнить все задачи бэклога спринта, они могут рассказать Ави, какие задачи бэклога вызывают у них беспокойство, не вдаваясь в ненужные подробности.

Это одна из основных причин успешной работы scrum-методологии, и именно поэтому так ощутима разница между высокой производительностью scrum-команды и результатом «лучше-чем-ничего».

ДЛЯ «КУРИЦ» НЕТ МЕСТА

В scrum-командах нет места для «куриц». Владельцы продукта — часть команды, а значит, они тоже должны быть «свиньями». Им это не всегда удастся, особенно если они чувствуют, что им пришлось работать в scrum-команде. Безусловно, большинство стейкхолдеров хотят быть «курицами», потому что работать с некоторой отстраненностью комфортнее. (Но иногда они хотят быть «свиньями», потому что это дает больше власти и возможности влиять на команду. Решение должен принять владелец продукта.)

Существуют способы, помогающие scrum-мастеру и команде вызвать у владельца продукта подлинное чувство ответственности. Наиболее важный — выслушать его мнение и признать, что он привносит в проект реальный опыт, необходимый команде.

Многие программисты считают, что программирование — это единственное, что нужно проекту, а все другие аспекты второстепенны. К сожалению, немало компаний разделяют эту точку зрения и выстраивают свои команды вокруг технических специалистов. Иерархическое разделение менеджеров проектов, владельцев продукта и технической команды позволяет разработчикам смотреть на всех остальных свысока и не прислушиваться к их мнению.

Владельцу продукта не обязательно разбираться в технических деталях. Каждый член scrum-команды привносит в проект навыки и знания, наиболее подходящие для него. У владельца продукта это глубокое понимание целей проекта. Чем активнее scrum-мастер и команда привлекают владельца продукта, чтобы понять эти цели и позицию владельца продукта, тем скорее он полностью посвятит себя проекту.

SCRUM ИМЕЕТ СОБСТВЕННЫЙ НАБОР ЦЕННОСТЕЙ

Каждая компания имеет собственную культуру, которая включает в себя конкретные ценности. Например, некоторые компании ценят разделение обязанностей, когда каждому отведена определенная роль и он защищен от ответственности за то, на что не может повлиять. Для других компаний важны прозрачность и свобода обмена информацией, а также то, что даже специалисты нижнего звена могут влиять на управленческие решения. Не существует единственно «правильного» способа управлять компанией. Каждая имеет культуру, которая со временем эволюционирует в зависимости от того, какой путь выбрала компания и какие решения принимала.

Каждая методология имеет заложенные в ней ценности. В главе 3 мы узнали, что конкретные agile-принципы часто связаны (или глубоко интегрированы) с определенными практиками, которые являются эффективным способом использования каждого принципа в проекте. Вам уже известно, что члены команды в компании, где решения принимают исключительно менеджеры, не могут полностью посвятить себя проекту. То же самое касается ценности или метода: если они конфликтуют с ценностями компании, то не могут быть приняты.

Но если культура компании соответствует agile-ценностям и принципам, то такая команда будет гораздо успешнее, чем командно-административная. (Это один из источников «удивительных результатов», о которых говорят agile-команды.)

Вы удивитесь, увидев, насколько agile-ценности и принципы соответствуют культуре вашей компании.

Первый шаг на пути внедрения Agile — разговор о ценностях, влияющих на культуру в вашей компании. Если обнаружилось, что внедрение гибких методов вызывает проблемы, поищите несоответствия между agile-ценностями и корпоративной культурой. Это поможет сгладить переход (или, по крайней мере, почувствовать себя лучше, понять, почему все пошло не так).

Самоорганизующиеся команды работают иначе, чем командно-административные, потому что имеют разные ценности. В уже упоминавшейся

книге Agile Project Management with Scrum Кен Швабер описывает пять scrum-ценностей: **мужество, приверженность, уважение, сосредоточенность и открытость**. Понимание того, что значит самоорганизация, начинается с изучения практической значимости принципов, которые могут быть учтены в ваших проектах.

*Каждый человек **ответствен** за цели проекта*

Такой уровень ответственности возможен, если для достижения поставленных целей команда имеет право принимать решения и каждый ее участник может высказать свое мнение о планировании и выполнении проекта. Команда проекта «Электронная книга», упомянутая в главе 3, изначально имела требование разработать шаблон для создания интернет-магазина. Для создания успешного продукта следовало проигнорировать это требование в заказе. Тогда они могли бы поставить более ценное программное обеспечение. Это можно было осуществить, если бы команда, scrum-мастер и владелец продукта сами принимали решения, избегая бюрократических процедур.

*Члены команды **уважают** друг друга*

Если в команде царит атмосфера взаимного уважения, то никто не сомневается, что каждый постарается выполнить свою работу максимально хорошо. Однако программистам и другим техническим специалистам трудно заставить себя уважать других членов команды. Многие программисты, особенно высококвалифицированные, ценят окружающих лишь за технические способности. Это может стать препятствием для эффективного внедрения Scrum. Если программист не уважает владельца продукта, то он не будет прислушиваться к его словам о цели проекта.

Хороший scrum-мастер находит способы повысить взаимное уважение в команде. Например, демонстрирует программистам, что владелец продукта глубоко разбирается в требованиях пользователей и компании в целом. Как только программисты поймут, насколько это знание полезно для успеха проекта, они начинают с уважением относиться к мнению владельца продукта.

*Все **сосредоточены** на работе*

В ходе спринта член scrum-команды полностью концентрируется на этой работе. Он может выполнять любые виды деятельности, необходимые для завершения бэклога спринта, а также создавать изменения,

внесенные в бэклог во время спринта. Когда каждый член команды фокусируется на целях спринта и имеет право делать все для их достижения, то вся команда способна самоорганизоваться и выбрать другое направление работы, если нужно что-то изменить.

В то же время команда, рассеивающая свое внимание, работает менее эффективно. Существует мнение, что современные работники, особенно программисты, успешнее справляются с многозадачной ситуацией, потому что в случае возникновения препятствий над одним проектом они могут переключиться на другой. В реальности это не так. Переключение между проектами или не связанными между собой задачами в рамках одного проекта вносит в работу хаос и требует дополнительных усилий, потому что переключение на другой контекст требует значительных когнитивных затрат. Чтобы отложить текущую работу и вернуться к предыдущей, необходимо серьезное умственное напряжение. Надо вспомнить события этого проекта и те проблемы, которые вы пытались решить. Члены команды рассказывают, что для перехода от одного задания к другому нужно время не только на его выполнение, но также и на переключение, и эти временные периоды практически равны.

Это вас не убедило? Тогда попробуйте провести мысленный эксперимент. Скажем, у вас две задачи, которые нужно выполнить в течение недели. Сделаем вид, будто благодаря удивительным причудам физических законов многофакторность не добавляет дополнительных издержек. Вы плавно переключаетесь между задачами, не тратя ни секунды на проволочки, и в этом случае работа займет ровно две недели. Даже в этих идеальных (но нереальных) условиях не стоит работать в многозадачном режиме. В обычной ситуации вы выполните одно задание за первую неделю, а другое — за вторую. Однако, работая над несколькими задачами одновременно, вы должны будете потратить хотя бы немного времени на выполнение второго задания в течение первой недели. В результате оно так и не будет выполнено на следующей неделе. Именно по этой причине многозадачность не приветствуется, даже если она вам вполне под силу (чего нельзя сказать о нас).

Многозадачность — не единственный фактор, отвлекающий команду. Придется часто посещать бесполезные собрания, участвовать в деятельности, не имеющей прямого отношения к проекту, а также оказывать помощь в работе над другими проектами. В хорошей scrum-команде участникам позволительно игнорировать эти отвлекающие факторы

без риска для карьерного роста*. (Поддержка текущего проекта может быть внесена в бэклог спринта, но только если из него предварительно что-нибудь удалили, чтобы его продолжительность не увеличилась.)

Команды ценят открытость

Члены scrum-команды всегда должны быть в курсе того, чем вы заняты и насколько это приближает проект к цели. Вот почему практики в базовой scrum-модели направлены на стимулирование открытости среди членов команды. Например, доски задач позволяют всем сотрудникам видеть, какой объем работ выполнен, а что еще предстоит сделать каждому участнику команды. Диаграммы сгорания помогают каждому оценить, насколько спринт близок к достижению целей. Успешные ежедневные scrum-совещания можно расценивать как тренировки открытости, потому что каждый в отдельности разделяет проблемы и успехи команды в целом. Все это поможет создавать атмосферу взаимной поддержки и одобрения.

Создание культуры открытости в scrum-команде — это звучит позитивно. Так оно и есть! Но зачастую это очень непросто, потому что scrum-ценности сталкиваются с уже существующей корпоративной культурой.

Во многих компаниях открытость не приветствуется, заменяется жесткой иерархией, зависящей от скрытности. Менеджеры, насаждающие такую корпоративную культуру, используют несколько способов. В отсутствие прозрачности гораздо легче сообщить команде о невыполнимой задаче («Меня не интересует, как вы это сделаете, мне нужен результат!»), заставляя людей работать сверхурочно. Это позволит менеджеру выйти сухим из воды, когда команда провалит проект («Я тут ни при чем, это они виноваты!»).

Вот почему открытость и самоорганизация при внедрении Scrum часто оказывается «третьим рельсом»**. Это центральное понятие, необходимое для правильного внедрения Scrum, но оно также требу-

* Эти советы кажутся нереальными? Задевают за живое? Если команда не имеет права игнорировать отвлекающие факторы, то, возможно, они не должны ее отвлекать. Потому что отвлекающий фактор, который нельзя проигнорировать, превращается в требование. Если вы работаете в команде, где игнорирование «отвлекающего фактора» — серьезная проблема для проекта, то мышление вашей команды может стать препятствием для внедрения Scrum и с этим придется что-то делать.

** Американский жаргонизм, означающий «вопрос, который не следует затрагивать или поднимать» (по аналогии с контактным рельсом в метро). *Прим. ред.*

ет от компании нового отношения к команде. Во всем, что касается подробностей разработки ПО, скрытный менеджер, спасающий свою шкуру, недопустим. Многие начинающие scrum-команды столкнулись с неудачей при попытке внедрить Scrum, потому что скрытные менеджеры стремились «залезть в душу» к каждому сотруднику.

Открытость угрожает «мультяшным» менеджерам, похожим на героя комикса с остатками волос на голове, торчащими вверх, и скрытным менеджерам. Но даже хорошим командам нелегко ее принимать. Взгляните на открытость с позиции разработчика, который рассматривается в качестве эксперта по некоторой части кода, или менеджера проекта — «хранителя» плана, или владельца продукта — единственного представителя команды, контактирующего с пользователями и решающего, какие функции войдут в ПО. Каждый из этих людей имеет полное право рассматривать это как свой вклад в проект. Бывает очень трудно открывать эти вещи команде и поощрять других ее членов делиться знаниями, имеющимися только у них, и вносить изменения без предварительного получения разрешения.

Это довольно распространенный способ противостоять открытости, и им нередко пользуются. Но когда этот этап пройден и каждый получает за это свою долю влияния, включая ответственность за неудачи, — подобные вещи помогают команде работать лучше, потому что это единственный способ довериться друг другу и быстро создать более ценное программное обеспечение.

*Члены команды имеют **мужество** отстаивать проект*

Когда вы отвергаете непрозрачность и выбираете открытость, вы делаете сильной команду, а не себя. Это требует мужества, но итогом становятся лучший продукт и оптимальные условия работы.

Scrum-команды отваживаются жить в соответствии с ценностями и принципами, приносящими проекту пользу. Не так-то легко отражать постоянное противодействие компании, чьи ценности не соответствуют принципам Scrum и Agile. Это требует бдительности со стороны всей команды, особенно scrum-мастера. Но важно, чтобы каждый верил: ценность поставляемого ими программного обеспечения поможет преодолеть это противодействие. Чтобы провести для руководства обзор проделанной работы, также приходится запастись мужеством. Еще оно пригодится, когда вы говорите себе: «Помощь команде в создании ценного программного обеспечения важнее, чем выпячивание моего личного вклада».

Так как же сформировать у команды мужество? Как помочь ей поверить в себя и в то, что Scrum позволит не только создавать более ценное программное обеспечение, но и продемонстрирует компании ценность новой методологии?



Ключевые моменты

Основополагающая модель scrum-проекта включает в себя следующие роли и практики: scrum-мастер, владелец продукта, команда, спринты, продукты и бэклоги спринта, ежедневные scrum-митинги, обзоры и ретроспективы.

Чтобы «осознать» Scrum, члены команды должны выйти за рамки простого внедрения практик и четко осознать, что означают самоорганизация и коллективная ответственность.

Команды рассуждают о «свиньях» и «курах», чтобы помочь разобраться в том, что это такое — «поставлять ценное программное обеспечение» — и по-настоящему почувствовать ответственность каждого за продукт, произведенный всей командой.

Чтобы стать успешными scrum-командами, необходимо глубоко усвоить scrum-ценности: приверженность, уважение, сосредоточенность, открытость и мужество.



Описание: команда, разрабатывающая приложение для мобильного телефона в небольшой компании

Роджер — руководитель команды, пытающийся использовать Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт II. Обновления статуса — это только для социальных сетей!

Вернемся к команде Lolleaderz.com. Роджеру и Ави требовалась помощь. Они знали, что в Hover Purrpu существует еще одна команда, имеющая большой опыт работы со Scrum. Роджер попытался поговорить с представителями этой команды, чтобы разобраться в непонятных вещах, но запутался

еще больше. Казалось, у них все точно так же, как и в его команде: спринты, ежедневные scrum-совещания, ретроспективы, бэклог, владелец продукта и scrum-мастер. На первый взгляд, обе команды делали одно и то же, но в одной получались отличные результаты, а другая медленно чахла. Роджер и Ави встретились с Эриком — scrum-мастером другой команды. Он достиг больших успехов в Scrum и был рад помочь товарищам найти причину их неудач.

Первым делом Эрик спросил: «Есть ли у вас коуч?» Роджер не сразу его понял. «Это наставник, — объяснил Эрик. — Человек, помогающий правильно внедрять Scrum. Лучшего способа не существует. Если бы не он, моя команда ничего бы не добилась». И здесь Роджер впервые смог оценить Ави как продавца, потому что к концу дискуссии тот убедил Эрика стать коучем в команде Lolleaderz.com.

В следующий понедельник во время scrum-митинга Роджер и Ави хотели представить Эрика команде. Но Эрик попросил, чтобы все шло как обычно — а он будет наблюдать за командой, а затем попытается придумать и сделать ей несколько небольших предложений. Хорошо, что он так сказал, потому что только половина команды успела к началу совещания — все знали, что ведущий разработчик всегда выступал первым и подробно рассказывал об обновлениях, — остальная же часть команды подошла в середине его доклада.

В оставшееся время члены команды по очереди рассказывали Роджеру о своих задачах. Все поведали ему о ходе выполнения заданий, которые должны были сделать, и спросили о следующих. Во время обновления ПО один из участников отметил, что все до сих пор ждут, когда сисадмины исправят конфигурацию на одном из веб-серверов, и спросил у Роджера, что он будет делать, чтобы это исправить. Роджер дополнил список препятствий, требующих устранения. Эрик молча наблюдал за происходящим.

На следующий день Эрик наблюдал за другим scrum-митингом, который проходил по тому же сценарию. Он заметил, что один из членов команды сообщил о выполнении задания на 95%, и вспомнил, что этот сотрудник ранее уже называл ту же цифру. Эрик сказал об этом Роджеру. «Да, похоже, он опаздывает. Не волнуйся, я контролирую ситуацию и уже обновил задание. Он постоянно затягивает сроки, поэтому я заранее учел непредвиденные обстоятельства. Если он чересчур забуксует, то я уверен, что об этом узнают Ави и генеральный директор».

В тот же день Эрик встретился с Роджером и Ави. Он начал с главного. «Роджер, ты используешь scrum-митинги для управления своим планом проекта. Что происходит, если член команды запаздывает? Ты обновляешь свой

график, и получается, что работа сделана, не так ли? За исключением того, что само по себе обновление диаграммы Ганта не уменьшает сдвиг окончания проекта на более поздний срок».

Роджеру неприятно было это услышать. Ави тоже, потому что он использовал этот график для корректировки деятельности остальных участников проекта. Эрик продолжал объяснять Роджеру, что тот использует ежедневные совещания для получения информации о задачах, выполненных командой. Ситуация накалялась. «Конечно, я использую их для получения обновлений! Ведь так и должно быть!» Роджер начал жалеть, что привлек Эрика в качестве коуча.

Вы можете объяснить, почему у Эрика возникли проблемы с Роджером и Ави, которые на ежедневных scrum-митингах получали обновленные статусы команды или вносили изменения в расписание? Если ежедневные митинги нужны не для этого, то для чего же?

Вся команда принимает участие в scrum-митингах

Ежедневные митинги — это один из наиболее эффективных инструментов, имеющихся в распоряжении scrum-команды. Они решают две важные задачи. Митинги помогают **выяснить**, справляется ли команда с работой, и адаптироваться таким образом, чтобы в результате был выпущен наиболее ценный фрагмент продукта. Это дает команде возможность принимать самостоятельные решения **в нужный момент**, придает ей гибкость в том, чтобы нужная работа делалась в правильное время надлежащим специалистом. Когда все члены команды используют ежедневные митинги, чтобы сфокусироваться на самой важной в данный момент работе, они занимаются только созданием следующего фрагмента программного обеспечения, и вся команда начинает соглашаться с тем, что митинги — это ценный инструмент, и использует его эффективно.

ОБРАТНАЯ СВЯЗЬ И ЦИКЛ «ОБЗОР-КОНТРОЛЬ-АДАПТАЦИЯ»

Многие новички в Agile привыкли считать, что мир вращается вокруг программирования. Эти разработчики проходят в офис, чтобы сделать то, для чего их наняли, и, выполнив задание, уходят домой. Это очень удобно, потому что дает возможность сосредоточиться в основном на решении технических проблем.

Но каждый программист знает, что это такое: потратить массу времени и усилий на разработку решения и вдруг в самом конце обнаружить

проблему, которую он не мог предвидеть, потому что никогда ни с чем подобным не сталкивался.

Особенно трудно бывает, когда традиционный проект выполняется по принципу «вначале детальные требования» (big requirements up front, BRUF). Представьте себе все эти многочисленные этапы, через которые должен пройти проект, прежде чем попасть к разработчику. В типичном водопадном проекте разработка BRUF выглядит примерно так.

- Руководитель проекта должен определить объем работы, чаще всего исходя из бизнес-требований или документированных целей.
- Руководитель проекта должен «подписаться» под объемом работ.
- Бизнес-аналитик должен изучить цели и объем работ, а затем выяснить у пользователей и других стейкхолдеров, как они планируют использовать создаваемый продукт и какая проблема решается.
- Бизнес-аналитик продумывает варианты использования, функциональные требования и т. д.
- Программист принимает требования и делает оценки работ.
- Менеджер проекта принимает требования и оценки, строит график и согласовывает его со стейкхолдерами и руководством.

Эта длинная цепь событий должна произойти, прежде чем начнется разработка. Неудивительно, что такая «игра в испорченный телефон» мешает разработчикам получить правильную информацию.

Такая проблема характерна не только для водопадных команд. Даже команды, практикующие личное общение, будут сталкиваться с непониманием и недостаточной коммуникацией. Ведь общение лицом к лицу эффективно, но менее конкретно, чем письменная коммуникация. В результате устных переговоров три человека могут подумать, что достигли консенсуса, а на самом деле имеют три разных мнения о предмете обсуждения.

Известная поговорка гласит: «Солнечный свет — лучшее дезинфицирующее средство». Она может казаться сомнительной с точки зрения медицины, но это очень хороший совет для проектных команд. Лучший способ узнать, создает или нет команда ценное программное обеспечение, — это как можно чаще поставлять работающее ПО пользователям. Это называется **обзором (или прозрачностью)** и также относится к коммуникации.

Ежедневные митинги — это эффективный инструмент создания прозрачности, потому что он решает проблемы коммуникации. Если эти проблемы происходят, то становятся видимыми. Возьмем, к примеру, ситуацию,

когда три разных человека уверены, что договорились, но все имеют разное представление о предмете договоренности. Что делать, если каждый из них начнет делать разные функции, которые нужно будет интегрировать в конце спринта? Такие мелкие неурядицы, вызывающие дополнительные проблемы, накапливаются, если их своевременно не устранять. Они возможны на протяжении всего проекта и приводят к постоянному ухудшению качества кода, который создает команда. Небольшое недоразумение становится причиной дефекта или даже двух. Если они обнаружены поздно, то придется делать заплатки либо большие изменения в коде. Именно так происходит во время длительного проекта: качество кода со временем ухудшается.

Но результаты будут другими, если этим трем людям достаточно 15 минут ежедневного общения, чтобы задать друг другу три вопроса.

- Что я сделал с момента нашей последней встречи?
- Что я планирую сделать к нашей следующей встрече?
- Какие препятствия есть на моем пути?

Когда члены команды каждый день рассказывают о своей работе, многие проблемы, вызванные недопониманием, успевают заявить о себе прежде, чем станут трудноустранимыми (и дорогостоящими!). Когда каждый член команды **контролирует** работу остальных, они сохраняют взаимопонимание по поводу целей проекта и того, как их достичь.

Во время ежедневных планерок каждый рассказывает, какую задачу сейчас решает, а остальные вносят предложения по улучшению работы. Если предложения удачные, то на следующий день работа будет сделана лучше. Команда также может обнаружить, что сотрудник трудится не над той задачей. Обычно это вызвано проблемами с коммуникацией. Команда меняет план работы в нужном направлении уже на следующий день. Такие изменения называются **адаптацией**. Ежедневный цикл обзора, контроля и адаптации позволяет командам непрерывно получать обратную связь о ходе проектов и улучшать качество ПО. Это одна из наиболее важных особенностей Scrum. Scrum-команды принимают решения, основываясь на опыте ведения проектов и на реальных, известных фактах*.

Такая обратная связь исключает из scrum-команды «посредника», которым может быть менеджер проекта, не принимающий в нем непосредственно-

* Студенты, изучающие теорию Scrum, относятся к этому как к *эмпирическому процессу управления*. Для получения дополнительной информации об эмпиризме см. с. 4 уже упоминавшегося выше Scrum Guide.

го участия. Таким образом снижается ущерб от игры в испорченный телефон, сохраняется время при одновременном повышении качества. Этот цикл называется **петлей обратной связи**, и команда может его использовать, чтобы поддерживать выполнение проекта и быть уверенной: все одинаково понимают суть вопроса.

В подобной ситуации программистам не всегда удобно «наблюдать» за работой товарищей по команде. Однако даже самые заядлые интроверты, как правило, к этому привыкают и с нетерпением ждут ежедневных митингов. Потому что это самый эффективный способ управления коммуникацией и формирования единого понимания командной работы.

ПОСЛЕДНИЙ ОТВЕТСТВЕННЫЙ МОМЕНТ

Речь идет о размышлениях умного, скептически настроенного командно-административного менеджера проекта, которые выглядят примерно так: «Хорошо, мы так сделали. Команды не утверждают, что наперед знают все о своих scrum-проектах. Коммуникация и общее понимание важны для команды. Но работу нужно выполнять и необходимо определить, кто этим займется. Как это происходит на практике? Какова реальная механика работы с задачами программирования, администрирования баз данных, тестирования или другими задачами, которые мы можем внести в свой список для работы?»

Существует распространенный метод, который используют scrum-тренеры и agile-коучи при обучении команд проведению ежедневных митингов. Суть его заключается в том, чтобы позволить команде потерпеть неудачу*. Люди, привыкшие к командно-административной системе, ждут от ежедневных встреч, что менеджер проекта или scrum-мастер возьмут управление в свои руки, выяснят у каждого статус проекта и укажут новые задачи. Команда считает такой способ работы естественным, она привыкла получать задание. Agile-коуч готов помочь команде проводить ежедневные митинги продуктивнее, поэтому он может посоветовать scrum-мастеру хранить молчание. Часто это приводит к неловкой паузе, которая длится минуту или две. Затем кто-нибудь начинает говорить о работе, которую проделал с момента последней встречи. Большая удача, если за этим рассказом последует вопрос «Какова моя следующая задача?».

Именно в этот момент scrum-мастер осознает свое отличие от менеджера проекта. Менеджер продолжит распределять между членами команды

* Эта и другие эффективные коучинг-практики описаны в уже упоминавшейся книге Лиссы Адкинс *Coaching Agile Teams*.

задачи, которые заранее подготовил. А scrum-мастер воспользуется ситуацией, чтобы организовать команде момент просветления, когда она наконец поймет, что надо делать дальше. Он мог бы это сделать, задавая вопрос «Что ты собираешься делать дальше?». Или он может просто молчать в зависимости от того, с какой командой имеет дело.

Дело в том, что сами члены команды — это ресурс, распределяющий задания. Каждый человек *самостоятельно берет следующую задачу, как только закончит предыдущую*.

Это происходит во время ежедневных scrum-митингов, когда остальная часть команды имеет возможность внести свой вклад и помогает исправить курс. Например, если разработчик берется за оптимизацию сложной базы данных, то администратор баз данных (DBA) может вмешаться и предложить отложить решение этой задачи на более поздний срок.

НО РАЗВЕ МЫ НЕ МОЖЕМ ЗАРАНЕЕ ИЗБЕЖАТЬ УЗКИХ МЕСТ ПРИ ПЛАНИРОВАНИИ РАБОТЫ?

Командно-административное управление проектом начинается с предположения, что задачи по развитию проекта должны быть выполнены определенным членом команды. Причина, как правило, в наличии специальных знаний (например, администратор баз данных с навыками оптимизации БД). Это выглядит как аргумент в пользу предварительного планирования: создает узкое место в потоке работ, и команда, имеющая фиксированный крайний срок выполнения работы, должна учитывать это при планировании.

Как ни странно, это самый распространенный источник проблем в области управления проектами. Очень сложные задачи оценить гораздо труднее, чем простые. Задания, требующие конкретного специалиста или ресурса, несут больше риска, чем те, которые могут быть выполнены любым членом команды. Не стоит удивляться, что сложные задачи, подвластные лишь конкретному квалифицированному человеку (как в примере с оптимизацией баз данных), скорее всего, будут реализованы с ошибками. Хуже всего то, что менеджеры проектов зависят от этого узкого специалиста как на этапе оценки, так и в процессе выполнения работ.

Невозможно все просчитать заранее. Существует несколько решений, принимаемых в начале проекта (Java или C#? Windows или Linux? Mac или PC?). И есть еще ряд задач, которые необходимо сделать конкретному человеку. Но вообще scrum-команды не распределяют задачи в начале проекта или спринта.

Они не пытаются придумать «окончательную» последовательность задач. Причина в том, что в большинстве случаев задач, и особенно задач по программированию, команда не знает точно, сколько ей потребуется времени, пока она не приступит к работе, и зачастую она обнаруживает зависимость только тогда, когда они становятся явными. Это также относится к заданиям, которые кажутся небольшими в начале, но становятся объемными в конце (и наоборот). Конечно, хотя перед командой и стоит общая задача обнаружить, что она пропустила во время спринта, это не снимает с нее ответственности за разработку настолько сложного или окончательного списка задач, насколько это возможно, во время планирования спринта.

Поэтому вместо декомпозиции работ на задачи в начале спринта, упорядочивания задач, распределения их между членами команды перед началом любой работы и отслеживания плана agile-команды следуют простому правилу планирования. Они принимают все решения в **последний ответственный момент***.

Вернемся к истории проекта Lolleaderz.com. Роджер и Ави столкнулись с проблемой на четвертом спринте, потому что администратор баз данных затянул с выполнением одной из тех специализированных задач, которая, казалось, была полностью и официально распланирована с первого дня. Эта общая причина провала проекта вызвана чрезмерным планированием. Руководитель проекта предполагает, что комплектованием задач будет заниматься один человек — как правило, эксперт, обладающий специальными навыками. Обычно именно эти задачи имеют больше шансов «провалиться». Когда это происходит, никто другой не может справиться с этими заданиями, поэтому начинаются каскадные задержки, сверхурочная работа, которая неизбежно снижает качество результата (и возможно, приводит к поиску таким специалистом новой работы!).

Scrum-команды, принимающие решения в последний ответственный момент, могут справиться с этим по-разному. Вместо того чтобы предполагать, что администратор баз данных вовремя выполнит все эти задачи, они выписывают их на карточки (или какой-нибудь электронный эквивалент) и вывешивают на доску задач в колонку «выполнить». Во время ежедневной планерки их видят все и кто-нибудь, как правило, задает вопросы, если предполагает, что задача может вызвать проблемы в ходе спринта.

Команды, работающие с открытым исходным кодом, используют поговорку «При достаточном количестве глаз все ошибки лежат на поверхности»

* Термин lean- и канбан-разработки, означающий критическую точку невозврата, до достижения которой система допускает разные варианты решения конкретной проблемы.
Прим. ред.

(закон Линуса). То же самое относится к плану. Ежедневные планерки — это способ команды полностью сконцентрироваться на деле. Когда проектная работа рассматривается так же внимательно, как и исходный код, появляется больше шансов найти ошибки в плане.

Команде гораздо проще обнаружить узкие места во время митингов, чем в ситуации, когда командно-административный руководитель проекта заранее все за всех продумал. И когда команда видит, что приближается к этим узким местам, у нее есть время, чтобы найти обходные пути. Благодаря бдительности всех участников часто обнаруживается, что последний ответственный момент для некоторых задач находится на более ранних этапах спринта, чем для других. В этом ценность цикла «обзор-контроль-адаптация»: он дает возможность выявить потенциальные проблемы и принимать решения всей командой.

Что бы произошло, если бы Роджер и Ави эффективнее проводили ежедневные митинги? Вместо распределения заданий они тратили бы время на совместную работу с командой, чтобы выявить проблемы графика и самостоятельной постановки задач. Работая как одна команда, они могли бы быстрее понять, что им нужен кто-то другой, кроме DBA, чтобы начать работать над хранимыми процедурами, чтобы успеть выполнить эту работу до окончания спринта. Или, по крайней мере, выяснили бы, что «взвалили на себя больше, чем могут унести», и успели бы вовремя уточнить, будет ли работающее программное обеспечение поставлено к концу спринта.

КАК ПРОВЕСТИ ЭФФЕКТИВНЫЙ ЕЖЕДНЕВНЫЙ SCRUM-МИТИНГ

Берите пример со «свиньи»

Во время этой встречи каждый член команды несет ответственность перед коллегами и должен объяснить, почему обязательство, взятое на предыдущей встрече, не выполнено. Представьте себя членом самоорганизующейся команды, действительно чувствующим ответственность за проект. Что нужно знать, чтобы делать нужную работу каждый день? Прежде всего — понимать, над чем вы работаете. Но если ваша команда действительно самоорганизующаяся, то вы не можете полагаться на некоего командно-административного менеджера проекта, все решающего за вас. Нужен другой механизм, чтобы получить текущее задание. Первое, что дают ежедневные scrum-митинги, — это ваша следующая задача. Если наступает очередь ответить на вопрос, что вы будете делать начиная с сегодняшнего дня и до следующей планерки, то в случае, когда вы уже выполнили все необходимые работы,

ваша обязанность — взглянуть на задачи, которые все еще находятся в столбце «выполнить», и выбрать наиболее значимую для вас и для проекта. Если вы сделали неправильный выбор, то тот, кто действительно предан позиции «свиней», скажет вам об этом.

Ведите дискуссии о задачах вживую

Ежедневные митинги нужны для выявления проблем, а не для их решения. Если не удалось решить проблему в течение пары минут во время дискуссии, запланируйте следующую встречу с теми, кто должен участвовать в решении. Многие повторные встречи будут посвящены тому, кто станет выполнять эти задачи. Именно так работают самоорганизующиеся команды: большинство задач может быть назначено самостоятельно, но некоторые из них требуют обсуждения. Только в процессе ежедневных митингов можно сказать, какие задачи требуют обсуждения, а какие нет.

Чередуйте право направлять работу команды

В проекте нет единственного «хранителя плана» или одного наиболее значимого человека. Очевидно, что некоторые разработчики опытнее других. Но хорошие идеи могут прийти в голову даже самому неопытному сотруднику, и не стоит отмечать их только на этом основании. Свежий человек иногда находит серьезную проблему в задачах, с которыми предстоит иметь дело команде. Чтобы люди внимательнее слушали друг друга и не пропускали мимо ушей ценные предложения, можно привлечь члена другой команды, чтобы он начал ежедневную планерку.

Не воспринимайте это как ритуал

Хотя мы и проводим эти встречи каждый день (и оттого некоторые scrum-команды считают их «церемониалом»), каждый должен присутствовать и активно участвовать. Довольно легко считать ответы на три вопроса (что я сделал с момента последнего scrum-митинга? что я буду делать дальше? какие есть препятствия на моем пути?) ритуалом, который просто нужно соблюдать, не задумываясь о причинах. Со временем ритуалы обычно исчезают, потому что приобретают формальный характер. Но три упомянутых выше вопроса — это основа ежедневных митингов. Они нужны, чтобы выявить проблемы на ранних этапах проекта. Например, эффективный способ определить узкие места, вызванные слишком большим количеством задач, возложенных на одного человека, — это выслушать его ответ на вопрос о том,

что мешает его успеху. Потому что он способен обнаружить это узкое место раньше, чем другие.

Каждый принимает участие

Каждый — это значит и тестировщики, и бизнес-аналитики, и остальные члены команды, и владелец продукта. Все они действительно имеют обязательства. Владелец продукта выполняет особенно важную работу, потому что держит всех в курсе задач бэклога, наиболее существенных для пользователей и компании. Чем лучше команда понимает значимость поставляемого продукта, тем точнее она может удовлетворить запросы пользователей. Владелец продукта вместе с остальной командой также должен ответить на три вопроса, потому что он может забыть о важности своей задачи, а ответы напомнят об этом. (Оказывается, что посвящать все свое рабочее время переговорам с пользователями, разбираться в их бизнесе и руководить бэклогом — это занятие, достойное уважения разработчиков, если они видят это воочию!)

Не относитесь к этому как к статус-митингу

Типичный статус-митинг — это отличный пример «ритуала», который мы должны совершать каждую неделю. Все мы знакомы с ритуалами и совершаем их, не задавая лишних вопросов. Предполагалось, что совещание служит двум целям: информировать команду и администрацию. Но для большинства команд это действительно всего лишь односторонняя связь, связка двух человек — члена команды и руководителя проекта. Вы можете избежать этого, убедившись, что каждый присутствующий на ежедневных scrum-планерках действительно слушает выступающих. (Это означает, что не нужно проверять электронную почту, мобильный телефон и контролировать выполнение работы!) Как только члены команды начинают понимать, что ежедневные scrum-митинги реально помогают выявлять проблемы на раннем этапе и сохранять время разработчика, они признают: в этих встречах нет бюрократической волокиты, они похожи на ориентированный на разработчика инструмент, который можно использовать, чтобы сделать код лучше.

Проверьте каждую задачу

Когда вы ищете препятствия, анализируйте не только текущую работу. Просчитывайте ситуацию на несколько ходов вперед, исследуя каждый элемент в колонке «сделать», чтобы выяснить, повлекут ли

они последствия. Если есть потенциальная проблема, то лучше обсудить это с командой. Иначе впоследствии можно «обжечься». Поэтому проверка задач требует доверия между всеми участниками команды. Если кто-то — намеренно или случайно — недостаточно точно описывает то, над чем работает и что планирует сделать, то остальные члены команды могут не заметить вовремя потенциальное препятствие, которое может привести к серьезным проблемам.

Измените план, если это необходимо

Адаптация как часть цикла «обзор-контроль-адаптация» — это эффективный способ самоорганизации. Допустим, команда выявляет препятствие во время ежедневного scrum-митинга и на следующей встрече понимает, что допустила серьезный просчет и не в состоянии выполнить поставку обещанной главной функции. Есть ли смысл продолжать придерживаться прежнего плана, который явно не будет работать? Конечно, нет. Бэклог и доска задач должны отражать реальный проект. Если проблема обнаружена, то вся команда должна работать вместе, чтобы исправить ситуацию. В этом случае очень кстати придется владелец продукта — «свинья». Именно он способен немедленно внести изменения в ожидания остальных людей. Помните: если люди негативно отреагируют на изменения, о которых узнали сегодня, то изменения, обнаруженные позже, они воспримут гораздо хуже.



Ключевые моменты

Во время ежедневных scrum-митингов каждый член команды отвечает на три вопроса: что я сделал с момента последнего митинга? Что я буду делать, начиная с сегодняшнего дня и до следующего митинга? Какие есть препятствия на моем пути?

Команда использует эти вопросы, чтобы каждый день совместно проверять план проекта и адаптироваться к изменениям, а также для получения постоянной обратной связи на протяжении всего проекта.

Успешные scrum-команды принимают решения в последний ответственный момент, поэтому они не делают лишнюю работу и легко адаптируются к изменениям.

Ежедневные scrum-митинги принадлежат всей команде, а не только scrum-мастеру или владельцу продукта, и все в равной степени в них участвуют.



Описание: команда, разрабатывающая приложение для мобильного телефона в небольшой компании

Роджер — руководитель команды, пытающийся использовать Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт III. Спринтерский забег в тупик

После затянувшегося обеда Эрик, Роджер и Ави разговорились о том, как использовать ежедневные scrum-митинги, и у Роджера появилась идея. На следующем митинге он попросил девушку — младшего разработчика начать встречу с ответов на три вопроса. Когда она спросила Роджера о следующей задаче, он промолчал. Молчание длилось приблизительно полминуты и вызвало всеобщее замешательство. Роджер уже засомневался, была ли эта идея действительно удачной, но тут один из старших разработчиков заговорил. После краткого совместного обсуждения младшая разработчица уже знала, что делать дальше. Она сняла с доски задач карточку, которая находилась в колонке «сделать», написала на ней свое имя и прикрепила ее в колонку «в процессе выполнения».

Последующие scrum-митинги прошли еще удачнее. Казалось, что для этого потребовалась всего одна дискуссия, и у команды вдруг начало получаться — все заговорили о задачах друг друга, и понадобилось лишь запланировать еще два обсуждения, чтобы определить, кто чем должен заниматься. Роджер был приятно удивлен, что ему придется принимать участие только в одном из них. Теперь ему предстояло заняться разработчиком, который регулярно выполнял задачи на 95%. Оказалось, что этому разработчику требовалась серьезная помощь, но он не решался попросить о ней, чтобы не занимать время коллег (кроме того, вероятно, стеснялся признаться, что у него проблемы).

После нескольких планерок Эрику удалось порадовать Роджера: члены команды начали привыкать работать вместе. В течение следующей недели стало очевидно, что идея самоорганизации реализовалась — и они сделали это вместе как команда. Ежедневно все члены команды определяли объем работы на следующий день, помогали друг другу придерживаться выбранной стратегии и решать проблемы. Они использовали scrum-митинги для корректировки курса как команда, которая берет за основу планирования ежедневный обзор выполняемой работы.

Казалось, все шло отлично. Так было вплоть до конца спринта. Команда представила новую версию работающего программного обеспечения, точно так же как это было после предыдущих шести спринтов.

Это была катастрофа.

Ави вернулся с очередного совещания стейкхолдеров крайне разочарованный. Он ожидал, что менеджеры по обслуживанию клиентов будут в восторге от возможностей новой версии редактора Lolleaderz.com, который позволял пользователям создавать записи о собственных достижениях и делиться ими в социальных сетях. Кроме того, команда обновила функционал рекламного баннера, чтобы любой аккаунт-менеджер имел индивидуальную страницу для каждого аккаунта, показывающую самую последнюю информацию о просмотрах и расходах на рекламу.

Вместо этого большинство менеджеров выразили крайнее недоумение. Их не проинформировали, что будет столько изменений. Неожиданно каждый из них получил десятки голосовых сообщений от клиентов, в которых те интересовались новыми функциями. Раньше они пользовались графиками Роджера, где было много предварительной информации, помогающей им продавать новый функционал. Но мир вокруг менялся слишком быстро, и они почувствовали, что не успевают за ним.

У Ави новости были еще хуже. Некоторые стейкхолдеры попросили вернуть старые графики и поинтересовались, не может ли команда перенести выпуск части опций на следующий квартал. Все выглядело так, будто компания решила полностью прекратить использовать Scrum и вернуться к водопадному процессу.

Но так ли это на самом деле? Эрик слышал те же новости, что и Роджер, но выказывал необычайный оптимизм. Как вы думаете почему?

Спринты, планирование и ретроспективы

Для некоторых проектов планирование спринта — это легко. Например, когда нужно в итоге создать то, о чем люди просят на протяжении нескольких месяцев. Если есть функция, которую требуют пользователи, и вы делаете ее высокоприоритетной, то это легкая победа. В этом случае планирование спринта — это просто следование здравому смыслу.

Но порой возникают сложности. Часто во время планирования нужно думать о том, чего хотят пользователи, что ценят, и при этом выясняется, что никогда раньше вам не приходилось об этом задумываться. Когда люди говорят, что Scrum сложен, то они имеют в виду именно это.

К счастью, успешные scrum-команды имеют для решения этой проблемы не самое секретное оружие — владельца продукта. Когда этот человек действительно тратит время на попытки понять, чего хотят стейкхолдеры и что представляет для них ценность, он может помочь команде определить цели каждого спринта и те проблемы, которые компания должна решать в первую очередь. Визуализируя эти ценности для команды и участвуя в составлении нового плана для каждого спринта, он превращает поэтапный процесс в непрерывный. И когда в конце каждого спринта команда проводит ретроспективу, владелец продукта помогает ей извлечь уроки, чтобы ожидания команды соответствовали тому, чего она на самом деле может достичь.

ИТЕРАТИВНЫЙ ИЛИ ИНКРЕМЕНТАЛЬНЫЙ?

Какова ценность каждого нового релиза, получаемого в конце спринта?

Если вы планируете ограниченные по времени спринты, то придерживайтесь такой стратегии: когда время спринта подходит к концу, останавливайте всю работу по разработке, поскольку если команда будет поставлять работающее программное обеспечение в конце каждого спринта, то подобная стратегия принесет существенную пользу. Вы получите стандартные контрольные точки, помогающие поддерживать высокое качество ПО. Это позволяет владельцу продукта, пользователям и заинтересованным сторонам увидеть функционально полные версии, в которых встроенные опции хорошо сочетаются друг с другом. Таким образом значительно снижаются риски, потому что команде не придется дожидаться окончания проекта, чтобы интегрировать функции, созданные разными людьми, и обнаружить, что они не работают вместе.

Проведем мысленный эксперимент, который поможет понять проблемы интеграции. Представим себе, что два члена команды трудятся над разными функциями программы, сохраняющими текущий рабочий файл пользователей, но делают это по-разному. Знаете ли вы, сколько вариантов «конфликтных ситуаций» может возникать между этими функциями? Вот лишь некоторые из них: одна функция использует значок «сохранить», а другая — «файл» в меню, две характеристики имеют несовместимые способы доступа к общим ресурсам, или сохраняют файлы в несовместимых форматах, или могут перезаписать общие данные, которые управляют приложением. Возможны и другие проблемы интеграции. Можете ли вы вспомнить иные потенциальные проблемы, возникающие в процессе интеграции? Если вы давно разрабатываете ПО, то вам не нужно ничего придумывать, так как наверняка приходилось не раз сталкиваться с подобными вещами.

Объединяя все разработки вместе в конце каждого спринта, не дожидаясь для этого окончания проекта, команда может распознать и предотвратить многие подобные проблемы. Существуют и другие преимущества такого подхода: улучшение коммуникации, более активное участие стейкхолдеров и легко измеримый статус проекта. Разбивка проекта на этапы называется **инкрементальной разработкой**. Scrum-спринты — еще один способ разделить ваш проект на инкременты. Вот почему Scrum — это инкрементальный подход.

Но Scrum — это нечто большее. Scrum-спринты — не просто поставка работающего программного обеспечения в ограниченные сроки. Это также понимание ценности, которую данный программный продукт обеспечивает, точно определяя, какая именно ценность будет поставлена, и осознание необходимости смены курса, если есть способ, позволяющий поставить большую ценность. Когда методологии или процессы, такие как Scrum, работают таким образом, это называется **итеративная разработка**. То есть Scrum — это одновременно и инкрементальный, и итеративный подход.

Майк Кон в своей замечательной книге «Пользовательские истории. Гибкая разработка программного обеспечения»^{*} хорошо объясняет главные отличия между понятиями «итеративный» и «инкрементальный»:

Итеративный процесс позволяет достигать прогресса за счет последовательных уточнений. Команда разработчиков берет первый отрезок системы, зная, что он неполный или слабый в некоторых (а возможно, во многих) областях. Затем она многократно совершенствует эти области до тех пор, пока продукт не станет удовлетворительным. С каждой итерацией программное обеспечение улучшается путем добавления дополнительных деталей.

В инкрементальном процессе программное обеспечение создается и поставляется частями. Каждая часть, или инкремент, представляет собой полный ряд функций. Инкременты могут быть маленькими или большими, возможно, это будет только логин системы на экране или очень гибкий набор управления данными экрана. Каждый инкремент полностью закодирован и протестирован, и считается, что работу, сделанную в этой итерации, уже не нужно будет пересматривать.

У нас уже есть план мероприятий для принятия первой части системы, пусть и известно, что она неполная или слабая в некоторых областях, и мы

^{*} Кон М. Пользовательские истории. Гибкая разработка программного обеспечения. М. : Вильямс, 2012.

в несколько приемов усовершенствуем эти области, используя цикл «обзор-контроль-адаптация». Scrum-команда, применяющая инкрементальную разработку, так же как она применяет цикл «обзор-контроль-адаптация» во время ежедневных scrum-митингов, использует эту технологию для проекта в целом. В этом цель планирования спринтов, управления бэклогом спринта и проведения ретроспектив.

Вот почему владелец продукта так важен для scrum-команды и ему отводится отдельная роль. Задачи владельца продукта:

- выявить главные потребности компании и транслировать их команде разработчиков;
- понять, какие именно функции программного обеспечения команда потенциально может поставить;
- выяснить, какие функции наиболее ценны для компании, а какие второстепенны;
- работать с командой, чтобы понять, какие функции создавать легче, а какие сложнее;
- использовать понятия ценности, сложности, неопределенности, комплексности и прочие, чтобы помочь команде выбрать нужные функции для создания продукта в каждом спринте;
- делиться этими знаниями с остальными работниками компании, чтобы они могли сделать все необходимое для подготовки следующей версии программного обеспечения.

ВЛАДЕЛЕЦ ПРОДУКТА ЗАПУСКАЕТ И ОСТАНАВЛИВАЕТ СПРИНТ

Владелец продукта выполняет очень специфические функции в проекте. Он владеет продуктовым бэклогом и выявляет наиболее приоритетные задачи, которые команда будет разрабатывать в следующем спринте. Вместе с командой участвует в планировании спринта и решает, какие из пунктов войдут в бэклог спринта, и принимает те задачи, которые были выполнены командой по поручению компании. Это означает, что владелец продукта должен иметь широкие полномочия. Если их нет (или он боится это делать), значит, человек занимает чужое место. Он также должен четко понимать, что ценно для компании. Разговаривая с людьми, владелец продукта выясняет их мнение о том или ином продукте, но окончательное решение обо всех приоритетах продуктового бэклога принимает только он.

Вот почему так важно, чтобы команда и владелец продукта с самого начала спринта договорились о том, из чего состоит каждый из этих пунктов. Когда они планируют бэклог спринта, все должны прийти к единому мнению

о том, что означает слово **«выполнено»** для каждого пункта — это не просто «дело сделано», а «Выполнено» с большой буквы. Бэклог задач считается «выполненным», когда его принимает владелец продукта и весь результат можно поставлять клиентам. Если нет однозначного определения понятия «выполнено» по каждому пункту, то неизбежны путаница и жаркие споры в конце спринта. Но когда все имеют четкое представление о том, что это означает для каждого пункта, то команда отчетливо видит, как продвигается спринт.

Спринты ограничены по времени — обычно 30-дневным сроком (хотя некоторые scrum-команды выбирают короткие спринты — длиной в две-три недели). Когда спринт заканчивается, все «выполненные» пункты принимаются владельцем продукта. Любые пункты в статусе «не выполнено» возвращаются в продуктовый бэклог — даже если команда сделала многое (или почти все), работая над ними. Это не значит, что команда должна все начать сначала, удалить исходный код или что-нибудь отменить. Просто работа над этим пунктом не закончена до тех пор, пока он действительно не будет «выполнен» и принят владельцем продукта.

Это важно, поскольку гарантирует, что пользователи никогда не будут заблуждаться относительно того, поставлена ли ценность командой или нет. Лучше осторожничать при принятии обязательств. Обзор спринта — это специальное мероприятие, когда вся команда выступает перед пользователями и заинтересованными сторонами, чтобы продемонстрировать результаты своей работы за последние 30 дней. Если обещания не выполнены, то каждый член команды должен напрямую объяснить пользователю, что он делал и почему не справился. Это очень мощный инструмент, помогающий каждому почувствовать коллективную ответственность. Кроме того, именно в этой ситуации пользователи и стейкхолдеры могут задавать вопросы. Такое взаимодействие помогает всем объединиться и гораздо лучше понять, что действительно ценно, — и они могут начать создавать чувство настоящего доверия. Чем чаще люди встречаются и говорят о программном обеспечении (как в этом случае), тем больше доверия и свободы будет в команде и ей легче будет создавать ПО. Хотя пользователи и стейкхолдеры присутствуют на встрече и обсуждают ситуацию с командой, только владелец продукта имеет право принимать работу от имени компании.

Изредка владелец продукта и команда обнаруживают, что либо спринт был плохо спланирован, либо произошли серьезные изменения, которые не могут ждать окончания спринта. В этом случае владелец продукта имеет право остановить спринт, прекратить все работы и переместить все пункты из бэклога спринта в бэклог продукта. Подобное явление должно быть чрезвычайно редким, так как оно может разрушить с трудом заработанное доверие между командой, пользователями и стейкхолдерами.

ОБЗОР И ЦЕННОСТЬ

Подумайте, что вас мотивирует, когда вы работаете. Как часто мысли, подобные перечисленным ниже, приходят вам в голову?

- «Опыт работы с этой технологией украсит мое резюме» (если вы разработчик).
- «Если я проявлю себя в этом проекте, то могу получить в команде более высокую должность» (если вы руководитель проекта).
- «Выполнение такой сложной задачи точно в срок прославит меня» (если вы менеджер проекта).
- «Если мне удастся удовлетворить такого крупного клиента, то я получу огромный бонус» (если вы менеджер по работе с клиентами, владелец продукта, стейкхолдер и т. д.).

Мы все так думаем. И это нормально.

У любого человека есть свои собственные интересы, и в этом нет ничего плохого. Но персональная мотивация не объединяет команду. Когда люди работают вместе над единой задачей, они могут достичь гораздо большего, чем каждый по отдельности. Так что, пока каждый из нас беспокоится только о том, что может дать ему работа, которую он выполняет, мы добиваемся намного меньшего, чем если бы все работали как одна команда.

Приведем пример, как личные интересы могут нанести ущерб проекту. Предположим, в команде есть человек, который может переложить неинтересные или раздражающие его задачи на другого, обычно ниже его по статусу. Большинство из нас сталкивались с такой ситуацией. Например, старшие разработчики настолько заняты созданием нового ПО, что сняли с себя обязанность исправлять ошибки. Ведь заниматься разработкой новых функций гораздо интереснее, тем более если есть возможность ознакомиться с новыми технологиями. К тому же обычно существует команда младших разработчиков (зачастую в другом городе, где меньше платят), готовая заняться исправлением ошибок. Вообще это довольно распространенная практика: набирается «команда» опытных разработчиков для создания новых функций и группа поддержки, куда входят менее опытные специалисты, которые исправляют ошибки и ставят «заплатки» в уже выпущенное программное обеспечение.

Это очень удобно для старшего разработчика, желающего «выстрелить и забыть». Ведь он уверен: допущенные им погрешности будут устранять другие люди, которых он и знать не желает. В краткосрочном плане это очень неплохой способ работы для одного человека, но с точки зрения команды

и в долгосрочной перспективе это неэффективно. Без всякого сомнения, ошибки должен исправлять тот, кто их сделал. Он уже знает детали кода, потому что писал его и хорошо в нем разбирается. Чтобы исправлением занялся кто-то другой, нужно затратить усилия на коммуникацию. Правда, иногда достаточно отправить письмо по электронной почте, но порой необходимы дополнительные документы (например, отчет об ошибке и обновленная спецификация). Человеку, который займется исправлением, потребуется время, чтобы разобраться в коде. Программист, создавший его, исправит погрешность за несколько минут, а другим понадобятся часы или дни (особенно если это новички или недостаточно опытные разработчики).

В scrum-командах редко встречается ситуация, когда грязную работу спихивают на другого, потому что все сотрудники привержены общему делу. Командная культура предполагает, что более опытный разработчик сам делает грязную работу за несколько минут, а не передает ее младшему коллеге, который потратит часы. Это и есть один из источников гиперпроизводительности и «удивительных результатов» применения Scrum, которые, похоже, недоступны многим командам.

Любая (даже не гибкая) команда может стать высокопроизводительной, если введет правила, запрещающие своим старшим членам передавать «скучные» задачи младшим по должности. Но по-настоящему успешной scrum-команде не нужно создавать специальных правил для подобных ситуаций. Причина в том, что все ее участники наделены подлинным чувством ответственности за каждый аспект проекта. Поэтому старшим членам команды никогда не придет в голову желание «свалить» технические задачи на младших коллег. Они поступят разумно, выполнив ту работу, которая необходима прямо сейчас (вспомните СЕО, который пошел за кофе для сотрудников)*. Исправление ошибок и другие задачи обслуживания *просто добавляются в бэклог спринта*, рассматриваются во время ежедневных scrum-митингов и выполняются надлежащими людьми в определенное время. (И последний ответственный момент для исправления ошибок, возможно, как раз именно сейчас, потому что разработчик еще ничего не забыл.)

В успешной scrum-команде все чувствуют свою сопричастность не только к коду, который создают, но и к бэклогу, и каждый старается делать все возможное, чтобы поставить работающее программное обеспечение. Бэклог спринта — это ответственность каждого, даже самый молодой разработчик ощущает, что именно он сделал для пользователей. Вот что подразумевает

* Наверняка кто-нибудь из представителей успешной scrum-команды, читая это, будет смущен или даже расстроен от мысли, что есть важная персона, «сваливающая» задачи на других, потому что это чуждая ему идея.

Кен Швабер под коллективной ответственностью в цитате, приведенной в начале этой главы: каждый член команды разделяет ответственность за бэклог и чувствует персональную ответственность за поставку наиболее ценного работающего программного обеспечения, необходимого пользователям, — включая все функции, а не только те, над которыми работает лично он.

Как же достичь такого чувства командной ответственности, чтобы каждый — от младшего разработчика, старшего технического руководителя, scrum-мастера и до владельца продукта — добровольно взял на себя решение неинтересных задач только потому, что заботится о проекте?

ПОВЫШЕНИЕ МОТИВАЦИОННЫХ ЦЕЛЕЙ КАЖДОГО ЧЛЕНА КОМАНДЫ

Вы когда-нибудь были волонтером? Содействовали проекту с открытым исходным кодом? Вступали в клуб, любительскую спортивную команду, рок-группу, церковный хор? Подумайте о том случае, когда вы присоединились к группе, не имеющей отношения к вашей работе или семье. Зачем вы это сделали?

Вы присоединились к группе и, вероятно, отдавали ей немало своего времени и сил, потому что вас интересовала главная цель ее существования. Если это был штаб избирательной кампании, то вас беспокоила явка людей на выборы. Если футбольная команда — то вас волновала победа (и качественная игра). Так почему же на работе должно быть по-другому?

Все мы стремимся к цели. По меньшей мере, работаем за деньги. Если работа перестает приносить доход, мы прекращаем ею заниматься. У нас есть счета, которые нужно оплачивать, и семьи, которые требуется кормить. Так что, когда нам платят и предоставляют комфортную, безопасную обстановку, в которой мы должны отработать положенные часы, предлагают другие элементарные блага, составляющие рабочую среду, — этого бывает достаточно, чтобы мы приходили в офис и выполняли свои служебные обязанности.

Но достаточно ли этого, чтобы нас по-настоящему волновало создание работающего программного обеспечения?

Если вам довелось работать в недостаточно мотивированной команде, то вы знаете, что ответ будет отрицательным. Дело в том, что многие из нас никогда не трудились в действительно мотивирующей обстановке. Но если вам с этим повезло, то, скорее всего, вы вспоминаете те времена с ностальгией. Когда каждый заботится о создании отличного программного

продукта, все вокруг приносит удовольствие: люди больше общаются, меньше спорят (или делают это продуктивно) и, похоже, легче достигают результата.

Существует много способов мотивировать команду: дать возможность поработать с новой технологией или в области, которая интересна, помочь продвинуться по служебной лестнице, платить премии, предоставлять работу на дому. Возможна и негативная мотивация: руководитель будет злиться, кричать (меньше заплатит, уволит). Эти позитивные и негативные стимулы мотивируют отдельных людей, но неспособны объединить команду.

Действительно сплотить может только **вдохновляющая цель**. Стив Макконел, эксперт и автор работ на тему управления проектами, в 16-й главе своей книги Beautiful Teams дал такое определение вдохновляющей цели:

Если вы копаете канавы, это вас не очень возвышает и вдохновляет. Но если вы копаете канавы, чтобы защитить свой город от врага, то это вдохновляет гораздо сильнее, хотя вы делаете то же самое. Поэтому задача лидера — представить работу таким образом, чтобы люди могли понять, в чем ее ценность.

Почти все программное обеспечение создано командой, а не отдельными людьми. Чтобы команда работала эффективно, нужно мотивировать всех сотрудников. Лучше всего вдохновляет и объединяет высокая цель, не оставляющая никого равнодушным.

Поставка ценности может быть очень эффективной вдохновляющей целью, мотивирующей всю команду. Когда у нее есть эта цель, она искренне верит в нее и готова самостоятельно решать, как ее достичь (и имеет возможность рисковать), команда будет усердно работать, используя имеющиеся инструменты, чтобы устранить любые препятствия на пути к этой цели.

Вдохновляющие цели направлены на ценность, но термин «ценность» может показаться абстрактным или оторванным от реальности. Для agile-команды ценность имеет вполне конкретный смысл: программное обеспечение ценное, если делает жизнь пользователей лучше. Что произойдет, если старший вице-президент сообщит команде: «Мы увеличили ваш доход в третьем квартале на 0,024%, потому что вы напряженно работали. Отлично, молодцы!» Это не особенно мотивирует большинство разработчиков, даже если среди них есть те, кто оплатил опционы на акции.

Другое дело, если этот же человек придет и скажет: «Обычно я тратил три часа только на то, чтобы разобраться в этих цифрах. А ваше программное обеспечение настолько просто в использовании и так хорошо работает, что

теперь я могу сделать все за десять минут. Спасибо!» Это гораздо сильнее мотивирует большинство разработчиков.

Разработчиков — а таковыми можно считать всех членов agile-команды, даже тех, кто не пишет код, — очень воодушевляет *гордость мастера*. Мы хотим создавать такое программное обеспечение, которое приносит пользу, нравится потребителям и вызывает у них желание заботиться о нем. Мы хотим, чтобы наше ПО работало эффективно и было создано максимально хорошо, так как тратим слишком много времени на споры о дизайне, архитектуре и технологиях. Все это действительно волнует команду. Сделать жизнь пользователей лучше — вот наиболее прямой путь, которым мы поставляем ценность. А это и есть настоящая, честная, вдохновляющая цель.

Этот принцип находится в верхней части списка принципов Agile-манифеста. Обратите внимание на слова, которые мы выделили.

Наш главный приоритет — удовлетворение заказчика посредством ранней и непрерывной поставки *ценного программного обеспечения*.

Причина, по которой это считается приоритетной задачей, в том, что поставка ценности пользователям — наиболее эффективный способ мотивации команд и движущая сила хорошего планирования спринта.

КАК ЗАПЛАНИРОВАТЬ И ЗАПУСТИТЬ ЭФФЕКТИВНЫЙ SCRUM-СПРИНТ

Начинать с бэклога — это значит начинать с пользователей

Почему мы планируем поставку специализированных функций в спринте, вместо того чтобы двигаться дальше? Потому что работали как команда, чтобы выяснить, какие функции наши пользователи считают наиболее ценными. Именно поэтому владелец продукта так важен — его задача понять потребности пользователей и своевременно информировать о них команду.

Будьте реалистичны в оценке того, что вы можете поставить заказчику

Многих менеджеров посещает безумная идея, что если не подбадривать разработчиков, то они будут бездельничать, выполнять минимум работы и устанавливать длинные сроки. Однако для большинства

команд это неверно. Просто в реальности разработчики слишком большие оптимисты. Именно поэтому мы знаем немало проектов, которые завершились с опозданием, и почти не видели случаев, когда они заканчиваются раньше времени. Не пытайтесь втиснуть в спринт слишком много функций. (В конце концов, пользователи должны всего лишь дождаться следующего спринта, чтобы получить рабочее программное обеспечение, которое включает в себя очередной набор функций.) Хороший scrum-мастер помогает команде оценить работу и понять, чем можно заниматься, а чем нет.

Измените план, если это необходимо

Используйте преимущество ежедневных scrum-митингов, чтобы выяснить, действительно ли команда собирается закончить разработку тех функций, которые были запланированы в спринте. Когда необходимо изменить план, команда делает это. Если становится понятно, что не удастся закончить все работы в бэклоге спринта, то команда должна перенести некоторые из них (начиная с наименее значимых) из бэклога спринта обратно в продуктовый бэклог. Scrum-мастер обязан убедиться в том, что все члены команды понимают суть изменений. Пользователи, привыкшие видеть работающее программное обеспечение, меньше нервничают, не дождавшись во время обзора спринта ожидаемых функций, если владелец продукта предварительно их к этому подготовил.

Заставьте всех говорить о ценности

В успешных scrum-командах, как правило, многие понимают, что на самом деле нужно пользователям и что для них ценно. Единственный верный способ добиться этого — сделать так, чтобы каждый член команды понимал, что именно он будет делать для пользователей. Как это облегчит их жизнь? Что позволит им делать то, что раньше было невозможно? Все это действительно важно для agile-разработчиков. Чем больше вы обсуждаете это, тем лучше будет программный продукт.



Ключевые моменты

Scrum одновременно инкрементальный и итеративный. Инкрементальный — потому что работа разбита на последовательные этапы, а итеративный — потому что команда адаптирует каждый новый спринт к изменениям, которые происходят во время проекта.

Когда успешные scrum-команды говорят, что они мотивированы поставлять ценность, они имеют в виду, что их важнейшая цель — создание программного обеспечения, которое улучшает жизнь пользователей.

Работа владельца продукта заключается в том, чтобы сохранить в команде мотивацию создавать ценность, помогая им понимать пользователей, разбираться в том, что они делают и для чего нужно ПО.



Описание: команда, разрабатывающая приложение для мобильного телефона в небольшой компании

Роджер — руководитель команды, пытающийся использовать Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт IV. Собака ловит автомобиль

С тех пор как Роджер и Ави принесли Эрику новости о том, что стали называть «встречей со стейкхолдерами из ада», они не могли понять, почему он полон оптимизма. Несколько дней спустя все трое отправились после работы в соседний ресторан, чтобы все обсудить.

Эрик спросил: «Как вы думаете, почему все эти менеджеры по работе с клиентами были расстроены?»

У Роджера и Ави не нашлось ответа. Роджер заговорил о том, как команда старалась стать более гибкой, что в его понимании означало возможность приспосабливаться к потоку запросов на добавление новых функций и создание новой продукции каждый раз, когда это необходимо. Ави чувствовал, что потратил много усилий, чтобы «присоединиться» к команде и передать ей массу великолепных идей, высказанных менеджерами по работе с клиентами. Они оба ощущали, что предоставили заинтересованным сторонам именно то, что те просили. «Посмотри, я получил письма, в которых каждый из них просит именно то, что мы им предоставили, — сказал Ави. — Что же их могло огорчить?»

Эрик объяснил, почему он как agile-тренер доволен увиденным, хотя проект столкнулся с неприятностями. Он сказал, что раньше они были похожи

на гигантское круизное судно, которому нужно проплыть много миль, чтобы сделать поворот. Теперь же они больше напоминают флот, состоящий из тесно взаимодействующих между собой быстроходных катеров. Пришлось больше общаться, но зато они могут разворачиваться на месте. А еще Ави избежал потрясений и появления некоторого антагонизма между ним и командой, он смог стать ее настоящим членом и успешным связующим звеном между командой и ее пользователями.

Помогая команде в организации эффективных ежедневных scrum-митингов, Роджер уполномочил ее самостоятельно организовывать свою работу согласно меняющимся приоритетам. Но появилась новая проблема, и Эрик объяснил, что это случается со многими командами, преодолевшими первый этап на пути к самоорганизации. Такая команда теперь имеет право эффективно менять направление деятельности. А входящий в состав команды владелец продукта может определять это направление.

Эрик предложил хорошую аналогию: «Вы когда-нибудь видели пожарную команду, которая учится тушить пожар при помощи пожарного шланга? Вам со стороны кажется, будто они просто целятся в огонь. Но они потратили недели или месяцы, пока не научились эффективно двигаться и общаться друг с другом, чтобы действовать согласованно. Раньше ваша команда была как садовый шланг, а теперь как пожарный.



Часто задаваемые вопросы

Как мне поступить с зависимостями между задачами в моем плане проекта?

Точно так же, как и сейчас, — вы говорите об этом команде и выясняете ее мнение.

Для командно-административного менеджера проекта, использующего планирование в стиле «вначале детальные требования» и создающего большие диаграммы Ганта, это очень часто возникающий вопрос.

Идея самоорганизующейся команды, где каждый участник решает вопрос о своей следующей задаче перед началом работы, звучит нереально. Часто это происходит потому, что менеджер проекта тратит много усилий на выявление всех зависимостей между задачами. Учебники на тему «Управление проектами» (в том числе те, которые

написали мы!) содержат разделы, объясняющие различные типы зависимостей (финиш-старт, старт-старт и т. д.) и то, как их определить и записать при планировании проекта. Так что нет смысла спрашивать, как именно анализ зависимостей проявляется в самоорганизующихся командах.

Почему руководителям проектов необходимы эти зависимости в начале проекта? Потому что они нужны для определения проектной деятельности, разделения задач на части, распределения их последовательности, выбора ресурсов и построения графика проекта. Упорядочить все это можно только одним способом: определив зависимости между отдельными задачами.

Предположим, что, пока член команды работает над задачей в общих чертах, он обнаруживает, что она зависит от другой задачи. Какой ужас! Теперь следующие задачи в плане будут сдвинуты назад, потому что принятая ранее последовательность задач не учитывала эту скрытую зависимость. Это приводит к каскадным задержкам — одной из самых распространенных причин изменения проектных планов. Еще хуже, если команда имеет фиксированный срок. Тогда руководитель проекта должен сделать трудный выбор и провести сложные переговоры о сокращении масштабов в конце проекта. Неудивительно, что менеджеры проектов заикливаются на зависимостях! Этот сценарий до поры неизвестных зависимостей — распространенное явление, он превращает хорошо продуманные планы в хаос. Казалось бы, полный анализ может дать руководителю ложное чувство безопасности благодаря уверенности, что команда берет на себя просчитанные риски, — и привести к задержкам, которые, как обычно, случаются в самый неподходящий момент. Команда также испытывает ложное чувство безопасности, потому что тратит много времени на размышления о зависимостях, после того как план рассмотрен и распределен по командам.

Самоорганизующиеся команды также выявляют эти связи, но делают это гораздо лучше. Разница в том, что они имеют дело с зависимостями в последний ответственный момент, когда у них гораздо больше информации о задачах и они способны выполнять более полный анализ.

Все это хорошо в теории, но будет ли это работать на практике?

Вспомните пример из предыдущей главы, когда командно-административный руководитель проекта проводил неэффективные ежедневные scrum-митинги и ежедневно назначал задачи каждому члену команды. Идя на собрание, руководитель проекта должен знать все зависимости

задач, чтобы распределить задания между сотрудниками. Сравните это с самоорганизующейся командой. Когда один ее участник назначает себе задачу, вся команда активно ищет зависимости во время летучки, а также на всех предыдущих и последующих встречах. Поэтому уже на третьем scrum-митинге любой член команды знает, какие существуют преграды. Каждая из них — это зависимость, которую пытается обнаружить команда. Все ее члены ищут зависимости ежедневно и поэтому находят. В результате они предотвращают многие каскадные задержки, с которыми сталкивается командно-административный проект.

Почему это удастся сделать? Когда командно-административный менеджер проекта выполняет анализ зависимостей для проекта на этапе его планирования, один из его наиболее важных инструментов — экспертное заключение. Это беседа руководителя проекта с членами команды, на чьи знания он готов положиться, чтобы помочь избавиться от зависимостей. Когда самоорганизующаяся команда ежедневно контролирует свой план во время scrum-митингов, она использует ту же экспертную оценку, но имеет гораздо больше информации, потому что участвуют все члены команды. Планирование на протяжении всего проекта дает ей гораздо более широкий обзор и позволяет принимать лучшие решения. Самоорганизующаяся команда имеет больше шансов найти критическую зависимость, потому что ищет ее во время проекта. А командно-административная имеет меньший обзор и более скудную информацию, потому что делает анализ за несколько недель или месяцев до начала работы. Улучшенный обзор и высокое качество анализа зависимостей, проведенного в нужное время, — важный источник производительности успешных scrum-команд.

Мне немного дискомфортно от выражения «в последний ответственный момент». Не лучше ли планировать заранее, даже если план придется изменить?

Руководители проектов любят цитировать президента США Дуайта Эйзенхауэра, который сказал: «В процессе подготовки к сражению я всегда находил, что планы бесполезны, но планирование необходимо». Когда вы садитесь вместе с командой и планируете проект, каждый думает о деталях работы и конкретных проблемах, с которыми придется столкнуться в ходе ее выполнения. Даже если расчеты руководителя проекта и команды неидеальны, после удачного планирования команда знает о проекте больше и все лучше подготовлена к работе.

Когда происходят изменения, команда выявляет их и получает опыт, позволяющий ей в будущем избежать любых ситуаций, вызывающих проблемы планирования. Так почему бы не планировать заранее и получить эти преимущества, особенно если есть возможность изменить план, когда это будет нужно?

Команда, которая планирует в последний ответственный момент, получает все эти преимущества и многое другое. Это не значит, что она не планирует заранее. Наоборот! Разница в том, что вы заранее планируете крупные работы: задачи в продуктивном бэклоге. Продуктивный бэклог содержит такие задачи (часто пользовательские истории), которые будут понятны владельцу продукта и другим нетехническим специалистам компании. До начала первого спринта команда и владелец продукта должны сесть и договориться о бэклоге спринта, а это требует от них большого продуктивного бэклога с точными оценками историй, то есть чтобы они уже фактически составили продуктивный бэклог заранее. Что ж, команда и впрямь уделяет много внимания предварительному планированию!

Вы можете привести пример того, как работает планирование scrum-спринта?

Конечно! Представьте, что вы член традиционной проектной команды, которая работает над проектом, где нужно разработать несколько новых функций, а также провести тонкую настройку производительности. Если бы вы планировали проект, то какую задачу сделали бы в первую очередь? Большинство команд, как правило, начинает с разработки новых функций, а настройку производительности делает в конце проекта. Если вы разработчик, то разработка новой функции кажется гораздо более интересной, дает возможность развития, а отслеживание и исправление ошибок производительности — это трудно и неинтересно, поэтому неудивительно, что такая работа часто располагается в самом конце плана.

Но что если пользователям действительно известны проблемы с производительностью? Что если эти проблемы становятся препятствием для тех пользователей, которые просто делают свою работу, и именно быстродействующее программное обеспечение, а не новые функции сделает их жизнь намного лучше? Значит, команда должна сделать тонкую настройку производительности своим приоритетом. В этом случае владелец продукта, который еженедельно встречается с командой и обсуждает наиболее ценные функции в бэклоге, имеет больше шансов в первую очередь улучшить производительность.

Напомните мне еще раз — какое это имеет отношение к принятию решений в последний ответственный момент?

Когда команда все планирует заранее, она займется тонкой настройкой прежде, чем новыми функциями, только в том случае, если с первого дня знает, что это необходимо пользователям. А если владелец продукта не привык постоянно говорить команде о том, что ценно для пользователей, и члены команды не привыкли слушать, что может произойти, когда владелец продукта в середине проекта сообщит, что пользователям в первую очередь действительно нужны именно эти настройки производительности? Менеджер проекта и команда, скорее всего, «замнут» эту тему, потому что она потребует серьезных изменений и приведет к неразберихе. Они также, вероятно, пожалуются, что бизнесмены, с которыми приходится работать, постоянно меняют свое мнение и будут мечтать о проекте без таких многочисленных изменений.

Такая команда все равно будет работать над внесением этих изменений. Но на их выполнение потребуется больше усилий, а также, как правило, реорганизация проекта. Реальность такова, что почти каждая группа пользователей имеет дело с проблемами, которые постоянно меняются. Такова жизнь. Не стоит думать, что изменения — это из ряда вон выходящее событие. Именно поэтому agile-команды любят использовать фразу «принять изменения».

Негибкое отношение к «планированию наперед» превращает внесение изменений в проект в переговоры между командой и владельцем продукта. А мы уже знаем, что agile-команды ценят сотрудничество с заказчиком выше, чем переговоры по условиям контракта. В переговорах кто-то выигрывает, кто-то проигрывает, и все идет на компромисс. Это не эффективный способ выполнения проекта, он приводит к потере производительности.

В то же время, когда команда сотрудничает с клиентом, все работают вместе и все в выигрыше — ведь каждый признает трудности, с которыми нужно справляться вместе. Планирование в последний ответственный момент поощряет такое отношение, потому что это помогает команде избежать установления произвольных ограничений, которые необходимо будет обсудить позже. Это позволяет команде оставаться открытой к изменениям и дает возможность (может быть, не всегда легко) изменить порядок работы, чтобы достичь меняющихся целей (в отличие от планирования работы и рабочего плана). Это облегчает задачу владельца продукта, который постоянно вовлекает команду в выполнение этих целей.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой).

Если вы работаете с командой, которая уже проводит ежедневные встречи, то попросите всех ответить на три вопроса, которые задаются во время ежедневных scrum-митингов.

Если вы работаете с командой, которая еще не проводит ежедневные митинги, выясните, можете ли вы провести хотя бы одно такое совещание.

Поговорите с вашей командой о последнем ответственном моменте. Напишите список решений, которые вы и ваша команда сделали. Не исключено, что их придется пересмотреть, потому что они были сделаны слишком рано.

Обсудите с командой задачи, которые могут появиться в продуктовом бэклоге. Какую работу вы еще не делаете? Можете ли вы составить список этих задач?



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

Вы можете узнать больше о самоорганизующихся командах и о том, как управлять scrum-проектами, в книге Кена Швабера *Agile Project Management with Scrum* (Microsoft Press, 2004).

Вы можете узнать больше о планировании scrum-проекта в книге Майка Кона *Agile Estimating and Planning* (Addison-Wesley, 2005).

Вы можете узнать больше о scrum-правилах, прочитав *The Scrum Guide* (Sutherland and Schwaber, 2011). Текст можно скачать на сайте Scrum.org.



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Одна из самых сложных задач внедрения Scrum — это поиск владельца продукта. Если вы работаете с командой, которая хочет внедрить Scrum, помогите ей найти человека, готового принимать решения в интересах бизнеса и обладающего полномочиями на это.
- Многие agile-тренеры считают, что их scrum-команды сталкиваются с проблемами, потому что выбрали владельца продукта из числа команды. Помогите им понять, что владелец продукта должен иметь полномочия принимать разработанные функции от имени компании.
- Проводят ли команды ежедневные встречи, именуемые scrum-митингами, которые по сути являются обычными статус-митингами? Помогите команде осознать разницу между командно-административным управлением и самоорганизацией.
- Помогите scrum-мастеру понять, что он не несет ответственности за ежедневную работу каждого члена команды и не должен отслеживать, выполнил ли тот свою работу. Помогите команде понять, что роль scrum-мастера состоит в том, чтобы помогать команде правильно выполнять scrum-правила и устранять любые препятствия, мешающие это делать.

ГЛАВА 5

Планирование и коллективные обязательства в Scrum

Каждый разработчик должен чувствовать себя комфортно, отвечая за работу, под которой он подписался. А поскольку команда исповедует принцип «это наше общее дело», комфортно должен чувствовать себя каждый ее участник.

Майк Кон*

Вы изучили механизмы Scrum и то, как использовать базовые шаблоны Scrum для совместной работы вашей команды. Но есть существенная разница между теорией Scrum и командной разработкой программного обеспечения в ходе реального проекта. Как вы настраиваете свою scrum-команду на то, чтобы добиться успеха? Как вы убеждаете ее членов стремиться к одинаковым целям? Иными словами, теперь, когда вы понимаете, что Scrum — это самоорганизация и коллективные обязательства, как вы управляете своей командой, чтобы обеспечить все это в реальной жизни?

В этой главе вы узнаете о практиках, которые используют многие scrum-команды для планирования своих спринтов. Вы увидите, как пользовательские истории помогут понять, что именно требуется пользователям от программы, и будете использовать очки историй и показатель скорости команды, чтобы определить объем работ, который команда способна сделать в каждом спринте. Кроме того, вы узнаете, как использовать два полезных инструмента визуализации — график сгорания и доску задач, чтобы все члены команды находились на одном и том же этапе.

Также вы поймете, почему этих практик и основной схемы scrum-проекта недостаточно, чтобы достичь «гиперпроизводительности» или «удивительных результатов». Мы вернемся к ценностям Scrum, и вы узнаете, как

* Кон М. Пользовательские истории. Гибкая разработка программного обеспечения. М. : Вильямс, 2012.

определить, соответствуют ли ваша команда и корпоративная культура этим ценностям. И что делать, если не соответствуют.



Рис. 5.1. Владельцам продукта часто чрезвычайно сложно угадывать мысли пользователей



Описание: команда, работающая над приложением для мобильного телефона в небольшой компании

Роджер — лидер команды, пытающийся применять Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт V. Не вполне ожидаемая неожиданность

На следующий день вся группа встретилась, чтобы обсудить, как начать двигаться в унисон и внести в проект больше предсказуемости. Они говорили о коллективной ответственности — что это значит на самом деле.

Эрик попросил нескольких членов команды вспомнить известные им случаи, когда большое количество пользователей реально работало с их приложениями.

Все оживились и стали охотно вспоминать подобные случаи — стало ясно, что больше всего их радовало, когда созданная ими программа использовалась клиентами. Затем Эрик поинтересовался ситуациями, когда выяснялось, что никто не применяет их ПО. Оказалось, что в прошлом году команда потратила четыре месяца на разработку системы отслеживания пользователей и управления учетными записями, а потом узнала, что старший вице-президент одобрил покупку подобной программы другого разработчика. После этого два человека уволились, а остальные чувствовали себя подавленными. Один из ведущих разработчиков, напряженно трудившийся над этим проектом, с возмущением сказал: «И ведь ничто не мешает этому повториться!»

Роджер подытожил: «Мы счастливы, когда люди используют созданные нами программы, и ненавидим, когда результаты нашего труда выбрасывают. Поэтому необходим способ убедиться, что мы создаем только такое программное обеспечение, которое будет применяться».

После этого оказалось несложно привлечь команду на свою сторону. Роджер объяснил, как проходит планирование спринта и как Ави, работая с ними и пользователями их ПО, добьется, чтобы в бэклог спринта попали действительно важные вещи. В ходе встречи Ави, Роджер и Эрик поняли, что почти вся команда готова воспринять идею коллективной ответственности и искренне хочет создать программное обеспечение, которое оценят пользователи.

После встречи Эрик, Роджер и Ави собрались, чтобы подвести итоги. Роджер и Ави поздравили друг друга с тем, что наконец получили команду, умеющую совместно размышлять над проблемами. И опять они были весьма удивлены реакцией Эрика: тот выглядел обеспокоенным.

Он сказал: «У вас команда, умеющая планировать спринты, и это здорово. Но это не решает ключевой проблемы — как обеспечить включение в бэклог спринта самых необходимых функций. Ведь если вы будете включать в него преимущественно второстепенные функции, то окажетесь в той же ситуации, что и сейчас, — пользователи будут продолжать жаловаться на отсутствие важных функций и перегруженность ненужными возможностями».

Роджер и Ави отнеслись к этим опасениям со скепсисом, который рассеялся после следующего обзора спринта, проведенного с пользователями. Ави

с гордостью перенес последнюю версию сайта Lolleaderz.com на свой тестовый сервер и начал выполнять новую функцию «создавай и достигай», над которой они работали. Она давала возможность пользователям устанавливать критерии для своих друзей, чтобы зарабатывать «достижения», загружая видео. Он установил достижение, названное им «получи козу», воспользовавшись новым редактором достижений. Ави перетащил слово «коза» из списка ключевых слов, используя новый виджет, позволяющий указать, какой будет награда после 500 просмотров страницы, и украсить ее всплывающей анимацией, которую они для этого нарисовали. В конце демонстрации в комнате стало очень тихо.

«Да, похоже, над этим изрядно потрудились, — сказал один аккаунт-менеджер. — Послушай, для чего ты это сделал?»

Они ожидали совсем не этого! Оказалось, что нужна была простая функция для одобрения видео друзей, чтобы те получали звездочку рядом с именем. Но команда неправильно восприняла это несложное требование и создала полноценный редактор критериев достижений со своим псевдоскриптовым языком и сервисной инфраструктурой. Никто из менеджеров не представлял, как продать клиентам эту функциональность. Команда потратила много сил и времени без всякой пользы, но еще хуже то, что бэклог оказался заполнен множеством важных функций, реализация которых оказалась отложенной.

Это было совсем не то, что ожидалось от разработчиков!

После встречи Роджер и Ави вернулись к Эрику, который совершенно не был удивлен. «Как же нам все исправить?» — спросил Роджер.

Ави заявил, что и так старался изо всех сил, стремясь дать другим менеджерам то, чего они хотели. Он поддерживал бэклог, приоритизируя его по ценности функций и передавая в команду. Что еще он мог сделать? Роджер даже не знал, с чего начать, но понимал: если они не внесут какие-либо изменения, то продолжат получать не совсем нужные результаты. Кроме того, произнеся зажигательную речь, Роджер и Ави заставили команду поверить в Scrum, и теперь ее постигло жестокое разочарование. Ави сообразил, что если не изменить что-то прямо сейчас, то команду можно потерять навсегда.

Эрик сказал: «Нужно научиться читать мысли ваших пользователей. Вы должны узнать, как они будут применять программное обеспечение. Но еще важнее понять в конце спринта, нужно ли клиенту то, что вы создали. Если вам все это удастся, то ваше программное обеспечение всегда будет пользоваться успехом».

Роджер был настроен скептически. Он по опыту знал: пользователи редко понимают, чего хотят, пока они этого не увидят. А Ави снова забеспокоился о своем понимании роли владельца продукта.

Он спросил: «Как нам научиться угадывать их мысли?»

Пользовательские истории, скорость работы команды и общепринятые практики Scrum

Стейкхолдеры и пользователи продукта ненавидят непредсказуемость. Если программное обеспечение, предоставляемое по итогам спринта, не похоже в работе на обещания, сделанные командой в начале, то пользователи и заинтересованные лица будут разочарованы, какое бы ценное ПО ни собирались создать разработчики. Другими словами, недостаточно просто предложить максимально ценное программное обеспечение. Вы также должны убедиться, что людей, для которых вы создаете программное обеспечение, не ждут сюрпризы при получении готового продукта.

Необходимо отметить, что приятные неожиданности могут быть столь же вредны, как и неприятные, поскольку порождают ожидания, что ваша команда всегда будет делать больше, чем обещала. Чтобы избежать таких сюрпризов, необходимо выполнить два условия.

Во-первых, в начале спринта команде следует хорошо поработать над формированием у стейкхолдеров правильных ожиданий. А во-вторых, в ходе выполнения спринта она должна держать всех в курсе изменений, вносимых во время ежедневных scrum-митингов. Вот почему присутствие пользователей и заинтересованных лиц на scrum-митинге ценно (хотя и не обязательно). Важно также, чтобы они наблюдали, но не вмешивались. Ежедневные scrum-митинги существуют для планирования работы на следующий день, а не для ответов на вопросы тех, кто не участвует в команде. Не всегда заинтересованные стороны могут принимать участие в scrum-митингах, иногда их и вовсе не следует приглашать. Это нормально. Хорошо, если заинтересованные стороны находят время, чтобы присутствовать, но это не обязательно. Пока владелец продукта будет держать стейкхолдеров в курсе любых изменений в плане, все будут находиться на одном и том же этапе развития продукта в течение всего спринта.

Но все это перестает иметь значение, если команда создает бесполезное программное обеспечение.

СДЕЛАЙТЕ ВАШЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПОЛЕЗНЫМ

Вспомним цитату из книги Кена Швабера, приведенную в начале главы 4. Если вы не добьетесь коллективной ответственности, то не получите Scrum. Но что такое «коллективная ответственность»? Какие именно обещания вы даете как команда в целом?

Коллективная ответственность означает искреннее стремление сделать продукт более полезным. Чтобы программное обеспечение оказалось полезным, вы должны понимать, что делают пользователи ПО. Надо помочь им выполнять свои задачи, и вам следует заботиться об этом больше, чем о чем-либо другом.

Этот принцип включен в Agile-манифест. Задумайтесь, что означают слова «сделать работающую программу» — какой смысл вкладывается в понятие «работающая»? Совсем несложно сделать программу, способную запускаться. Нетрудно создать программное обеспечение, которое выглядит как работающее, но сводит пользователей с ума, если они пытаются применить его на практике. Но создание действительно работающего ПО означает предоставление инструмента, который реально помогает людям решать стоящие перед ними задачи. Наиболее эффективный способ написать востребованную программу — сотрудничать с клиентами. Это еще одна причина, по которой мы ценим работающее ПО выше исчерпывающей документации, а сотрудничество с заказчиком — больше, чем согласование условий контракта.

Пока Agile не изменил мир разработки программного обеспечения, команды нередко выпускали никому не нужные продукты, чему можно привести немало примеров. Практически в любом учебнике конца 1990-х — начала 2000-х годов по разработке программного обеспечения вы найдете ссылку на доклад Standish Group's CHAOS. В Standish Group начали готовить такие доклады в середине 1990-х, когда стало известно, что огромное количество проектов завершаются неудачей (лишь около трети были признаны успешными*). Ежегодные исследования этой компании неоднократно показывали: команды разработчиков имеют основание считать, что многие функции в написанных ими программах не используются. В исследовании 2002 года** этот показатель невероятно высок — 64% (45% не применяются вообще, а 19% — редко).

Это могло оказаться шоком для научного сообщества, но разработчики считали такой результат очевидным. На самом деле то, что в отчетах

* J. Laurenz Eveleens and Chris Verhoef. The Rise and Fall of the Chaos Report Figures. IEEE Software, vol. 27, no. 1 (2010).

** 2002 CHAOS Report. The Standish Group International, 2002.

расценивалось как провал, для большинства команд было обычным явлением при разработке ПО. Созданный продукт перебрасывали клиентам в надежде, что кое-что из сделанного будет работать. Многие обсуждения (споры) с пользователями завершались фразой разработчиков «это не ошибка, а особенность нашей программы». Это означало, что программа работает именно так, как было задумано, а клиент должен принимать ее такой, как есть. Такое отношение к пользователям — дурной тон.

Роджер и Ави попали впросак со своим редактором достижений, потому что создали чрезмерно усложненный инструмент для решения простой задачи. К сожалению, команды разработки нередко выпускают менее полезный для клиента продукт, чем могли бы. Для этого явления придуман специальный термин — золочение. Команды стремятся приукрасить программное обеспечение, то есть добавить никем не востребованные функции, причем делают это с самыми лучшими намерениями. Разработчики хотят помочь клиентам и думают, что создают нечто очень ценное. Вполне естественный порыв для разработчиков, любящих свое дело, но в то же время — одна из главных причин тех самых 64% неиспользуемого функционала.

Еще один фактор появления большого количества неиспользуемых функций — отсутствие тесного взаимодействия с клиентами. Когда время общения пользователей с командой разработчиков сильно ограничено, они стараются затребовать как можно больше новых функций. В результате стейкхолдеры могут включить в план массу функций, которые не будут применяться на практике.

Эффективные agile-команды практически не имеют подобного опыта. В создаваемых ими продуктах лишь немногие (а вовсе не 64%) функции остаются неиспользованными. Вот почему в этих командах зачастую воспринимают выводы доклада CHAOS как ошибочные. Нередко активные приверженцы Agile стараются поставить под сомнение результаты этого отчета. Так что же позволяет agile-командам создавать столь полезные и востребованные программы?

ПОЛЬЗОВАТЕЛЬСКИЕ ИСТОРИИ ПОМОГАЮТ СОЗДАВАТЬ ТЕ ФУНКЦИИ, КОТОРЫЕ ВАШИ КЛИЕНТЫ БУДУТ ИСПОЛЬЗОВАТЬ

Agile-команды начинают с попытки узнать, чего хотят их пользователи. У них есть для этого очень эффективный инструмент. Кажущаяся простота такого инструмента, как **пользовательская история**, обманчива: это краткое описание того, как именно будет применяться программное

обеспечение. Большинство пользовательских историй — не длиннее четырех предложений. Команды в основном придерживаются эмпирической закономерности, согласно которой пользовательская история должна помещаться на лицевой стороне стикера размером 7×12 сантиметров.

Многие команды пишут свои истории пользователей в формате Mad Libs по следующему шаблону:

Я как <роль> хочу <конкретные действия, которые я предпринимаю>, чтобы <цель>.

Вот история, которую команда Lolleaderz.com использовала в качестве отправной точки для разработки функции контроля достижений.

НОМИНИРОВАТЬ ВИДЕО НА ДОСТИЖЕНИЕ
КАК ПОСТОЯННЫЙ ПОСЕТИТЕЛЬ САЙТА, ИМЕЮЩИЙ
ДЛИННЫЙ СПИСОК ДРУЗЕЙ, Я ХОЧУ НОМИНИРОВАТЬ
ВИДЕО ОДНОГО ИЗ НИХ НА ДОСТИЖЕНИЕ, ЧТОБЫ
ВСЕ НАШИ ОБЩИЕ ДРУЗЬЯ СМОГЛИ ЗА НЕГО
ПРОГОЛОСОВАТЬ.

Рис. 5.2. История пользователя, написанная на карточке

Эта пользовательская история эффективна, поскольку делает очевидными три важные особенности:

- Кто наш пользователь: «постоянный посетитель сайта, имеющий длинный список друзей».
- Что хочет сделать пользователь: «номинировать видео одного из своих друзей на достижение».
- Почему пользователь хочет это сделать: «чтобы все наши общие друзья смогли за него проголосовать».

Она также имеет название («Номинировать видео на достижение»), которое дает возможность команде легко сослаться на эту конкретную историю, когда о ней заходит речь.

Здесь представлено много информации, упакованной в одну пользовательскую историю. Она подразумевает, что нужно сделать многое: пользовательские интерфейсы для выдвижения и голосования, способ хранения номинаций и голосов, обновление системы, отображающей видео, чтобы можно было просматривать достижения и показывать звездочки, и т. д.

Не менее важно и то, чего в истории нет. Например, в ней ничего не говорится про какой-либо редактор критериев (добавленных командой без необходимости) или иных ненужных функций. Вот почему истории пользователей — эффективный инструмент для борьбы с золочением. Что происходит, когда команда описывает пользовательские истории, прорабатывает их с владельцем продукта (а также пользователями и другими заинтересованными лицами — всеми, у кого есть мнение о данном функционале) и ориентируется на них в ходе разработки? Если они применяют такую процедуру, то вряд ли «раздуют» программное обеспечение лишними функциями.

Истории пользователей также дают командам простой способ управлять бэклогом. У многих эффективных agile-команд в бэклоге содержатся почти исключительно пользовательские истории.

Поскольку каждая из них написана с точки зрения потребителя, владельцу продукта легко анализировать с пользователями и заинтересованными сторонами эти истории, чтобы выяснить, какие из них наиболее ценные. К тому же эти истории очень короткие, что позволяет легко добавлять новые и в любой момент изменять их порядок в бэклоге. А когда приходит время начать спринт, владелец продукта и команда берут несколько пользовательских историй из бэклога для реализации.

Затем команда может обсудить выбранные истории с владельцем продукта, чтобы убедиться, что она правильно понимает смысл каждой из них и сможет проверить корректность ее реализации. После этого большинство команд разделит истории на задачи и оценит длительность их выполнения. Задачи каждой пользовательской истории перейдут в колонку «сделать» доски задач, где будут ждать, когда кто-нибудь начнет работу над ними. Когда кто-то из команды готов взять новую задачу, он выбирает самую ценную из тех, которые способен выполнить, пишет на карточке свое имя и перемещает ее в столбец «в процессе».

КРИТЕРИИ УДОВЛЕТВОРЕННОСТИ

Один из способов убедиться в полезности того, что вы создаете, — умение представить себе конечный результат. Разработчик обычно получает большое удовлетворение, когда видит свою работу завершенной. **Критерии удовлетворенности** пользователя эффективно помогают разработчикам представить, как будет выглядеть законченный продукт, и на каждом спринте оценить, насколько они далеки от завершения. (Некоторые команды рассматривают критерии удовлетворенности как «критерии приемки».)

Критерии удовлетворенности, как и пользовательские истории, кажутся очень простыми, но выполняют сложную задачу. Большинство команд формулируют их для каждой пользовательской истории, вписывая конкретные операции, которые пользователь должен иметь возможность делать с программой, уже после создания такой истории. Обычно критерии удовлетворенности помещаются на задней части той же самой карточки (размером 7 × 12 сантиметров), что и пользовательская история. Владелец продукта обычно имеет право формулировать критерии удовлетворенности или высказывать свои замечания, если критерии уже сформулированы.

Критерии удовлетворенности для пользовательской истории «достижения» могут выглядеть следующим образом.

КРИТЕРИИ УДОВЛЕТВОРЕННОСТИ	
✱	ПОЛЬЗОВАТЕЛЬ МОЖЕТ НОМИНИРОВАТЬ ВИДЕО НА ДОСТИЖЕНИЕ.
✱	ДРУГ ПОЛЬЗОВАТЕЛЯ ПОЛУЧАЕТ УВЕДОМЛЕНИЕ, КОГДА ВИДЕО ПОЛУЧАЕТ ДОСТИЖЕНИЕ.
✱	ПОЛЬЗОВАТЕЛЬ МОЖЕТ ПРОСМОТРЕТЬ СПИСОК ВСЕХ ВИДЕО, НОМИНИРОВАННЫХ ЕГО ДРУЗЬЯМИ.
✱	ВИДЕО С ДОСТИЖЕНИЕМ ПРЕДСТАВЛЕНО НА ЭКРАНЕ С ОТМЕТКОЙ В ВИДЕ ЗВЕЗДОЧКИ.

Рис. 5.3. Критерии удовлетворенности, написанные на обратной стороне карточки с пользовательской историей

Эти условия имеют значение для разработчиков, потому что помогают им избежать преждевременного празднования победы. Программистам свойственно создавать различные части объекта, но вплоть до окончания спринта не приступать к его сборке. Например, разработчик может нарисовать страницы для номинаций и изменить код, отображающий видео, чтобы можно было добавить звездочку для достижения. Но пока он не свяжет все это воедино, чтобы каждое звено находилось на своем месте, а весь новый код был полностью интегрирован в существующую базу кода, добавление функциональности не будет завершено. Но гораздо эффективнее сделать все сразу, пока различные фрагменты кода свежи в памяти.

Критерии удовлетворенности дают команде конкретное определение понятия «готово». Они позволяют команде и владельцу продукта понять, что обработка истории и ее реализация завершены. История может считаться готовой, только если выполнена вся работа, необходимая, чтобы ее понять, построить, испытать и развернуть. После этого пользователь может открыть рабочее программное обеспечение и проверить, что все критерии выполнены именно так, как описано в обзоре спринта. Пока член команды не может этого сделать, разработка пользовательской истории не считается завершенной и он продолжает трудиться над ней (помните о сфокусированности как одной из ключевых ценностей Scrum?). Когда работа закончена, история на доске задач перемещается из столбца «сделать» в столбец «готово». Дело сделано!

ОЧКИ ИСТОРИЙ И СКОРОСТЬ КОМАНДЫ

Во время планирования спринта команда совместно определяет, сколько они смогут сделать в текущем цикле, чтобы создать программное обеспечение для поставки по его итогам. Но как именно команда это делает?

У каждой команды свой способ оценивать, какой объем работы она может сделать в спринте. Один из методов оценки пользовательских историй, подтвердивший на практике свою эффективность, — использование очков историй. Очки историй — это способ понять, сколько усилий вам потребуется, чтобы создать функцию для конкретной истории пользователя. Команда выставляет эти очки, сравнивая текущие пользовательские истории с затратами труда на реализованные ранее.

Нет никаких жестких правил, определяющих, сколько баллов (очков) следует присваивать той или иной истории. Некоторые команды назначают от 1 до 5 баллов за любую пользовательскую историю. Максимальное значение «пять» выбрано произвольно. Другие команды дают своим историям

от 1 до 10 баллов или используют другие пределы, не меняющиеся от спринта к спринту. Некоторые команды используют числа из последовательности Фибоначчи либо значения экспоненциальной зависимости. Вы можете выбрать любую из работающих схем, при которой все в команде чувствуют себя комфортно. Одна история, оцененная в 3 очка, должна требовать столько же работы, сколько и другая история, оцененная так же. Когда ваша команда оценивает в очках все истории, над которыми она работает, ее члены начинают понимать, на сколько очков они могут выполнить (или «сжечь») историй в текущем спринте. Если ваша команда завершает за один спринт в среднем историй на 25 очков, то говорят, что ее скорость составляет 25 очков историй за спринт.

Команды, как правило, работают с примерно постоянной скоростью, если оценивать по нескольким спринтам. Поскольку до того, как команда начнет работать, трудно предсказать, какой будет скорость, вы можете использовать прошлый опыт, чтобы точнее спланировать следующие спринты.

Если вы сталкивались с инвестиционными фондами, то знаете, что прошлые показатели не гарантируют будущих доходов. То же самое применимо и к очкам историй. Даже если ваша команда «сожгла» 32 очка в прошлом спринте и 29 в позапрошлом, нет никакой уверенности, что в очередном спринте эти результаты удастся повторить. В каждом конкретном спринте люди могут неправильно истолковывать отдельные истории, сталкиваться с неожиданными техническими проблемами, кто-то уходит в отпуск, заболевает, увольняется и т. д. Но несмотря на это очки историй и скорость команды с течением времени становятся наиболее надежным ориентиром для большинства scrum-команд, и вы сможете воспользоваться им при планировании спринтов.

Сессия планирования спринта при помощи очков историй может проходить так.

1. Начните с самых ценных историй пользователей из бэклога продукта.
2. Возьмите историю из этого списка — лучше самую маленькую, это удобно для сравнения. Затем найдите историю такого же размера из предыдущего спринта и назначьте первой из них то же самое количество очков, какое было у второй.
3. Обсудите с командой, можно ли считать эту оценку точной: обнаружив дополнительные проблемы, работы или технические вопросы, повысьте оценку, и наоборот, выявив упрощающие факторы, такие как повторное использование существующего кода или уменьшение объема необходимой функциональности, — понизьте ее.
4. Продолжайте переходить от истории к истории, пока не наберете достаточное количество очков, чтобы заполнить спринт.

Не перегружайте бэклог текущего спринта. Полезно оставлять в нем запас и нежелательно увеличивать нагрузку сверх той, что имела место в прошлом. Если средняя скорость команды — 28 очков за спринт и на очередной спринт запланировано новых историй на сумму в 26 очков, то вы можете добавить только одну историю размером в 2 очка или две — в 1 очко. Так как разработчики — оптимисты по натуре (так и должно быть, поскольку мы созидатели), команда захочет добавить еще 3 очка, превысив свой план на один балл. Не поддавайтесь этому соблазну: нет более верного способа разочаровать пользователей при следующем обзоре спринта.

Но что делать, если вы впервые взялись за такое планирование? Вам потребуется время, чтобы наработать архив историй и сформировать у команды понимание того, насколько велика трехочковая история. Что ж, тогда для начала вам придется руководствоваться предположениями. Выберите одну историю, которая, по вашему мнению, находится примерно в середине шкалы трудоемкости, и произвольно назначьте ей 3 очка. Затем найдите самую большую историю и оцените ее в 5 очков. Теперь возьмите самую маленькую и назначьте ей 1 очко. Используйте их все в качестве отправной точки для оценки историй и наполнения своего спринта. К концу второго спринта вы будете иметь целый набор историй для сравнения, а также неплохую оценку средней скорости команды.

ПОЧЕМУ ОЧКИ ИСТОРИЙ — ЭТО РАБОТАЮЩИЙ МЕТОД

Как и сами пользовательские истории, очки историй не так просты, как кажутся. Командам легко начать применять их в работе (хотя есть множество нюансов их использования, которые мы здесь не будем рассматривать). Но почему они так эффективны?

Они просты

Новому члену команды легко объяснить, как их применять. Кроме того, их сумма за проект оказывается не очень большой. Это десятки или сотни, но никак не тысячи, что не забивает головы членам команды огромными числами.

У них нет волшебных свойств

Многие разработчики и менеджеры проектов видели немало неправильных оценок, не подтвердившихся впоследствии, и считают, что разработку программного обеспечения просто невозможно оценить

более или менее точно. Присуждение очков историй исходя из реального опыта конкретной команды не делает тайны из того, откуда возникла та или иная оценка.

Команда контролирует их

Когда руководитель говорит вам, что вы должны соответствовать жестко заданным требованиям, это воспринимается как давление и увеличивает нагрузку на команду. Но когда команда самостоятельно принимает решение, как ей измерять свою работу и использовать для планирования только отобранные ею методы, такой подход становится для нее ценным инструментом.

Они побуждают вашу команду оценивать свою работу

Когда разработчик впервые попадает в scrum-команду, он, вероятно, впервые начинает открыто говорить об оценках, во всяком случае, его мнением интересуются. Выставление оценок — это навык, и единственный способ улучшить его — практика, которая в большинстве случаев предполагает обсуждение того, сколько сил понадобится на решение конкретных задач.

Разработчики их не боятся

Уже довольно много разработчиков имеют опыт выставления оценок в процессе работы. Затем эти оценки передаются руководителю проекта, после чего устанавливается жесткий дедлайн плана проекта. Подобное никогда не случится с очками историй, потому что они не превращаются в часы или конкретные сроки. Единственная зафиксированная величина — это дата окончания спринта, а запланированный объем работ может быть изменен во время ежедневного scrum-митинга в ходе цикла «обзор-контроль-адаптация».

Они помогают команде выяснить, в чем заключается смысл истории

Если вам кажется, что перед нами пятибалльная история, а я оцениваю ее в 2 балла, то мы, скорее всего, расходимся в понимании ее смысла. После обсуждения* может оказаться, что я полагал, будто для этой

* Еще один пункт в поддержку утверждения о пользе личного общения в сравнении с созданием всеобъемлющей документации!

истории достаточно создания простого инструмента командной строки, а вы считали необходимым создание инструмента с графическим интерфейсом. Хорошо, что мы выяснили это во время планирования спринта. Иначе нас мог ждать неприятный сюрприз уже во время работы, когда двухбалльная история превратилась бы в пятибалльную!

Они помогают всем членам команды стать по-настоящему вовлеченными

Очки историй и скорость команды дают каждому участнику «точку опоры», от которой он может отталкиваться: например, команда считает, что раз в последнем спринте было «сожжено» 26 очков, то одна из его историй заслуживает более высокой, четырехбалльной оценки. Новички нередко склонны уклоняться от участия в подобном обсуждении, но позже выясняется, что в этом случае решения принимаются без них. Когда вы осознаете, что у вас была информация, которая могла бы помочь команде оценить историю в 3 балла, а не в 1 и тем самым избежать ошибки, вы наконец начнете заботиться о планировании спринта и станете по-настоящему вовлеченным в дело.

ДИАГРАММЫ СГОРАНИЯ РАБОТ

Диаграмма сгорания — это способ наглядно показать текущий прогресс спринта в сравнении с ранее достигнутой скоростью команды. Вот как можно построить этот нисходящий график на основе очков историй (этот способ также работает с другими единицами измерения, например часами, но мы будем использовать очки историй).

1. Начните с пустого графика. По оси *X* отсчитываются даты, начиная с первого дня спринта и заканчивая последним его днем. По оси *Y* — очки историй с максимумом примерно на 0–20% больше, чем общее количество очков в бэклоге спринта. Поставьте первую точку на графике: количество очков в спринте в день начала работы. Нарисуйте прямую («направляющую») линию от стартовой точки до конца периода — ноль очков должно остаться к моменту завершения спринта.
2. Как только первая пользовательская история завершена и перешла на доске задач в колонку «сделано», нарисуйте следующую точку на графике: количество баллов, оставшихся в спринте на текущий день. По мере завершения вами следующих историй и «сжигания» большего числа очков историй бэклога заполняйте последующие дни в этом нисходящем графике.

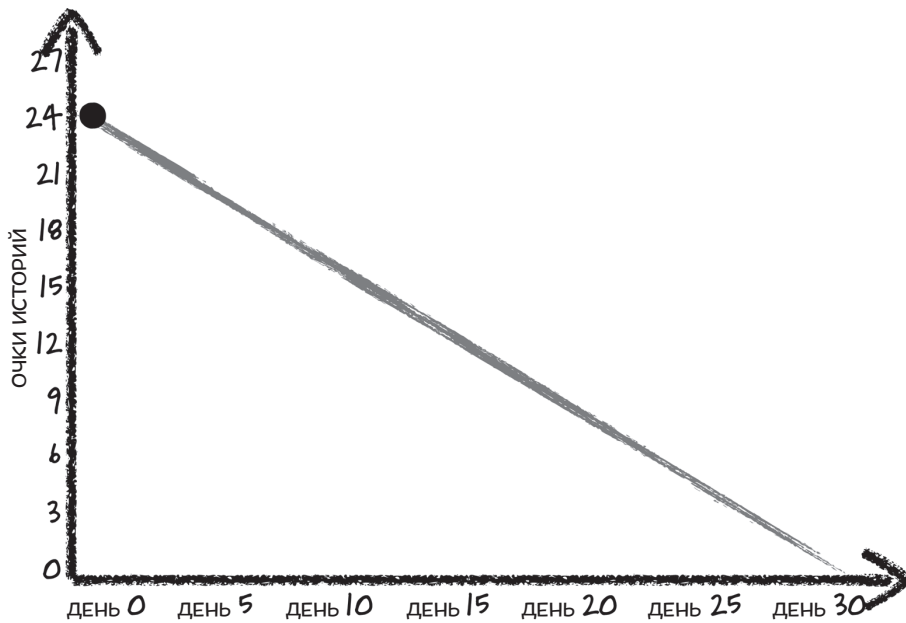


Рис. 5.4. Диаграмма сгорания в момент начала спринта. Мы создали этот нисходящий график, используя очки историй, но вы также можете использовать часы, дни или иные единицы измерения

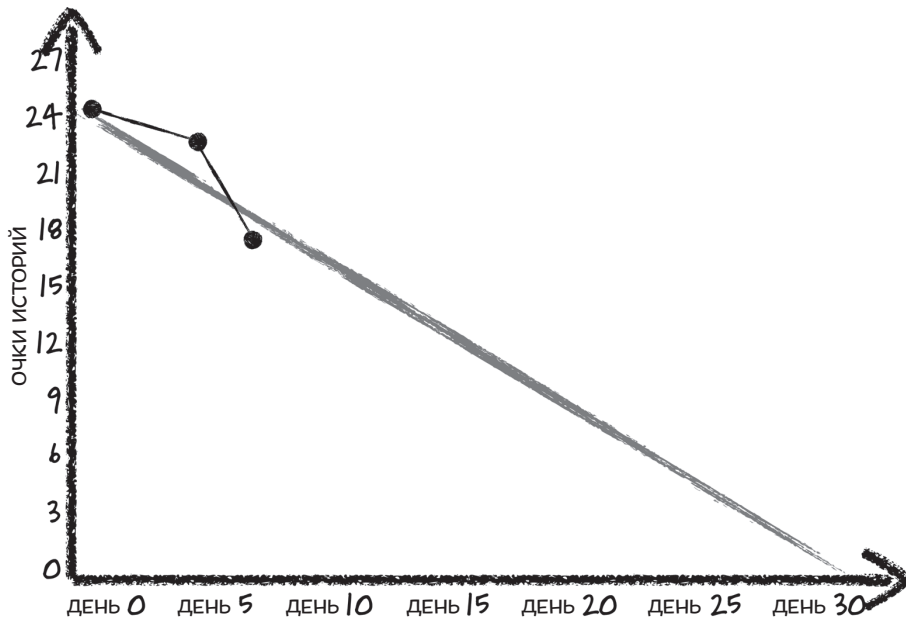


Рис. 5.5. «Сгорели» две истории на сумму в 7 баллов

3. Во время ежедневного scrum-митинга вы можете обнаружить, что необходимо увеличить объем работ. Иногда вы будете делать это, так как команда «сжигает» больше баллов, чем она ожидала, и в результате работы будут закончены раньше срока окончания спринта. Или появится новая важная задача поддержки, и при этом команда и владелец продукта согласятся, что она должна быть добавлена в спринт, но объем необходимой работы пока неизвестен. Поэтому команда не знает, сколько работы надо убрать из спринта, чтобы его сбалансировать. При добавлении карточки данной работы на доску задач запланируйте встречу команды, чтобы оценить в баллах (очках историй) каждую задачу и добавить их в диаграмму. Полезно также провести дополнительную линию, которая будет показывать момент, когда новые пункты были добавлены к графику, — не бойтесь оставлять на диаграмме свои заметки!

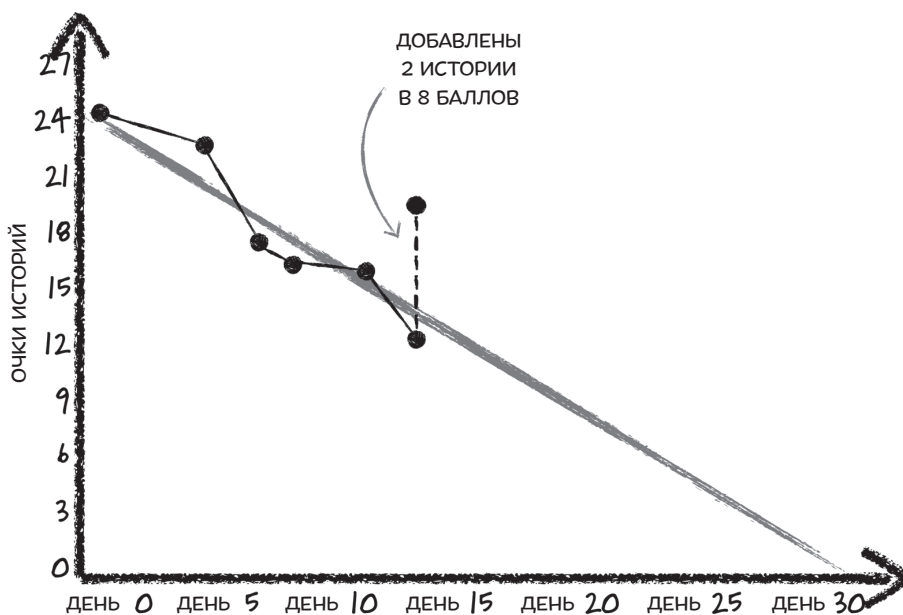


Рис. 5.6. Владелец продукта добавил пользовательские истории в середине спринта

4. По мере приближения к окончанию спринта «сгорает» все больше очков, что отражается на графике. Следите за интервалом между направляющей и вашим текущим «сгоранием», потому что он может означать, что в спринте осталось слишком много очков историй и какую-то из них нужно убрать.

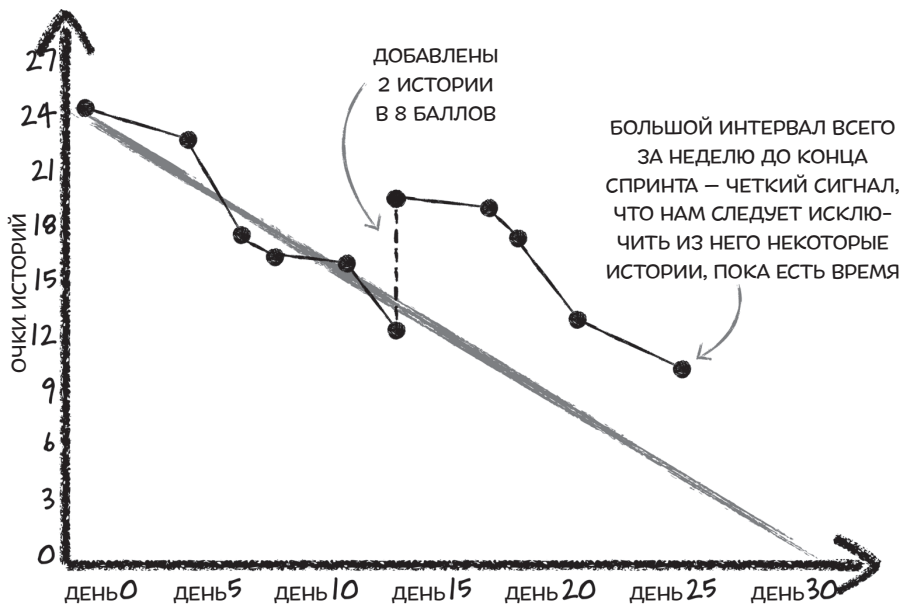


Рис. 5.7. Интервал между диаграммой сгорания и направляющей означает серьезный риск не успеть закончить все истории к концу спринта

Существует много программных решений для управления бэклогом, историями пользователей и очками историй, которые могут строить диаграммы сгорания автоматически. Однако многие команды предпочитают рисовать диаграммы сгорания вручную и размещать их на той же стене, что и доску задач. Это наглядно демонстрирует всем, как ведет себя проект в ходе спринта. Разработчикам особенно приятно видеть, как график обновляется сразу после завершения ими очередной истории пользователя и перемещения ее карточки в колонку «сделано».

Планирование и выполнение спринта с использованием историй, очков, задач и доски задач

Планирование спринта проводится, чтобы сформулировать развернутые ответы на два вопроса.

- Что команда поставит заказчику в этом спринте?
- Как команда сделает эту работу?

Мы только что продемонстрировали, как можно использовать очки историй и скорость команды для определения, что будет включено в спринт. Это распространенный способ, при помощи которого scrum-команды

выполняют первую часть планирования спринта. Но каким образом они определяют ответ на второй вопрос?

Один из самых распространенных для scrum-команды способов спланировать свои действия — это добавить карточки индивидуальных задач разработки. Это могут быть любые задачи, которые выполняет команда: написание кода, создание дизайна и архитектуры, разработка тестов, установка операционных систем, проектирование и создание баз данных, развертывание программного обеспечения на рабочих серверах, проведение юзабилити-тестов и всех прочих действий, которые команды выполняют повседневно в процессе сборки и выпуска программного обеспечения.

Вот примерная последовательность действий команды.

1. Команда проводит вторую сессию по планированию спринта. Scrum-мастер берет первую историю и обсуждает с командой, что они должны сделать, чтобы ее выполнить.

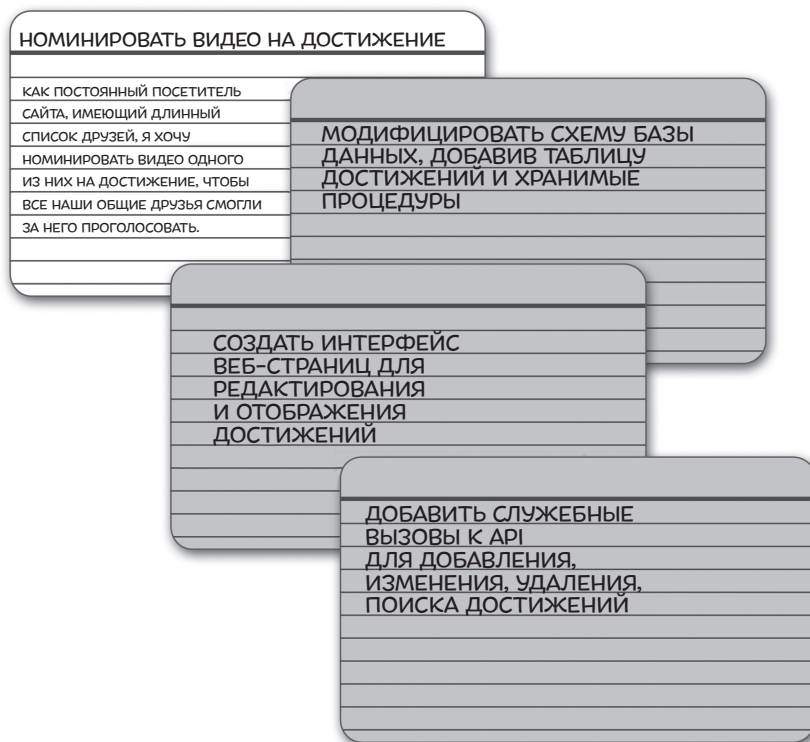


Рис. 5.8. Вторая половина сессии по планированию спринта состоит в разбивке историй на задачи, которые члены команды будут выполнять во время спринта

Команда составляет список индивидуальных задач с таким расчетом, чтобы каждая занимала не больше одного дня. Все задачи при этом выписываются на отдельные карточки. Некоторые команды используют карточки различных цветов для историй и задач, чтобы легче отличать их друг от друга. Карточка пользовательской истории размещается на доске рядом с карточками ее задач. Процедура продолжается, пока не будут спланированы все истории. Если время планирования спринта истекает прежде, чем все истории разбиты на задачи, то каждая незапланированная история получает отдельную карточку с заданием «спланировать историю».

2. Истории и их задачи группируются и добавляются в колонку «сделать» на доске задач.

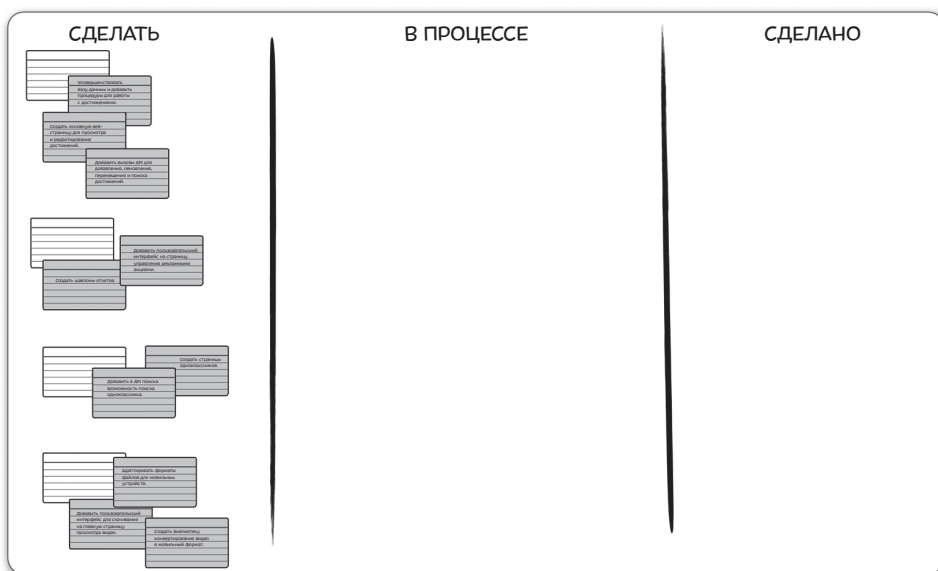


Рис. 5.9. Карточка каждой пользовательской истории, включенной в спринт, группируется вместе с карточками своих задач и добавляется в колонку «сделать» доски задач.

3. Когда член команды завершает задачу и готов двигаться дальше, он перемещает карточку выполненной задачи в колонку «сделано». Затем он вытягивает следующую карточку из колонки «сделать», пишет на ней свое имя и клеит ее обратно на доску в колонку «в процессе». Если история все еще находится в колонке «сделать», то он также перемещает ее в колонку «в процессе» (но не пишет на ней свое имя,

потому что кто-нибудь из коллег по команде может работать над задачами для той же истории).

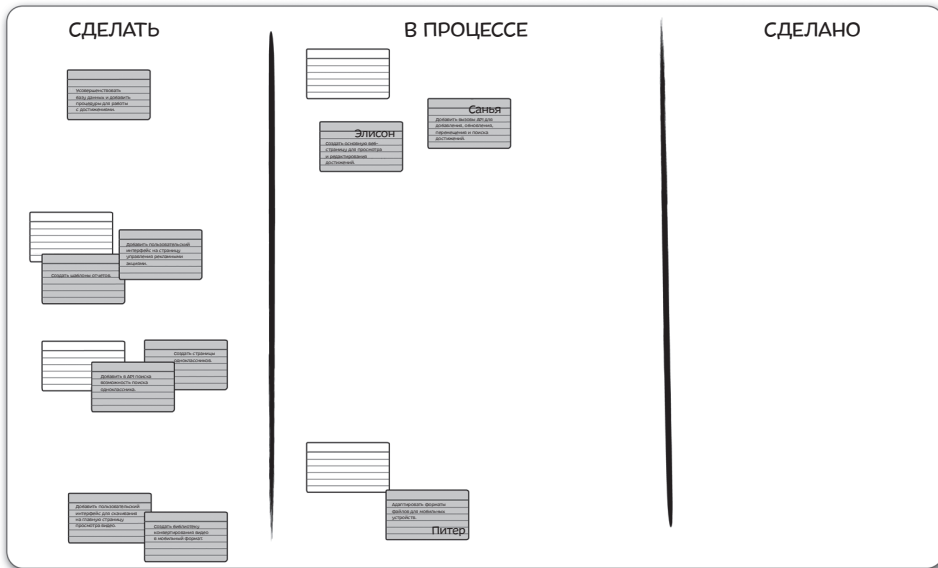


Рис. 5.10. Каждый член команды одновременно работает только над одной задачей, указав свое имя на ее карточке и перенеся ее в раздел «в процессе» доски задач

4. По мере того как продвигается работа над спринтом, команда переносит задачи из колонки «сделать» в колонку «в процессе» и затем — в «сделано». Нередко члены команды обнаруживают, что им необходимо выполнить дополнительные задания, чтобы закончить историю. Когда такое происходит, новая задача заносится на доску, а член команды сообщает об этом на ежедневном scrum-митинге, чтобы все понимали ситуацию и могли помочь определить потенциальные проблемы.
5. Как только член команды заканчивает последнюю задачу в истории, он снимает карточку задачи с доски, проверяет, выполнены ли все условия, и переносит ее в колонку «сделано» так, чтобы она располагалась рядом со всеми своими задачами. (Но помните: история не считается «сделанной» до тех пор, пока владелец продукта не примет ее для передачи заказчику от имени компании!) Если он обнаружит, что один из критериев удовлетворенности не был выполнен, — он перемещает историю обратно в столбец «сделать» и добавляет задачи, позволяющие завершить эту работу и переместить историю в колонку «сделано».

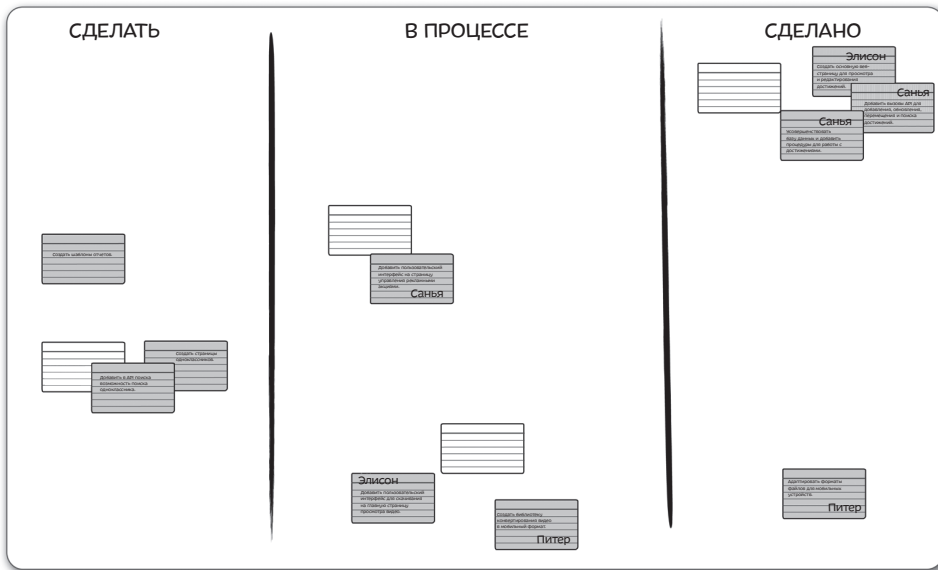


Рис. 5.11. Когда член команды заканчивает задачу, он перемещает ее в раздел «сделано» доски задач и берет следующую задачу. И если все условия удовлетворенности выполнены, то карточка пользовательской истории тоже перемещается в раздел «сделано»

- Спринт выполнен, когда его временные рамки исчерпаны. Однако при этом могут оставаться карточки историй и задач, находящиеся в столбцах «сделать» и «в процессе». Эти истории возвращаются назад, в бэклог продукта, и могут быть объявлены приоритетными в следующей сессии по планированию спринта*. При этом команда может уменьшить назначенное им количество баллов в зависимости от числа задач этой истории, которые остались невыполненными. История не засчитывается команде как выполненная, пока ее карточка и все карточки ее задач не будут перемещены в столбец «сделано».

* Существуют споры среди scrum-тренеров о допустимости перемещения невыполненной работы обратно в бэклог продукта и ее повторной оценки. Некоторые тренеры считают, что это нормально, в то время как другие полагают, что не следует изменять оценку (потому что стабильность оценок помогает команде лучше делать следующие оценки). Есть также те, кто считает, что это не имеет значения, потому что к концу проекта все подобные отклонения сглатываются.

ОБЩЕПРИНЯТАЯ ПРАКТИКА SCRUM

В упоминавшейся книге по Scrum Кена Швабера Agile Project Management with Scrum ничего не говорится о пользовательских историях и очках историй. Многие команды узнали о них из других источников, например книги Майка Кона «Пользовательские истории. Гибкая разработка программного обеспечения» (о которой мы говорили в главе 4).

Есть много замечательных практик, которые команды используют, чтобы улучшить свой уровень владения Scrum. Это не должно вызывать удивления. Мы применяем прошлый опыт, чтобы улучшить текущие разработки. Если мы решим ограничиться тем, что один человек написал в своей книге, и откажемся расширять свои горизонты, то нет смысла искать пути совершенствования.

Именно поэтому многие команды используют дополнительные техники и методы, которые Кон назвал общепринятой практикой Scrum (Generally Accepted Scrum Practices — GASPs)*. Например, многие команды приходят к выводу, что ежедневные scrum-митинги более эффективны, если проходят в виде standup-совещаний (когда все участники стоят, а не сидят). Этот прием не относится к числу основных scrum-практик, но широко применяется scrum-командами.

Вспомните последний принцип Agile-манифеста:

Команда постоянно ищет способы стать более эффективной путем настройки и коррекции своих действий.

Вы и ваша команда должны помнить об этом, когда проводите ретроспективные обзоры. Попробуйте найти пути для улучшений, не пытаясь изобретать велосипед. Если вы сталкиваетесь с проблемами в Scrum, то наверняка кто-то уже нашел их решение. Хороший наставник (например, такой как Эрик для Роджера и Ави) наверняка поможет найти решение, которое устранит проблему.



Ключевые моменты

Пользовательская история помогает команде понять желание клиентов, объясняя их специфическую потребность, а также то, как программа будет ее удовлетворять.

* См.: GASPing About the Product Backlog (по состоянию на 26 июля 2014 г.).

Каждая пользовательская история имеет критерии удовлетворенности, которые помогают команде понять, что история реализована.

Команды применяют очки историй и скорость, чтобы оценить, сколько историй они могут включать в каждый спринт.

Публичное размещение диаграммы сгорания работ помогает каждому из участников команды всегда быть в курсе того, что завершено, что требует доработки и укладываются ли они в график.



Описание: команда, работающая над мобильным приложением для телефона в небольшой компании

Роджер — руководитель команды, пытающийся использовать Agile

Ави — владелец продукта

Эрик — scrum-мастер в другой команде

Акт VI. Круг почета

Была половина двенадцатого ночи, все еще находились в офисе, но никто и не думал работать. На днях команда запустила сайт Lolleaderz.com, и на него уже поступили отличные отзывы. Число пользователей росло. СЕО наконец-то одобрил идею вечеринки по поводу этого удачного релиза, которую настойчиво предлагали Роджер и Ави. Он позвал диджея, организовал шведский стол с закусками и спиртными напитками. Роджер и Ави намеревались хорошенько выпить.

Роджер наткнулся на Эрика, который увлеченно болтал с несколькими членами команды, и сказал ему: «Мы бы никогда не сделали этого без тебя!»

Эрик немного подумал и ответил: «Очень мило с твоей стороны. Но сам посуди, что особенного я сделал? Я просто указал на некоторые проблемы и предложил решения, которые в свое время помогли мне самому. И время от времени я позволял вам столкнуться с некоторыми из этих трудностей, чтобы вы поняли, в чем их суть».

Подошел Ави со словами: «Так как насчет пары сложных моментов, с которыми все-таки пришлось столкнуться? Почему мы не смогли их избежать? Не подумай, что я жалею, ведь полученные результаты говорят сами за себя».

К этому моменту возле них оказалась бóльшая часть команды, а также несколько менеджеров. Эрик сказал: «Если бы у тебя не было той тяжелой встречи с другими аккаунт-менеджерами, то сумели бы вы продвинуться вперед и объединить в проекте все пользовательские истории? Удалось бы вам оценить скорость работы над проектом и начать планировать свои спринты?»

Роджер задумался. «Так ты считаешь, что единственный способ чему-нибудь научиться — это нарваться на неприятности?»

На это Эрик ответил: «Я бы предложил взглянуть на ситуацию с другой стороны — позитивной — и стать более эффективными. Когда вы ищете способы совершенствования, вы растете как команда».

Все присутствующие заулыбались и закивали. Откуда-то из-за спин собравшихся прозвучало: «Именно это я и хотел услышать!»

Это сказал CEO, выглядевший очень довольным. «Я вижу, что у этой команды большое будущее. Отличная работа!»

Переосмысление scrum-ценностей

В ходе проекта Lolleaderz.com Роджер, Ави и команда встретились с множеством трудностей. Многие команды, столкнувшись с подобными проблемами, провалили бы проект. Как же этой команде удалось превратить проблемы в возможности? Как они смогли извлечь опыт из своих ошибок и достичь успеха?

Одна из важных причин — наличие эффективного наставника, Эрика, который помог им преодолеть препятствия. Прежде всего Эрик помог Роджеру, Ави и остальным членам команды вникнуть в ценности Scrum: мужество, приверженность, уважение, сосредоточенность и открытость. Он рассказывал им про «свиней» и «кур», чтобы они поняли, что такое «приверженность». Он помог им понять, как сделать ежедневные scrum-митинги эффективнее и стать более сосредоточенными. И он показал им, как научиться взаимному уважению, начав прислушиваться друг к другу. Эрик помог улучшить планирование и организацию спринтов и бэклога с использованием таких инструментов, как пользовательские истории и скорость. Это сделало команду более открытой для всех, включая других аккаунт-менеджеров и CEO. И когда настали трудные времена, Эрик показал им, как важно иметь мужество говорить правду, даже если это на какое-то время приводит к неприятным последствиям.

Каждый проект имеет свои проблемы и вызовы. Когда вы пробуете новую методику впервые, эти проблемы могут усугубляться из-за отсутствия

опыта, непонимания или повторения старых ошибок. Хороший наставник, такой как Эрик, знает это. Вместо того чтобы пытаться избежать ошибок, он воспользовался ими, чтобы создать учебные ситуации для команды. Это называется *позволить команде неудачу*, и такой прием — один из основных в арсенале коучинга.

Лисса Адкинс пишет об этом в уже упоминавшейся здесь книге *Coaching Agile Teams*:

Позвольте команде неудачу: конечно, это не значит хладнокровно наблюдать, как команда приближается к краю пропасти. Но используйте десятки возможностей, которые возникают в каждом спринте, чтобы позволить команде провалить одно из заданий. Команды, попадавшие в трудные ситуации и выходившие из них благодаря в том числе сплоченности, становятся крепче и работают быстрее по сравнению с теми, которых оберегали от неудач. Такая команда может удивить вас. То, что должно было причинить ей вред, сделало ее сильнее. Наблюдайте и ждите.

Одно дело — читать такие слова, как «мужество», «приверженность», «уважение», «сосредоточенность» и «открытость», и соглашаться, что они звучат как призыв. И совсем другое — пойти к руководству и сказать, что команда не успеет поставить ПО в срок, потому что ей не хватает времени в спринте. Трудно проявлять мужество, когда ситуация грозит вам увольнением.

Вы не сможете успешно внедрить Scrum, не придерживаясь этих ценностей. Но дело в том, что только часть компаний имеют культуру, совместимую со Scrum, а у остальных ее нет. Возможно даже, что ваш руководитель требует от вас командно-административного стиля управления проектами и вы будете уволены, если попытаетесь создать самоорганизующуюся команду.

Scrum-команды учатся на своих ошибках — благодаря этому они развиваются и двигаются вперед. Команда нуждается в культуре, при которой совершать ошибки — это норма. Именно поэтому мы так тесно сотрудничаем внутри scrum-команд и взаимоуважение так ценно. Но даже если команда терпима к ошибкам и способна учиться на них, может случиться, что она работает в компании, которая их не приемлет. Как пишет новатор в области разработки программного обеспечения Грэди Буч в своей книге *Beautiful Teams*:

Один из известных мне признаков здоровой компании — ее терпимость по отношению к возможному неудаче. Компании, которые крайне негативно относятся к провалам, имеют тенденцию

быть менее инновационными, и работать в них не очень-то весело, потому что люди настолько боятся неудач, что всегда излишне осторожничают. А компании, допускающие бóльшую свободу действий (хотя и не настолько, чтобы подорвать бизнес) и позволяющие сотрудникам совершать ошибки, оказываются более продуктивными, потому что в них карьере работника не угрожает случайная оплошность, допущенная в коде.

Так что же делать, если вы обнаружили, что оказались в компании, где неудача означает увольнение? Возможен ли в ней хоть какой-нибудь Scrum?

ПРАКТИКИ СПРАВЛЯЮТСЯ И БЕЗ ЦЕННОСТЕЙ (НО НЕ СТОИТ НАЗЫВАТЬ ЭТО SCRUM)

Не каждая команда и не в любой компании может самостоятельно организовать за один день, и это нормально. Если культура вашей компании не совпадает с принципами Scrum, то вы, как адепт этого подхода, должны разъяснять окружающим ее ценности. Лучше всего делать это на личном примере: показывать, что значит открытость в работе, уважение к другим членам команды, а также умение проявлять мужество при столкновении с трудностями. Найти хорошего наставника — отличный способ продемонстрировать людям, как работают принципы Scrum, и помочь им начать менять свое отношение к делу. Часто команды пытаются использовать консультантов как наставников. Это очень удобно для руководителей: следовать чужим советам в том, какие изменения надо внести.

Но иногда, даже если вы делаете все правильно, сложившаяся культура компании, в которой вы работаете, может быть слишком далека от ценностей Scrum. В этом случае самоорганизация и коллективные обязательства могут оказаться для вашей команды недостижимыми.

А по мнению Кена Швабера, не добившись самоорганизации и коллективной приверженности, вы не получите Scrum.

И это нормально!

Даже помимо безоговорочно принятых ценностей, самоорганизации и коллективной приверженности Scrum содержит множество полезных методов. Они очень просты, легки в применении, и, как правило, вы можете внедрить их внутри команды, не спрашивая разрешения у руководства (или позже извинившись за самодеятельность).

Даже если вы работаете в среде, где невозможно изменить существующие ценности, внедрение отдельных приемов Scrum может дать полезный

результат. Вы должны так поступить, потому что лучше что-то делать, чем сидеть сложа руки. Просто убедитесь, что вы и ваша команда знаете, что делаете и где проходят границы ваших возможностей по внесению изменений.

Однако не стоит просто применять отдельные методы, а потом называть это полным внедрением Scrum. Поступив так, вы обрекаете себя на борьбу в дальнейшем. Сотрудники компании, читавшие о «гиперпродуктивных командах» и «потрясающих результатах», будут спрашивать, где все это. А у вас не найдется достойного ответа. Еще неприятнее, если команда, увлекшись Scrum и рассчитывая на радикальные улучшения, обнаружит, что для нее почти ничего не изменилось. В некотором смысле это равнозначно сигналу о том, что они достигли предела возможных изменений и дальнейшие улучшения производить нельзя. Это может расстроить и даже демотивировать работников. Такие проблемы создадут препятствия для внесения улучшений в будущем. Ведь если планка установлена слишком низко, то люди будут считать, что дальнейшие улучшения находятся вне их компетенции.

Иными словами, поздравляя окружающих с использованием новой методологии, вы будете создавать у них впечатление, что Scrum — это всего лишь переименование статус-митингов в ежедневные scrum-митинги, а требований — в бэклог. Это опасно для внедрения Agile, потому что фундаментальные проблемы, которые вы пытаетесь решить за счет Scrum, никуда не денутся. Большинство людей будет считать, что Scrum не может их исправить и даже что эти проблемы существуют везде и не имеют решения. Вспомните, сколько раз вы слышали фразы «Разработчики не могут заранее оценить объем работ», «Программное обеспечение постоянно глючит» или «Проекты никогда не выполняются в срок», произнесенные таким тоном, словно это законы бытия.

Хуже всего то, что, утверждая, будто Scrum может решить те проблемы, которые многим кажутся неразрешимыми в принципе, вы производите впечатление фанатика. Отсутствие явных результатов будет отталкивать людей от Scrum. Это повредит вашей карьере и настроит коллег против Scrum.

Существует еще один способ!

Дайте команде не только новые методы, которые они внедрят сегодня, но и видение того, чего они смогут достичь завтра. Тогда люди охотнее поверят в то, что делают Scrum, потому что уже освоят несколько простых методов.

Что делать, если вы реализовали практики Scrum, но еще не приступили к работе по самоорганизации и коллективной ответственности и понимаете это? Вместо того чтобы утверждать, что практики Scrum полностью внедрены, мы можем честно признать: впереди еще длинный путь по освоению ценностей Scrum, самоорганизации и превращению в команду, по-настоящему стремящуюся к результативности.

Первое отличие такого подхода заключается в том, что вы не давали обещаний, которых не сможете сдержать, а просто заявили людям, что сделали бы некоторые вещи немного лучше. Поэтому когда они видят улучшение результатов, то вместо недовольства тем, что не все проблемы устранены, испытывают удовлетворение от того, что команда производит программное обеспечение быстрее и лучшего качества. Гораздо легче начать разговор о новой ценности для компании, если у вас есть улучшения, пусть и небольшие. И ваш руководитель, вместо того чтобы слышать критику в свой адрес за недостаточное внедрение Scrum, получит от вас хорошие результаты сегодня и радужные перспективы на завтра — если только он разрешит команде реализовать свои устремления.

СОВМЕСТИМА ЛИ КУЛЬТУРА ВАШЕЙ КОМПАНИИ С ЦЕННОСТЯМИ SCRUM?

Не каждая компания готова отказаться от командно-административного управления проектами и сделать ставку на самоорганизацию, и не все команды способны принимать коллективные обязательства. Чтобы выяснить, готовы ли к ним ваша команда и компания, необходимо оценить, совместима ли культура компании с ценностями Scrum. Но как эти ценности Scrum внести в практику реальной жизни?

Постарайтесь честно ответить на следующие вопросы. Обсудите это с вашей командой и менеджером. Если все в порядке, то ваша компания и команда готовы к ценностям Scrum. А если нет, то такие обсуждения помогут понять, как правильно внедрить Scrum в вашу команду.

Чтобы понять, готовы ли вы к **обязательствам и ответственности**, спросите команду и руководителя, согласны ли они:

- отказываться от контроля над проектом и доверить команде самой принимать решения;
- не заниматься самодеятельностью и не создавать изолированных элементов в надежде на то, что удастся интегрировать их в конце проекта;

- следить, чтобы в команде не появился «мальчик для битья», на которого валятся все шишки*;
- прислушиваться к мнению коллег, а не навязывать свое представление о том, как надо работать;
- действительно брать на себя ответственность. Все ли в команде готовы к этому.

Чтобы понять, готовы ли вы к **уважению**, спросите команду (себя в том числе) и руководителя, согласны ли они:

- доверять команде работать самостоятельно и устанавливать сроки сдачи функций исходя из их относительной ценности и того, как развивается проект, даже если он займет больше времени, чем вы ожидали;
- давать команде достаточно времени на выполнение задач и не требовать от нее сверхурочных работ;
- доверять команде выбирать задачи, которые ей подходят и нужны для проекта, а не полагаться на строгое распределение ролей, матрицы RACI и т. д.;
- никогда не говорить, будто вы понятия не имеете, почему команда приняла то или иное решение.

Чтобы понять, готовы ли вы к **сосредоточенности**, спросите команду (себя в том числе) и руководителя, согласны ли они:

- не просить никого в команде сделать работу, которая не входит в текущий спринт;
- не просить никого из членов команды выполнить работу, которая не была включена в ее планы, только потому, что «мне нужно, чтобы это срочно было сделано»;
- ставить наиболее важные интересы компании выше всех прочих проектов и работ;
- не требовать, чтобы члены команды работали над конкретными задачами в наперед заданном порядке.

Чтобы понять, готовы ли вы к **открытости**, спросите команду (себя в том числе) и руководителя, согласны ли они:

* Будьте осторожны со словами «мальчик для битья», потому что во многих компаниях так называют владельца продукта. Команда правильно поймет это значение благодаря самоорганизации и коллективной ответственности, но остальные сотрудники почувствуют себя сконфуженными, услышав такое выражение.

- действительно прислушиваться к мнению других и размышлять над ним;
- думать о планировании проекта и пользователях, если раньше этого не делали;
- задумываться о технических деталях;
- интересоваться, над чем работает сидящий рядом программист и служит ли это общей цели;
- что программист, сидящий рядом с вами, должен действовать аналогичным образом.

Чтобы понять, готовы ли вы к **мужеству**, спросите команду (себя в том числе) и руководителя, согласны ли они:

- никогда не обвинять менеджера проекта в отсутствии планирования;
- никогда не сетовать на «эти дурацкие требования», не попавшие в оценку, сделанную владельцем продукта или старшими менеджерами;
- тратить время на то, чтобы действительно понять пользователей;
- не добиваться совершенства там, где клиенту нужен просто добротный продукт.

Если на большинство этих вопросов прозвучит ответ «да», то ваша команда, менеджеры и компания, скорее всего, имеют культуру, соответствующую ценностям Scrum. Но если хотя бы в одной группе вопросов будут два ответа «нет», то это причина обсудить данную тему с командой и руководством. В том случае, если открытой дискуссии не получится, вам наверняка придется заняться проблемой открытости, чтобы в дальнейшем получить максимальную отдачу от Scrum.



Часто задаваемые вопросы

Не являются ли бэклог и доски задач простой разновидностью графика проекта? Не делают ли scrum-команды то же самое, что и моя команда?

Можно применять инструменты и практики Scrum способами, очень похожими на традиционный проект. Например, некоторые команды пишут пользовательские истории, близкие к сценариям использования.

Но команда, не меняющая приемы работы, не изменит и своих результатов. Это означает, что команды, не изменившие собственного мышления, что необходимо для Scrum, не получают существенного прироста производительности труда. Иными словами, их работу можно охарактеризовать как «лучше-чем-ничего».

Одно из основных различий между эффективной scrum-командой и традиционными проектными командами заключается в том, что план scrum-команды не отправляется на полку сразу после составления, чтобы быть извлеченным оттуда только в случае срыва сроков. Они проверяют его выполнение на ежедневном scrum-митинге, чтобы сразу выявить возникающие проблемы. Более того, многие эффективные scrum-команды резервируют не менее часа в неделю для работы с владельцем продукта над обновлением бэклога (некоторые называют это «причесывание»). Владельцы продукта все время открывают для себя новые возможности. Во время еженедельных встреч по бэклогу команда вместе с владельцем продукта обсуждает создание новых карточек пользовательских историй с критериями удовлетворенности. Они присваивают очки новым пользовательским историям, а затем вместе с владельцем продукта составляют приоритеты новых историй в бэклоге продукта, отталкиваясь от того, насколько они важны и как долго их делать.

Это дает команде два очень мощных инструмента. Один из них — это получение обширного опыта по оценке трудоемкости, так что, когда приходится вносить изменения, она точно знает, как спрогнозировать их влияние.

Другой дает командам возможность обсуждать с владельцем продукта, какие из пользовательских историй он считает наиболее ценными. И если команда хочет вести диалог об этом, она должна понять истинные цели проекта. Внимание к целям важно, потому что, поскольку владелец продукта проделал большую работу, объясняя командам, для чего пользователям необходимо это ПО, им будет гораздо проще выяснить, какие истории в бэклоге продукта наиболее значимы.

Этот подход полезен, когда команда взаимодействует с владельцем продукта, чтобы понимать значение каждого элемента невыполненной работы. Ведь члены команды постоянно планируют и оценивают каждый из этих элементов. Таким образом, когда появляются изменения, любой член команды может сразу оценить их воздействие на общий план работ. Именно этот процесс обозначают словами «планирование в последний ответственный момент»: почти непрерывное выполнение, для которого команда специально резервирует время, поскольку считает это естественным.

Но не терпят ли программисты неудач при планировании, особенно в отношении проектов разработки ПО, трудоемкость которых невозможно просчитать в принципе?

Вероятно, это самое распространенное заблуждение о планировании или даже самоисполняющееся пророчество. Это не означает, что только программисты бездарны в планировании. Любой новичок в этом деле будет совершать ошибки. Если бы все люди были профессионалами в составлении планов, то бизнес всегда приносил бы прибыль, рынок двигался вверх, а каждый брак длился вечно (ведь никто не планирует развод в день своей свадьбы).

Планирование — это не предсказание будущего. Многие команды и менеджеры верят в миф об идеальном плане проекта, для которого разработчики могут дать точные оценки, а руководители проектов в состоянии спланировать любой риск и добавить оптимальный резерв на случай чрезвычайных ситуаций. Планы таких команд и менеджеров меняются не реже, чем у всех остальных, а порой даже чаще, потому что они включают в них слишком много мелких деталей, которые очень часто оказываются неточными. Недаром их проекты сталкиваются с проблемами планирования, и поэтому создается впечатление, что программисты бездарны в планировании, а программные проекты непредсказуемы!

Scrum-команды способны планировать более эффективно, потому что идут от общего к частному. Сначала они обрисовывают план в общих чертах, начиная с бэклога продукта, который содержит достаточно информации от владельца продукта и пользователей, чтобы можно было решить, что в нем главное. Команды начинают детальное планирование только в начале спринта и лишь для тех невыполненных работ бэклога, которые собираются включить в спринт. Основная часть работы по планированию на ближайшие дни и часы приходится на ежедневные scrum-митинги. Если какой-то фрагмент подробного планирования должен быть сделан в начале спринта, то команда его выполнит. Но такое случается редко, так что на подробное планирование не требуется отводить много времени, да и команда может пересмотреть эти планы во время очередного scrum-митинга.

Разве не более эффективно спланировать все наперед, а позже при необходимости просто уточнить план?

Нет. Многие команды убеждаются на практике, что, откладывая решение на последний ответственный момент, они получают дополнительную гибкость и это помогает эффективнее планировать или не менять планы. Этот стиль хорош тем, что дает возможность людям принимать

решения, когда они лучше к этому подготовлены, а не наспех и не имея полной информации.

Одна из важных причин провала многих крупных проектов — в чересчур подробном изначальном планировании, подробности которого далеки от реальности, потому что у разработчиков просто недостаточно информации. Это приводит к знакомому нам по предыдущим главам СУА-циклу, когда руководители проектов винят разработчиков в неточной оценке объемов работ, а программисты менеджеров — в плохом планировании. Люди обвиняют друг друга, но положение дел не меняется. Страдает только качество программного обеспечения.

Scrum дает нам свободу от СУА-цикла благодаря тому, что планирование производится с той степенью детализации, которая доступна команде в данный момент и позволяет отложить те подробности, которых мы пока не знаем, на последний ответственный момент. Мы ежедневно пересматриваем и адаптируем план, чтобы быть в состоянии быстро отреагировать, когда он оказывается ошибочным, что рано или поздно обязательно случится. Взамен Scrum требует от нас ответственности в правильном понимании того, что является ценностью создаваемого ПО.

Разве не является нереальным обещание, что в конце каждого спринта вы будете иметь работающее программное обеспечение, пригодное для демонстрации? Как быть, если команда работает над тем, что нельзя показать наглядно?

Этот вопрос очень часто задают новые scrum-команды. Некоторые функции, например новые страницы для веб-приложений, дополнительные кнопки, изменение в действиях конечного пользователя, очень легко продемонстрировать. Просто возьмите историю пользователя, запустите программу и продемонстрируйте выполнение каждого пункта условий удовлетворенности. Но некоторые функции показать не так-то просто. Как быть, если команда несколько недель оптимизировала базу данных, улучшала программные сервисы или выполняла другие нефункциональные изменения?

Любое изменение программного обеспечения можно продемонстрировать, и для программистов крайне полезно найти способ сделать это. Скажем, спринт был направлен на внесение изменений в базу данных. Разработчики не могут просто зайти в нее, внести изменения, а затем двигаться дальше, не проверив, работают ли они, — никто так не делает! Поэтому они написали код, сделали вставки, обновления, изменения, удаления и прочее, чтобы убедиться: изменения базы данных работают корректно. И вероятно, они также внесли изменения в другие части кода, которые используют эту базу.

Именно эти изменения команда и должна показать во время обзора спринта владельцу продукта, пользователям и заинтересованным сторонам. Разработчики часто пишут одноразовый код для проверки работоспособности изменений, внесенных в базу данных. Но если они знают, что им нужно наглядно показать работу изменений, внесенных в ходе спринта, то они могут взять этот код за основу и, слегка доработав его и выполнив небольшую отладку, создать модуль для демонстрации сделанного. Это вряд ли потребует много дополнительного времени, но зато у команды будет способ проверить работу изменений, который может пригодиться в будущем, если потребуется внести другие изменения или поменять тестовый сценарий.

Такой же подход полезен и в случае других видов нефункциональных изменений (которые не влияют на способы взаимодействия пользователей с программой, но требуют внесения правок в код). Команда, работающая над изменениями производительности, может продемонстрировать тесты «до и после», которые показывают достигнутое улучшение. Изменения, внесенные в архитектуру, можно продемонстрировать при помощи простой программы, которая показывает различные данные, полученные от нового API-сервиса. Любое изменение можно показать наглядно, и создание таких презентаций держит команду в тонусе. Это один из способов, который помогает scrum-командам планировать и развиваться.

Когда команда демонстрирует нефункциональные изменения, она делает это при помощи нескольких отдельных скриптов, тестов, программ или другого кода, написанного для проверки изменений. Разработчикам следует говорить о своей работе на языке, понятном клиентам. На самом деле не так уж мало пользователей способны понять то, что до них хотят донести программисты. Но главное, что такая демонстрация дает клиентам и заинтересованным сторонам реальное представление о том, что именно создает команда. Без показа могло бы сложиться впечатление, будто она провела спринт впустую, занимаясь всякой ерундой. Демонстрация же позволяет показать всем, в чем причина такой, казалось бы, неспешной работы, и это чрезвычайно ценно для команды, потому что в дальнейшем она сможет ставить реальные сроки, не подвергаясь давлению со стороны руководства, ничего не понимающего в ПО.

Когда возникает какая-то ошибка в уже готовом программном обеспечении, команда вынуждена приостановить текущую работу ради ее исправления, потому что невозможно отложить это до конца спринта. Но разве Scrum учитывает необходимость работ по поддержке?

Всем командам разработчиков постоянно приходится бороться с неожиданно возникающими проблемами. Поддержка — прекрасный пример этого. Команда, которая работает над сайтом, должна найти возможность исправить ошибку и немедленно помочь пользователям, и ей приходится потрудиться, чтобы это осуществить. Если исправление не может ждать до конца спринта, то команде остается только один вариант — заняться им немедленно.

Эффективная scrum-команда в этом отношении гораздо лучше, чем команда, придерживающаяся административно-командного стиля с «большими требованиями в начале проекта» (BRUF). Причина в том, что scrum-команды встречаются каждый день, чтобы оценить изменения и адаптировать к ним план работы. Эта команда может справиться и с проблемой изменения сроков, и с любым другим возникшим вопросом. Если это одна из тех экстремальных ситуаций, когда нет возможности дождаться следующего спринта, и владелец продукта согласился с этим от имени компании, то команда добавит эту срочную задачу в бэклог спринта, определит ее приоритет (скорее всего, установив его максимальным) и использует свои инструменты планирования — ежедневный scrum-митинг, доску задач, диаграмму сгорания, пользовательские истории, очки историй и скорость, — чтобы держать всех в курсе выполнения задачи и убедиться, что она будет сделана.

Основная разница заключается в том, что Scrum требует от нас быть честными перед самими собой и пользователями в оценке влияния этих изменений на проект. Мы не можем делать вид, что добавление такой работы происходит без затрат. Появится подъем в диаграмме сгорания, и станет очевидно: нужно принять меры для успешного завершения спринта. Впрочем, поскольку scrum-команды обычно выполняют свои обязательства по поставке ценного программного обеспечения, пользователи склонны доверять им. А раз команда демонстрирует работающее ПО, клиенты гораздо лучше понимают, как построена программа и что команда способна сделать за спринт.

Вряд ли у владельца продукта будет достаточно полномочий для принятия решений, необходимых связей с клиентами и компанией, а также свободного времени, которое он мог бы ежедневно проводить с командой. И это может создать впечатление, что Scrum не работает.

Хотите верьте, хотите нет, но существует масса эффективных scrum-команд в самых разных отраслях, которые работают именно так. Это происходит, потому что многие компании понимают: выделив команд-де работника с большим стажем работы и широкими полномочиями,

они получают огромное преимущество и действительно оптимизируют деятельность по развитию в долгосрочной перспективе. Опытный специалист, помогающий команде понять реальную ценность программного обеспечения, которое она создает, — это ключ к достижению тех замечательных результатов, о которых любят рапортовать многие scrum-команды. Как правило, они трудятся в компаниях, вложивших достаточное усилие в назначение в команды эффективных владельцев продуктов.

Известна реальная цена такого назначения. Появление в команде успешного владельца продукта означает, что компания должна поручить его текущие обязанности кому-то другому, но продолжать платить ему зарплату не меньше прежней. Когда компания доверяет Scrum, ее менеджеры видят, что этот метод обеспечивает необходимые результаты благодаря работе команд, которые способны соответствовать постоянно меняющимся требованиям бизнеса. Назначение хорошего менеджера по продукту на такой проект окупается в долгосрочной перспективе.

Если все это кажется нереальным для той компании, в которой работаете вы, то это проблема не Scrum, а существующей в организации системы поставки ценностей. Многие компании четко предписывают своим командам не беспокоить из-за дня в день руководство, бизнес-подразделения или продавцов вопросами о создаваемом программном обеспечении. Это один из наиболее распространенных видов конфликтов Scrum с ценностями компании.

Это также объясняет, почему многие компании предпочитают менее эффективный водопадный процесс «с большими требованиями в начале». Они готовы вложить больше времени, усилий и денег в создание этих документов, лишь бы снизить количество времени, которое их продавцы и руководители проведут в общении с командами. Ежедневная работа с командой — это очень трудоемкое занятие, руководителю гораздо проще прочитать документ, внести поправки, а затем смириться с не совсем удачным программным обеспечением. И компании часто пытаются исправить ситуацию, добавляя бизнес-аналитиков (улучшая тем самым качество документации), а также дополнительных инженеров по качеству и тестировщиков (чтобы убедиться, что требования были выполнены). Такой подход увеличивает количество дополнительной работы, компенсирующей нежелание исполнительных директоров утруждать себя. Но таков стиль работы во многих компаниях, и он все-таки позволяет создать хорошее программное обеспечение (правда, рано или поздно такие компании все равно окажутся в ситуации, характерной для отчетов CHAOS, и риск создания ими ненужных продуктов резко возрастет).

Если компания ценит программное обеспечение и время создающей его команды гораздо меньше, чем усилия бизнес-подразделений и продавцов, то несложно сделать анализ затрат и выгод — и тогда BRUF оказывается логичным способом построения программного обеспечения. Если вы работаете в компании именно такого типа, то успешная попытка использования Scrum поможет вашим менеджерам и остальным сотрудникам понять ценности Scrum и Agile. Но если это не сработает, вы, по крайней мере, сможете попрактиковаться в Scrum и пополнить свой послужной список неплохими (хотя и не выдающимися) результатами.

Я могу понять, как планирование спринта работает после того, как команда придумала свои оценки, но мне неясно, из чего эти оценки исходят. Как команды оценивают задачи?

Есть много различных способов оценки задач. Наиболее популярный из числа используемых scrum-командами — это покерное планирование, одна из общепринятых практик Scrum (GASPs)*. Такое планирование изобрел Джеймс Греннинг, соавтор Agile-манифеста. Метод стал популярным благодаря книге Майка Кона Agile Estimating and Planning. Вот как он работает.

В начале покерного планирования все оценщики получают по колоде карт. На каждой карточке написана одна из допустимых оценок. Оценщикам может быть, например, дана колода карт, которая включает 0, 1, 2, 3, 5, 8, 13, 20, 40 и 100. Карты должны быть подготовлены до проведения собрания по планированию, и цифры следует написать достаточно крупно, чтобы можно было видеть их на расстоянии. Карты можно использовать для повторных сессий покерного планирования.

Модератор поочередно читает описание каждой пользовательской истории или темы, которые должны быть оценены. Эту роль обычно поручают владельцу продукта или аналитику. Однако можно назначить на эту роль и любого другого человека, поскольку она не дает никаких особых привилегий. Владелец продукта отвечает на любые вопросы, которые задают оценщики.

После того как все вопросы заданы и все ответы получены, каждый оценщик выбирает из своей колоды карту, которая отражает его мнение. Карты не открывают, пока каждый оценщик не сделает свой выбор. Затем все карты одновременно

* Если вы хотите узнать больше о покерном планировании, рекомендую книгу Planning Poker PDF, написанную соавтором Agile-манифеста Джеймсом Греннингом.

переворачиваются, чтобы все участники могли увидеть оценки, данные другими.

Очень может быть, что разброс оценок будет велик. Это действительно хорошая новость. Если оценки разные, то оценщики объясняют, почему поставили высокие или низкие оценки. Важно, чтобы это не вызывало атак на авторов этих оценок. Вместо этого вы просто хотите узнать, о чем они думали.

Майк Кон. *Agile Estimating and Planning* (Pearson Education, 2005)

Покерное планирование очень эффективно, поскольку помогает людям воспользоваться накопленным опытом и объединить мнения и оценки всей команды в одно число. Значения карт не являются жестко фиксированными. Например, в прошлом некоторые команды использовали карточки с числами из последовательности Фибоначчи — 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, однако они не совсем удобны, потому что разница между соседними значениями довольно велика. Необходимо, чтобы выбор был легче, чем решение, займет задача 13 или 21 час, а история — 5 или 8 очков.

Как вы управляетесь с территориально разнесенными и большими командами?

Это непростой вопрос. Распределение команды по офисам в разных точках мира создает большие сложности. Нельзя назвать их непреодолимыми — многие agile-команды создают программное обеспечение, даже если они не сидят в одном помещении. Но когда вы снижаете возможности общения лицом к лицу и полагаетесь на менее надежные формы коммуникации, вы рискуете получить в своем проекте большее число дефектов, вызванных искажениями при передаче информации, как в детской игре «испорченный телефон».

Один из способов справиться с этой проблемой состоит в том, чтобы разделить крупные разнесенные scrum-команды на небольшие scrum-группы, работающие в одном офисе. Помимо ежедневных scrum-митингов они также планируют дополнительные ежедневные совещания (scrum-of-scrums), где члены каждой команды, собравшись вместе, отвечают на три вопроса. Используя бэклог своей команды, они самоорганизуются для работы над глобальным проектом и могут создавать друг для друга новые задачи, чтобы процесс интеграции работы каждой группы в общий проект был легким. Этот способ эффективен, потому что формирует для всей большой команды ее укрупненный план работ, который проверяется каждый день.

Scrum-of-Scrums — это не «серебряная пуля» и не решит всех или хотя бы большинства проблем больших территориально разнесенных команд. Самая большая трудность для такой команды заключается в поддержании каждого члена команды мотивированным общей целью. Как нам уже известно из главы 4, это главным образом достигается тем, что члены команды становятся «свиньями», а не «курами». Хотя безусловно можно сделать это независимо от удаленности членов команды друг от друга, гораздо более сложно для владельца продукта сделать так, чтобы у всех существовало единое видение задач. Потому что он должен передать его другим через разные часовые пояса при помощи телеконференций и электронной почты, а это гораздо труднее, чем при личном общении.

Я понимаю, что ежедневные scrum-митинги обеспечивают работу людей над правильными задачами. Но даже добросовестные разработчики способны увязнуть во второстепенных делах. Может ли нечто подобное произойти со scrum-командами?

Да, может. Основатель Scrum Джефф Сазерленд совсем недавно опубликовал вместе со Скоттом Дауни документ* под названием «Метрики для сверхпроизводительных scrum-команд», касающийся именно этой темы. Они работали со многими scrum-командами, чтобы определить набор параметров, позволяющий точно выяснить, насколько в действительности продуктивны «сверхпроизводительные» команды. Обнаружилось, что эффективная scrum-команда может выполнять проекты до пяти раз быстрее, чем обычная. И удержание разработчиков на главном направлении разработки имеет много общего с получением этого улучшения.

Сазерленд и другие scrum-лидеры постоянно ищут способы улучшить эту методику. Одна из многих интересных мыслей в этой статье заключается в том, что небольшая корректировка ежедневного scrum-митинга, направленная на удержание разработчиков в рамках наиболее ценных задач, создает значительную разницу в том, насколько успешно работает команда. Внесенные ими изменения состояли в том, что команда отвечала на различные вопросы по каждому элементу бэклога спринта, начиная с первого по порядку:

- Чего мы вчера достигли для истории первого элемента?
- Каков был наш вклад в историю первого элемента, выраженный в очках историй?

* Scrum Metrics for Hyperproductive Teams: How They Fly like Fighter Aircraft (документ представлен на 46-й Гавайской международной конференции системных наук, Мауи, 7–10 января 2013 года).

- Каков наш план по выполнению истории первого элемента сегодня?
- Что делать, если сегодня что-то блокирует нашу работу или замедляет ее?

Команда работает вместе, чтобы ответить на эти вопросы. Затем, вместо того чтобы поочередно опрашивать каждого ее члена, она движется по бэклогу спринта в порядке убывания важности задач, останавливаясь, когда закончится список или выйдет время ежедневного scrum-митинга.

Так почему этот процесс работает? Благодаря ежедневным scrum-митингам он удерживает членов команды от появления у них отстраненности (сквозь зевоту: «Я вчера работал над моим кодом, сегодня продолжаю над ним работать и завтра буду делать то же самое»). Митинги поддерживают всех в тонусе и фокусируют на приносимой командой ценности. Даже эффективные гибкие команды могут ненароком оступиться, если программисты чересчур увлекутся задачами, которые кажутся интересными им, отложив в сторону работу, важную для команды. Именно об этом заставляют думать всю команду приведенные выше вопросы.

Это возвращает нас к основной мысли, что scrum-митинг — фактически формальная совместная инспекция (подобная тем, о которых вы, возможно, слышали в университетском курсе по разработке ПО), цель которой — улучшить качество командного планирования и коммуникаций. Просто она проводится так, чтобы быть интересным и полезным занятием для команды.

Это одна из наилучших составляющих Scrum, потому что она постоянно развивается, а ведущие специалисты по этой методике могут вносить свой вклад в ее постоянное совершенствование.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой).

- Если вы до сих пор не применяете пользовательские истории, то возьмите одну функцию, которую вы в настоящее время создаете, и напишите пользовательскую историю для нее. Поделитесь ею с остальными членами команды — что они думают об этом?

- Попробуйте покерное планирование уже сегодня и выделите время вместе с командой, чтобы попробовать оценить несколько задач текущего проекта. Вы можете напечатать карточки сами или купить их (свои мы приобрели в компании Mountain Goat Software — можно также попробовать их онлайн-овую игру по покерному планированию).
- После того как вы сформировали оценку, воспользуйтесь доской или прикрепите большой лист бумаги на стену и приступите к созданию диаграммы сгорания работ. Как долго вы сможете поддерживать ее актуальной? Обратите внимание на то, что происходит на графике в конце проекта или итерации.
- Можете ли вы проанализировать ваш последний проект или итерацию и оценить скорость работы?



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше о Scrum и коллективных обязательствах в книге Кена Швабера *Agile Project Management with Scrum* (Microsoft Press, 2004).
- Вы узнаете больше о пользовательских историях и общепринятой scrum-практике в книге Майка Кона «Пользовательские истории. Гибкая разработка программного обеспечения» (М. : Вильямс, 2012).
- Узнайте больше о планировании scrum-проекта в книге Майка Кона *Agile Estimating and Planning* (Addison-Wesley, 2005).
- Вы сможете узнать больше о ретроспективах в книге Эстер Дерби и Дианы Ларсен «Agile-ретроспектива. Как превратить хорошую команду в великую» (М. : Издательство Дмитрия Лазарева, 2017).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Одна из главных задач коучинга состоит в том, чтобы помочь команде по-настоящему понять, что такое коллективные обязательства. Выявляйте ситуации, когда некоторые члены команды не решаются давать оценки, не принимают деятельного участия в планировании или пытаются заполучить scrum-мастера, который выполнил бы это за них.
- Обсудите с отдельными членами команды цели проекта. Отметьте случаи, когда люди проявляют узость мышления и думают только о конкретной функции или пользовательской истории, которую они воспринимают как зону своей ответственности. Рекомендуйте им брать на себя задачу другой пользовательской истории или функции, а другим членам команды — помогать им в этом.
- Приучайте scrum-мастеров делать все данные доступными. Поместите диаграммы сгорания и доски задач на стену и проводите ежедневные scrum-митинги там, откуда эта диаграмма хорошо видна.
- Оцените влияние внутренней политики компании: одной из причин, почему командам порой не хочется раскрывать реальный прогресс в работе над проектом, может оказаться неприятный опыт в прошлом, когда кто-то из руководства компании активно проявлял неудовольствие недостаточно успешным, на его взгляд, ходом работ. Ваша задача — не решать эту проблему за руководителей, а помочь им распознать ее, а также понять, как можно отчасти изменить корпоративную культуру, чтобы внедрение Scrum проходило эффективнее.

ГЛАВА 6

XP и охватывающие изменения

Люди ненавидят перемены... и это потому, что люди ненавидят перемены... я хочу быть уверен, что вы понимаете меня. Люди действительно ненавидят перемены. Они воистину их ненавидят.

Стив Макменагин. The Atlantic Systems Guild (1996)*

Вы когда-нибудь замечали, как часто программисты жалуются на своих пользователей? Немного побродив по специализированным программистским форумам, вы наверняка наткнетесь на ветку, где разработчики сетуют, что пользователи сами не знают, чего хотят, когда просят написать для них программу. Это стало частью культуры программистов: клиенты никогда не говорят, чего они хотят на самом деле, и осложняют жизнь программиста, то и дело меняя свои требования.

Выше уже говорилось, как Scrum предлагает решать эту проблему: непосредственные контакты с пользователями позволяют понять, что для них ценно, а частая поставка работающего ПО дает представление о том, как меняются потребности. Это дает руководителям проекта и владельцам бизнеса возможность постоянно пересматривать цели проекта, раз это обеспечивает максимальную отдачу. Значит, чтобы поспевать за изменением требований пользователей, команда должна постоянно корректировать код. Но разработчики знают, что внесение изменений в код приводит к ошибкам. Чем больше их вносится, тем более хрупким получается код. Любые ли правки кода приводят к ошибкам и нестабильности?

Это одна из тех проблем, с которыми справляется гибкая методология XP (Extreme Programming — экстремальное программирование). Как и Scrum,

* Демарко Т., Листер Т. Человеческий фактор: успешные проекты и команды. М. : Символ-Плюс, 2014.

она состоит из практик, ценностей и принципов. Практики просты в освоении и весьма эффективны — они могут изменить ваш подход к работе. Но, как и в Scrum, эти изменения происходят, если члены команды воспринимают их правильно.

В этой главе вы узнаете об основных практиках XP и о том, как они могут быть применены командой программистов (или, наоборот, использованы неверно). Вы узнаете о ценностях и принципах XP и о том, как они способствуют формированию правильного мировоззрения у каждого члена команды, помогающего создавать более качественный код. Вместо того чтобы ненавидеть изменения, каждый человек в команде учится их принимать.

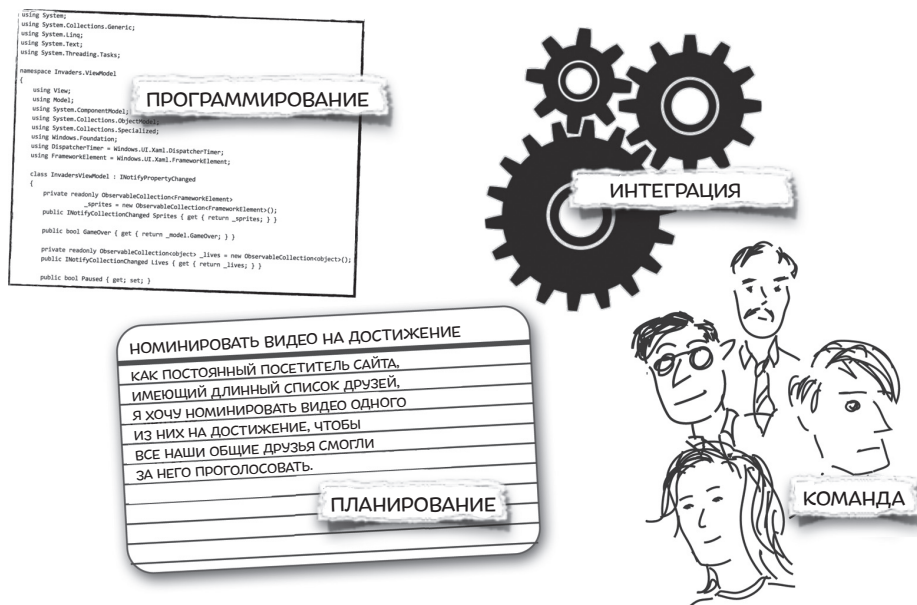


Рис. 6.1. Мы разделили 10 основных XP-практик на четыре категории: программирование, интеграция, планирование и команда



Описание: команда занимается разработкой баскетбольного сайта

Джастин — первый разработчик

Даниэль — второй разработчик

Бриджит — менеджер проекта

Акт I. Сверхурочная работа

Джастина всегда охватывало странное чувство, когда он задерживался в офисе допоздна. Ему казалось, что он выпил слишком много кофе, хотя на самом деле это было не так. Но он никогда не испытывал ничего подобного, работая по ночам дома.

— Похоже, предстоит одна из таких ночей, — сказала Даниэль, напарница Джастина. Они подружились в колледже. Она училась на два курса старше на том же самом факультете информатики — одном из лучших в стране. Кроме того, они были напарниками на занятиях в химической лаборатории. Даниэль дала ему рекомендацию, когда он устраивался разработчиком онлайн-фантазийной баскетбольной игры в Chalk Player Online. Эта компания занимается разработкой сайтов на спортивную тематику, и она оказалась его первым работодателем.

Джастин никогда не имел намерений задерживаться в офисе. Не то чтобы у него были проблемы с работой по ночам. Руководитель проекта Бриджит тоже ничего не имела против того, чтобы он трудился допоздна, особенно если он делал это из дома. Но как-то вечером Джастин обнаружил, что уже 10 вечера, а он все еще сидит на рабочем месте и отправляет письмо с извинениями подружке, а соседу по комнате — просьбу выгулять собаку.

Забавно, что незадолго до этого они с Даниэль обсуждали, что неплохо было бы уйти сегодня домой вовремя. Но в конце дня Бриджит сообщила плохие новости: состоялась встреча с менеджерами продукта и появилась необходимость внести в код большое количество изменений. При запуске проекта предполагалось, что он будет включать только команды NBA. Но теперь менеджеры продукта решили, что добавление европейских баскетбольных команд принесет намного больше денег.

И вот Джастин и Даниэль снова задерживаются на работе.

— Это действительно нечестно, — сказал Джастин. — Мы твердо согласовали с ними более трех месяцев назад, что будем использовать только команды NBA.

— Мог бы и не напоминать мне об этом. Та встреча была моей идеей, — добавила Даниэль.

— И после всего этого они хотят, чтобы мы добавили эти команды. Нам придется выдрать много кода и сделать довольно серьезные изменения в базе данных. — Он замолчал на секунду. — Знаешь, я виню Бриджит. Я так зол на нее. Она просто не понимает, как это будет тяжело.

Даниэль испуганно смотрела куда-то мимо Джастина.

— Бриджит стоит у меня за спиной?

— Да, — сказала Бриджит. — Послушай, я прекрасно понимаю, насколько это серьезные изменения.

— Но это не просто изменения, — ответил Джастин. — Мы объяснили им, что именно собираемся сделать. Это было несколько месяцев назад. Они подписались под этим. Если бы они предупредили о европейских лигах тогда, то мы бы разработали нечто совершенно другое. Система, которую мы программируем, не дает возможности использовать несколько лиг, поскольку речь шла только об одной.

— Таким было исходное предположение, — добавила Даниэль, — и его изменение полностью меняет подход к проектированию системы. Мы собираемся выдирать большие куски кода, а это приведет к появлению проблем.

— Каких проблем? — спросила Бриджит.

— Вы когда-нибудь чинили автомобиль при помощи скотча и скрепок? — поинтересовался Джастин. — Так вот, это похожий случай.

Все переглянулись и поняли: впереди много бессонных ночей.

Основные практики XP

Существует 13 **основных практик экстремального программирования**, которые могут помочь команде найти входы и выходы в разработке программного обеспечения и создавать легко изменяемый код. В отличие от scrum-практик, многие XP-практики имеют особенности и направлены на решение самых распространенных проблем, провоцирующих создание плохого кода. В этих практиках все настолько тесно связано с программированием, что многие люди ошибочно считают: XP — это только для высококвалифицированных специалистов.

В этой главе мы поговорим о 10 основных практиках XP. Эти 10 практик разделены на четыре категории — программирование, интеграция, планирование и команда, — чтобы их легче было понять и удержать вас от заблуждения, будто методика XP пригодна только для суперразработчиков.

ПРАКТИКИ ПРОГРАММИРОВАНИЯ

Две основные практики XP — разработка через тестирование и парное программирование — предназначены специально для программистов и помогают им писать более качественный код. Они сосредоточены на нескольких

важных аспектах разработки программного обеспечения. Созданием автоматизированных тестов перед написанием кода и его последующим тестированием команды повышают качество программного обеспечения. Работая вдвоем на одном компьютере, разработчики более тщательно вычитывают код в поиске ошибок.

Когда программист делает **разработку через тестирование**, которую обозначают аббревиатурой TDD (test-driven development), он создает автоматизированный тест перед тем, как начнет писать код. И пока код продукта не написан, тест будет завершаться провалом. Программист понимает, что код работает, когда тест оказывается пройден. Это создает петлю жесткой обратной связи, помогающую предотвратить появление дефектов: сначала написание проверочных тестов, затем создание кода, способного их пройти, поиск проблем и путей их решения и написание дополнительных проверочных тестов. Такие автоматизированные тесты обычно называют **модульными**. Слово «модуль» используется неслучайно: почти во всех языках программирования код очевидным образом разделяется на модули (классы, методы, функции, подпрограммы и т. д.), и практически любой язык предоставляет как минимум один способ создания и запуска автоматических тестов, которые привязаны к каждому из этих модулей. Создав вначале тесты, программист гарантирует, что все написанные им модули будут работать корректно.

Разработка через тестирование позволяет убедиться лишь в том, что каждый отдельно взятый модуль работает. Это также помогает командам предотвратить некоторые из самых распространенных и серьезных проблем, которые могут создать трудности в ходе поддержки кода. Можно внести изменения в одну часть кода и обнаружить, что неожиданно появились ошибки в другой части, которая, казалось бы, никак не связана с первой. И это только потому, что разработчик не подозревал о существовании зависимости между ними. Когда программист пишет модульные тесты, использующиеся каждый раз при сборке кода, эти автоматизированные тесты помогают избежать появления зависимостей, которые, оставшись невыявленными, приведут к немедленному сбою. Тесты помогают программисту заметить проблему, прежде чем она окажется встроенной в общий код и ее будет трудно удалить. Кроме того, модульные тесты помогают разработчикам писать код, который легче использовать повторно. Например, программист может создать класс Java, работа с которым чрезмерно усложнена: использовать в нем сбивающие с толку имена, неудобную инициализацию или другие структурные проблемы, которые смогут легко пробраться в код любого разработчика. Когда программист создает модульный тест, использующий такой класс, изъян может стать очевидным и, так как код класса еще не написан, разработчику несложно устранить этот изъян.

Затем модульный тест становится частью исходного кода, так что другие программисты могут опираться на него, чтобы увидеть, как этот метод предполагалось использовать.

Вторая основная XP-практика, ориентированная на написание кода, — это **парное программирование**. В команде, занимающейся парным программированием, два разработчика пишут код, сидя за одним компьютером. В большинстве случаев один программист сидит за клавиатурой, а другой наблюдает и они постоянно обсуждают, что делают. Команды, работающие в парах, вносят гораздо меньше ошибок, потому что за процессом внимательно следят две пары глаз. Парная работа также помогает уменьшить усталость, потому что один программист может печатать, когда другой устал. Эффективные команды парного программирования обсуждают идеи и подходы, постоянно устраивают мозговые штурмы и поэтому более инновационны. Они также несут ответственность друг перед другом за правильное применение практики: одному проще словчить, а под наблюдением партнера сделать это гораздо сложнее. Многие разработчики рассказывали нам, что программисты в парах пишут код быстрее и лучшего качества, чем работая по отдельности.

ПРАКТИКИ ИНТЕГРАЦИИ

Существуют две основные XP-практики, которые мы относим к категории «интеграция». Первая — это **десятиминутная сборка**. Команда создает автоматизированную сборку для всей кодовой базы, которая длится 10 минут. Эта сборка включает в себя автоматический запуск всех модульных тестов и генерацию отчетов с положительными и отрицательными результатами.

«Десять минут» может звучать как произвольная величина, но с точки зрения команды она имеет смысл. Если сборка проекта занимает более 10 минут, то члены команды запускают ее гораздо реже. Но для команды очень важно использовать ее часто, потому что это быстро выявляет проблемы. Например, в сборке запускаются модульные тесты, поэтому после ее завершения команда знает, достигла ли она нужного качества.

Иными словами, она позволяет быстро получить ответ на вопрос «Работает ли еще наш код?». И благодаря тому, что сборка довольно короткая, команда будет запускать ее часто, а каждый член команды получит постоянно обновляемую картину качества кода.

Другая интеграционная практика — это **непрерывная интеграция**. Она реализуется благодаря набору инструментов, позволяющих нескольким

членам команды одновременно работать с одной версией исходного кода. Если несколько членов вашей команды должны одновременно работать с одними и теми же файлами, то они не могут использовать одну физическую копию файла на всех. Иначе они бы постоянно затирали изменения друг друга. Поэтому команда будет применять специальную систему контроля версий, которая имеет централизованный (или децентрализованный) основной набор файлов. Отдельные разработчики будут **выписывать из него**, или копировать, нужные файлы, которые будут использоваться только этим программистом (что часто именуют помещением в так называемую **песочницу**). Разработчик будет трудиться над копией кода в своей «песочнице» и периодически отправлять изменения обратно в общее хранилище.

Проблема в том, что слишком часто разработчик, внося измененную версию программы, обнаруживает конфликт своего кода с изменениями, незадолго до этого внесенными другим программистом. В большинстве случаев этот конфликт выявляется сразу же при попытке выгрузить личную копию в общую базу кода, которая перестает компилироваться. Но временами проблемы оказываются более серьезными и скомпилированная программа ведет себя неправильно. Причина в том, что один код был заменен конфликтующим с ним кодом, взятым из другой «песочницы». Все эти проблемы проявятся, когда будет проведена **интеграция** системы из различных частей, написанных разными программистами.

И вот тут вступает в дело практика непрерывной интеграции: команда должна достаточно часто собирать код и отслеживать ошибки компиляции или сбои модульных тестов. Многие команды настроят сборочный сервер для периодической компиляции кода из хранилища, запуска полученной сборки и уведомления команды, если обнаружатся ошибки. Но создание сервера непрерывной сборки — это лишь одна часть непрерывной интеграции. Непрерывная интеграция означает, что каждый член команды постоянно содержит свою копию исходного кода в актуальном состоянии. Члены команды периодически интегрируют последнюю версию кода из хранилища обратно в свои «песочницы». Команды, применяющие парное программирование, порой пользуются физическим **маркером сборки**. Это может быть плюшевая или резиновая игрушка (забавная и заметная, чтобы сразу было видно, у кого она находится). Она переходит от пары к паре. Когда пара получает этот символ, наступает ее очередь интегрировать последнюю версию кода из хранилища к себе, исправить все найденные проблемы интеграции, а затем передать маркер сборки следующей паре. Это гарантирует, что проблемы интеграции будут обнаружены на ранних стадиях и исправлены.

ПРАКТИКИ ПЛАНИРОВАНИЯ

Для управления своими проектами XP-команды используют итеративную разработку. Как и в Scrum, практики XP-планирования основаны на большом цикле долгосрочного планирования, который разбит на короткие итерации. В практике **недельного цикла** XP-команды используют итерации длиной в одну неделю совместно с практикой **историй** (это то же самое, что и пользовательские истории, о которых вы уже знаете). Каждый цикл начинается с совещания, посвященного планированию, где члены команды проводят обзор достигнутых результатов, работают с клиентами, чтобы выбрать истории для итерации, а затем разбивают их на задачи, которые оцениваются и передаются разработчикам.

Это вам уже знакомо, потому что это очень похоже на scrum-планирование. Действительно, многие XP-команды в точности перенимают практики scrum-планирования (которые считаются популярными гибридами Scrum и XP, описанными в отчете VersionOne State of Agile, упоминавшемся в главе 2). Закончив планирование, команда посвящает первую часть итерации написанию автоматических тестов для историй и задач, а остальное время итерации пишет код, способный пройти эти тесты. Но вместо процесса самоорганизации некоторые XP-команды помещают все задачи на данную итерацию в очередь. Каждый разработчик берет следующую задачу из этой очереди после завершения текущей. Это гарантирует, что разработчики не выбирают свои любимые задачи и они распределяются равномерно между всеми.

Для долгосрочного планирования XP-команды используют практику **квартального цикла**. Раз в квартал команда собирается, чтобы проанализировать ход работ над проектом. На этих встречах XP-команды обсуждают **темы**, то есть те идеи из реального мира, которые они могут использовать, чтобы связать воедино истории своего проекта. Такие дискуссии помогают выяснить, какие истории необходимо добавить в проект, и поддерживают связь команды с реальными проблемами бизнеса, которые должно решить программное обеспечение. Также обсуждаются внутренние и внешние трудности, которые испытывает команда, повторяющиеся ошибки и еще не внесенные исправления. Оцениваются результаты, которых добилась команда, насколько хорошо удалось удовлетворить потребности клиентов и как в целом продвигается проект. Некоторые XP-команды склонны использовать ту же практику ретроспектив, что и scrum-команды.

И последняя из XP-практик, посвященных планированию, — **временной запас**. Она позволяет команде добавлять мелкие, менее приоритетные истории в каждый недельный цикл. Во время планирования эти истории

разбивают на задачи, но к ним не приступают до конца итерации. И тогда, если команда сталкивается с неожиданными проблемами, она исключает из итерации эти «запасные истории» и все-таки поставляет работающее ПО в конце цикла. Как и в других методах итеративной разработки, ХР-команды поставляют в конце итерации только полностью законченный функционал. Это означает, что он работает, все тесты пройдены и его можно продемонстрировать пользователям.

КОМАНДНЫЕ ПРАКТИКИ

ХР затрагивает как программирование, так и совместную работу команд. Эта методика включает в себя две основные практики, помогающие командам сплотиться, — «командные практики». Первая — **коаллокация**, когда команда располагается в одном помещении. Люди в команде работают лучше, когда сидят рядом друг с другом и могут легко коммуницировать. Многие люди не осознают, что, хотя программирование — это индивидуальный труд и зачастую выполняется в одиночку (за исключением парного программирования), работа в команде разработчиков требует высокой социальной активности.

Члены команды постоянно консультируются друг с другом, советуются и предупреждают о возможных проблемах. Если команда располагается в помещении типа open space, такая социализация обеспечивается естественным образом. По поводу того, каким должно быть открытое рабочее пространство, ведется много споров. Чтобы работать эффективно, программисты должны быть защищены от отвлекающих факторов. Многие из них ценят уединенность и размещают свои мониторы так, чтобы проходящие мимо люди не могли видеть экран. Популярное решение этой проблемы — модель планировки офиса «пещеры и поляны», описанная Стюартом Брэндом в книге *How Buildings Learn**. В его схеме имеются как отдельные, так и общие рабочие места для программистов, которые одной стороной выходят в общий зал со столами для проведения совещаний и компьютерами для работы парами.

Следующая основная практика ХР-команд — это **информативное рабочее пространство**, в котором производственная среда команды устроена таким образом, чтобы важная информация о проекте автоматически передавалась всем участникам. Один из наиболее популярных способов для этого — применение большой доски задач и диаграммы выполняемых работ (диаграммы сгорания), которые висят так, чтобы каждый мог их видеть. Наглядное

* Steward Brand. *How Buildings Learn: What Happens After They're Built* (London: Penguin Books, 1995).

размещение постоянно обновляемой информации о проекте позволяет каждому члену команды знать, на какой стадии находится проект, и принимать правильные решения. Такие диаграммы и другие способы, которые команда использует для отображения данных, делающие рабочее пространство более информативным, называются **излучателями информации**. Они автоматически передают текущую информацию о проекте всем, кто на них взглянет.

Большие, наглядные диаграммы — не единственный способ сделать рабочее пространство информативным. Когда у членов команды появляются вопросы, они сталкиваются с проблемами или им необходимо обговорить какие-то детали, они организуют обсуждение. Если такие дискуссии происходят в общем рабочем пространстве, а не в закрытых конференц-залах, то окружающие невольно слышат их и впитывают информацию о происходящем. Когда члены команды автоматически вбирают информацию о проекте — неважно, из диаграмм, разговоров или иным способом, — это называется **осмотической коммуникацией**. Командам необходимо найти баланс между пользой такой коммуникации и ее отвлекающим влиянием, и XP-команды становятся успешными, если им удастся его отыскать.

Почему важны информационные рабочие пространства? Суть в том, чтобы помочь команде принимать лучшие решения и дать ей больше контроля над ходом проекта — подобно тому, как доски задач помогают scrum-командам. Главное, что *информация предназначена для всей команды*, а не только для ее руководителей или менеджеров. Это демократизирует распространение информации и обеспечивает ею всю команду без какой-либо фильтрации. Чем больше сведений о проекте доступно команде, тем сильнее каждый ее участник может влиять на ход событий.

ПОЧЕМУ КОМАНДЫ ПРОТИВОСТОЯТ ИЗМЕНЕНИЯМ И ЧЕМ МОГУТ ПОМОЧЬ ПРАКТИКИ

Никто не любит менять уже написанный код по требованию заказчика. Внесение изменений в конце проекта может вызывать различные эмоции — от раздражения до озлобления и даже до ощущения надвигающейся катастрофы, если в ближайшие полтора месяца предстоит работа без выходов только потому, что клиент не продумал все до конца.

Именно поэтому разработчики часто жалуются, что бизнес-пользователи постоянно меняют свои решения. Очень часто, когда проект уже начался, можно услышать такие разговоры: «Так вы хотите это изменить? Если бы я знал, то написал бы код совершенно иначе! Почему в начале проекта вы не объяснили нам, чего именно хотите?»



Рис. 6.2. Каждый руководитель проекта знаком с ощущением беспомощности из-за необходимости внесения изменений, которые разработчики не в состоянии выполнить. Каждому разработчику знакомо чувство тревоги, когда, казалось бы, небольшие изменения в коде разрастаются как снежный ком и требуют слишком много времени

И разве можно их за это винить? Команды сопротивляются этим изменениям, особенно в конце проекта*, потому что им предстоит дополнительная работа. И не просто работа — а самый плохой ее вариант, самый раздражающий. Изменения заставляют вас возвращаться к тому, что вы считали законченным, разносят сделанное в пух и прах и вынуждают переписывать большие куски кода заново. И это не просто разочаровывающая работа: ваша команда вложила в нее много душевных сил, решила массу проблем и нашла наиболее элегантные решения. А теперь вам говорят, что нужно разрушить созданный командой Тадж-Махал, потому что на самом деле нужна была Эйфелева башня.

* Вернемся к принципам Agile-манифеста в главе 3. Какой из этих принципов применяется здесь?

Но у медали есть и другая сторона. Пользователь или руководитель, просивший в начале проекта одно, а к концу изменивший свои требования, делает это вовсе не с целью дезорганизовать команду. Мы уже знаем, что раздробленное видение приводит к плохим результатам, так что давайте попробуем взглянуть на эту проблему со стороны пользователя. Ему нужно программное обеспечение, позволяющее выполнять свои задачи, а это означает, что он должен сотрудничать с командой разработчиков. Теперь его просят ответить на вопросы, которые ему не совсем понятны: «Кто должен иметь доступ к этой функции?»; «Сколько людей будут использовать эту систему одновременно?»; «Каково допустимое время между нажатием кнопки “начать” и отображением результатов?»; «Какова правильная последовательность действий?»

То, что начиналось как встреча, на которой пользователь пытался объяснить свою проблему, каким-то образом превратилось в бесконечный поток обращенных к нему технических вопросов, на которые он не знает ответов. А время-то идет, и он чувствует: если не дать ответы на все вопросы, то команда отложит выполнение проекта и обвинит в этом его. (Будем честны: именно так она, скорее всего, и поступит.) Все знают, что создание ПО — дорогое удовольствие, а пользователь — не специалист в этом, поэтому дает самый лучший ответ, на который способен. И когда встреча заканчивается, он не мешает команде двигаться вперед и сжигать бюджет.

Как решить эту проблему? Кто прав?

Гибкие методологии, и в частности XP, с готовностью признают следующее: невозможно заранее знать в точности, что мы собираемся создавать, а наиболее эффективный способ это описать — начать работу над проектом. В данных методологиях исчерпывающую документацию заменяет работающее ПО, поскольку самый эффективный способ получить от пользователей обратную связь — это написать часть программы и отдать им для практического применения.

Поэтому, вместо того чтобы настраивать команду против изменений, возможно, стоит создать условия, при которых она сможет справиться с этими изменениями без шумных распрей и работы в выходные? Никто не любит вносить изменения, но, возможно, вам удастся найти способ ограничить влияние на проект каждого из них, причем не только в том, что касается целостности кода, но и в отношении морального состояния команды и пользователей.

Именно так поступают команды XP и называют такой стиль **адаптацией к изменениям**. Команды XP признают, что изменения — это не только неизбежное зло. Лишь в случае частого получения обратной связи

от пользователей и быстрого реагирования на нее они смогут создать действительно лучшее ПО для своих клиентов. Эти команды готовы к изменениям и даже планируют их. Они не воспринимаются как плохие новости, которые всегда сопровождаются поиском виноватого, прозевавшего часть требований. Правильное мышление для ХР означает не только принятие изменений, но и желание иметь их.

Адаптация к изменениям имеет два аспекта: степень охвата проекта и собственно база кода. Влияние изменений на код иногда бывает значительным — изменение, которое первоначально казалось маленьким, может ошеломить команду, потребовав существенного переписывания. Команды будут говорить о необходимости «разодрать код» и заделать дырки «скотчем, скрепками и жевательной резинкой». Мы рассмотрим тему адаптации к изменениям в главе 7. А пока обсудим, как ХР-команды приспособливаются к изменениям масштабов проекта, функциональности программного обеспечения и его поведения.

Так как же основные практики ХР помогают командам делать это?

Планирование

Когда команды итерационно планируют и поставляют работающее программное обеспечение в конце каждой итерации, они постоянно получают обратную связь. Все в команде знают, что они активно просят об обратной связи. Ведь намного проще справиться с изменениями, если вы тот, кто попросил об этом в первую очередь.

Команда

Решать проблемы гораздо проще, если поблизости есть люди, готовые помочь. Когда команды располагаются рядом, они выявляют проблемы намного раньше. А информационное пространство помогает сделать так, чтобы все работали над одной и той же проблемой. Это позволяет всем членам группы взять под контроль происходящие изменения и не быть застигнутыми врасплох возможными неожиданностями. Отдельные члены команды получают автономность, которая нужна им для принятия решений, влияющих на их код и даже жизнь.

Интеграция

Одна из самых неприятных проблем, вызванных неожиданными изменениями, — это то, что они распространяются по всей базе кода. Если

вы работаете с постоянными изменениями, то вам важно узнать о них как можно раньше — вот почему командные практики действительно полезны. Но человек все равно будет застигнут врасплох, если изменение, внесенное в базу кода его коллегой, вызовет неожиданную ошибку в той части кода, над которой работает он сам. Благодаря непрерывной интеграции (которую осуществлять легче, потому что наша сборка работает быстро) мы узнаем об этой проблеме очень быстро и исправим ее раньше — до того, как она станет нашей головной болью.

Программирование

Как непрерывная интеграция помогает выявлять изменения? Мы создали набор модульных тестов, которые запускаем каждый раз при выполнении интеграции кода. Поэтому, когда один из членов команды добавляет изменения, он запускает все тесты, в том числе и для той части кода, над которой работают все остальные ее участники. Если один из этих тестов терпит неудачу, то мы сможем работать вместе — и даже парами, — чтобы решить эту проблему в обеих конфликтующих частях кода.

Когда все эти практики работают вместе, то изменения — даже крупные — испытывают меньшее негативное влияние. Это помогает команде понять, что с изменениями можно справляться легче.



Ключевые моменты

Когда XP-команды используют разработку через тестирование, первым делом они создают модульные тесты, выражающие поведение будущего кода, а затем пишут код, который сможет их пройти. Это создает петлю обратной связи, помогающую предотвратить появление дефектов.

Команды создают десятиминутную сборку, или автоматизированную сборку, которая выполняется не более чем за 10 минут.

Отдельные разработчики используют непрерывную интеграцию, чтобы постоянно учитывать изменения коллег, поэтому у каждого из них «песочницы» соответствуют текущему состоянию проекта.

XP-команды применяют итерации с циклами в неделю и квартал, а также истории так, как это делают scrum-команды.

Команды добавляют незначительные, с низким приоритетом задачи в каждый недельный цикл, чтобы создать временной запас.

Когда XP-команды сидят вместе, они периодически впитывают информацию о проекте через осмотическую коммуникацию.

XP-команды работают в информационном рабочем пространстве, которое использует излучатели информации, такие как настенные диаграммы выгорания, автоматически передающие ее людям, находящимся поблизости.



Описание: команда занимается разработкой фантазийного баскетбольного сайта

Джастин — первый разработчик

Даниэль — второй разработчик

Бриджит — менеджер проекта

Акт II. План игры изменился, но мы все равно проигрываем

Шесть недель назад Джастин решил, что дела пошли в гору. Команда только что закончила шлифовку первого альфа-релиза фантазийного баскетбольного сайта, включающего NBA и европейские лиги. Но возникли проблемы.

Он вспомнил слова Даниэль: «Проблемы — это мягко сказано. Там столько ошибок, что черт ногу сломит». Все члены команды были расстроены, поэтому провели большое совещание, чтобы обсудить качество кода. Бриджит сказала: «Надо срочно что-то предпринять!» От участников встречи посыпались предложения: «нужно проводить больше оценок кода и самого проекта», «давайте устроим масштабное тестирование, для которого привлечем на несколько часов всех менеджеров продукта, чтобы разом улучшить все программное обеспечение», «нужно нанять тестера на полную ставку».

И тогда Даниэль рассказала им об XP.

— Нам нужно начать ее использовать, — сказала она. — Эта методика позволит решить наши проблемы с кодом.

Казалось, что все члены команды согласились, и Бриджит договорилась о выделении некоторых ресурсов, чтобы внедрить XP. Она получила задание дать команде несколько недель на апробирование новой практики, начиная

с недельной разработки модульных тестов. Поскольку никаких модульных тестов у них не было, вся команда целую неделю писала множество тестов для разных частей программы, чтобы успеть сделать как можно больше.

К сожалению, Бриджит не смогла убедить офис-менеджера позволить им сидеть вместе. Но возле кофейного автомата была пустая стена, и Даниэль вывесила на ней описания основных практик ХР, напечатанных крупным шрифтом на большом листе бумаги. Достав контрольный лист, она поставила галочку против пункта «информационное пространство». И после недели разработки модульных тестов отметила как выполненный пункт «разработка через тестирование». Джастин установил сборочный сервер и запустил на нем ежечасный контроль изменений кодовой базы, сборку кода и выполнение модульных тестов с рассылкой результатов всем членам команды. Появилась отметка о выполнении пункта «непрерывная интеграция».

Это было шесть недель назад. С тех пор был проделан большой объем работы. Так почему же Джастин считает, что ничего не изменилось?

Он и Даниэль только что вышли от Бриджит. Встреча не удалась, Бриджит очень разозлило, что исправление ошибок в базе данных управления игрой заняло больше времени, чем ожидалось.

Незадолго до этого Джастин и Даниэль поговорили и пришли к выводу, что не успеют закончить работу в срок. Но на общей встрече всей команды они заверили Бриджит, что закончат все вовремя, и совершенно ее успокоили. «Ничего, мы потом извинимся», — сказала Даниэль. Джастин знал, что им приходилось халтурить, чтобы уложиться в срок, но и этого было недостаточно. Они все равно не успевали. Но он был согласен с Даниэль, что пока не время посвящать в это Бриджит. Это только усугубит ситуацию и еще сильнее задержит проект. Он не был уверен, что всему виной именно ХР-практика, но и пользы от нее не было заметно.

Был перерыв, они с Даниэль пили кофе, и Джастин колебался, стоит ли выносить эти проблемы на общее обсуждение. Кроме того, он хотел знать ее мнение на этот счет.

— Что ты думаешь о том, как у нас внедряется ХР? — спросил он.

Даниэль с грустью сказала:

— Я не знаю, в какой момент мы сбились с пути. Эти практики должны были исправить наш код, но я только что потратила четыре часа на поиск ошибок в совершенно новом коде, который мы писали уже после внедрения ХР. Мне кажется, что мы повторяем прежние ошибки. Ничего не меняется!

— Так в чем же мы ошибаемся? — воскликнул Джастин.

Даниэль немного подумала и сказала: «Ну, например, мы больше не работаем парами. Первое время мы активно использовали этот метод, но спустя несколько недель он стал казаться неэффективным. Я не думаю, что кто-то распорядился перестать так работать. Просто один человек ушел в отпуск, и поэтому пара распалась».

Даниэль была права, но, похоже, никто из команды «слона-то и не приметил». Хотя никто не говорил о прекращении парного программирования, оно почему-то остановилось. То же самое произошло и с разработкой через тестирование: Джастин написал лишь несколько модульных тестов, а Даниэль поняла, что у нее на это вообще нет времени.

Несколько дней назад Даниэль все же начала переживать из-за отказа от парного программирования, поэтому решила взглянуть на код Джастина. Это был тихий ужас, все оказалось завалено закомментированными блоками кода — настоящая мешанина. Даниэль не побоялась спросить его прямо:

— Куда делся тот аккуратный код, который ты писал прежде?

Джастин вздохнул:

— Я бы с удовольствием писал сейчас хороший код, но сроки поджимают. У меня просто нет времени подумать.

И Даниэль поняла, что ему действительно не хватает времени — важна каждая минута. Да и Бриджит была уверена, что они считают недопустимым потерю хотя бы одного дня из-за ошибок. Так что, хотя они оба знали, что могли сделать работу лучше, срок сдачи приближался слишком быстро.

— Знаешь, почему так случилось? Потому что мы перестали использовать парное программирование. Ты и я работали в паре до тех пор, пока не обнаружилась серьезная ошибка в работающей части программы, — сказал Джастин. — Мне пришлось заняться ее исправлением, а ты не могла ждать. Поэтому код базы данных, над которым мы работали вместе, ты дописывала в одиночку. У нас просто не хватило времени на работу в паре. Нам вообще некогда заниматься всеми этими новшествами!

Даниэль сказала:

— Может да, а может нет. — Немного помолчав, она добавила: — Скажу честно, я никогда не писала модульные тесты перед основным кодом. Я программировала, как обычно, и добавляла их после завершения каждого объекта. Ведь не должно быть никакой разницы, правда?

Джастин задумался на минуту, а затем произнес:

— Значит, мы работаем так же, как и до начала внедрения XP.

XP-ценности помогают команде изменить мышление

Сами по себе практики непродуктивны. Вне связи с поставленными целями они становятся пустым звуком. Например, парное программирование не имеет смысла, если делать его для галочки. Работать парами ради того, чтобы угодить руководителю, означает просто разочароваться в этом методе. Парное программирование ради общения, получения обратной связи, упрощения системы, поиска ошибок и поддержания вашего мужества имеет гораздо больше смысла.

Кент Бек. Extreme Programming Explained: Embrace Change

В главах, посвященных Scrum, была приведена цитата Кена Швабера, объясняющая, что без коллективной ответственности и самоорганизации вы не сможете внедрить этот подход. Теперь вы знаете, как команда, не выполнившая этих требований, идет по пути применения отдельных scrum-практик, но не занимается самоорганизацией, — в итоге получает результат «лучше-чем-ничего».

Почти то же самое происходит с XP. Если ваша команда сопротивляется переменам, отстраняется от пользователей, которым нужна программа, непохожая на ту, что вы написали, и не верит в реальность создания продукта, который легко (или хотя бы возможно) изменить, то у вас не будет XP. И если вы не добьетесь простоты — в архитектуре и коде, корпоративной культуре и командной атмосфере, которые влияют на способность создавать простой код и предотвращать ошибки, — то вы также не получите XP. Оставшаяся часть этой главы посвящена важной теме — как научиться принимать изменения. В главе 7 будет рассказано о простоте и о том, как она поможет разработать программное обеспечение, которое проще изменить. Но чтобы лучше понять это, разберемся сначала, откуда берутся ошибки.

Так откуда же *появляются* ошибки?

Загляните в любой учебник по разработке программного обеспечения, и вы найдете ответ: **переделка***. Источников ошибок много, но чаще всего дефекты привносятся в ПО (термин *привнесение дефектов* вы найдете во многих учебниках по разработке программного обеспечения) именно при переделке. Команда разрабатывает ПО для решения определенных задач, но потом кто-то приходит и говорит, что нужны изменения. Приходится все переделывать: вырезать старый код, вносить изменения в суще-

* Загляните и в нашу первую книгу Applied Software Project Management. Мы были правы. Характерное мировоззрение, присущее традиционной разработке программ и управлению проектами, — избегать изменений, что радикально отличается от менталитета их принятия. Но все-таки это обоснованное мышление, и оно может оказаться совместимым с гибкой разработкой ПО.

твующий и ставить заплатки нового кода. Именно так чаще всего ошибки попадают в программный продукт.

Но разве нельзя предотвратить такие изменения? Теоретически можно. В том же учебнике по разработке программ есть ответ: документируйте требования к программному обеспечению. Если вы добросовестно собираете требования, записываете их и оцениваете вместе с заказчиками, то можете предотвратить необходимость изменений. Команда способна разработать продукт, наиболее соответствующий всем требованиям пользователей, клиентов и заинтересованных сторон. Конечно, изменения будут, но минимальные!

Однако это в теории.

Но, оказывается, это может работать и на практике. Многие команды создали достаточное количество превосходных программных продуктов, используя четкие требования*.

Но это очень жесткий способ создания ПО, потому что он работает только при условии, что люди, собирающие спецификации, действительно могут выяснить все (по крайней мере, все важные) требования в начале проекта. Но это подходит не для каждого случая. Поэтому проекты, использующие такие требования, нуждаются в системах управления изменениями или иных подобных средствах.

Именно для таких случаев особенно подходит ХР. Оно помогает команде создавать программы таким образом, чтобы внесение изменений наносило наименьший вред исходному коду. Переделка и внесение изменений вполне приемлемы и даже приветствуются, потому что команда пишет код так, чтобы его можно было править.

Но еще лучше то, что ХР предотвращает серьезную проблему — невозможность привносить действительно хорошие идеи. Очень часто они появляются в середине проекта. В самом деле, ранние версии ПО рождаются в ходе лучших мозговых штурмов. Но когда команда использует уже упоминавшийся метод разработки BRUF, разговоры, подобные приведенному ниже, заканчиваются безрадостно.

Программист: у меня есть отличная идея для кода!

Руководитель команды: идея хорошая, но это потребует слишком больших изменений. Запиши ее, и мы проведем ее оценку, когда сможем уложить наш бэклог в шесть месяцев.

* У нас тоже были такие проекты! И мы этого совершенно не стыдимся.

Программист: *но я могу изменить это за десять минут!*

Менеджер проекта: *это слишком рискованно. Мы видели, как даже небольшое изменение может вызвать волну ошибок во всей программе. Не волнуйся, мы не говорим тебе «нет», просто не сейчас!*

Услышать такое неприятно. Наверняка в следующий раз, если у этого программиста появится отличная идея, он не станет ею делиться.

Команда, боящаяся вносить изменения, сама душит собственные инновации. Это не значит, что она не сможет создавать ПО, — она будет это делать, и порой даже качественно. Если вы используете метод BRUF, то сможете выполнить заявленные требования. И это хорошая цель для многих команд. Но данный путь, безусловно, имеет свои минусы.

XP ПОМОГАЕТ ПРОГРАММИСТАМ НАУЧИТЬСЯ РАБОТАТЬ С ПОЛЬЗОВАТЕЛЯМИ

Многие команды, работающие с BRUF в водопадном подходе, получают успешный продукт после первого или второго релиза. И это не должно удивлять. Перед началом проекта, как правило, бывает много обсуждений. Существует конкретная потребность, которую все хорошо понимают, и достаточно небольшого толчка в нужном направлении, чтобы заставить компанию вложить ресурсы в данный проект. Это означает, что первая версия требований часто отражает все прошедшие обсуждения и включает лучшие идеи. Все уже устали от обсуждений, и команда с облегчением приступает к реальной разработке. Это отличный способ предотвратить изменения и доработки, потому что все продумано до мелочей и коррективы должны быть несущественными.

Более того, вторая версия наверняка содержит больше полезных функций, потому что у команды просто не было времени, чтобы включить их в первую. Так что для BRUF-команд характерны начальные успехи — создание первой версии программного обеспечения, которая хорошо работает и не требует значительных правок.

Проблема в том, что, когда проект достигает второго, третьего и четвертого релиза, люди уже давно используют это ПО. А поставка работающего программного обеспечения — отличный способ получить обратную связь от пользователей (именно поэтому agile-команды отдают предпочтение работающему ПО перед всеобъемлющей документацией). Теперь от пользователей поступила полезная обратная связь, а в большинстве случаев учет их пожеланий требует переделок.

Что происходит, если команда создала программу таким образом, что ее сложно изменять? Она начнет планирование, оценку запросов и установит, что внесение этих изменений влечет за собой высокий риск появления дефектов в коде и поэтому потребует много времени на реализацию. Это сформирует у членов команды вполне объяснимый страх перед изменениями. И, в свою очередь, заставит клиентов и стейкхолдеров искать способы изменить свои требования, чтобы они могли быть реализованы на базе существующего кода. Партнерство между пользователями и командой, возникшее после первой версии ПО, может легко превратиться в антагонизм — тому, кто превращает выявление требований в переговоры по контрактам, трудно сотрудничать с клиентами.

XP помогает командам избежать этой проблемы, предоставляя практики, которые помогают создавать базу кода, легко поддающуюся изменениям. (Вы больше узнаете об этом в главе 7.)

Но XP также изменяет представление каждого члена команды о качестве. Качество — не просто способ исключить ошибки. Это возможность предоставить пользователям такое программное обеспечение, которое отвечает их потребностям, даже если это не совсем то, о чем они изначально просили. Традиционные BRUF-команды отделяют проектирование функций (что делает программа) от разработки внутреннего устройства (как она устроена). Функциональность записывается в спецификации, которые затем передаются команде для реализации. Если программистам что-то непонятно или функциональность кажется неверной, они могут задать вопросы тому человеку, который подготовил спецификацию. Затем начинается «игра в испорченный телефон» — автор спецификаций опрашивает пользователей, менеджеров и тех, с кем еще не говорил, и переводит их требования на язык, понятный программистам. BRUF-команды считают это очень эффективным способом, потому что разработчики «не убивают» время на разговоры с клиентами. (Всем известно, что у программистов слабо развиты навыки общения, не так ли?)

Особенность XP в том, что она заставляет разработчиков думать как пользователи — и действительно, все начинается с реального разговора с ними. Если программисты и не имеют навыков общения с клиентами, то они поневоле начнут их получать. XP-практики помогают в этом. Например, возьмем разработку через тестирование. Программисты пишут тесты до создания кода. Это предотвращает появление ошибок и заставляет каждого разработчика задуматься над тем, что должен делать код, прежде чем его написать. Члены команды, выработавшие привычку думать о функциональности, прежде чем создавать любой код, начинают поступать так же при планировании недельных циклов.

XP также позволяет избежать ситуации, присущей спецификациям, в которых каждый просто спасает свою шкуру. Разработчикам легче реализовать то, что прописано в спецификации, не задумываясь о реальных потребностях пользователя. Но если он привык перед началом работы над кодом задаваться вопросом «Что должна делать программа?», то начинает с выявления проблем и будет чувствовать себя комфортно, требуя ответов, потому что они нужны для написания тестов. Программист не будет сдвигать ответственность на того, кто подготовил и передал ему спецификацию. Когда все в XP-команде поступают именно так, они постоянно обращаются за разъяснениями к пользователям и стараются выяснить, что им нужно от ПО, чтобы выполнять свою работу. Коммуникация происходит непосредственно, без промежуточных звеньев, что экономит массу времени.

Другими словами, BRUF позволяет создавать то, что вы *задумали*, а XP — то, что действительно *нужно* пользователям. И все счастливы: члены команды — потому что могут тратить время на решение реальных проблем, а не бороться с кодом, внося в него правки, а пользователи — потому что получают то, что им действительно нужно (а не то, что они хотели полгода назад). В результате формируется команда, включающая пользователей, клиентов и стейкхолдеров. Им не нужно предугадывать, что потребуется через полгода, а можно сосредоточиться на создании ПО, соответствующего текущим нуждам.

Так как же XP добивается этого?

ПРАКТИКОВАТЬ ТОЛЬКО МЕТОД КНУТА, ЕСЛИ КОМАНДА ДЕЙСТВИТЕЛЬНО В НЕГО ВЕРИТ

XP-практики помогают концентрироваться на качестве кода, который вы создаете, еще *до того*, как будет написана его основная часть. Если Scrum всецело направлена на то, чтобы пользователи знали предел ваших возможностей в создании ПО, то XP позволяет вносить изменения и удалять дефекты максимально быстро. И это помогает всем членам группы выработать такое отношение к качеству, которое проходит красной нитью через все работы в проекте.

Это новый способ мышления для многих технических специалистов и даже высококвалифицированных разработчиков. Обычно несложно переключить технарей на Scrum, особенно на версию «лучше-чем-ничего», если они чувствуют, что плохое планирование приводит к проблемам в проекте. Scrum действительно помогает командам лучше планировать. Но что произойдет, когда разработчик, изучая практики XP, обнаружит, что они вынуждают его *изменить методы своей работы*?



Рис. 6.3. Очень распространенная реакция на внедрение ХР заключается в том, что, хотя в теории это звучит хорошо, на практике парное программирование и разработка через тестирование «непригодны для нашей команды». Это не обязательно аргумент против данных практик, а скорее предчувствие сложности их реализации

Основополагающий принцип создания программного обеспечения — существование практически неограниченного количества способов, которыми можно написать программу. Попросите двух программистов создать одинаковый модуль (класс, функцию, библиотеку и все, что относится к использованию языка или рабочей среды), и практически наверняка они напишут разный код. Сложно ли вообразить, что один из них создаст более качественный код, чем другой?

Очевидно, что навык программирования очень важен, но это еще не все. Многие команды, состоявшие из отличных разработчиков, поддерживали неудачный код. Чаще всего код становится таким, когда в него спешно вносят множество незапланированных изменений. И в итоге масса ошибок, а также страх переделок, который сковывает инициативу команды.

Это довольно распространенная причина, по которой команды обращаются к ХР. Поэтому многие из них вместе с руководителями будут уделять основное внимание ХР-практикам, направленным на поиск и предотвращение ошибок. Парное программирование — удачная идея, потому что позволяет следить за кодом в четыре глаза, что помогает находить больше ошибок. Разработка через тестирование означает, что команда будет писать и поддерживать тесты для выявления ошибок, которые всегда появляются при

программировании, а непрерывная интеграция подразумевает, что эти тесты будут запускаться все время. Отличная идея, не так ли? И это, безусловно, поможет выявить больше ошибок.

Команда, получающая от XP результат «лучше-чем-ничего», обнаружит, что парное программирование предотвращает попадание дефектов в ПО, а разработка на основе тестирования и непрерывная интеграция находят дефекты, когда они уже попали в код. Но подобные команды обычно не готовы прикладывать много усилий для внедрения таких практик. Необходимый минимум для получения готового проекта — написать код.

Если команды отстают от графика, то они стремятся делать только то, что крайне необходимо, отбросив всякие «украшательства». Обычно они говорят: «Мы опаздываем, поэтому у нас нет времени писать модульные тесты» или «Было бы здорово, если бы мы могли назначить на каждую задачу по два программиста, но тогда мы сорвем сроки сдачи проекта».

Это характерное мышление для команды, которая впервые применяет XP. Проблема заключается в том, что такой настрой сродни желанию сесть на диету или заняться спортом: мысль отличная, но где взять на все время? Даже если люди скрупулезно выполняют все рекомендации этих методик, но рассматривают их как дополнение к обычному процессу, они с легкостью отбросят их в случае отставания от графика.

Именно поэтому вредно, когда команды трактуют XP-практики как список мероприятий. Еще хуже, если они начинают оценивать, на сколько процентов внедрились XP, которая в этом случае рассматривается как результат, а не как средство достижения цели. Это надо рассматривать как красный сигнал светофора, показывающий, что люди не готовы к применению XP.

Команда, неправильно воспринимающая XP, должна изменить образ мышления. Смена мышления может показаться серьезным достижением, особенно если команда не допускает мысли, что этот метод оправдывает затраченное на него время. Так что же заставляет команду трактовать XP-практики как «первоклассные» и жизненно важные для создания отличного ПО, а не в качестве необязательного дополнения?

Эффективное мышление начинается с ценностей XP

Команды, имеющие правильное мышление, всегда с готовностью используют отличные практики, потому что *точно знают*, что они приводят к улучшениям. Так же как Scrum, XP имеет пять **ценностей**. И так же они помогают команде правильно внедрять XP и избегать результата

«лучше-чем-ничего». Эти ценности позволяют разработчикам изменить мышление, чтобы XP-практики казались естественным путем создания хорошего программного обеспечения. И наоборот, правильное XP-мышление означает понимание того, что эти практики обеспечивают создание отличного кода.

ЦЕННОСТИ XP

Коммуникация

Каждый член команды знает, что делают остальные.

Простота

Разработчики стараются создать максимально простое и прямое решение.

Обратная связь

Постоянное тестирование и обратная связь держат качество продукта под контролем.

Мужество

Каждый член группы нацелен на выбор лучших решений для проекта, даже если это означает отказ от неудачных решений или требует иного подхода.

Уважение

Каждый член команды ценен для проекта.

Все это абстрактные азбучные истины. Разве можно с ними не соглашаться?

(Не исключено, что вскоре вы обнаружите *свое* несогласие с ними... и это нормально, пока ваш разум открыт для них.)

ПУТЬ, ВЫМОЩЕННЫЙ БЛАГИМИ НАМЕРЕНИЯМИ

Когда команда начинает внедрять XP, все настроены оптимистично. Они знают, что есть проблемы с качеством кода, устали от суеты и ошибок, и им кажется, что XP-практики смогут это исправить.

Потом приходит реальность. Нам кажется излишеством, когда кто-то сидит рядом с нами, пока мы обдумываем и пишем код. Если единственная

польза от парного программирования — дополнительная пара глаз, высматривающая ошибки, то для нее есть более эффективное применение. Анализ кода после его написания позволяет отлавливать ошибки с такой же вероятностью (точно такой же анализ можно провести после сессии парного программирования). Но если у вас мало времени из-за приближающегося срока сдачи, то легко убедить себя в том, что обзор кода почти так же хорош, как и парное программирование. «Это разновидность парного программирования, потому что я подключаю кого-то еще!»

Почти каждая XP-практика имеет схожее упрощение. Вам необязательно сидеть вместе, достаточно ежедневных встреч. Членам вашей команды не-зачем постоянно интегрировать код в своих песочницах, можно настроить сервер, чтобы он делал это за вас автоматически. Нет необходимости начинать с написания тестов, можно сделать это после написания кода и все-таки обнаружить ошибки.

Все эти утверждения справедливы. Но в каждом случае это означает делать «лучше-чем-ничего» и, соответственно, получать такие же результаты.

Вернемся к цитате Кента Бека, приведенной выше: «Сами по себе практики непродуктивны. Вне связи с поставленными целями они становятся пустым звуком». Примерно то же самое Кен Швабер сказал о Scrum, самоорганизации и коллективной ответственности: каждая XP-практика делает гораздо больше, чем просто обеспечивает лишний взгляд на код, заставляет людей сидеть в одной комнате или добавлять тесты в базу кода. Они помогают команде привнести XP-ценности в проект.

Вот почему упрощения, которые делают, казалось бы, то же самое, что XP-практики, дадут лишь результат «лучше-чем-ничего». Они меняют методику работы команды, но не влияют на ее намерения и образ мышления.

Обычно при знакомстве с XP команды не обращают внимания на ценности, потому что практики кажутся им интереснее. Многие разработчики открывают книгу об XP, быстро просматривают раздел, посвященный ценностям, а затем ищут практики.

Практики *конкретны*. Оценив практику, вы можете представить себе, как это делается, и понять, какое влияние она оказывает на проект. Практики *изолированы* — можно добавить отдельную практику в свой проект, не меняя понимание того, что делаете вы и ваша команда.

Воспринять ценности командам гораздо сложнее. Они более абстрактны, оказывают влияние на весь проект и на то, как каждый его воспринимает. Можно механически добавить практики в проект, не понимая ценности

(что дает результат «лучше-чем-ничего»). Но разобраться в ценностях без предварительного опыта работы с практикой — труднее.

Для XP в этом нет ничего особенного. Это же относится и к Scrum. Для вводных курсов по Scrum типично показать ценности на паре слайдов и рассказать о них в двух словах, а затем сразу перейти к «сути» тренинга — практикам. Но не стоит слишком многого требовать от тренеров. Гораздо проще обучать применению отдельных практик XP или Scrum, чем пытаться изменить мировоззрение слушателей. Большинство тренеров знают, что, если они уделяют много внимания ценностям, то слышат от разработчиков такие упреки: «слишком много теории» и «это не дает им информации, которую мы можем применить в своих проектах». Эта реакция вполне объяснима: трудно понять, почему ценности имеют практическое значение, пока вы их не усвоите. И невозможно сразу понять, почему практики не работают по-настоящему без ценностей, так что разработчики попадают в ловушку.

Вернемся к Джастину, Даниэль и их фантазийному баскетбольному проекту. Они сразу занялись внедрением XP-практик, не найдя времени разобраться с ценностями. Даже если вы не знаете деталей их ежедневной работы, попробуйте установить, что они делали неправильно, пытаюсь внедрить XP.

Вот несколько примеров.

Коммуникация

Даниэль и Джастин не говорили о проблемах с применением нужной практики. И они точно не рассказывали Бриджит о реальном состоянии дел, так что в случае проблем с проектом она легко обвинила бы их в том, что они скрывали правду.

Простота

Код Джастина становится все сложнее и запутанней. Он способен работать лучше, но у него не хватает времени.

Обратная связь

Когда Джастин и Даниэль перестали работать в паре, они почти прекратили общаться. Если бы Даниэль продолжала просматривать его код и давала ему обратную связь, то, возможно, не было бы такого беспорядка. И несмотря на то, что рабочее пространство дает больше

информации, она не помогала людям принимать более обоснованные решения. Выяснение, «насколько глубоко команда освоила XP», не делает программное обеспечение лучше.

Мужество

Если команда не надеется закончить проект вовремя и понимает это, то требуется много мужества, чтобы рассказать правду, — особенно если за это грозит увольнение. У Даниэль и Джастина не хватило мужества.

Уважение

Требовать от команды выполнить проект в заведомо нереальные сроки — это верх неуважения к ней*. Бриджит неоднократно устанавливала невероятно жесткие сроки. Хуже то, что она не чувствовала потребности быть рядом с людьми и знать, чем они живут. Например, как в случае, когда она назначила встречу вечером в пятницу и потребовала от команды внести множество исправлений к утру понедельника. И пока разработчики все выходные трудились в поте лица, она отдыхала. (Это не вошло в главу, но нам доподлинно известно, что так все и было.)

Когда XP-ценности по-настоящему не усвоены каждым членом команды, она в лучшем случае может получить результат «лучше-чем-ничего». Люди будут говорить, что некоторые практики — чаще всего парное программирование и разработка через тестирование — «непригодны для нашей команды». Команда не выдвинет реальных аргументов против них, но у ее участников создастся ощущение, что изменить это трудно, и они предпочтут просто составить список мероприятий для внедрения, в котором программирование и разработка через тестирование будут на последнем месте.

Если у XP-команды правильный настрой, то она действительно принимает эти ценности. И когда она так поступает, происходит изменение в восприятии XP: практики начинают обретать смысл.

Например, когда команда приступает к внедрению непрерывной интеграции, происходит примерно следующее. Поначалу многие команды этого избегают, считая, что им необходимо настроить сервер сборки (требующий времени и дополнительного компьютера). Но при непрерывной

* Как ни странно, некоторые менеджеры обманывают себя, думая, что команды работают продуктивно только благодаря такому давлению. [Они получают такую производительность команды, которой заслуживают.]

интеграции можно обойтись только маркером сборки, например забавной игрушкой. Представьте, что команда — новичок в ХР начинает это делать. Первый человек, получающий маркер, тратит несколько минут, чтобы интегрировать код из системы контроля версий в свою песочницу, проверяет, работает ли программа, а потом вносит измененный код обратно в систему контроля версий. Маркер передается от одного разработчика к другому, и каждый из них проделывает описанную выше процедуру.

Как это помогает команде познакомиться с ХР-ценностями?

Часто маркер сборки остается у одного разработчика довольно долго, потому что ему кажется, что он не может прервать текущую работу. Команда должна дать этому программисту понять, что непрерывная интеграция важна для нее, — это научит его коммуникации и поможет глубже понять ценности. Этот упрямый разработчик усвоит нужность уважения к другим, потому что ему придется поставить непрерывную интеграцию в интересах команды выше любой текущей задачи, даже если он считает ее высокоприоритетной. Есть и другие знания о ценностях, которые усваиваются при внедрении непрерывной интеграции. Команда может понять значение обратной связи (когда очередная интеграция выявляет большие проблемы, однако экономит общее время), мужества (ведь если в этой проблеме виноват кто-то другой, ему необходимо сообщить об этом, что не всегда легко) и многого другого.

Вот как использование ХР-практик помогает формировать мышление команды.

Но вам не обязательно приступать к внедрению практик, чтобы узнать, совместимо ли мышление команды с ХР. Для этого достаточно ответить на несколько вопросов.

- Как воспримет команда ситуацию, когда часть написанного кода придется выбросить, потому что он не работает? Как отнесется к этому руководство?
- Если руководитель хочет сократить сроки, то действительно ли команда будет считать, что создание модульных тестов — это кратчайший путь к цели, даже если для этого придется написать больше кода?
- Если младший программист получает задачу, то дает ли ему команда полномочия довести ее до конца? Что если она не согласна с его подходом или кто-то считает, что он выполнит ее быстрее? Допустимо ли, что разработчик может не справиться с заданием, но извлечет из этого урок?

Так что же вы предпринимаете, когда ваша команда не до конца освоила ХР-ценности?



Описание: команда занимается разработкой фантазийного баскетбольного сайта

Джастин — первый разработчик

Даниэль — второй разработчик

Бриджит — менеджер проекта

Акт III. Динамика изменений

— Джастин, произошло что-то очень странное.

Джастин полностью погрузился в работу, занимаясь алгоритмом ранжирования игроков по нескольким характеристикам, но отвлекся и взглянул на Даниэль.

— Помнишь, парное программирование нам не очень помогло?

— Да, — сказал Джастин. — Сначала ты смотрела, а я писал код, потом мы менялись. Работы было много, а толку мало. Я посчитал количество строк, которые мы писали за час, и оказалось, что наша производительность падала примерно вдвое. Думаю, именно поэтому мы постепенно перестали им заниматься.

— Вообще-то сейчас мы очень хорошо поработали в паре с Тайлером, — сказала Даниэль.

— Тайлер? Этот новичок? Прошло чуть больше месяца, как он окончил колледж. Неужели из вашей затеи вышло что-то интересное?

Даниэль кивнула:

— Я сама удивилась. Мы начали работать в паре, потому что я решила, что так он быстрее войдет в курс дела. Но когда мы дошли до кода, описывающего кэш данных игроков...

Джастин перебил:

— О боже! Это был тяжелый труд.

— Я помню. А знаешь, он поинтересовался, почему мы не стали хранить ключи и хэши вместе с объектами игрока.

Джастин от удивления разинул рот. В системе имелся кэш данных, которые они с Даниэль написали, чтобы исправить некоторые серьезные проблемы с производительностью. Этот код был очень сложный, поэтому ему потребовалось несколько минут, чтобы снова его просмотреть.

— Постой, так значит, у нас в кэше будут неверные данные!

— Да, — сказала Даниэль.

— И Тайлер, этот желторотик, во всем разобрался?

— Да, мы провели небольшое тестирование и выявили проблему. Когда он это обнаружил, внести исправления было уже нетрудно. Не сделай мы правку вовремя, нас в дальнейшем ждала бы серьезная головная боль. Игрок бы видел неправильные параметры. Не исключено, что мы обнаружили бы эту ошибку только после выхода игры.

У Джастина была копия списка ценностей ХР, которую он в свое время распечатал, прикрепил на стену рядом с собой и благополучно забыл о ней. Теперь он перечитал ее и сказал:

— Знаешь, я действительно не уделял этому внимания. Думаю, что сейчас я кое-что узнал о коммуникации.

— И об уважении, — добавила Даниэль. — В дальнейшем я обязательно буду интересоваться мнением Тайлера. И думаю, что продолжу работать с ним в паре.

Понимание принципов ХР поможет вам принять изменения

Существует разница между ценностями и практиками ХР. Ценности — понятие довольно широкое, они направляют вас в сторону размышлений о совместной работе. Но новичкам в ХР бывает трудно применить их на практике.

К счастью, ХР имеет ряд **принципов**, которые сориентируют вас, как применять практики в реальных проектах. Эти принципы подсказывают не только как организовать жизнь в команде, но и как довести проект до конца.

Ниже приведен список принципов ХР. Он может показаться длинным, но имейте в виду, что их не обязательно запоминать. Ценности ХР охватывают широкий круг вопросов, поэтому в принципах отражены многие детали того, как члены ХР-команды воспринимают возникающие проблемы. Это придает им значимость, помогая выяснить, обладает ли ваша команда мышлением, необходимым для использования ХР.

Гуманизм

Помните, что программное обеспечение создается людьми и существует баланс между потребностями каждого члена команды и самого проекта.

Экономика

Всегда есть тот, кто платит за разработку программного обеспечения, и каждый должен учитывать размер бюджета.

Взаимная выгода

Ищите практики, которые одновременно приносят пользу отдельному программисту, команде и клиенту.

Сходство

Месячный, недельный и дневной циклы строятся по одному шаблону.

Улучшение

Выполняйте максимально хорошо сегодняшнюю задачу и думайте о том, как сделать так, чтобы завтра работать еще лучше.

Разнообразие

Объединяйте различные мнения и взгляды, чтобы получить наилучший результат.

Рефлексия

В ходе разработки программного обеспечения хорошие команды постоянно обсуждают правильность своих действий по реализации проекта.

Поток

Непрерывная поставка означает постоянную доставку результатов труда разработчиков, не разделяемую на этапы.

Возможность

Каждая проблема, встающая перед командой, — это шанс узнать что-то новое о разработке программного обеспечения.

Избыточность

Хотя на первый взгляд это может показаться расточительным, избыточность позволяет избежать серьезных проблем с качеством.

Неудача

Неудачи могут многому научить. Нет ничего предосудительного в том, чтобы пробовать подходы, которые не работают.

Качество

Нельзя обеспечивать скорость поставки за счет снижения качества продукта.

Принятие ответственности

Если кто-то берет на себя ответственность, то он должен иметь полномочия для выполнения обещанного.

Маленькие шаги

Лучше делать маленькие шаги в правильном направлении, чем вносить грандиозные изменения при внедрении новых методов.

ПРИНЦИПЫ ХР

Эти принципы полезны, чтобы понять некоторые конкретные практики, которые использует ХР. А когда вы начинаете изучать практики, они помогают разобраться в принципах.

Так же как в случае с ценностями, у тех, кто впервые сталкивается с ХР, возникает стремление перейти сразу к практике, минуя принципы. По ряду причин это оказывается путем наименьшего сопротивления. Вы можете добавить практику, не осознавая, что с проектом что-то не так. («Мы все делаем отлично, но можем сделать еще лучше!») Миновав изучение ценностей и принципов и сразу перейдя к практикам, вы можете просто составить их список и отмечать галочкой принятие каждой из них. Если ваша цель — «полное» внедрение всех ХР-практик, то этот подход сработает отлично. Но если вы хотите помочь команде создать лучшее программное обеспечение, то «списочный» подход к ХР закончится провалом. Как максимум вы получите результат «лучше-чем-ничего», а со временем практики начнут исчезать и команда вернется к старому способу работы.

Очень знакомая картина для ХР- и scrum-команд. Мы говорили со многими разработчиками, прошедшими через это. Люди, как правило, начинают обвинять методологию: «Мы прилагаем все усилия для написания тестов (парного программирования, ежедневных scrum-митингов и т. д.), но это не убеждает нас. Да такая методология и не способна работать».

Принципы помогают понять, почему это происходит. Принципы — это самоанализ. Они заставляют вас думать о том, как вы и ваша команда делаете свою работу. Когда вы тратите время, чтобы понять ценности и принципы, вы начинаете узнавать о проблемах команды. Такой опыт редко доставляет

удовольствие. Например, вы можете взглянуть на принцип, касающийся неудач, и осознать, что культура вашей команды не признает права на ошибку. Не исключено, что вы сталкивались с проблемой, которая заставляла вашего руководителя накричать на вас. Члены команды могут перестать уважать вас как человека, который постоянно ошибается. Во многих компаниях факт признания собственной ошибки имеет серьезные последствия*.

Вот почему люди во многих командах поступают вполне разумно, когда игнорируют ценности и принципы или поддерживают их на словах, ничего не меняя на деле. Поставьте себя на место тех, кто понимает, что культура их команды находится в противоречии с принципом XP, признающим право на ошибку. Что произойдет, если вы поднимете этот вопрос и попросите коллег изменить стиль работы?

Либо вы сможете изменить культуру команды, либо коллеги и руководитель так разозлятся на вас, что останется только уволиться. Причем последний вариант встречается намного чаще, чем первый. Это одна из причин, по которой люди считают, что Agile — это трудно.

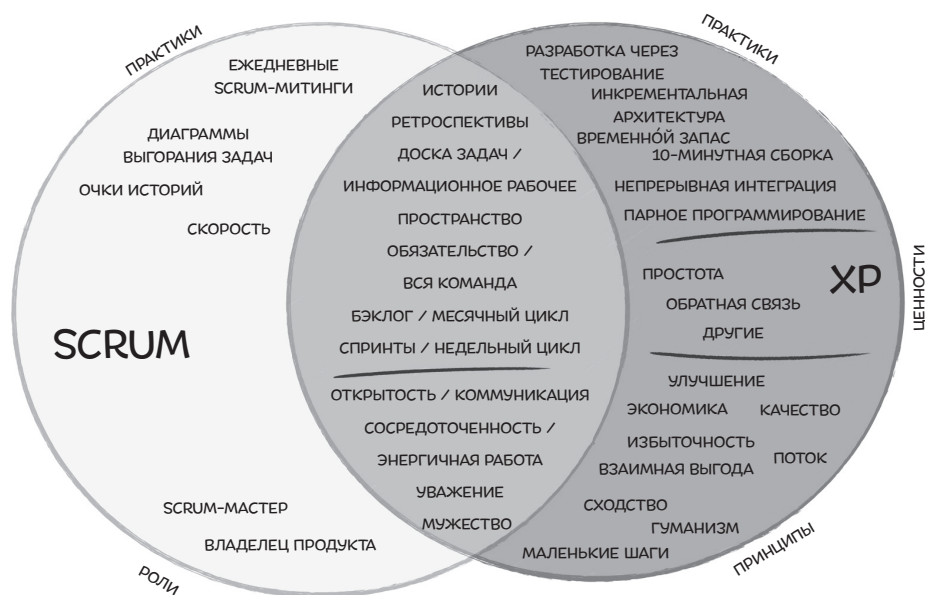


Рис. 6.4. Scrum и XP похожи, но все-таки различаются

* Некоторые специалисты по внедрению Agile считают ситуацию, когда команда «не смогла», забавной, что приводит к неприятию этой методологии. То, что подходит одной команде, может развалить другую. Постарайтесь с сочувствием отнестись к тем, кто приступил к внедрению Agile.

XP-ПРИНЦИПЫ ПОМОГАЮТ ВАМ ПОНЯТЬ ПЛАНИРОВАНИЕ

В зрелой XP-команде нет фиксированных ролей. Цель в том, чтобы каждый человек вносил в успех команды все лучшее, на что он способен. В начале освоения жесткое распределение ролей может помочь в изучении новых навыков, дав возможность технарям принимать технические решения, а менеджерам — бизнес-решения. После того как между членами команды установятся доверительные и уважительные отношения, фиксированные роли мешают каждому максимально проявить себя.

Кент Бек. Extreme Programming Explained: Embrace Change

Неслучайно проблемы мышления, вытекающие из столкновения между командой и ценностями, влияют на Scrum и XP. У этих методологий много общего, но есть и различия. Например, роли: Scrum выделяет роли менеджера продукта и scrum-мастера, в то время как зрелая XP-команда не имеет фиксированных ролей. Чтобы узнать об XP, полезно сконцентрироваться на принципах, которых нет в Scrum. Особенно ценно изучить, как XP-команды планируют проекты.

Значительная часть Scrum посвящена различным видам планирования: общей картины с использованием бэклога продукта, итераций — при помощи бэклога спринта, а также вовлечению всех членов команды в сотрудничество по составлению планов при помощи ежедневных scrum-митингов и других практик. XP использует похожие итерационные практики: ежеквартальный цикл для планирования общей картины, недельный цикл — для управления итерациями. Команда также использует информационное пространство, которое содержит инструменты отслеживания, например доску задач.

Но планирование и отслеживание XP-проекта отличается от этих же действий в проекте Scrum, и принципы помогают нам понять почему. Отчего XP-проект не имеет конкретных ролей? Это тот случай, когда XP-команды очень серьезно относятся к принципам **«возможность»** и **«разнообразие»**. Бывает, хотя и редко, что в scrum-команде владелец продукта или scrum-мастер переключаются на разработку программной архитектуры и проектирование, а член команды берет на себя руководство работой с пользователями или занимается бэклогом. XP-команды отвергают разделение ролей по двум причинам. Первая заключается в том, что люди, ограниченные одной ролью, упускают возможность расширить свой вклад. А вторая — в том, что каждый в команде может взглянуть на проблему с неожиданной стороны, что может стать ключом к ее решению. Иногда высококвалифицированный разработчик предлагает хорошую идею по работе с пользователями или руководитель вносит ценный вклад в создание архитектуры именно потому, что они смотрят на проект иначе, чем те, кто непосредственно занимается этими вопросами.

Такое понимание принципов меняет практику. Занимаясь фантазийным баскетбольным проектом, Даниэль узнала, что парная работа с новым программистом, который никогда не видел код, может дать совершенно иное видение. Это вносит в работу разнообразие: хорошие идеи приходят отовсюду, даже от новых членов команды. Вот почему многие XP-команды чередуют партнеров во время парного программирования, вместо того чтобы назначать постоянные пары. Это также вносит **разнообразие** и в работу пар, помогает взглянуть на код свежим взглядом, выявлять больше проблем и стимулировать инновации. Принцип **гуманизма** тоже играет свою роль. Привлечение разных людей и их совместная работа помогают создавать среду, где все дают друг другу обратную связь, что помогает каждому научиться принимать мнение коллег и даже их критику, сохраняя при этом взаимное уважение. Это дает мужество молодым членам команды высказывать свое мнение о проблемах, даже если они работают в паре с опытными программистами. Такие принципы, как разнообразие и гуманизм, в сочетании с такими ценностями, как коммуникация, уважение и мужество, помогают команде превратить парное программирование в эффективный инструмент.

Принципы также помогают понять, почему XP не имеет конкретной практики, позволяющей проанализировать сделанную работу, как при ретроспективах в Scrum. XP опирается на такие ценности, как **улучшение** и **рефлексия**, и XP-команды постоянно обсуждают то, как они делают работу, и способы ее улучшения. Но они склонны к самокопанию, углубляясь в рефлексию и отвлекаясь от непрерывного выполнения задач. Это тот случай, когда парное программирование способствует тому, чтобы люди *в процессе работы* обсуждали свои действия.

Принцип **маленьких шагов** очень полезен: вместо того чтобы сразу устанавливать глобальную цель («давайте внедрим все XP-практики»), команда может сделать к ней небольшой шаг («давайте начнем парную разработку, и тогда каждый день мы будем становиться лучше»).

XP-ПРИНЦИПЫ ПОМОГАЮТ ПОНЯТЬ ПРАКТИКИ — И НАОБОРОТ

XP-команды используют **истории**, и если вы посмотрите на любую из них, то она окажется точно такой же, как scrum-история. Но хотя XP- и scrum-практики похожи, можно многое узнать, наблюдая, как XP-команды используют истории, применяя принципы.

На рисунке 6.5 показана типичная история, которую XP-команда может использовать в проекте. XP, так же как Scrum, не вводит универсального формата для пользовательских историй. В данном случае команда решила

применить свободную форму предложений (а не стандартный повествовательный формат) и писать на лицевой стороне карточки такое количество часов, какое они посчитали необходимым для реализации истории.

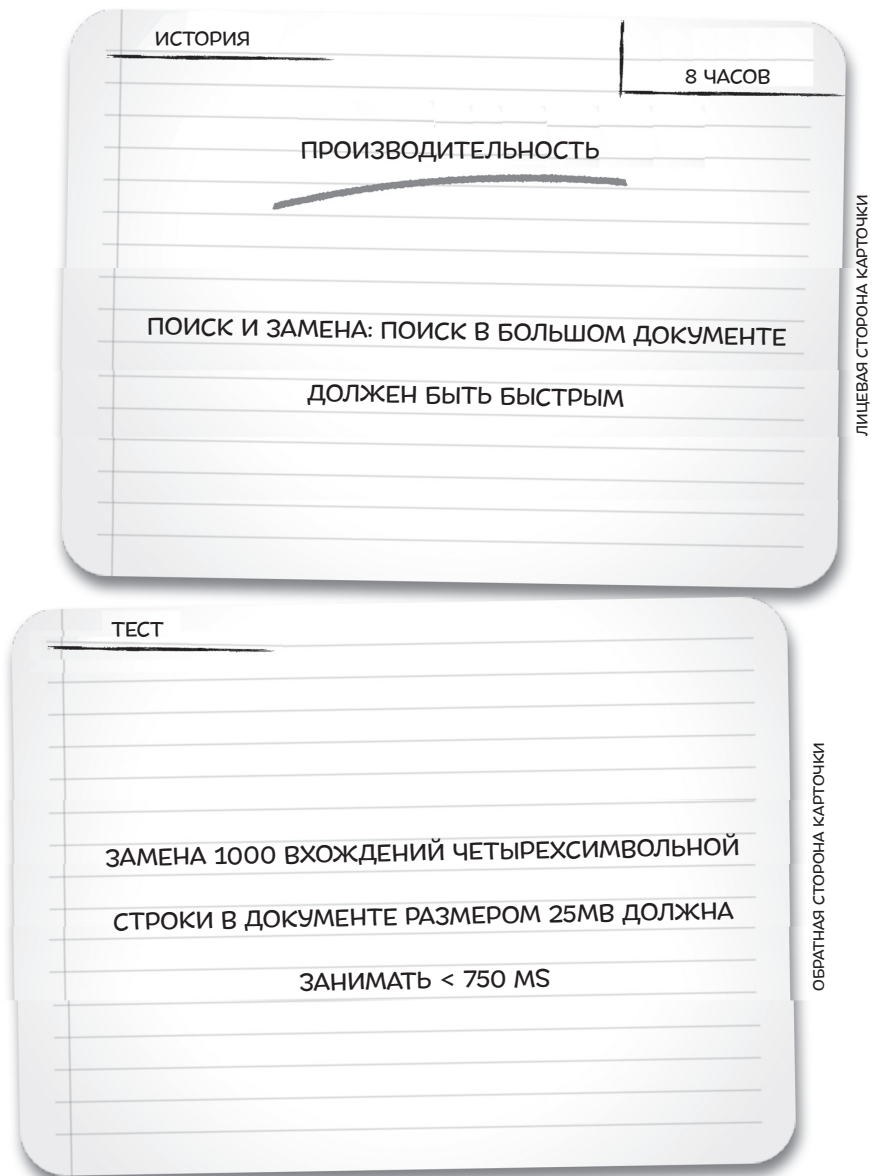


Рис. 6.5. XP-команды используют те же истории, что и scrum-команды, но применяют к ним свои ценности и принципы

Вы уже знаете, как scrum-команда использовала бы эту историю: создала бы ее как часть бэклога, встроила бы в спринт и разместила на доске задач. А как бы использовала это XP-команда? Она поступила бы аналогично, но применила бы еще и XP-принципы, чтобы удачнее включить ее в проект.

Экономика

Клиенты подбирают следующие истории так, чтобы это помогало в разработке, что позволяет команде проекта сосредоточиться на создании только самых ценных историй.

Неудача

Истории небольшие и не связаны между собой. Команда может быстро создать историю и передать ее клиенту, поэтому, если программное обеспечение окажется неудачным, она в сотрудничестве с пользователями сможет внести необходимые изменения.

Принятие ответственности

Программист прочитал карточку задачи, оценил, сколько времени потребуется на выполнение работы, и тем самым взял всю ответственность на себя.

Коммуникация

Записывая историю на языке, понятном для пользователя, проще определить приоритеты в работе.

Качество

Обдумывая заранее, как вы будете тестировать историю, вы способствуете поставке продукта высокого качества.

Благодаря тому, что вы уже понимаете, как работают пользовательские истории и как применяют их проектные команды, вы сможете отталкиваться от них, чтобы лучше понять перечисленные принципы.

ПЕТЛИ ОБРАТНОЙ СВЯЗИ

Обе методологии придают большое значение **обратной связи**. Мы уже видели, как Scrum использует петли обратной связи, чтобы получить непрерывную коммуникацию между командой и заказчиком. Команды, применяющие эти методологии, разделяют очень похожие ценности — **открытость**

(Scrum) и **коммуникацию** (XP). Они сходным образом воспринимают способы общения разработчиков в команде.

Но XP придает гораздо большее значение обратной связи. XP-команды нацелены на очень короткие итерации (недельный цикл) для улучшения качества обратной связи и укорочения петли. Изучая Scrum, мы узнали, что для эффективной итерации команда должна активно интересоваться потребностями пользователей и понимать, что для них ценно. Чтобы сделать петли обратной связи короче, программисты постоянно анализируют ход проекта.

Если возникает проблема, то команда будет ее обсуждать. Осмотическая коммуникация, происходящая тогда, когда люди сидят рядом, помогает распространять эту обратную связь внутри команды. Если совместить все эти приемы, то команда может достичь в своей работе состояния **потока**. Этот XP-принцип говорит о том, что прямой и эффективный способ работы над проектом заключается в обеспечении постоянного потока функционального и высококачественного ПО.

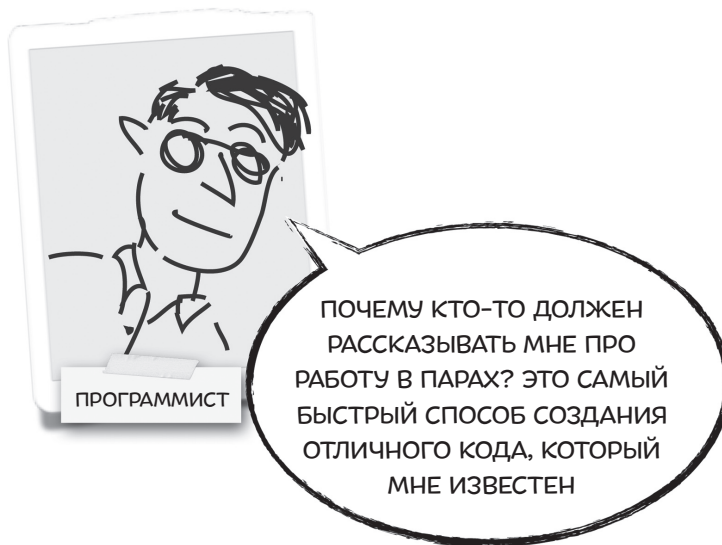


Рис. 6.6. Когда мышление команды соответствует XP, ее практики помогают эффективнее создавать программное обеспечение

Различных XP-практик очень много — размещение рядом друг с другом, осмотическая коммуникация, петли обратной связи, коммуникация, рефлексия. Их объединяют, чтобы помочь командам найти эффективный путь выполнения проекта. Но поток происходит только тогда, когда много компонентов работают вместе: методы, принципы, идеи и, прежде всего,

хорошие архитектура программного обеспечения и кодирование. Когда команды достигают состояния потока, они реализуют бесперебойную поставку программного обеспечения. Активная коммуникация и постоянная обратная связь приводят к тому, что люди начинают видеть узкие места на пути создания ПО и учатся обходить препятствия как единая команда. Они ликвидируют эти узкие места, и поток становится свободным.

Этого нельзя достичь в одночасье. Команда должна затратить время не только на создание программного обеспечения, но и на обсуждение того, как она его пишет, и изучение, как каждый принцип и практика могут помочь улучшить работу. Кроме того, принцип маленьких шагов даст команде ощущение комфорта при постановке целей, так же как короткие петли обратной связи ведут к потоку. Методология как целое становится дорожной картой: команда может добавлять практики по одной, руководствуясь ценностями и принципами при каждом внедрении. Например, одну неделю они могли бы искать способ увеличить рефлексивность или улучшить свое окружение, сидя вместе, или повысить эффективность работы в парах. Пока вся команда предпринимает шаги в правильном направлении, она знает, что улучшает планирование, отслеживание и выполнение проекта. А это значит, что она с каждым днем становится совершеннее — не только работая с XP, но и создавая программное обеспечение.



Ключевые моменты

XP имеет пять ценностей, которые помогают командам сформировать правильное мышление и избежать результата «лучше-чем-ничего».

Коммуникация как XP-ценность означает, что каждый член команды в курсе дел коллег и постоянно с ними общается.

XP-команды ценят простоту, пишут четкие и ясные решения, избегают сложностей.

Благодаря постоянному тестированию и интеграции XP-команды создают петли обратной связи, помогающие поддерживать высокое качество.

Члены XP-команд имеют мужество принять лучшее решение, которое потребует большего объема работы в краткосрочной перспективе, чтобы в дальнейшем было проще обслуживать код.

Каждый член XP-команды заслуживает уважения коллег, менеджеров и пользователей.



Часто задаваемые вопросы

Так с чего же начать? Как узнать, какие шаги делать в первую очередь?

Многие команды, впервые сталкивающиеся с ХР, задают эти вопросы, но на них нет однозначного ответа. Он зависит от конкретных проблем. На первой странице этой главы приведена цитата, в которой говорится о том, как люди ненавидят перемены. Не стоит это недооценивать.

Принятие ХР означает, что вы меняете способ разработки программного обеспечения. Восторг сотрудников в отношении новой практики улетучится, как только люди поймут, что работу надо выполнять не так, как прежде.

Итак, каким должен быть первый шаг? Если вы рассматриваете ХР в качестве инструмента для решения проблем, то логично предположить, что они у вас есть. Каковы эти проблемы? Какие ХР-практики могут помочь вам их решить?

Если проблемы связаны с ошибками, то попробуйте начать с разработки через тестирование. А может, члены вашей команды не общаются между собой или удивляются изменению требований? Тогда добавьте истории, но пробуйте практики по одной — и не только для галочки. Соберитесь всей командой и *поговорите о ценностях и принципах ХР*. Убедитесь, что все понимают, чего вы пытаетесь добиться при помощи этой практики. Если каждый человек понимает, в чем заключается проблема, которую вы пытаетесь решить, и как она влияет на него самого, то вы сможете оценить, насколько хорошо эта практика будет работать.

Это поможет вам найти решение, но, что более важно, каждый человек в команде поймет, как эта практика улучшает его жизнь. И у них не возникнет чувства, что они тратят свое время впустую (из-за чего команды порой перестают использовать практики).

Меня беспокоит, что я не понимаю, как применить парное программирование к моему проекту. Что оно может дать?

Когда команда пытается внедрить ХР, но в конце концов возвращается к прежним методам работы, практика парного программирования, как канарейка в угольной шахте, погибает первой. Так случается часто, потому что эта практика наиболее явно демонстрирует разницу между мышлением команды и ценностями ХР. Поэтому если вы

не видите смысла в парном программировании, то обдумайте ценности XP заново.

Коммуникация

Считаете ли вы нормой общение с коллегой в ходе программирования? Как вы отнесетесь к тому, что в этот момент с вами заведут разговор? Допускаете ли вы, чтобы ваш код обсуждали без вас?

Простота

Считаете ли вы нормальным, если кто-то говорит, что код, который вы только что написали, очень сложный? Готовы ли вы выделить время, чтобы исправить его немедленно, или оставите комментарий «сделать» либо «решить позднее»? А если поджимают сроки, вы все равно готовы заняться упрощением кода?

Обратная связь

Хорошо ли вы себя чувствуете, если кто-то говорит вам — возможно, проявляя бестактность, что характерно для многих программистов, — что ваш код не очень хорош?

Мужество

Решаетесь ли вы, просмотрев чей-то код, указать на ошибки, даже если это спорный вопрос? Спокойно ли вы воспринимаете свои ошибки в оценке чужого кода? Нормально ли вы воспринимаете ситуацию, когда допускаете ошибки на глазах коллег, которые указывают вам на них? А если этот человек пользуется вашим уважением или выше вас по должности?

Уважение

Когда кто-то говорит, что ваш код не слишком хорош, готовы ли вы прислушаться к этому мнению и пересмотреть свою работу, чтобы понять, кто прав? Не затаите ли вы обиду?

Если в вашей команде не все гладко с реализацией этих принципов, то их анализ поможет вам понять, что необходимо делать, прежде чем внедрять XP. Какой из приведенных выше вопросов беспокоит всех больше всего? Обсудите это, чтобы выяснить, что создает дискомфорт в команде. Подумайте, что нужно сделать, чтобы избавиться от него.

Существует наихудший сценарий, когда образ мыслей не позволяет команде использовать парное программирование, но дает внедрить какую-то иную практику XP. Однако недостаточно просто внедрить практику, кажущуюся бессмысленной. Нужно убедиться, что вся ваша

команда обсуждает XP-ценности и серьезно относится к принципам. Есть надежда, что спустя некоторое время вы вернетесь к парному программированию и обнаружите, что ваше мышление изменилось благодаря маленьким шагам, сделанным в применении других практик.

Почему программисты должны писать все эти тесты? В моей команде нет лишних сотрудников, чтобы сидеть и отлавливать ошибки. Разве не дешевле нанять тестеров?

Разработка через тестирование — это не просто добавление тестов и проверка того, что они выполняются. Чувствуете ли вы дискомфорт от идеи разработки через тестирование? Если да, то это показатель того, что вам нужно пересмотреть свои взгляды на ценности и принципы. Вероятно, вы обнаружите, что не поняли один из них. Может быть, у вас проблема с принципом качества, который означает, что каждый член команды отвечает за качество проекта. Вы не можете просто «перекинуть» код на этап тестирования и назначить ответственного за поиск ошибок. Возможно, вас тревожит избыточность — кажется, что эти тесты добавляют дополнительный, потенциально дублирующий код. Потратьте немного времени на переосмысление принципов и ценностей и попытайтесь сформулировать, что именно вам не вполне понятно.

Действительно, написание модульных тестов требует времени, и очень часто разработчики считают, что это напрасный труд*. Но самое забавное происходит, когда программисты все-таки начинают заниматься разработкой через тестирование. В какой-то момент модульный тест дает сбой, тем самым выявляя маленькую, но неприятную ошибку, которая потом могла бы отнять массу времени. Несколько подобных случаев переубедят любого скептика. Когда это происходит, многие agile-разработчики говорят, что вы «заражаетесь тестами», и разработка через тестирование становится естественным способом создания программного обеспечения.

Необходимо помнить еще об одном: в разработке через тестирование гораздо больше смысла, чем кажется на первый взгляд. В главе 7 вы узнаете, что написание модульных тестов до создания кода может иметь огромное влияние на то, как команда разрабатывает программное обеспечение. Те, кто эффективно использует разработку через тестирование, создают более качественное ПО не только потому, что отлавливают ошибки, но и благодаря тому, что начинают думать иначе.

* Эти разработчики говорят так: «У меня нет времени, чтобы искать ошибки в нашем коде, — я слишком занят тем, что сам их туда вношу!»

Все еще сомневаетесь? Это нормально. Вам просто надо найти другой подход к ХР. К счастью, есть много иных методов, с которых можно начать. Как и в случае с парным программированием, если вам не удастся убедить себя и команду, что разработка через тестирование — это верный путь, то попробуйте другой метод.

Я программист, и все это кажется мне чем-то неопределенным. Обычно я получаю задание, поэтому представляю, чем должен заниматься. Откуда же я узнаю, что мне делать дальше?

Если вы привыкли работать по плану-графику или диаграмме Ганта, то ХР-подход к ежедневному планированию может показаться вам необычным. Ежедневно вы получаете список задач, каждую из которых нужно выполнить и после этого отметить галочкой. Так работать удобно, потому что вы всегда видите свой прогресс. Руководителю и менеджеру проекта это также подходит — они легко могут видеть состояние проекта по проставленным галочкам.

Разработчику, привыкшему к такому режиму, поначалу трудно приспособиться к зрелой ХР-команде. Потому что в командно-административной среде кто-то уже проанализировал исходные требования и составил на их основе список задач, которые команда должна выполнить. Это задает строгий порядок действий каждого программиста, поскольку все решения были приняты в начале проекта.

ХР-команды не занимаются предварительным планированием. Они обсуждают темы и истории при планировании квартального цикла, но работают по очень коротким, недельным итерациям. Индивидуальные задания планируются только на ближайшую неделю. В этом и заключается «экстремальность» экстремального программирования — в принятии решения в последний ответственный момент. И разработчику, привыкшему, чтобы все было запланировано заранее, кажется, что решения принимаются слишком поздно.

Есть еще одна ситуация, когда ХР-команда может вызвать ощущение «дезорганизации» у тех, кто раньше использовал гораздо более жесткое планирование: пары постоянно меняются и люди не определяют заранее, кто какое задание должен делать. Решение о том, кому выполнять ту или иную работу, принимается непосредственно перед ее началом. Это отличается от методов scrum-команд, у которых есть ежедневные совещания, оценивающие ход работ по спринту, и назначение задач происходит способом самоорганизации. Поскольку ХР-команда имеет короткие итерации и петли обратной связи, она может «свалить» свои задачи в кучу, и, когда пара готова к следующему заданию, она просто вытягивает его оттуда.

Разве можно ждать хорошей работы от команды, если каждая пара наобум выбирает задание? Может быть, лучше планировать работу, учитывая навыки каждого программиста?

Нет. И это еще одна область, в которой XP помогает команде совершенствоваться. В начале недельного цикла команда выбирает истории, которые собирается делать, и разбивает их на задачи. Поскольку разработка ведется итеративно, в конце цикла они поставляют работающее программное обеспечение, которое **действительно сделано**, а все частично завершенные истории переносятся в следующий цикл. Но они не планируют, кто какую задачу будет делать. Вместо этого задачи «сваливаются» в кучу, ставятся в очередь или организуются иным образом. После того как пара заканчивает текущее задание, она выбирает следующее из очереди.

Значит ли это, что существует правило, будто люди должны выбирать следующее задание случайным образом, даже если в команде есть тот, кто справится с ним лучше? Конечно, нет. Члены команды не роботы. Они будут делать то, на что способны, и принимать самые правильные решения в процессе работы. Возможно, стоит подбирать пары таким образом, чтобы один участник имел глубокие знания в определенной области, а другой стремился в ней усовершенствоваться. Тогда в следующий раз, когда появится задание, требующее аналогичного навыка, у команды будет из кого выбирать, потому что нужные знания есть уже у двух человек.

Существует веская причина, по которой XP не имеет закрепленных ролей. Вот что Кент Бек говорит об этом в своей книге *Extreme Programming Explained: Embrace Change*.

После того как между членами команды устанавливаются новые, уважительные отношения, закрепленные роли мешают им действовать наилучшим образом. Программисты могут писать историю, если сейчас это полезнее. Менеджеры проектов могут предложить архитектурные улучшения, если они видят, как их сделать.

Это отражает два важных XP-принципа.

Принятие ответственности

После того как пара берется за задачу, она обязана завершить ее. Если она столкнется с проблемами, то сделает все возможное, чтобы справиться с ними. Но эти программисты попросят помощи у команды,

даже если это может задеть их самолюбие. (Хотелось бы надеяться, что, поскольку они работают вдвоем, кто-то услышит их дискуссию и сам предложит помощь.)

Возможность

Каждая новая задача — это возможность для человека научиться еще чему-то, выполняя при этом свою работу. Если есть технология, которую кто-то не знает, то ее изучение становится частью задачи. Это помогает распространять знания внутри команды и открывает больше возможностей на будущее.

Есть еще одна идея, применимая в такой ситуации: коллективное владение. Помимо 13 основных XP-практик существует также 11 вытекающих из них практик-следствий.

Истинная вовлеченность клиентов

Реальное вовлечение клиентов в квартальные и еженедельные сессии планирования и учет их мнений.

Инкрементальное развертывание

Развертывание небольших частей системы по отдельности, а не «одним ударом» (с верой в то, что внедрение можно выполнить таким путем).

Целостность команды

Объединение эффективных команд.

Сокращение команд

Когда команда улучшается, она может выполнять работу быстрее. Но вместо того чтобы увеличивать объем еженедельной работы, исключите из команды одного человека (и используйте его для переноса XP-культуры в другую команду).

Анализ первопричин

Когда что-то идет не так, выясните, в чем дело и что привело к этому, после чего устраните проблему так, чтобы она больше не возникала.

Общий код

Каждый человек в команде чувствует себя комфортно, работая с любой частью кода, и все коллективно владеют кодом.

Код и тесты

Команда поддерживает только код и тесты, документация генерируется автоматически из исходного кода, и история сохраняет

жизнеспособность благодаря передаче из уст в уста (потому что люди редко открывают пыльные папки со старыми планами).

Единая база кода

Не поддерживайте несколько версий кода.

Ежедневное развертывание

Развертывайте новые версии в производственную среду каждый день.

Согласованный объем контракта

Как принято в консалтинговых компаниях, вместо того чтобы фиксировать объем и время переговоров (часто жертвуя качеством, чтобы уложиться в срок), фиксируйте время и регулярно договаривайтесь об объемах.

Плата за использование

Это еще одна практика, заимствованная из консалтинга, — вместо того чтобы выставить счет за разработку, взимайте плату с клиента за использование системы. Это позволяет получать в режиме реального времени постоянную обратную связь и знать, какие функции востребованы, а какие нет.

В этой книге больше не будут упоминаться практики-следствия, но одна из них — общий код — имеет особое значение. В XP-команде любой ее член — даже менеджер проекта — имеет право вносить изменения в код. Потому что он принадлежит всем. Это означает, что если обнаружится ошибка, то исправление может внести любой член команды, а не только тот, кто ее допустил. Такой подход отличается от работы традиционных команд, где каждый ответственен за собственный кусок кода.

(Существует еще одна практика-следствие, о которой мы будем много говорить, — анализ первопричин и метод пяти «почему», который поможет уловить за внешними проявлениями суть проблемы. Мы вернемся к этому в главе 8.)

Разрушение барьеров владения кодом помогает объединить команду, потому что при возникновении проблемы все чувствуют ответственность за ее решение. Поэтому каждый член команды отвечает за весь код и соответственно способен решить любую задачу в очереди или, по крайней мере, научиться ее решать. Команда понимает, что процесс обучения — это часть проекта и на него стоит тратить свое время.

Практик-следствий действительно 11? Почему их так много?

Если количество практик в XP кажется вам огромным, то это признак того, что вы формально относитесь к этой методологии. Ранее мы

видели, что такое мышление может привести вас только к результату «лучше-чем-ничего». Практики-следствия существуют потому, что помогают решать проблемы, с которыми сталкиваются команды во время создания лучшего программного обеспечения. Если вы беспокоитесь о том, как сможете выполнять все эти практики одновременно, значит, вы склонны к формальному подходу к ХР. Помните ХР-принцип маленьких шагов: если вы тратите время, чтобы понять, какие практики нужны вам и вашей команде, то будет проще включить их в свою деятельность и постепенно улучшать совместную работу.

В главе 7 вы узнаете о том, как ХР-практики работают вместе и оказывают значительное влияние на то, как команда проектирует и создает код. Пока вы будете ее читать, постарайтесь найти варианты совместной работы практик и понять, как они при этом взаимно усиливаются. Это позволит вам выйти за рамки формального мышления и воспринять ХР как целостную систему.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой).

- Попробуйте парное программирование, даже если на первый взгляд оно кажется вам немного странным. Нет нужды брать на себя обязательство делать это всегда, просто позанимайтесь этим несколько часов и оцените впечатления. Возможно, вы поймете, что такая работа намного удобнее, чем кажется!
- Если вы разработчик, попробуйте непрерывную интеграцию. Вам даже не нужно привлекать к этому остальную команду (пока). В следующий раз, когда вы достигнете ключевой точки, в чем бы она ни заключалась, возьмите последний код из хранилища контроля версий, поместите его в вашу песочницу и проведите тестирование, чтобы убедиться: код интегрирован. Сделайте это снова через несколько часов. Насколько часто вы сталкиваетесь с проблемой интеграции, которую легко решить сейчас, но гораздо сложнее позднее?
- Попробуйте разработку через тестирование. В следующий раз, когда вы создаете новый класс (модуль или подпрограмму — все, что считается единицей в языке, который вы используете), сначала напишите

модульный тест. Не беспокойтесь, если он не будет охватывать все возможные случаи. Создайте простой работающий тест. Возможно, вам придется потратить немного времени на изучение того, как модульное тестирование работает с языком, — это также полезное упражнение.



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше о методах, ценностях и принципах XP в книге Кента Бека и Синтии Андрес *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 2004).
- Узнайте больше о модульном тестировании и разработке через тестирование в книге Энди Ханта и Дэйва Томаса *Pragmatic Unit Testing* (Pragmatic Bookshelf: Java version 2003, C# version 2007).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Это один из самых сложных аспектов agile-коучинга, особенно для agile-тренеров, которые не имеют опыта программирования. Сосредоточьтесь на XP-ценностях и принципах и помогите команде выяснить, как они влияют на код.
- Команде легко потерять мужество и уважение. Помогите найти примеры, когда они колебались, сообщать ли кому-то вне команды реальное состояние качества кода. Например, подправляла ли команда когда-нибудь демоверсию, чтобы обойти известные ей ошибки. Меняла ли она отчет об ошибке, чтобы та казалась менее серьезной, чем на самом деле? Помогите ей найти способы, чтобы было легче делиться такой информацией.
- Как найти лучшие возможности для коммуникации команды? Помогите ей создать информационное табло.

XP, простота и инкрементальная архитектура

*Я не великий программист, я просто хороший программист
с замечательными привычками.*

Кент Бек, создатель XP

Цели XP выходят за рамки простого стремления научить команду работать лучше. Масштабная цель XP-практик, ценностей и принципов — помочь командам **создавать программное обеспечение, которое можно легко изменять и расширять**, а также работать, планировать и расти вместе в среде, легко принимающей изменения.

Внедрение XP означает больше, чем просто использование парного программирования для постоянной проверки кода или разработки через тестирование для увеличения тестового покрытия. Повышение качества и улучшение отношений с коллегами — это замечательные побочные эффекты XP, и они помогают достичь основной цели. Эти практики изменяют технологию создания ПО, *но не вносят кардинальных изменений в его проектирование.*

Повторим еще раз: важная цель XP — это программное обеспечение, которое может быть легко изменено. Команды примут изменения охотнее, если создаваемый ими программный продукт несложно изменять. Это оказывает глубокое влияние на подход хорошей XP-команды к разработке и проектированию программ.

Здесь также следует отметить один из ключевых моментов приобретения правильного XP-мышления: искреннюю веру в то, что практики, изученные в главе 6, такие как разработка через тестирование, парное программирование и временной запас, помогают вам и вашей команде изменить

подход к разработке ПО. Игнорирование этих практик может привести к созданию некачественного, трудно изменяемого исходного кода.

В этой главе вы узнаете, как талантливые программисты иногда все же создают код с серьезными проблемами в нем самом и архитектуре. Вы изучите три оставшихся основных XP-практики и то, как они помогают избегать упомянутых проблем. Вы узнаете о многих замечательных привычках, которые появляются у членов XP-команд (подобных той, о которой говорится в цитате Кента Бека в начале главы). Мы также расскажем, как все XP-практики формируют экосистему, в которой создается лучший, легкий в обслуживании, гибкий и изменяемый код.



Описание: команда занимается разработкой фантазийного баскетбольного сайта

Джастин — первый разработчик

Даниэль — второй разработчик

Бриджит — их менеджер проекта

Акт IV. Работа в сверхурочное время, часть 2: снова сверхурочные

Джастин обычно старался уходить из офиса с таким расчетом, чтобы успеть к поезду, отходящему в 17:42. Почему-то это случалось редко. Сегодняшний день не стал исключением. Поздним уходам всегда предшествовала одна и та же ситуация: примерно за час до окончания рабочего дня он приступал к исправлению небольшой ошибки или внесению незначительной настройки. Но маленькие изменения непонятным образом превращались в большую работу.

Это всегда происходило по одной и той же схеме. Внеся запланированное изменение, Джастин обнаруживал, что необходимо подправить и другую часть кода. Он шел дальше и вносил второе изменение. Но это вмешательство влекло за собой необходимость корректировать третью и четвертую части кода. Одно из этих изменений могло, в свою очередь, потребовать, чтобы он перешел к еще одной части исходного кода и т. д. Иногда, к тому времени, когда он заканчивал вносить цепочку исправлений, он уже забывал, зачем делалась первая правка.

Однажды изменения оказались такими кардинальными, что он просто не мог их внести. Он поговорил с Даниэль и Бриджит, и они решили, что требуемые

изменения были настолько рискованными, что могли дестабилизировать код. Ситуация оказалась деморализующей: была потрачена масса времени на внесение изменений, а потом все пришлось возвращать в исходное состояние.

Сегодняшний вечер ничем не отличался от предыдущих и развивался по обычной схеме: изменения вызывали изменения. Джастин вспомнил, как говорил Даниэль: «Я обещал своей девушке, что буду дома рано, поэтому просто собираюсь сделать последний кусок задания на сегодня». Надо было всего-то добавить опцию в раскрывающемся списке на странице «Настройки». Это было просто, точнее, ему так казалось пять часов назад.

Если бы он знал, что дело настолько затянется, то поработал бы вечером из дома. Или вообще оставил бы это на потом.

Джастин пытался исправить раскрывающееся меню, в котором хранятся параметры конфигурации созданных им команд. Все, что ему нужно было сделать, — это добавить параметр в раскрывающемся меню, чтобы пользователь мог скрыть рейтинг для игрока. Он добавил этот параметр, но всякий раз, когда новый пункт был уже выбран, на странице возникало сообщение: «Вы должны выбрать пункт из списка». И то, что казалось простой манипуляцией, отняло у него часы.

Когда он занимался отладкой кода страницы валидатора, то увидел, что тот читает параметры из кэшированной копии таблицы базы данных. Поэтому он добавил вызов обновления кэша, когда пользователь нажимает кнопку ОК, и теперь этот пункт проверялся, что потребовало от него перейти в другое место кода и внести еще три изменения. Кроме того, еще три страницы обращались к той же странице настроек. Поэтому их тоже понадобилось исправить.

В результате Джастину пришлось сделать более десятка различных изменений в коде, но в итоге он заработал, хотя и благодаря отвратительному читерскому клуджу*. Он попросил совета у Даниэль, и она придумала обходной путь решения неприятной проблемы. «Это обновление кэша использует только ID-номер пользователя. Попробуй создать фейк-класс пользователя, который имеет только его идентификатор, а возвращает ноль для всего остального». Это было некрасиво, но сработало.

Наконец Джастин добавил пункт в раскрывающийся список. И ощутил давно знакомое чувство, вызванное пребыванием в офисе допоздна. Он сказал Даниэль:

* Клудж — это инженерный сленг, обозначающий быстрое и «грязное» решение, которое выполняет свою задачу, но выглядит безобразно и его трудно поддерживать. В этом слове гораздо больше тонкостей, чем мы можем перечислить. Если вы незнакомы с этим термином, то займитесь его изучением прямо сейчас.

— Наконец-то дело сделано. Ухожу домой. Я уже весь издерган изменениями, которые пришлось внести ради этой дурацкой настройки. Думаю, я теперь понимаю, что чувствует пинбольный шарик во время игры.

Она ответила:

— Я знаю, что ты имеешь в виду. Казалось бы, нужно просто поменять кое-что на странице авторизации, но это привело к необходимости править код в трех других местах. Два изменения потребовали дополнительных изменений в других классах, а один из этих классов вызывает сервис, выдающий непонятный ответ. И я не знаю, что с этим делать.

Джастин спросил:

— Почему всегда так происходит?

Даниэль ответила:

— Такова сущность программирования. Фактически вопрос заключается в том, действительно ли ты уверен, что только что написанный код работает?

Ей не хотелось задавать этот вопрос. Несколько секунд они пристально смотрели друг на друга и чувствовали себя неуютно.

— Я лучше потрачу еще несколько минут на тестирование этого, — сказал Джастин. Он надел наушники и начал печатать для своей подруги еще одно послание с извинением.

Код и архитектура

В ходе работ над проектом Chrysler Comprehensive Compensation (C3) Кенту Беку было необходимо изменить корпоративную культуру команды, переключив ее с создания «умного» кода на простые решения, а это весьма сложная задача. Один из методов, который он использовал, — это церемония давления со стороны коллег. Например, группа торжественно шествовала к человеку, придумавшему самое заумное решение, ему надевали на голову шапочку с пропеллером, а затем раскручивали его и комментировали эту заумь. Негативное внимание со стороны команды заставило людей отойти от усложненных решений и перейти к простой архитектуре и простым решениям. Но все люди разные, и не все в команде с готовностью приняли ХР. Одному человеку не понравился новый стиль работы и тесное сотрудничество, поэтому он покинул проект.*

Алистер Коберн. Быстрая разработка программного обеспечения

ХР-команды создают код, который легко изменить. Но как они это делают? Ведь никто не собирается создавать код, который изменить трудно.

* Единая система расчета зарплат автоконцерна Chrysler. Прим. перев.

Многие разработчики — даже очень умные! — стремятся делать наоборот. Они будут настраивать свои команды на создание кода, пригодного для повторного использования, и потратят много времени, пытаясь спроектировать компоненты, которые можно применять многократно. Но планировать повторное использование очень трудно, и совсем непросто довести до конца разработку чрезмерно абстрактного и обобщенного кода. Ведь объем работы, который придется выполнить для создания общей структуры, ровно такой же, какой необходим для выполнения проекта под ключ. Архитектура для многократно используемой библиотеки, созданная сегодня, вскоре превращается в препятствие, которое команда должна обойти или боится трогать.

Многие люди считают, что хрупкий, трудно изменяемый код — это ошибка, типичная для новичков. Но нередко даже лучшие разработчики создают код, который непросто модифицировать. (Неопытные программисты редко создают достаточно большой исходный код, чтобы в нем проявились такие проблемы.) И необязательно написанное ими программное обеспечение будет плохим или неудачно спроектированным. В большинстве случаев это сводится к вопросу, какой код написан: «заумный» или простой.

Для разработчиков (особенно для умных) вполне естественно пытаться решить не только сегодняшние, но и завтрашние проблемы. Большинство из нас присутствовали на сессиях по планированию, которые казались нескончаемыми, потому что каждый думал лишь о своих проблемах и о том, как их решить. И чем крупнее проблема, которая возникла перед командой, тем больше усилий она требует.

Scrum-команды избегают трудностей, связанных с бесконечным планированием, разбивая проект на спринты. Они сосредоточены на сегодняшних задачах и оставляют все остальное на потом, принимая решение о планировании в последний ответственный момент. И когда XP-команды занимаются итерационным планированием, используя квартальные и недельные циклы, они действуют таким же образом.

Принятие решений в последний ответственный момент — это инструмент не только планирования. Он ценен также при проектировании и кодировании. И это главное средство, при помощи которого XP-команды реализуют принцип **простоты**.

Если вы работали программистом в нескольких командах, то наверняка видели код, который трудно изменить. Создавался он вовсе не с этой целью, а для того, чтобы выполнять что-то полезное. А потом его стали приспособливать к другим задачам. И после нескольких циклов перепрофилирования код стал сложным. Так как же это произошло?

КОД «С ДУШКОМ» И АНТИПАТТЕРНЫ, ИЛИ КАК ПОНЯТЬ, ЧТО ТЫ ПЕРЕБОРЩИЛ СО СЛОЖНОСТЬЮ

Простота начинается с понимания того, как не делать слишком много.

Вот пример, как сложность (отсутствие простоты) может повлиять на продукт. Когда один из авторов этой книги учился в колледже, он получил в подарок устройство, сочетавшее в себе функции блендера, миксера и кухонного комбайна. Не все эти функции работали хорошо: он был неплох как блендер, средненький как миксер и совершенно непригодный как кухонный комбайн. Кроме того, в нем проявились проблемы, которые не хочется видеть в подобных приборах. Например, его сменные части было неудобно хранить и практически невозможно отмыть после использования.

Тем не менее он пользовался им десять лет. Но поскольку весь этот сложный набор приспособлений работал не слишком хорошо, приходилось избегать рецептов, для которых требовался кухонный комбайн. Когда же наконец он засунул агрегат на дальнюю полку (потому что рука не поднималась выкинуть «весьма неплохой» электроприбор) и купил самый дешевый кухонный комбайн, он обнаружил, что теперь может готовить по рецептам, которые раньше были недоступны. Оставалось только удивляться, чего он ждал целых десять лет.

Этот прибор был более чем бесполезен. Вместо помощи в приготовлении пищи по разнообразным рецептам, для чего он и был нужен, комбайн только мешал этому. Но его наличие препятствовало покупке нового устройства, поэтому в течение многих лет хозяин избегал рецептов, связанных с использованием кухонного комбайна. И не потому, что у него не было возможности приготовить эти блюда. Просто с прибором было связано слишком много дополнительных проблем. Но долгие годы ему не приходило в голову купить замену, потому что старый был «весьма неплох».

Причина того, что этот прибор был неудобным, заключалась в его **сложности**. Чтобы соединить в одном приборе три, нужны технические компромиссы, которые привели к тому, что его было сложно использовать (например, было сложно использовать миксерную насадку, потому что она не хотела работать, если не была идеально закреплена). Замена сложного прибора на три **простых** — дешевые блендер, миксер и кухонный комбайн — экономит деньги, они займут меньше места и позволят готовить больше блюд.

Проблемы, подобные этой, объясняют, почему простота — это одна из ценностей XP, о чем говорилось в главе 6. XP-команды признают, что такая проблема, как сложность, может повлиять на планирование проектов.

В этой книге мы уже упоминали, как scrum-команды сталкивались со схожими трудностями.

Например, несложно убедить сотрудников, что для команды, стремящейся быть продуктивной и выполнить проект, не имеет смысла держать людей, сидящих сложа руки. Однако некоторые менеджеры, исповедующие командно-административный стиль руководства, доводят эту идею до крайности: они приносят план проекта, где все 100% ресурсов уже распределены*, и требуют от разработчиков отчета за каждую минуту работы. Постоянный контроль за 100%-ной загрузкой персонала запланированной работой перегружает управление проектом. Люди должны следить за использованием своего времени и вносить данные в систему контроля (что несложно сделать в теории, но на практике отнимает уйму времени и сил). Менеджерам необходимо составлять, обновлять и пересматривать планы, и невозможно избежать множества совещаний по «решению» проблем, потому что только кажется, что разработчик занимается делом 95% своего времени.

Получается сложная система управления проектом, но дополнительное усложнение *не добавляет ценности в проект*. Теперь команда тратит около 20% своего времени на отслеживание и разговоры о том, на что уходят остальные 80%, а это увеличивает длительность проекта. Похоже, команда тратит все больше времени на планерки, обсуждая такие вопросы, как «На сколько процентов завершено дело?» и «Можете ли вы оценить, сколько минут потратите на эту задачу?».

XP-команды, так же как scrum-команды, используют итерации для принятия решений по планированию проекта в последний ответственный момент. Такой способ планирования проектов гораздо проще. XP-команды, как правило, не следят постоянно за временем каждого сотрудника, как показано в нашем примере, потому что мониторинг не помогает выполнять проект. Несмотря на благие намерения, собранная информация редко используется для прогнозирования проекта и почти никогда не пересматривается после его завершения. Как уже было сказано, ситуацию ухудшает то, что проекты со временем меняются. Планирование 100%-ного распределения ресурсов потребует от команды принятия всех решений в начале проекта. Если какое-то из них оказывается неверным (а это обязательно случится, таков нормальный ход проекта), то вся команда должна будет пересмотреть планы, задачи и варианты распределения.

В этом примере есть более значимый урок, помогающий понять XP. Он связан с признанием паттернов и их применением для улучшения работы.

* Менеджеры с командно-административными взглядами имеют привычку называть членов группы «ресурсом».

Если вам доводилось работать в команде, где строгий руководитель командно-административного типа требует от каждого 100%-ного распределения и собирает бесконечные совещания, чтобы в этом убедиться, требует от каждого постоянно обновлять свой список задач в соответствии с этим надуманным распределением, то, прочитав несколько последних абзацев, вы, возможно, вспомнили эту атмосферу нескончаемых совещаний. Наверняка вы сразу распознали **антипаттерн** — модель поведения, которую команда демонстрирует в качестве причины, вызывающей проблемы проекта. Оpozнание антипаттерна в управлении проектом дает возможность обозначить проблему и надежду на отыскание решения, которое поможет упростить эту работу, что позволит команде вернуться к созданию продукта.

ХР-КОМАНДЫ ИЩУТ КОД «С ДУШКОМ» И ИСПРАВЛЯЮТ ЕГО

Код тоже может содержать антипаттерны, и его распознавание — это первый шаг к упрощению вашего кода. Когда антипаттерн относится к структуре кода или архитектуре решения, это называется **код «с душком»**. ХР-команды всегда ищут такой код и вырабатывают привычку исправлять его сразу же, как только найдут, не позволяя распространяться на другие модули.

Мы собираемся потратить некоторое время на обсуждение наиболее распространенных вариантов кода «с душком», чтобы вы могли лучше понять образ мышления успешного ХР-разработчика. Этот конкретный список составлен на основании опыта многих разработчиков, обсуждавших проблемы, всплывавшие в их собственных проектах на протяжении ряда лет. Термины «код «с душком»» и «антипаттерн» впервые стали популярны среди разработчиков во второй половине 1990-х годов во многом благодаря сайту WikiWikiWeb (самый первый из Wiki-проектов, созданный автором Agile-манифеста Уордом Каннингемом). Этот сайт был (а для некоторых остается) одним из самых популярных мест, где разработчики обсуждают проектирование ПО. Чем больше людей собиралось, чтобы поговорить на эту тему, тем чаще они обнаруживали, что существует ряд общих симптомов, с которыми сталкивался каждый из них*.

Многие разработчики программного обеспечения (и мы в том числе), впервые сталкиваясь с такими проблемами, реагируют почти на интуитивном уровне.

* Мы настоятельно рекомендуем читателям найти информацию о разновидностях кода «с душком» и антипаттернах. На наш взгляд, один из лучших ее источников — оригинальный сайт Wiki, созданный Уордом Каннингемом.

Это весьма странное ощущение, когда вы мучаетесь непонятным разочарованием и обнаруживаете, что его в точности описывает неизвестный вам человек. И в ходе дальнейшего обсуждения важно иметь в виду, что эти проблемы *созданы*, признаны и могут быть предотвращены или исправлены людьми. Даже если симптомы носят технический характер, причина в человеческом факторе, поэтому решение должно быть как техническим, так и командно ориентированным.

Когда мы проводим тренинги для разработчиков и говорим о коде «с душком», большинство людей соглашается, что это напоминает **«стрельбу дробью»**. Таким термином описывается ситуация, когда вы хотите сделать небольшое изменение в одной части кода, но обнаруживаете, что оно затрагивает несколько других частей, казалось бы, не связанных с первой. Вы пробуете внести эти изменения и видите, что одно из них требует дополнительной правки в местах, связь которых с первой правкой неочевидна. Если исходный код особенно «плохо пахнет», то нередко программист пытается изменить то, что *должно* быть простым. Он делает десяток или более переходов в коде и в конце концов сдаётся, потому что уже не в силах отследить нарастающие каскадом изменения.

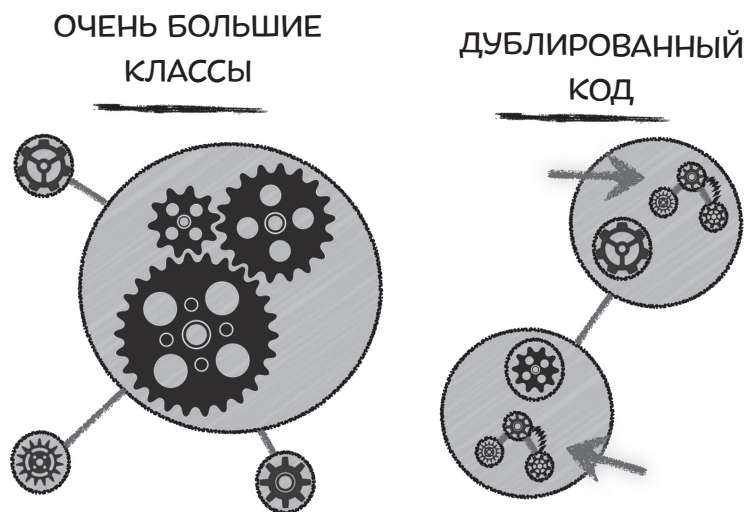


Рис. 7.1. Успешные XP-разработчики имеют привычку искать и исправлять код «с душком», например **очень большие классы** или **дублированный код**. Этим антипаттернам следует придавать структуру отдельных модулей. В этой главе мы будем использовать шестеренки в качестве визуальной метафоры для кода на рисунках

Когда разработчик «закапывается» в код, являющийся причиной «стрельбы дробью», он часто натывается на код, издающий другой «запах», — **сырой код** (незаконченные объекты, что не редкость в объектно ориентированном программировании). Сырой код появляется, когда программист, чтобы использовать один объект (модуль, блок и т. д.), также должен инициализировать другие объекты в заранее определенном порядке. Например, после инициализации некоторой библиотеки разработчик должен установить значения определенных переменных по умолчанию, а также инициализировать другую библиотеку, от которой зависит первая. Об этом можно узнать только из документации или примеров кода, но неправильная инициализация приведет к сбою или непредсказуемому поведению.

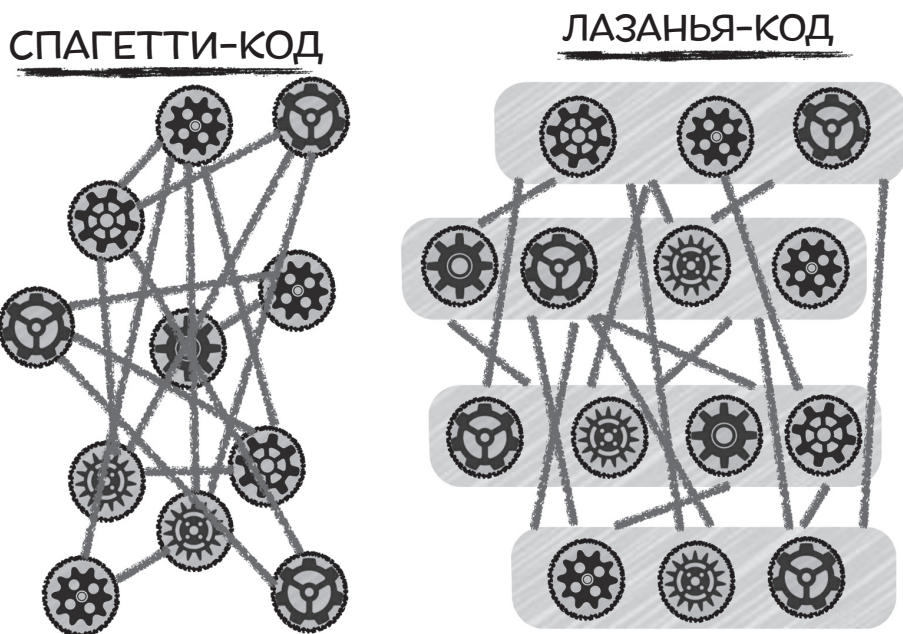


Рис. 7.2. Некоторые примеры кода «с душком» возникают как следствие громоздкой архитектуры системы и сложного взаимодействия модулей

Некоторые примеры кода «с душком» появляются тем же способом, что и нормальный код. Когда ваш код включает **очень большие классы** (или, в случае не объектно ориентированного кода, — громоздкие процедуры, функции, модули и т. д.), его трудно читать и поддерживать. Но часто это признак того, что код пытается выполнять слишком много задач и может быть разделен на более мелкие модули, которые легче понять. В то же время **дублируемый код** идентичен (или почти идентичен) блоку программы,

который встречается в нескольких местах. Это весьма вероятный источник ошибок, особенно если программист вносит изменения в три копии, но забыл про четвертую.

Другой пример кода «с душком» — это неправильная организация взаимодействия отдельных модулей друг с другом. **Спагетти-код**, или код со сложной и запутанной структурой, — один из самых старых типов кода «с душком», он известен по крайней мере с 1960-х годов. Часто спагетти-код можно распознать по пугающим комментариям других разработчиков, которые ранее безрезультатно пытались распутать его. А другая разновидность — **лазанья-код** — может оказаться более коварной проблемой. Современная архитектура программного обеспечения, как правило, делит код на *слои*. Каждый из них имеет определенную цель или выполняет определенную роль, и эти слои составляют общую архитектуру. Однако когда слоев слишком много и шаблоны, по которым их писали, оказались несогласованными, становится трудно понять, что должен делать каждый слой. Эта ситуация может усугубляться **протечками** между слоями, где код, типы данных или идеи, которые должны находиться в одном слое, просачиваются в соседние.

ХУКИ, КРАЙНИЕ СЛУЧАИ И КОД, ДЕЛАЮЩИЙ СЛИШКОМ МНОГО

Приведенные примеры кода «с душком» обусловлены его *структурой*. Но бывают также проблемы, связанные с тем, что *делает код*, то есть с его функциональностью. И успешные XP-разработчики понимают, что они должны заботиться о поведении кода не меньше, чем о его структуре.

Приведем пример. Команда будет прогнозировать способ использования модуля кода (например, класса Java) в будущем и добавит некий хук, или заполнитель, который впоследствии будет заменен кодом. Хук кажется «бесплатным», но на самом деле имеет цену: он связывает команду принятым сегодня решением, которое можно было бы отложить на будущее. Когда наконец приходит время для использования хука, команда глубже понимает проблему и он должен быть другим. Команда приходит в замешательство, обнаружив, что ее предположения о том, как будет использован хук, привели к необходимости создать еще одну часть кода, которая будет обходить этот хук. Поэтому его стало гораздо труднее изменить (несмотря на то, что он пуст!) — ведь он уже используется в другом месте кода. Программист, привыкший добавлять слишком много хуков, забывает, где они расположены. Он будет постоянно сталкиваться с неприятностями при попытке использовать код, написанный несколько недель назад. Ведь окажется, что нужный код так и не был написан, а вместо него стоит комментарий со словами «надо сделать».

Это антипаттерн: добавление такого количества хуков, что становится трудно точно отследить, что именно делает код.

Другой антипаттерн, создающий сложности в понимании кода, — это за-
цикленность на исключениях. **Исключение** — редко встречающаяся ситу-
ация, происходящая при определенном наборе обстоятельств*. Например,
программа, которая загружает данные из файла, должна обработать слу-
чай, когда файл не найден. Ей может также понадобится отреагировать
на ситуацию, когда не удастся найти папку, содержащую файл, вероятно,
реагируя иначе, чем в первом случае. Возможно также, что файл существу-
ет, но не может быть прочитан или его удаляют во время чтения либо дан-
ные в нем имеют неверную кодировку. Хороший программист придумает
еще множество возможных проблем в такой простой ситуации, как чтение
данных из файла. Ну и на чем ему остановиться?

Многие вязнут в обработке слишком большого числа исключений. Но она
действительно необходима, и многие ситуации требуют обработки не-
скольких различных вариантов. Но чем больше обрабатывается вариантов
исключений, тем меньше польза для проекта от добавления еще одного.
Некоторые программисты попадают в замкнутый круг, пытаясь написать
«непрошибаемый» код и тратя на обработку исключений столько же уси-
лий, сколько и на основную часть кода. В этот момент исключения ста-
новятся сходными с избыточным планированием и начинают отвлекать
усилия от основной работы. Код для обработки ошибок срабатывает ред-
ко. Если программа выглядит как запутанный клубок исключений, то ее
труднее понять и изменить.

Хорошие разработчики, как правило, относятся к тому типу людей, которые
зацикливаются на деталях, и исключения — это как раз такая деталь. Имен-
но глубокий ум может превратить программиста в человека, одержимого
исключениями. Когда такое случается, на командных совещаниях начина-
ется бесконечное обсуждение исключений для особых ситуаций, в ходе ко-
торого каждый участник неожиданно приобретает твердое убеждение, что
значимое для него исключение обязательно должно быть обработано.

Если разработчики чрезмерно увлечены планированием исключений или
добавляют слишком много хуков, это можно считать умничаньем. Вместо
создания простого, легко изменяемого кода они размышляют о написании
усложненного, чересчур абстрактного кода или занимаются решением за-
втрашних, а не сегодняшних задач.

* Некоторые люди используют термины *edge case* и *corner case*, взаимно заменяя их. Другие считают, что термин *corner case* используется реже, чем *edge case* (потому что угол — это место, где встречаются два края).

Это подводит нас к так называемой **платформенной ловушке** — антипаттерну, вытекающему из умствования разработчиков. Платформенная ловушка — это ситуация, когда разработчику нужно решить некую проблему или выполнить задание. Но вместо того чтобы написать код для решения именно этой задачи, он создает более обширную платформу, которая может быть использована в дальнейшем для работы в схожих ситуациях.

**«Я ТРАЧУ МНОГО ВРЕМЕНИ НА ЭТУ ЗАДАЧУ.
Я ДОЛЖЕН НАПИСАТЬ ПРОГРАММУ
ДЛЯ ЕЕ АВТОМАТИЗАЦИИ».**

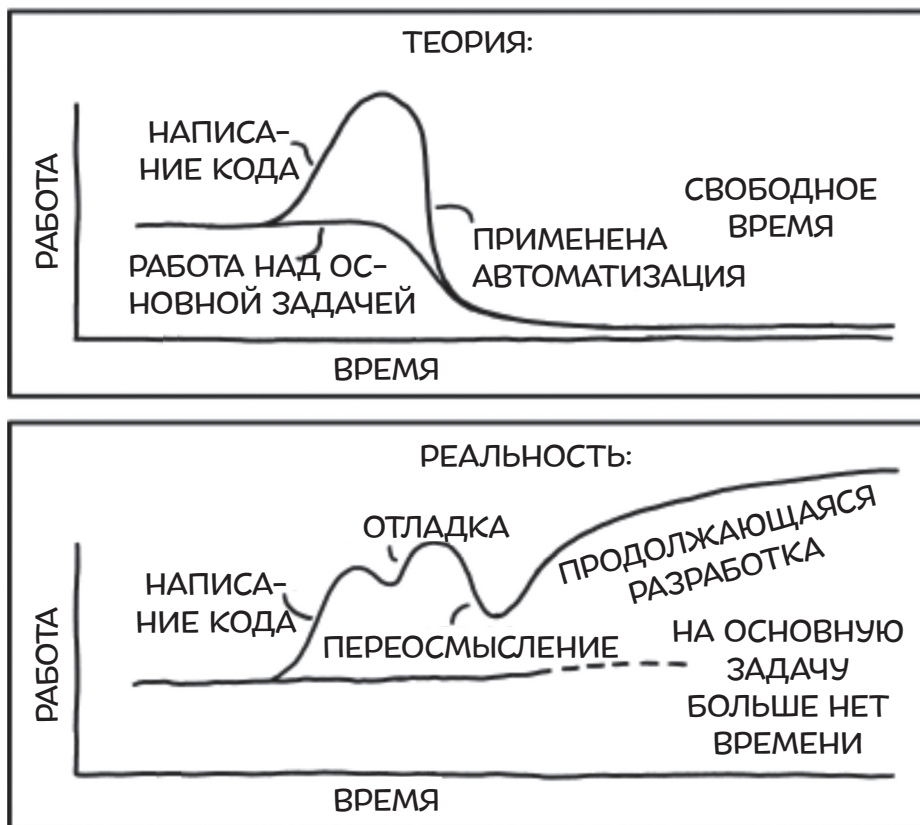


Рис. 7.3. Блестящий веб-комикс *hksd* показывает, что происходит, когда разработчики попадают в платформенную ловушку

Рон Джеффрис, партнер Кента Бека по созданию ХР, описал, как избежать платформенной ловушки. «Реализуйте то, в чем вы действительно

нуждаетесь в данный момент, и не стремитесь предугадывать, что вам может понадобиться в будущем»*. Некоторые XP-команды любят использовать аббревиатуру YAGNI (You Ain't Gonna Need It — «вам это не понадобится»), если речь заходит о таких ситуациях. Пытаясь предугадать будущие потребности и создавая дополнительный код, чтобы их удовлетворить, вы легко попадаете в платформенную ловушку.

Вместо того чтобы сосредоточиться на текущей задаче, чересчур умные разработчики замахиваются на более глобальные цели. Они размышляют примерно так: «Я могу написать это простое решение, но гораздо лучше сделать код, автоматизирующий такую работу, чтобы никому больше не пришлось ею заниматься». У программиста, начавшего писать веб-страницу, в результате получается конструктор для их создания. Попытка решить конкретную проблему производительности приводит к появлению большого кэша общего назначения. Простая программа для загрузки файлов по расписанию непонятным образом приобретает скриптовый движок.

Нетрудно понять, почему разработчики идут по этому пути. Они думают так: «Если бы кто-то другой создал платформу для работы, которую я сейчас выполняю, то я бы просто ее применил. Так почему бы не потратить немного времени на решение проблемы в общем виде?» Это очень благородно со стороны разработчиков и объясняет, почему так много команд попадают в платформенную ловушку.

ПОСТОЙТЕ, А ЧТО НЕ ТАК С ПОВТОРНО ИСПОЛЬЗУЕМЫМИ ПЛАТФОРМАМИ?

Все нормально, и, конечно, в XP нет ничего, что исключает использование или разработку таких платформ. Одна из наших наиболее популярных книг Head First CM учит программистов использовать .NET Framework, технологию Microsoft, для создания Windows-приложений. Существуют отличные готовые платформы для разработки веб-приложений, рендеринга 3D-графики, создания сетевых сервисов и т. д.

Это не значит, что у вашего проекта должна быть установка на создание сетевого сервиса.

Поговорим еще немного о возможности повторного использования, потому что платформенная ловушка — это наше название для антипаттерна, связанного с созданием повторно используемого кода, чего XP-команды учатся избегать. **Многократно используемый код** (код, который носит

* https://en.wikipedia.org/wiki/You_aren't_gonna_need_it, проверено 12 марта 2017 г.

достаточно общий характер и может применяться в разных частях системы) — это цель многих разработчиков. И в этом есть определенный смысл. Кнопки в диалоговых окнах в текстовом процессоре похожи друг на друга и работают примерно одинаково. Было бы странно, если бы команда, создающая текстовый редактор, писала отдельный код для каждой кнопки. Программисты напишут код кнопки один раз и будут повторно использовать его по мере необходимости. Но кнопки в диалоговых окнах текстового редактора работают и выглядят так же, как в браузере и других приложениях. Так что код для этих кнопок (вероятно) находится за пределами любой из этих программ — ваша операционная система предоставляет библиотеку для основных кнопок. Код кнопки был написан однажды и повторно используется тысячи раз. Это явно экономит время.

Программисты редко создают что-либо с нуля. В их распоряжении библиотеки, применяемые для чтения и записи файлов, доступа к сетевым ресурсам, графике и т. д. Это основной факт программирования: хороший разработчик, сталкиваясь с незнакомой проблемой, прежде всего ищет решение в веб-библиотеке. Поэтому, решив проблему, с которой могут столкнуться другие, он часто старается поделиться вариантом кода*. Это продуктивный способ создавать программное обеспечение.

В ЧЕМ РАЗНИЦА МЕЖДУ БИБЛИОТЕКОЙ И ПЛАТФОРМОЙ?

Вы знаете, что можете использовать библиотеку как независимую часть своего проекта.

Разница в том, что в библиотеке хранится код, разделенный на небольшие компоненты, которые можно применять повторно. Если цель заключается в создании простого кода, то каждый компонент должен делать только одну функцию. Компонент, выполняющий несколько функций, следует разделить на меньшие, более независимые модули. Это так называемое **разделение ответственности**; эффективные XP-разработчики признают, что это важный элемент обеспечения простоты кода.

А платформа — это *объединение* множества мелких, повторно используемых частей в одну большую систему. Изучение языка C#, например, помимо его синтаксиса означает также освоение .NET Framework.

Эта платформа включает в себя библиотеки для доступа к файлам, выполнения математических расчетов, подключения к сетям и многого другого

* Мы и сами так поступаем. Если вы зайдете на наш сайт, то увидите несколько библиотек с открытым кодом, которые мы опубликовали, чтобы другим программистам не приходилось заново решать те же проблемы.

(в том числе использования кнопок). Когда вы начинаете создавать свою программу при помощи C# и .NET, вы бесплатно получаете все упомянутые выше модули, потому что они часть платформы .NET. Из нее невозможно выделить нужный кусочек кода. И действительно, не имеет смысла пытаться использовать одну часть платформы .NET в программе, которая написана без ее использования. Как и в большинстве платформ, в ней многие части были объединены в большую единую систему.

Платформы гораздо сложнее библиотек. Команда, пытающаяся создать платформу, когда для данной задачи больше подходит библиотека, вынуждена использовать чрезмерно усложненную архитектуру. Часто эта ошибка встречается из-за того, что программисты видят, как другие платформы экономят им массу времени, и хотят, чтобы их код делал то же самое.

Недостаток такого подхода в том, что, желая получить все эти компоненты даром, вы должны использовать платформу целиком. Иногда это заставляет вас взглянуть на проблему по-новому. Разработчик .NET действительно будет воспринимать проблему иначе, чем Java-разработчик (разница не всегда велика, потому что у этих платформ много общих концепций), и абсолютно по-другому, чем программист на C++, Python, Perl или Lisp. Так происходит, потому что, когда у вас есть набор инструментов для устранения проблем, вы ищете решение, отталкиваясь от этих инструментов. Это только примеры языков и платформ (думая о разработке ПО на Java, Python или Perl, вы наверняка будете учитывать имеющиеся в этих языках стандартные библиотеки), поэтому в них не принимаются во внимание различные ограничения, с которыми вы столкнетесь, решая свои проблемы.



Рис. 7.4. Создание сложных платформ для решения простых задач иногда воспринимается как дело, которым надо заниматься именно сейчас (источник: xkcd)



Рис. 7.5. Платформа, которая сегодня кажется отличным решением, завтра может превратиться в обузу

Приведем типичный пример мышления, ориентированного на платформы. Команды используют скрипты для автоматической сборки программного обеспечения. (В главе 6 мы узнали, что в XP-командах эти скрипты выполняются менее чем за 10 минут.) Для создания скриптов существует

немало инструментов, и многие из них включают в себя скриптовые языки. Но для разработчика с платформенным мышлением, работающего над несколькими проектами, недостаточно просто написать отдельный скрипт для каждого проекта. Похоже, в этих скриптах дублируется слишком много кода (часто так оно и есть), поэтому такой программист напишет единую сборочную платформу для всех проектов. («Если вы используете мою платформу, то вам не придется писать сборочный скрипт. Вы получите его готовым!») Поэтому вместо недлинных скриптов для каждого проекта создается большая платформа, которая требует отдельной поддержки. Этот разработчик пошел на сделку: он предотвратил повтор нескольких десятков строк сборочного скрипта, заменив их на платформу, которая содержит сотни, даже тысячи строк кода. Когда во время сборки что-то пойдет не так (по закону Мерфи это происходит в ночь перед выпуском), вместо устранения небольшой неполадки в сборочном скрипте разработчику придется искать ошибку в большой платформе. Это нельзя назвать *простым* решением.

Объединение значительного числа функций в большой платформе для решения определенной проблемы может действительно дать результат. Но если проблема меняется, то внести изменения в платформу труднее.

Иными словами, объединяя множество разных частей в большие платформы, члены команды думают, что экономят время, но на самом деле они лишь ограничивают способность реагировать на новые запросы. Мы называем это явление *платформенной ловушкой* и можем использовать как пример для изучения простоты и ее влияния на проект. Причина, по которой нам нравится приводить в качестве примера такую ловушку, заключается в том, что она отражает важный аспект мышления, который не всегда совместим с XP: некоторые разработчики предпочитают объединять все функции в один большой блок, а не разделять их на множество мелких модулей.

КОД «С ДУШКОМ» УВЕЛИЧИВАЕТ СЛОЖНОСТЬ

Мы только что описали несколько разновидностей кодов «с душком», которых XP-команды обычно стараются избегать. Все эти коды объединяет одна черта: они *увеличивают сложность проектов*. Верно и обратное: если вы обнаружили, что вам необходимо запоминать, как инициализировать объект, или трудно удерживать в голове каскад вынужденных изменений, или вы подолгу пытаетесь разобраться в спагетти-коде, значит, исходный код вашего проекта излишне усложнен. Именно поэтому XP-команды стремятся как можно раньше найти и исправить код «с душком».

Речь идет не только о коде «с душком», который, возможно, не такой уж частый гость в ваших проектах. Тем не менее мы выбрали этот пример,

потому что за долгие годы общения с разработчиками обнаружили, что почти все сталкиваются с похожими проблемами в своем коде. Они возникают неслучайно и не из-за низкой квалификации разработчиков или их лени. Это является результатом плохих привычек программистов, особенно когда они попадают в ситуацию цейтнота или вынуждены обходить ограничения (в том числе те, что заложили в код в начале проекта).

Так что же вы делаете, когда в вашем проекте появляется код «с душком»? Можно ли это предотвратить?



Ключевые моменты

XP-команды выявляют код «с душком» (или антипаттерны), чтобы избежать сложности и сохранить простоту архитектуры.

Код «с душком», такой как сырой код и очень большие классы, помогает командам обнаружить отдельные модули, которые можно упростить.

Такой код «с душком», как лапша-код, помогает им искать проблемы в более крупных конструкциях.

Особенно неприятно, когда код «с душком» вызывает эффект «стрельбы дробью», то есть разработчик пытается исправить проблему, но обнаруживает, что внесенные изменения требуют целой цепочки правок в других частях кода.

Команды, чрезмерно увлекающиеся планированием в стремлении предусмотреть все особые случаи, добавляют слишком много хуков или создают большие платформы для решения отдельных проблем, что приводит к усложнению кода.

Даже высококвалифицированные специалисты, подверженные плохим привычкам, могут столкнуться с этими проблемами.

Принимать решения по коду и проекту в последний ответственный момент

Переход к проектированию в XP-стиле — это смещение дат принятия проектных решений. Проектирование откладывается до тех пор, пока не появится возможность использовать накопленный опыт и сразу же внедрять принятые решения. Это дает возможность команде:

- разворачивать программное обеспечение как можно раньше;
- принимать решения с уверенностью;
- избегать последствий плохих решений;
- сохранять темпы разработки на первоначальном уровне, несмотря на изменения.

Продолжать инвестировать в проектирование на протяжении всего проекта и вносить изменения маленькими шагами, чтобы сохранить поток новых ценных функциональных возможностей, — цена этой стратегии заключается в том, что она требует дисциплины.

Кент Бек. Extreme Programming Explained: Embrace Change

Мы уже знаем, что scrum-команды принимают решения по планированию проекта в последний ответственный момент. Это позволяет им иметь *гораздо более простой план* — зачастую обычный набор историй и карточки задач в сочетании с бэклогом — и дает возможность проводить ограниченные по времени совещания, потому что обсуждается только актуальная информация. Иначе говоря, scrum-команды используют *идею последнего ответственного момента, чтобы привнести простоту в планирование проекта*. Это помогает повысить производительность и принимать более взвешенные решения, избегая распространенных недостатков управления проектами.

Когда дело доходит до планирования технической стороны проекта, XP-команды используют похожий прием — применяют две основные ценности простоты к архитектуре, дизайну и коду. Так же как scrum-команды, они принимают решения в последний ответственный момент. В большинстве случаев этот момент наступает после того, как код написан.

Вам не кажется странным этот подход? XP-разработчики так не считают, потому что постоянно занимаются **рефакторингом** своего кода: изменением структуры кода без изменения его поведения. Рефакторинг используется не только в XP, это распространенная и очень эффективная практика в программировании. Действительно, большинство IDE (интегрированная среда разработки — программы, которые применяют разработчики для редактирования, запуска и отладки кода) имеют встроенные инструменты для рефакторинга.

Приведем пример рефакторинга — даже если вы не разработчик, то наверняка заметите, как он делает код чище и проще для понимания. Когда мы писали книгу «Изучаем C#»*, мы включили приведенный ниже блок кода в качестве решения для одного из проектов (симулятор улья, поэтому все

* Стеллман Э., Грин Дж. Изучаем C#. М. : Питер, 2016.

названия переменных связаны с пчелами), чтобы наши читатели увидели, как он построен.

```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl)) {
            fieldForm.Controls.Remove(beeControl);
            beeControl.Size = new Size(40, 40);
            hiveForm.Controls.Add(beeControl);
            beeControl.BringToFront();
        } else if (hiveForm.Controls.Contains(beeControl)) {
            hiveForm.Controls.Remove(beeControl);
            beeControl.Size = new Size(20, 20);
            fieldForm.Controls.Add(beeControl);
            beeControl.BringToFront();
        }
        beeControl.Location = bee.Location;
    }
}
```

ЭТИ ЧЕТЫРЕ СТРОКИ
ПЕРЕНОСЯТ УПРАВ-
ЛЕНИЕ ПЧЕЛОЙ
ИЗ ФОРМЫ «ПОЛЕ»
В ФОРМУ «УЛЕЙ»

ЭТИ ЧЕТЫРЕ
СТРОКИ ПЕРЕНО-
СЯТ УПРАВЛЕНИЕ
ПЧЕЛОЙ ИЗ ФОР-
МЫ «УЛЕЙ»
В ФОРМУ «ПОЛЕ»

Рис. 7.6. Это был оригинальный фрагмент кода из одного проекта, включенного в нашу книгу «Изучаем C#»

В ходе технического обзора один из наших рецензентов отметил, что этот код слишком сложен — в нем используется очень большой метод. Поэтому мы сделали то, что должны предпринять большинство XP-команд: переработали этот код, чтобы он стал простым и легким для понимания. В данном случае мы взяли блок из четырех строк кода и перенесли их в метод, который назвали `MoveBeeFromFieldToHiveQ` («перенеси пчелу с поля в улей»). Затем сделали то же самое с другим блоком из четырех строк, извлекли их в метод, который назвали `MoveBeeFromHiveToFieldQ` («перенеси пчелу из улья в поле»). Вот как выглядел код, когда он наконец отправился в печать (два новых метода появились в коде позже).

```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl))
            MoveBeeFromFieldToHiveQ(beeControl);
        } else if (hiveForm.Controls.Contains(beeControl))
            MoveBeeFromHiveToFieldQ(beeControl, bee);
        beeControl.Location = bee.Location;
    }
}
```

Рис. 7.7. Мы провели рефакторинг кода путем извлечения двух методов. Новый код стал проще, и было легче понять, что он делает

Так гораздо понятнее. До рефакторинга для понимания того, что делает этот код, требовалось больше знаний о структуре программы, поэтому нам пришлось добавить примечания, помогающие читателям понять два блока

кода. Их перемещение в именованные методы сделало работу кода понятнее. В переработанной версии видно, что эти блоки делают: в одном пчела перемещается с поля в улей, а в другом — из улья обратно в поле.

Этот рефакторинг не только сделал блок кода более понятным, но и снизил сложность всего проекта. Разумно предположить, что где-то в другом месте проекта программисту может понадобиться переместить пчелу между полем и ульем. Если эти методы уже существуют, то он, скорее всего, будет использовать именно их — это путь наименьшего сопротивления. Но даже если поначалу он поступит иначе и заметит дублирующийся код позже, то наверняка выполнит быстрый рефакторинг, чтобы удалить повторяющийся код и заменить его вызовом упомянутого метода.

ИСПРАВЛЕНИЕ ТЕХНИЧЕСКОЙ ЗАДОЛЖЕННОСТИ ПУТЕМ БЕСПОЩАДНОГО РЕФАКТОРИНГА

Поставка кода в первый раз подобна влезанию в долги. Небольшой долг ускоряет разработку до тех пор, пока он погашается быстрой доработкой... Опасность возникает, если задолженность не оплачивается.

Уорд Каннингем, автор Agile-манифеста

Плохое проектирование и плохое кодирование вызывают дополнительные затраты времени. Даже наиболее профессиональные разработчики пишут код, который может быть улучшен. Чем дольше проблемы с архитектурой и кодом остаются нерешенными, тем сильнее они сцепляются друг с другом, что в итоге приводит к эффекту «стрельбы дробью». Эти застаревшие проблемы в архитектуре и коде называются **техническим долгом**. Успешная XP-команда в каждой итерации оставляет время для «погашения долга». Это правильное использование **временного запаса** — одной из основных практик, рассмотренных в главе 6. При этом команда добавляет истории и задачи в каждый недельный цикл в качестве буфера для поглощения непредвиденных работ.

Любой хороший финансовый консультант скажет вам, что лучший способ избежать проблем с деньгами — не делать долгов. То же самое касается технического долга. Именно поэтому XP-команды занимаются **беспощадным рефакторингом**, постоянно выискивая код «с душком» и способы его упрощения. Чем чаще программисты употребляют рефакторинг, тем больше узнают, как фактически применяется их код и чем это отличается от ожиданий. Благодаря постоянному пересмотру каждый модуль исходного кода

все лучше соответствует тому, как он фактически используется. Этот итеративный характер постоянного кодирования и пересмотра заменяет многим командам необходимость планировать в начале проекта. Но хотя беспощадный рефакторинг требует дополнительного времени, на самом деле он его экономит, поскольку с простым исходным кодом работать проще, чем со сложным.

Когда все члены команды постоянно занимаются рефакторингом, они создают *легко изменяемый* исходный код. И если они находят, что необходимо реализовать новую историю, или оказывается, что они поняли одну историю неверно и работу программы надо изменить, то переделать простой код гораздо легче. Они готовы принять изменения (это основная цель каждой XP-команды), потому что не борются против этого.

РАЗВЕ РЕФАКТОРИНГ — ЭТО НЕ ПЕРЕДЕЛКА? ВЕДЬ ОНА ОДИН ИЗ ГЛАВНЫХ ИСТОЧНИКОВ ОШИБОК

Да, это переделка, и она способствует появлению дополнительных ошибок. Если пользователям нужно что-то изменить в конце проекта, то в ходе доработки команда может случайно их допустить. Но рефакторинг — это переделка особого рода, которая предотвращает ошибки. Рефакторинг на протяжении всего проекта дает вам исходный код, состоящий из более мелких *естественных многоцветных* модулей. Альтернатива хорошо знакома большинству опытных программистов — это трудно изменяемый исходный код.

Переделка — это традиционный источник ошибок в водопадных проектах. Одна из причин в том, что архитектура становится более сложной, хрупкой, код приобретает «душок», что затрудняет внесение изменений. Но что если ошибки добавляются в процессе рефакторинга? Предположим, блок кода выполняет две разные функции, а разработчик извлекает фрагмент кода в процедуру, обеспечивающую только одну из них. Ответный ход XP-команды — разработка через тестирование, одна из основных XP-практик. Когда у программиста есть набор тестов для модуля, который подвергается рефакторингу, то эта процедура оказывается безопаснее. И действительно, гораздо спокойнее заниматься серьезным рефакторингом, имея модульные тесты. Поскольку рефакторинг — это, по определению, изменение структуры кода, не влияющее на его поведение, модульные тесты должны успешно завершаться и до рефакторинга, и после него. И на практике тесты в самом деле отлавливают почти все ошибки, которые могут быть внесены даже при обширном рефакторинге.

РАЗВЕ НЕ ЛУЧШЕ ПРЕДОТВРАТИТЬ ПОЯВЛЕНИЕ ТАКИХ ПРОБЛЕМ ЗА СЧЕТ ХОРОШЕГО ПРОЕКТИРОВАНИЯ В НАЧАЛЕ ПРОЕКТА? МОЖЕТ БЫТЬ, БЕЗОПАСНЕЕ СРАЗУ СОЗДАВАТЬ ПРАВИЛЬНЫЙ КОД?

Да, безопаснее. И это было одной из основных задач разработчиков ПО, системных аналитиков и менеджеров проектов в течение многих лет. Но очень маловероятно, что код сразу получится правильным, потому что понимание командой проблем эволюционирует по мере развития проекта и написания модулей. Обычно команда со временем меняет свое понимание проекта. Это естественный результат постоянной поставки пользователям работающего ПО (а оно более подходящий инструмент для оценки возникающих проблем, чем исчерпывающая документация).

Этим также объясняется, почему XP-команды используют итеративную разработку и включают *квартальные* и *недельные циклы* в свои основные практики. Они выделяют достаточно времени в каждой недельной итерации для учета создания модульных тестов и рефакторинга. И каждая поставка работающего ПО помогает взаимодействовать с пользователями, чтобы улучшить понимание решаемой задачи и уточнить истории. Благодаря непрерывному поиску кода «с душком» и совершенствованию архитектуры они пишут и сохраняют исходный код, который легко поддается изменению.

Но не будем слишком строги к командам, которые в прошлом разрабатывали программное обеспечение при помощи обширных требований и предварительного проектирования. У них не было необходимых инструментов. Под этим словом мы понимаем не только программные утилиты, но и командные практики, которые разрабатывались на протяжении многих лет. И эти инструменты облегчили рефакторинг и модульное тестирование. Даже развертывание и выпуск были весьма сложными. Простая компиляция кода занимала дни, а то и недели, компьютеры не были объединены в сеть, поэтому приходилось копировать программы на компакт-диски, дискеты и даже ленты. Переделка требовала больших затрат, поэтому приходилось сначала разрабатывать документацию и тщательно ее оценивать, прежде чем приступать к созданию кода.

ИСПОЛЬЗОВАНИЕ НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ ДЛЯ ПОИСКА ПРОБЛЕМ ПРОЕКТИРОВАНИЯ

Мы узнали о практике *непрерывной интеграции* в главе 6, и это один из тех мощных инструментов, которые современные команды имеют в своем распоряжении. Одна из причин, по которой непрерывная интеграция улучшает архитектуру и предупреждает проблемы с ней, заключается в том, что

она позволяет команде обнаруживать их на ранних этапах (при возникновении проблем с интеграцией).



Рис. 7.8. Непрерывная интеграция выявляет проблемы на ранних стадиях

Способ, разработанный для немедленного информирования о сбоях, называется **системой быстрых неудач**. Если вы хотите получить отказоустойчивую систему и вам важно обнаружить первопричину возможных проблем, то нужно, чтобы вероятный сбой случился как можно раньше (поэтому метод и назвали «быстрые неудачи»). Быстрые неудачи обеспечивают петлю обратной связи, которая позволяет использовать полученные знания в проекте. Эту идею можно применить не только к способу разработки, но и к создаваемому программному продукту.

Неудача — это один из принципов XP, который мы обсуждали в главе 6. Непрерывная интеграция обеспечивает быструю неудачу в проекте в том случае, когда два члена команды одновременно добавляют несовместимый

или конфликтующий код. Если члены команды имеют совместимое с XP мировоззрение, то они будут положительно воспринимать интеграционные неудачи, потому что это помогает выявлять и устранять проблемы на ранних этапах, когда сделать это намного легче.

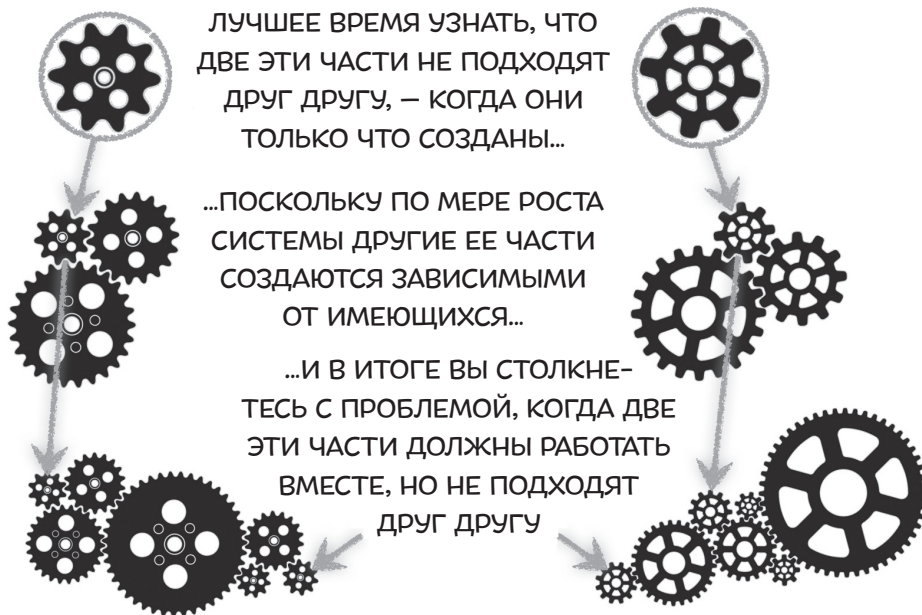


Рис. 7.9. Когда проблемы проектирования выявлены в самом начале проекта, их легче исправить, и это предотвращает появление запутанных клуджей в дальнейшем. Таков один из способов, которым непрерывная интеграция улучшает общую архитектуру

Разработка через тестирование помогает команде создавать небольшие независимые части, которые легче интегрировать, поэтому непрерывная интеграция — это меньшее бремя для XP-команды. Она постоянно тестирует каждый модуль, чтобы убедиться, что части работают. Когда все члены команды непрерывно интегрируют свой код в общую базу кода, это не дает им углубляться в свою задачу и создавать модули, которые не вполне подходят друг другу.

ИЗБЕГАЙТЕ МОНОЛИТНОЙ АРХИТЕКТУРЫ

Допустим, вы трудитесь над проектом программного обеспечения и в процессе работы многое узнаете о проблеме, которую это ПО должно решить.

Большинству команд такой сценарий хорошо знаком, и ваша команда должна принимать решения в последний ответственный момент, чтобы обойти эту проблему.

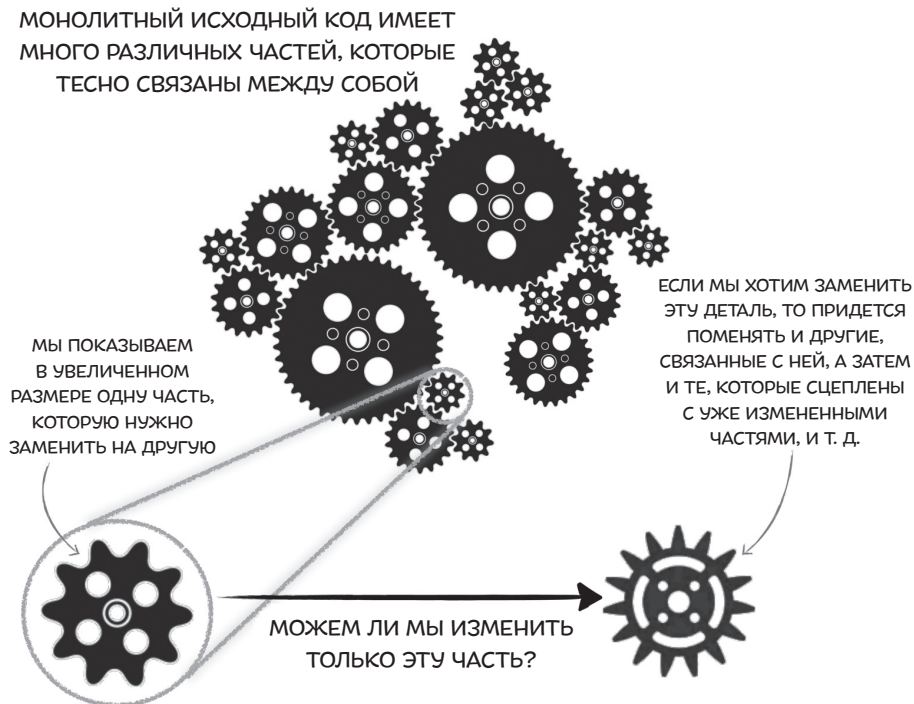


Рис. 7.10. Когда разработчики не привыкли создавать простой несвязанный код, в итоге они получают монолитную архитектуру

Традиционные водопадные команды часто сталкиваются с проблемами, развивающимися по такому сценарию. Предварительное выяснение требований, их рассмотрение большой аудиторией, а затем создание всего кода сразу приводят к тому, что его трудно изменять. У команды нет особых стимулов менять архитектуру решения, потому что такие изменения находятся под жестким контролем. Разработчики никогда не выработают привычки (наподобие постоянного рефакторинга или отслеживания кода «с душком»), заставляющие их создавать легко изменяемый код.

Это почти автоматически ведет к **монолитной архитектуре**, то есть дизайну, при котором программное обеспечение состоит из больших взаимосвязанных модулей, имеющих много взаимозависимостей и трудноотделимых друг от друга.

В начале книги мы говорили о том, чем ПО отличается от строительных работ. Самое опасное, что вы можете натворить в строительстве, — это взять кувалду и разрушить стены. В программном обеспечении удаление кода не нанесет большого ущерба — его можно восстановить из хранилища контроля версий. Гораздо опаснее, если вы напишете плохой код, а потом создадите другой, который зависит от первого. Добавление зависимостей (иногда называемых «связками», потому что они соединяют вместе два куска кода) делает сложным изменение одной из частей без того, чтобы по крайней мере оценить влияние этого изменения на другую часть.

Монолитная архитектура часто характеризуется **сильно связанным кодом**, где типичный модуль имеет много связей с другими частями системы. Попытка изменить сильно связанный код становится затруднительной — часто эта связка дает эффект «стрельбы дробью» и другие антипаттерны. Именно такие (обычно недокументированные) связки приводят к ошибкам при переделке.

Связанный код можно сделать **несвязанным**, разорвав соединения между модулями (а лучше вовсе не добавлять эти связи в начале работы).

Код «с душком» и рефакторинг помогут создать несвязанный код. Вернемся к примеру рефакторинга с кодом улья, где мы извлекли два метода из блока кода `C#` для перемещения пчел между полем и ульем. Мы увидели, что эти два метода могут повторно использоваться в другой части программы, которая должна перемещать пчел таким же образом. Но что если эти методы — часть более крупного модуля, который нуждается в большой инициализации? Программист с хорошими навыками, попытавшись повторно использовать эти простые методы, почувствует непродуманный код и проведет рефакторинг, чтобы удалить дополнительную инициализацию. Теперь этот код можно применить в двух различных частях программы, и для этого не нужно «знать», как его вызывать, потому что он отделен от тех частей кода, которые к нему обращаются. И если необходимо, чтобы он был вызван из третьей части кода, то он будет отделен и от этой части.

Систему легче поддерживать, если она собрана из небольших независимых модулей, то есть каждая часть кода отделена от любой другой, насколько это возможно (степень этой зависимости может быть удивительной), так что между ними существует очень мало зависимостей. Это главный принцип создания программы, которая может быть изменена без серьезной доработки. Если ваша архитектура не отягощена лишними связями, то вам редко придется прибегать к «стрельбе дробью» или производить целую череду взаимосвязанных изменений. И когда вы начнете так поступать и работаете в себе привычку не откладывать рефакторинг на завтра, чтобы

удалить эти сцепления, код будет становиться все лучше и лучше: каждый модуль станет выполнять только одну задачу и окажется отделен от постоянных модулей.

Инкрементальная архитектура и целостные XP-практики

В главе 6 мы говорили о 10 из 13 основных XP-практик, разделив их на категории: программирование, интеграция, планирование и командные практики. Существуют еще три основные XP-практики, и мы придумали для них отдельную категорию: *целостные практики*. Мы намеренно отложили их рассмотрение, поскольку о них надо говорить как о комплексе мер и в контексте всех прочих практик. Название «целостные» было выбрано, потому что они тесно взаимосвязаны и не работают одна без другой (в отличие, скажем, от парного программирования или недельных циклов, где команды могут продвигаться маленькими шажками).

Первая целостная XP-практика, о которой мы будем говорить, — это **инкрементальная архитектура**. Она самая трудная для понимания. В этой главе мы постараемся помочь вам понять основы этой практики и то, как она влияет на проект и команду в целом.

Вы можете найти замечательные примеры инкрементальной архитектуры во многих зрелых, качественных проектах с открытым исходным кодом, таких как Linux, Apache HTTP Server и Firefox. Эти проекты создаются вокруг стабильного ядра. Разработчики используют архитектуру, основанную на плагинах (или других способах отделения кода от ядра), чтобы создавать дополнительные функции, и только самые повторяющиеся, наиболее стабильные из них встраиваются в ядро. Такая архитектура приводит к весьма слабо связанному коду, и программисты, работающие над этими проектами, взяли за правило часто делать рефакторинг, писать много тестов и выполнять непрерывную интеграцию. (Следует отметить, что многие практики, включенные в XP, возникли или были отточены разработчиками, участвующими в проектах с открытым исходным кодом.)

Каждый из таких проектов создан программистами, добавлявшими отдельные модули, работающие независимо друг от друга и созданные специально для взаимодействия с ядром проекта. Создается впечатление, что исходный код проекта постепенно вырастает из этого ядра. Любой разработчик может добавить свои собственные модули независимо от остальной команды и рассчитывать на то, что все пишут почти полностью раздельный код, поэтому риск столкновения новых модулей со старыми

невелик. Однако работа ведется не в вакууме, и члены команды хорошо осведомлены об этом. Все дополнения происходят в том же исходном коде, где эти команды проводят автоматизированную или ручную непрерывную интеграцию. (Некоторые непрерывные интеграции и серверы сборки, доступные сегодня, возникли в ходе этих проектов.) Кроме того, каждый разработчик старается избежать кода «с душком» и антипаттернов и чувствует большую ответственность за их исправление, если они будут выявлены.

В результате применения этих отличных практик наряду с правильным поведением разработчиков исходный код растет **инкрементально**. Это пример, как очень большая команда с участниками, находящимися в разных странах, может создать качественное программное обеспечение*. Фактически инкрементальная архитектура — это *создание проектного решения в последний ответственный момент*, что дает возможность избежать одной из самых распространенных ловушек, в которую попадают даже опытные, высококвалифицированные программисты, — попытки создать все и сразу.

Команды, попадающие в такую ловушку, выработали дурные привычки, приводящие к монолитной или замысловатой архитектуре программного обеспечения. Например, они могут сосредоточиться на решении проблемы, которая крупнее текущей задачи или вовсе с ней не связана (фокусируя внимание на крайних случаях, а не на работе каждого конкретного модуля). Или посвящать массу времени обсуждению будущих дополнений и добавлять слишком много хуков. Они могут также создавать большие абстрактные платформы для решения мелких конкретных задач. Что объединяет все эти ситуации? Речь идет о командах, которые принимают чересчур много архитектурных решений и делают это слишком рано.

Однако когда команда выработала привычки, которые ведут к созданию несвязанного кода, состоящего из небольших, надежных, независимых модулей, ей не нужно продумывать архитектуру заранее. Они могут создавать архитектуру высокого уровня, не думая обо всех возможных сценариях, которые могут в нее не вписаться. И уверены, что если обнаружится непредвиденный пограничный случай, то найдутся инструменты, при помощи которых можно будет решить проблему позднее. Это раскрепощает людей и позволяет им свободно обдумывать проблемы, а также дает возможность создавать код по частям, по мере того как накапливается понимание задачи. Такая идея лежит в основе инкрементальной архитектуры,

* О командах, работающих с открытым исходным кодом, можно прочитать в книге Эрика Реймонда *The Cathedral and the Bazaar* (O'Reilly, 1999). О повседневной жизни команд, работающих с открытым исходным кодом, читайте в книге Карла Фогеля *Producing Open Source Software*, которую можно бесплатно скачать на сайте <http://producingoss.com/>.

что приводит к очень гибкому, поддающемуся изменениям коду. Но такой подход эффективен, когда *каждый разработчик по-настоящему верит* в проектирование кода и принятие решений в последний ответственный момент. Любой член команды должен быть убежден, что быстрее создать небольшую несвязанную часть системы сегодня и доверить команде внесение в будущем исправлений, если они понадобятся.

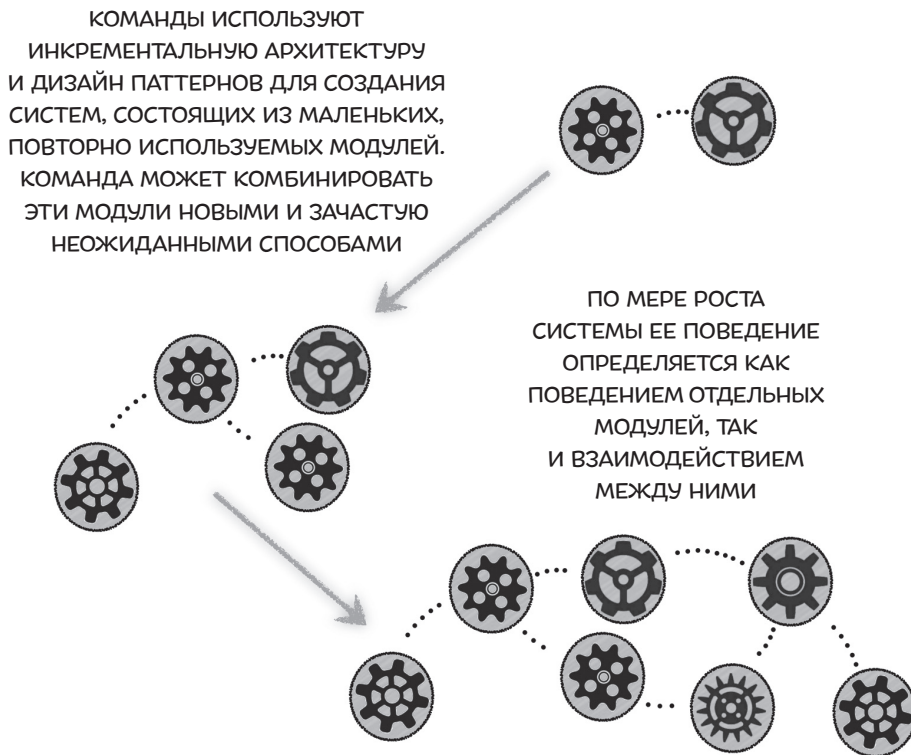


Рис. 7.11. Инкрементальная архитектура приводит к более надежной и пригодной к обслуживанию системе

Речь идет не только о том, как программисты проектируют и собирают код. Имеет значение и атмосфера внутри команды: как люди взаимодействуют, в какой обстановке работают и, главное, как относятся друг к другу.

Многие команды испытывают дефицит времени. Еще хуже, когда они понимают, что он искусственно создан руководством, которое думает, что невыполнимые сроки — этот лучший мотиватор. Команда с таким мировоззрением воспринимает любую работу, не добавляющую строки в код, как избыточную.

Когда вы чувствуете, что вам не хватает времени выполнять работу как следует, *почти не имеет значения, насколько хороши ваши навыки*. Чем сильнее сроки и план давят на вас, тем вероятнее, что вы откажетесь от таких полезных практик, как рефакторинг, разработка через тестирование и непрерывная интеграция. Например, вы не будете заниматься рефакторингом, потому что код и так работает, а у вас нет времени улучшить его. Ваш код будет выполняться и с модульными тестами, и без них, поэтому каждая минута, потраченная на их написание, расценивается как недобавленная новая функция. Умом мы понимаем, что эти практики помогают писать код быстрее, но, находясь под сильным давлением, мы подсознательно начинаем считать, что *на них не хватает времени*.

Рассмотрим случай, когда команда находится под прессингом руководителя. Возможно, он пообещал пользователям законченный продукт, но согласился на нереальные сроки и хочет уложиться в них. Поэтому он требует от разработчиков отказаться от модульного тестирования, рефакторинга и всего того, что напрямую не связано с добавлением новых функций. В такой команде обычное явление, когда программисты, сталкиваясь с кодом «с душком», думают: «У меня нет времени разбираться! Мне нужно как можно скорее закончить эту новую функцию и перейти к следующей». Конечно, они будут создавать сильно связанный код (потому что затрата нескольких минут, чтобы развязать его, расценивается как избыточная работа, ведь у них *просто нет времени на обдумывание!*). Руководитель такой команды фактически мешает ей эффективно внедрять XP, и в результате появляется код «с душком», который трудно изменять и поддерживать. Их проекты практически всегда выполняются с опозданием, а некоторые полностью проваливаются*.

В XP есть действенное средство против этого, поэтому поговорим о двух оставшихся целостных практиках: **энергичной работе** и **единой команде**.

КОМАНДЫ РАБОТАЮТ ЛУЧШЕ, КОГДА У НИХ ЕСТЬ ВРЕМЯ ПОДУМАТЬ

Разработка программного обеспечения — это игра на понимание, и прозрение приходит к подготовленному, отдохнувшему, расслабленному разуму.

Кент Бек. Extreme Programming Explained: Embrace Change

Энергичная работа означает создание среды, в которой каждый член команды имеет достаточно времени и свободы делать свою работу. Это

* Вспомним цитату Грэди Буча об инновациях и страхе провала из главы 5. Здесь приведен хороший пример того, как команда совершенно не испытывает удовольствия от работы.

поддерживает в них ментальное состояние, в котором они способны выработать и использовать хорошие привычки, ведущие к лучшему, естественно изменяемому коду. В таком состоянии они могут производить гораздо больше кода и поставлять более ценные функции потребителям за меньшие отрезки времени.

Разработка программного обеспечения — это преимущественно умственная деятельность*. Любой хороший разработчик тратит часы, размышляя над проблемой. Озарение может прийти внезапно — во время обеда, принятия душа, катания на велосипеде и т. д. Каждый программный проект опирается на ряд небольших инноваций, следующих одна за другой.

Чтобы разработчик вошел в состояние «потока», включающее в себя высокую концентрацию и максимальную продуктивность**, требуется время (обычно от 15 до 45 минут). Перерывы и отвлекающие факторы могут вывести его из этого состояния. Но когда он находится под давлением необходимости поставить код в немыслимые сроки, войти в «поток» невозможно. Он так стремится поскорее разделаться с кодом, что ему некогда думать.

Неуважение, установка нереальных сроков и другое негативное поведение руководителя может вызвать состояние пассивности, при котором люди не имеют возможности принимать решения или внедрять инновации.

Когда менеджеры создают атмосферу постоянного отставания от сроков, назначаемых произвольно, они неизбежно будут халтурить, создавая код. Обычно это сопровождается отказом от многих полезных XP-практик. Кроме того, в такой ситуации разработчики с трудом входят в состояние «потока», необходимое для использования в проекте более простых решений.

XP-команды стремятся создать совсем иную рабочую среду, наполненную энергией, означающую, что каждый член команды ощущает свою *автономность*.

Каждый сам решает, как выполнять свою работу, и понимает, что имеет право вносить изменения не только в код, но и в план проекта. XP-команды делают это путем применения практик, ориентированных на планирование: используя недельные циклы. Поэтому они не принимают заранее

* Не кажется ли вам странным, что программирование — это преимущественно умственное занятие? Вот небольшой эксперимент, чтобы доказать это самому себе. Вспомните кусок кода, для написания которого потребовались часы. Теперь перепечатайте его. Работа будет идти намного быстрее. Набор текста — это физическая часть написания кода, явно не самая трудоемкая.

** Некоторые соображения о том, как разработчики достигают и поддерживают состояние «потока», можно найти в книге Тома Демарко и Тимоти Листера «Человеческий фактор. Успешные проекты и команды» (М. : Символ-Плюс, 2014).

конкретных решений, если есть возможность сделать это позже. Кроме того, они используют временной резерв, добавляя работы с низким приоритетом, которые легко перенести в следующий цикл. Эти практики повышают чувство командной автономии, предоставляя больше гибкости при планировании. Автономия может исходить от самого кода: избегая антипаттернов и собирая легко изменяемый код, команда открывает для себя возможность выбора в будущем.

В главе 3 мы узнали об agile-принципе **устойчивого темпа**: гибкие процессы способствуют такой же разработке. Спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп в течение неопределенного времени. Считаете ли вы, что мужество как XP-принцип также требуется для того, чтобы поддерживать устойчивый темп?

Многие XP-практики используют термины «энергичная работа», «40-часовая рабочая неделя» и «устойчивый темп» как синонимы, поэтому XP-команды создают среду, где они могут энергично работать путем выделения разумного количества рабочих часов. Идее 40-часовой рабочей недели много лет — в свое время профсоюзы требовали «восемь часов на работу, восемь часов на отдых и восемь часов на сон»*. К 1950-м годам после бесчисленных исследований производительности и изучения диаграмм, показывающих ее снижение и увеличение проблем качества при рабочей неделе, превышающей 40 часов, многие отрасли приняли этот принцип. XP-команды знают: при устойчивом темпе работы и возможности жить полноценной жизнью вне офиса люди трудятся качественнее и испытывают чувство удовлетворенности.

ЧЛЕНЫ КОМАНДЫ ДОВЕРЯЮТ ДРУГ ДРУГУ И ПРИНИМАЮТ РЕШЕНИЯ ВМЕСТЕ

*Людам нужно чувство команды, мы принадлежим к ней.
Мы вместе делаем одно дело. Мы поддерживаем друг друга
в работе, развитии и обучении.*

Кент Бек. Extreme Programming Explained: Embrace Change

Хорошие команды вместе могут сделать гораздо больше, чем те же самые люди, работая по отдельности. Почему? Приглашая специалистов, обладающих необходимыми навыками и взглядами, и создавая для них среду, поощряющую открытое общение и взаимное уважение, вы помогаете рождению инноваций. Члены команды делятся своими идеями, а это создает еще больше новых идей.

* Подробнее об истории 40-часовой рабочей недели и 8-часового рабочего дня: https://en.wikipedia.org/wiki/Eight-hour_day, проверено 12 марта 2017 года.

Практика **единой команды** помогает отдельным ее членам объединяться. Когда они сталкиваются с препятствиями, то преодолевают их совместно. Важные решения, влияющие на направление развития проекта, также принимаются совместно. Все члены команды учатся доверять друг другу и определять, какие решения можно принимать самостоятельно, а какие — командой в целом.

В единой команде каждый принимает участие в обсуждении того, какие функции ценны для пользователей, какую работу команда возьмет на себя и как будет создано программное обеспечение. Если команда доверяет любому своему участнику принимать решения о кодировании, которое будет поставлять наибольшую ценность, то риск, что некоторые программисты начнут тратить время на дополнительный код, невелик.

Обратная сторона доверия — это понимание того, что каждый может ошибиться. Когда «единая команда» работает динамично, ее члены не боятся совершить оплошность, потому что знают: коллеги их поймут. Ведь единственный способ продвинуться вперед — вместе учиться на неизбежных ошибках.

ХР-ПРОЕКТИРОВАНИЕ, ПЛАНИРОВАНИЕ, КОМАНДА И ЦЕЛОСТНЫЕ ПРАКТИКИ ФОРМИРУЮТ ЭКОСИСТЕМУ, СТИМУЛИРУЮЩУЮ ИННОВАЦИИ

Единая команда, имеющая энергичную рабочую среду, проектирует лучше, чем разобщенная, в которой царит атмосфера подавленности.

Инновации могут казаться высокими ценностями, но для эффективной ХР-команды это ежедневная реальность. Разработчики, регулярно достигающие состояния «потока» и работающие в информационном пространстве, которое стимулирует осмотическую коммуникацию, часто подпитывают друг друга идеями. А инкрементальное проектирование ПО дает каждому программисту свободу писать новый код с меньшим числом ограничений. Хорошие привычки, выработанные командой, помогают ей избегать антипаттернов, поэтому, когда она наращивает исходный код постепенно, она не накладывает на себя ограничений, который могут сказаться в будущем.

Если менеджер такой команды доверяет ей, то работа идет лучше. Он помогает команде понять ценность, которую она предоставляет, не устанавливает нереальных сроков и создает атмосферу, в которой можно сосредоточиться на создании лучшего продукта. Команда с таким менеджером работает быстрее и имеет больше шансов создать продукт, соответствующий требованиям пользователей, в который легко вносить изменения.

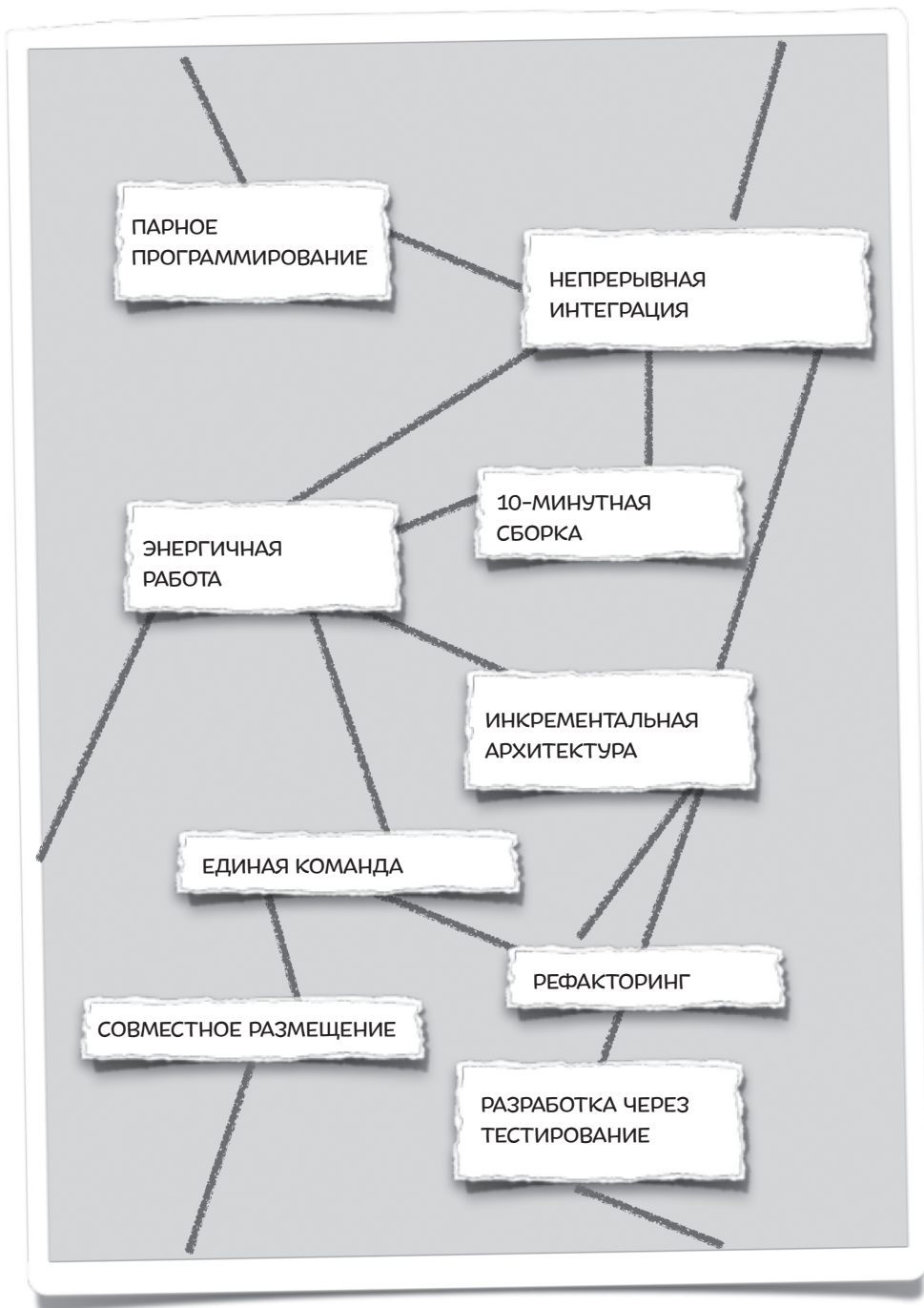


Рис. 7.12. Все практики взаимосвязаны и усиливают друг друга

Мы отложили рассмотрение целостных практик до главы 7, потому что в сочетании с другими основными XP-практиками они становятся понятнее, и команда не сможет освоить их по-настоящему, не имея образа мыслей, совместимого с XP-ценностями и принципами. Если рассматривать практики по отдельности и реализовывать их одну за другой, то, скорее всего, получится результат «лучше-чем-ничего», который не окажет глубокого влияния на архитектуру продукта.

В хорошей XP-команде каждый усвоил принципы и ценности, поэтому практики формируют *экосистему*, в которой они усиливают друг друга. Создание тестов до написания кода помогает писать небольшие, независимые, несвязанные модули, благодаря которым проблемы в коде и антипаттерны проще исправлять.

Члены команды работают в паре, что помогает им стать единой командой, заставляет быть более собранными, создает энергичную рабочую среду и дает время обдумать архитектуру. Они постоянно занимаются интеграцией, поэтому, когда одна пара вносит в код дефект, влияющий на другую часть программы, модульное тестирование быстро выявляет ошибку. Это позволяет команде быстро устранить проблемы еще до того, как они окажутся погребены под дополнительными слоями кода, и сделать его чище. Непрерывная интеграция не представляет сложности, потому что существует 10-минутная сборка, которая, помимо прочего, облегчает команде возможность сосредоточиться и позволяет меньше отвлекаться.

Разработчику с правильным XP-мышлением все это кажется очень простым. Он считает, что просто создает ПО именно так, как надо, а все эти принципы реализуются автоматически.

Команда, не обладающая таким мышлением, будет наткаться на барьеры при внедрении XP. Она застрянет в реализации отдельных практик и почувствует, что внедрила XP, как только все начнут писать тесты, кодировать в парах, работать недельными циклами или использовать сервер непрерывной интеграции. Это простые, осязаемые вещи, выполнение которых можно отметить галочками в списке. Но существует разрыв между применением практик и получением эффекта «экосистемы», о котором XP-команда, возможно, читала, но так и не достигла. Так же как *scrum*-команды, реализовавшие все практики, но так и не добившиеся поразительных результатов или гиперпродуктивности.

Как команде сформировать правильное мышление? Как совершить переход от реализации практик к коренным образом измененным способам проектирования и создания программного обеспечения?

ИНКРЕМЕНТАЛЬНАЯ АРХИТЕКТУРА В СРАВНЕНИИ С ПРОЕКТИРОВАНИЕМ ДЛЯ ПОВТОРНОГО ИСПОЛЬЗОВАНИЯ

Эндрю. Итак, проектирование для повторного использования — это не всегда здорово?

Ауке. Это не очень удобный подход к разработке программного обеспечения. Обычно, работая с открытым кодом, ты сначала создаешь нечто для одного пользователя.

Затем кто-то другой берет это решение и начинает приспосабливать к своим нуждам. И он может использовать его повторно. И как только вы сделали это — применили, а затем использовали повторно, — вы знаете, что в этих решениях общего, и можете выполнить рефакторинг общей функциональности. Вы видите довольно много таких случаев. Проектирование для повторного использования слишком статично. Оно не работает таким образом и очень плохо для этого подходит. Его можно пытаться к чему-то применить, но это не имеет особого смысла в обозримом будущем. Повторное использование подходит для разработки гораздо лучше.

Из интервью с Ауке Джилдерда, Beautiful Teams (глава 8)

Мы уже говорили о том, как команда может начать формировать правильное мышление для XP, внедряя его практики.

Во-первых, она могла бы получить результаты «лучше-чем-ничего». Но в ходе использования практик люди начинают понимать, как те взаимодействуют, и представление каждого члена команды о том, как создавать ПО, начинает меняться. Это справедливо не только для Scrum, но и для XP. И ключ к обладанию XP лежит в инкрементальной архитектуре.

Набор утилит Unix — классический пример инкрементальной архитектуры, и это то, что позволило тысячам разработчиков добавлять к нему маленькие и большие части на протяжении многих лет. Утилиты Unix состоят из множества маленьких кусочков*, разработанных различными программистами, намеревавшимися решить свои конкретные проблемы (они не старались создать огромную, монолитную операционную систему). На примере набора утилит Unix мы рассмотрим, как инкрементальная архитектура может помочь множеству людей (большинство из которых никогда не встречались) внести свой вклад в создание большой, устойчивой, высококачественной системы, продолжающей развиваться на протяжении десятилетий.

Вот пример того, как работают утилиты Unix. Допустим, у вас есть масса больших текстовых файлов с множеством данных — может быть, это адреса для списка рассылки, конфигурационные файлы операционной системы или числа, которые должны быть обработаны. Нужно быстро придумать

* Полный список утилит: https://en.wikipedia.org/wiki/List_of_Unix_commands, проверено 12 марта 2017 года.

способ их преобразовать (например, вытащить только имена и номера телефонов, изменить определенные разделы конфигурации нужным образом или найти значения данных, соответствующих шаблону). Как бы вы это сделали?

Можно написать программу, которая считывает данные из файла, обрабатывает строки, разделы, последовательности и формирует выходные данные. Но если вы знакомы с Unix-утилитами (такими как `cat`, `ls`, `cut`, `awk`, `sed` и т. д.), то знаете, что они созданы для решения подобных проблем. Вы пишете скрипт (или даже просто команду, которую выполняете в командной строке), считывающий данные из файла, выполняющий преобразования и записывающий результаты в другой файл.

Например, у вас есть большая, разделенная запятыми адресная книга в файле `addr.txt`. В ней восемь столбцов: имя, должность, адрес электронной почты, номер телефона, почтовый адрес, город, штат и почтовый индекс. Вы хотите найти имя, должность и номер телефона адресатов, проживающих в Миннесоте. Если полное имя и должность записаны в первых двух столбцах, номер телефона — в четвертом, а штат и почтовый индекс в последних столбцах, то эта команда в Unix будет производить правильный вывод и записывать его в файл `output.txt`*:

```
EGREP „MN,[0-9]{5}([-][0-9]{4})? $" ADDR.TXT | CUT-D,-FL,2,4 > OUTPUT.TXT
```

Разве Unix-утилиты создавали люди, желавшие упростить процесс обработки адресных книг? Конечно, нет. Утилиты для Unix основаны на *философии простоты*: каждая выдает результат, который любая другая утилита может использовать в качестве входных данных, и каждая делает одну конкретную работу. Утилита `cut` последовательно читает строки ввода, вырезает определенные поля или столбцы и вставляет их в выходные данные. Утилита командной строки `grep` проверяет каждую строку на входе и передает на выход только те, которые соответствуют заданному шаблону. Но эти две утилиты, особенно в сочетании с другими средствами Unix, могут выполнять практически неограниченное количество задач. Например, системный администратор может комбинировать их с такими утилитами, как `find` (она ищет файлы в файловой системе) или `last` (называет пользователей, последними работавших на этом компьютере), чтобы выполнить множество сложных задач системного администрирования.

* Вы действительно обеспокоены тем, что существует множество крайних случаев, которые не обрабатываются в нашем примере? Например, предполагается, что все адреса находятся в США, строка заголовка не обрабатывается, не принимаются в расчет закавыченные строчки с вложенными запятыми. Не надо обращать внимание на такие случаи — это всего лишь учебный пример.

Эти утилиты — основа того, благодаря чему Unix стала наиболее популярной операционной системой для интернет-сервисов и бизнес-применений. И в течение многих лет общая архитектура системы утилит развивалась среди людей, использующих эти утилиты каждый день, вместе с культурой системного администрирования. Со временем появлялись новые утилиты и возможности их использования. Так, сжатие файлов получило широкое распространение в 1980–1990-х годах. Когда в 1992 году gzip добавилась в набор утилит для Unix, для этого уже существовал простой шаблон. Еще до написания первой строки было ясно, как она будет получать данные и отдавать их (конвейер), выполняться (из командной строки) и даже в каком виде появится документация (страницы man). Утилиты могут легко расширяться, а поскольку все инструменты не связаны друг с другом, возможность сжатия и распаковки файлов легко добавляется, не требуя каких-либо изменений в других частях системы.

Уже более сорока лет расширение набора утилит Unix происходит именно таким способом. Тысячи разработчиков внесли свой вклад в отдельные утилиты или улучшили существующие. Unix-система росла естественным образом. А вместе с ней развивались культура и база знаний. Это позволило людям использовать Unix для выполнения все более сложных задач, что, в свою очередь, помогло найти новые способы ее совершенствования.

КОГДА МОДУЛИ ВЗАИМОДЕЙСТВУЮТ ПРОСТЫМ СПОСОБОМ, СИСТЕМА МОЖЕТ РАСТИ ИНКРЕМЕНТАЛЬНО

Все утилиты Unix придерживаются очень строгого шаблона входных и выходных данных.

Символ «|» в командной строке обозначает конвейер, он передает выходные данные одной утилиты и вход другой. Символ «<» выполняет ввод из файла, а символ «>» — вывод в файл. Все стандартные утилиты Unix создают выходы, работающие с этими конвейерами, — такова часть **договора**, которого каждая из них придерживается. Договор между утилитами Unix очень простой, что позволяет легко расширить всю систему. Пока каждая дополнительная утилита придерживается того же договора, она будет соответствовать другим утилитам.

Важный инструмент для оказания помощи команде в создании системы, использующей поэтапный подход, — это очень простой договор между ее модулями. Как они взаимодействуют друг с другом, передают ли сообщения? Вызывают функции или методы? Предоставляют ли услуги доступа

в сети? Чем проще и последовательней коммуникационный механизм между модулями, тем легче команде добавить новые.

Как вы делаете договор простым? Так же, как и модули: принимаете решения о том, как они взаимодействуют, в последний ответственный момент. И у нас уже есть инструмент, чтобы помочь в этом: разработка через тестирование. Когда программист создает модульные тесты, это помогает убрать сложность и заставляет его использовать этот модуль, прежде чем он написан. Можно применять один и тот же модуль тестирования утилит или платформ, чтобы написать простые **интеграционные тесты**, проверяющие, как различные модули взаимодействуют друг с другом. И разработчик напишет эти тесты, прежде чем добавит код для взаимодействия. Например, если есть взаимодействие между модулями, где выход одного из них передается в качестве входных данных в другой, выход которого поступает в третий, то разработчик напишет тест, который имитирует, как они «общаются» — то есть *интегрируются*.

Причина, по которой это помогает сохранить контракт между модулями простым, заключается в том, что легко понять, когда он начинает усложняться. При простом контракте этот вид интеграционного теста очень легко написать. Но если контракт сложный, то интеграционный тест становится для разработчика кошмаром. Сначала ему надо инициализировать множество разных, казалось бы, несвязанных объектов, затем кое-как преобразовать данные из одного формата в другой и вертеться, как уж на сковороде, чтобы обеспечить взаимодействие модулей. И так же, как создание модульных тестов, написание в первую очередь интеграционных тестов поможет избежать этих проблем, прежде чем они будут встроены в программное обеспечение.

ОТЛИЧНАЯ АРХИТЕКТУРА ВОЗНИКАЕТ ИЗ ПРОСТЫХ ВЗАИМОДЕЙСТВИЙ

Мы используем термин «возникает», когда сложное поведение появляется из простых систем. Возникновение — это распространенное явление: отдельные муравьи ведут себя примитивно, а весь муравейник демонстрирует гораздо более сложное поведение, возникающее в результате взаимодействия между муравьями. Таким образом, муравейник — это больше, чем сумма всех его обитателей.

Когда система разработана так, что ее поведение формируется из *взаимодействия* отдельных модулей, работающих вместе, и кажется, что они «вылупились» из одного модуля, это называется **возникающей архитектурой**.

Системы, созданные с использованием возникающей архитектуры, практически всегда состоят из маленьких, независимых, несвязанных модулей (подобных утилитам Unix или муравьям). Эти части объединяются для выполнения сложных задач, и поведение системы зависит как от взаимодействия между этими модулями, так и от них самих.

Глубокая иерархия модулей, вызывающих один другой, применяется редко. Вместо этого система будет использовать сообщения, очереди или другие способы коммуникации модулей, не требующие централизованного управления. Утилиты Unix — хороший пример такой организации работы*. Входной сигнал передается от одной утилиты к другой, что облегчает их объединение в цепочку. Это приводит к очень простому, последовательному использованию: вызывается первая утилита, потом вторая, затем третья. Можно создать и более сложное взаимодействие между инструментами, но это приводит к простой, неглубокой структуре вызовов (в отличие от глубокой, вложенной иерархии вызовов со слоями модулей).

Если система спроектирована подобным образом (простые модули, взаимодействующие простым способом), то можно получить интересный эффект. Команде начинает казаться, что *поведение* возникает из системы в целом, а не из отдельных модулей. И возникает ощущение, что сложное поведение не имеет единственного источника. Это очень похоже на поведение муравейника, которое тоже не определяется отдельными особями. Команда обретает уверенность, что каждая часть системы проста и поведение ее как целого возникает из взаимодействия между этими модулями.

Людам, потратившим годы на создание набора утилит Unix, знакомо это чувство возникающего поведения. Можно получить первое *поверхностное* представление о возникающем поведении, наблюдая за тем, как сочетаются утилиты в приведенном примере с адресной книгой: `cut` умеет только извлекать символы из строки, а `grep` — искать соответствие шаблонам, но их сочетание обеспечивает обработку адресов без специальной команды, которая пояснит системе, что такое адрес и как его обработать.

XP-команды, работающие с полной отдачей, считают естественным применение возникающей архитектуры. Все начинается с простоты: каждый модуль предназначен для конкретного использования. Разработка через тестирование гарантирует, что он остается простым и имеет только одно предназначение. Программист пишет тесты, чтобы убедиться: модуль выполняет одну функцию, — а потом создает код. И если он успешно проходит

* Муравьи — это также хороший пример несвязанных модулей, использующих простую коммуникацию (при помощи феромонов), что приводит к сложному поведению системы в целом.

тесты, то его разработка прекращается. Поэтому никакого дополнительного поведения у модуля не появляется, и в нем нет кода, написанного без основания. Весь код в модуле необходим.

Команда избегает глубоких вложений вызовов, когда один модуль вызывает другой, тот, в свою очередь, третий, и т. д. Вместо этого структура вызова программы выглядит плоской и широкой, что снижает зависимость между модулями. Такие взаимодействия просты настолько, насколько это возможно: когда один блок нуждается в данных, он получает их из другого блока или из очереди сообщений, поэтому не нужно знать, откуда они взялись. Чтобы сохранить простоту системы, команда избегает многоэтапных, сложных взаимодействий между большим количеством модулей (подобно тому, как Unix-утилиты передают данные от одной утилиты другой по конвейеру).

Новая архитектура возникает, когда команда обнаруживает изменения, которые необходимо сделать. Чтобы добавить новое поведение в систему или переменить способ ее работы, команда должна изменить отдельные модули или способ их взаимодействия. Но поскольку каждый модуль несвязан и взаимодействия между ними просты, эти изменения, как правило, не носят каскадного характера по всей системе. Если встречается код «с душком» (скажем, программист обнаруживает, что занимается «стрельбой дробью», продирается через спагетти-код или сырые объекты), то это серьезное предупреждение, что во время работы были допущены ненужные усложнения. И разработчик знает (а что еще важнее — *чувствует*): стоит потратить время на рефакторинг, чтобы разделить код на более простые модули.

Система, построенная на возникающей архитектуре, может развиваться годами, оставаясь простой для поддержки и изменений. Успешные XP-команды имеют инструменты для поддержания простоты исходного кода. Стиль их работы поощряет постоянно отслеживать появление сложного кода и выполнять его рефакторинг. Это, в свою очередь, позволяет им без напряжения вносить изменения, ведущие к возникновению архитектуры. Такой эффективный цикл приводит к простому, стабильному и очень качественному коду — и в результате команды обнаруживают, что таким образом можно создавать и поддерживать программное обеспечение намного быстрее. Когда система нуждается в поддержке (исправить ошибку или внести изменения), редко требуется серьезная модификация, затрагивающая много различных частей кода.

Поскольку члены команды знают, что система гибкая, и им известно, что можно расширять, а что — нет, они успешнее принимают решения в последний ответственный момент. Как у единой команды у них появляется

понимание того, когда и какие архитектурные решения можно отложить на завтра. А это, в свою очередь, помогает сегодня сохранять простоту кода. Команда менее напряжена, а ее члены лучше работают вместе. Они производят качественный код очень быстро и, когда его требуется изменить, делают это легко и без ошибок.

Другими словами, команды сохраняют код простым, что позволяет им принимать изменения.

В этом суть ХР.



Описание: команда занимается разработкой фантазийного баскетбольного сайта

Джастин — первый разработчик

Даниэль — второй разработчик

Бриджит — менеджер проекта

Акт V. Окончательный результат

Даниэль и Джастин были заняты парным программированием, когда Даниэль вдруг воскликнула:

— Ты понимаешь, что мы работаем в паре уже почти три месяца?

Прошло около года с тех пор, как Даниэль рассказала команде об ХР. Они делали выпуски каждую неделю, но последний недельный цикл был особенным. Компания объявила о рекламной акции в одной из крупнейших телевизионных сетей, и команда просто вытолкнула код в производство. Пару минут Даниэль и Джастин вспоминали, как шла работа на протяжении нескольких последних циклов.

Джастин сказал:

— Ты права. Все это время мы работали парами, но меняли партнеров так часто, что, по-моему, я успел поработать с каждым членом команды.

Даниэль спросила:

— Когда ты в последний раз засиживался допоздна?

Джастин задумался.

— Знаешь, мне придется поискать в телефоне последнее сообщение с извинениями. Давненько это было.

— Думаю, это потому, что мы стали лучше, — сказала Даниэль. — Я знаю, что теперь пишу более простой код, чем шесть месяцев назад.

— Ты права. Когда я впервые прочитал о разработке через тестирование, такая идея показалась мне чересчур абстрактной. Теперь же я вижу в ней смысл, — ответил Джастин. — Раньше мы рассуждали об этом всей командой, пытаясь убедить друг друга, что это пустая трата времени. Я не припомню, когда в последний раз мне нужно было спорить на эту тему.

Даниэль добавила:

— Самое забавное — это отсутствие ощущения, что я стала работать совсем по-другому.

Проходившая мимо Бриджит остановились и прислушалась.

— А я эту разницу чувствую, — сказала она. Джастин и Даниэль удивленно уставились на нее. Они не заметили, как она подошла. — Раньше моя работа в основном заключалась в том, чтобы убедить менеджеров продуктов дать нам больше времени. Мы всегда отставали от графика, и казалось, что любая мелочь требует нескольких месяцев работы.

Джастин спросил:

— А как теперь?

— Совершенно иначе! Раньше мне все время приходилось говорить «нет». А теперь я обычно соглашаюсь, даже если меня просят о действительно крупной задаче. Вы сделали мою работу намного проще.

— Я догадываюсь, почему вы больше не кричите на нас, — сказала Даниэль. — И думаю, именно поэтому мы не вкалываем в выходные, пытаясь наверстать упущенное время.

С минуту Бриджит смотрела на Джастина и Даниэль.

— Пойдите, так, значит, вы могли бы работать больше?

Джастин решил, что она пошутила. Даниэль не была уверена в этом.

— Я думаю, нам еще надо подучиться, — сказала она.



Ключевые моменты

*Разработчики исправляют код «с душком» путем его постоянного **рефакторинга** или улучшения структуры кода без изменения его поведения.*

За счет беспощадного рефакторинга XP-команды избегают **технического долга** или проблем в коде, которые известны, но еще не были исправлены.

Временной резерв помогает убедиться в том, что в каждом недельном цикле достаточно времени для оптимизации и погашения технической задолженности.

Непрерывная интеграция помогает командам найти проблемы интеграции на ранней стадии, когда их легче исправить.

Когда команда занимается рефакторингом, оплачивает технический долг и исправляет проблемы кода, в итоге она получает код, который легко изменить.

XP-команды используют **инкрементальную архитектуру**, создавая несвязанный код, который состоит из маленьких независимых модулей.

Команды инкрементально собирают только те модули, которые непосредственно необходимы для решения текущей проблемы, и доверяют друг другу принимать работающие решения в последний ответственный момент.

Энергичная работа позволяет XP-командам создать среду, где каждый сотрудник может программировать в стабильном темпе и располагает достаточным временем, чтобы сделать все правильно.

Члены XP-команды доверяют друг другу и чувствуют, что они единое целое.



Часто задаваемые вопросы

Инкрементальная архитектура — это звучит слишком абстрактно. Неужели кто-то действительно создает программные продукты таким образом?

Когда люди называют что-то «абстрактным», зачастую это означает: «я еще не пробовал так делать, но, по-моему, будет очень непросто».

В XP нет ничего абстрактно-теоретического. Каждая практика здесь — это определенный навык, которому программисты должны учиться,

практиковаться в нем и совершенствоваться. Чем лучше они усваивают такие практики, тем быстрее превращают их в полезные привычки при написании кода.

Ценности и принципы — это вполне практические идеи, которые помогут улучшить навыки работы каждого в отдельности и команды в целом.

Вот почему при работе с ХР нужна практика. Невозможно, купив гитару, ноты и самоучитель, стать за ночь великим музыкантом. Точно так же нельзя, начав применять ХР-практики, сразу их полностью освоить.

Когда бы я ни добавлял элементы технического долга в качестве временного резерва, они просто не выполняются. Как же мне погасить этот долг?

Если вы обнаруживаете, что ваша команда редко принимается за выполнение тех элементов в недельном цикле, которые были добавлены в качестве временного запаса, значит, вы закладываете в каждый цикл слишком много задач. Элементы временного резерва не относятся к числу необязательных. Они всего лишь не препятствуют команде выпустить готовое ПО в конце итерации. Это важные элементы, и большинство из них должно быть выполнено. Если их не удастся сделать, то попробуйте уменьшить общее количество элементов при планировании недельного цикла.

К тому же, если эти элементы технического долга так и не выполняются, то, возможно, у вашей команды есть проблемы с мировоззрением. Действительно, существует ловушка, связанная с техническим долгом, в которую попадают многие команды. Они будут выявлять технический долг, но вместо его безотлагательного погашения добавляют эти задачи в бэклог, используя карточки, электронные таблицы, тикеты и другие механизмы отслеживания. Как только техническая задолженность определена, она получает более низкий приоритет, чем развитие новых возможностей. Вряд ли команду, которая никак не соберется погасить технический долг, а просто признает его и помечает как задачу на будущее, можно назвать хорошей.

Команда, не устраняющая технический долг, воспринимает его как естественный побочный продукт разработки. Для нее важнее выпустить новую функцию — даже если это приводит к читерским манипуляциям или клуджам, — чтобы проект мог двигаться дальше.

И действительно, нет ничего предосудительного в том, чтобы считать поставку ценного программного обеспечения более важной, чем создание красивого технического решения. Причина, по которой успешные ХР-команды постоянно устраняют технический долг, заключается

в том, что таким образом легче продолжить поставку ценного ПО в будущем.

Если вы считаете нормой устранение технического долга во время добавления дополнительного кода, а также проведение постоянного рефакторинга в ходе работы, то вы чувствуете, что делаете важное дело. Каждый в вашей команде знает, что нужно гасить технический долг, как только он обнаружен, потому что это дает возможность легко изменять исходный код. Вот почему члены XP-команды резервируют в недельном цикле достаточно времени (и действительно чувствуют, что его хватит), чтобы устранять проблемы.

Если вы обнаружите, что технический долг накапливается, то следует обсудить это с командой. Постарайтесь выяснить, не связано ли это с внешним давлением, цель которого — добавить как можно больше новых функций. Или дело в том, что вы не считаете погашение настолько важным, чтобы заняться им немедленно. А может, это происходит потому, что вашу команду поощряют за добавление новых «крутых» функций, а не за пломбирование дыр. Если погашение технического долга требует столько же усилий, сколько создание новых функций, но вы собираетесь продемонстрировать серьезные улучшения и руководство знает, что вы пишете новый «крутой» модуль, от которого пользователи будут в восторге, то есть смысл отодвинуть технический долг на нижние позиции в списке приоритетов. Независимо от причины обсуждайте это друг с другом, чтобы принять верное решение.

Некоторые XP-принципы звучат сомнительно. Не потому ли здесь говорится о самоподобии, что это звучит внушительно, или дело в том, что методология создавалась в 1990-е, когда все обсуждали самоподобие, фракталы и теорию хаоса?

Вовсе нет. Легко отказаться от принципа, если вы еще не видели, насколько успешно он улучшает команду и ее понимание XP. Но каждый из этих принципов действительно помогает правильно понять XP. Самоподобие не исключение.

Оно означает, что одни и те же паттерны проявляются в различных масштабах — от очень больших до совсем маленьких. В хорошей архитектуре программного решения существует большое количество самоподобия. Основательное, надежное и простое в использовании программное обеспечение часто состоит из слоев с точно такими же характеристиками, а те в свою очередь, — из таких же модулей. И когда вы открываете один из этих модулей и смотрите на код, то что вы там увидите? Правильно — основательный, надежный и легкий для понимания код.

Самоподобие присутствует и при энергичной работе: команды достигают состояния потока, когда мировоззрение отдельных разработчиков также соответствует этому состоянию. Самоподобие можно найти и в планировании. Разработчик планирует тесты, затем пишет код, а команда планирует функции и выполняет недельный цикл, она также определяет тематику работы и начинает квартальный цикл.

Так нужно ли искать самоподобие в каждом аспекте вашей работы? Конечно, нет, это просто один из принципов. Но это поможет выявить закономерности, которые дадут лучшее понимание того, как именно идет работа над вашим проектом. И любой другой принцип делает то же самое. Так что не отмахивайтесь от них — это ценные инструменты, которые помогут понять методологию. Если они кажутся лишними, то постарайтесь подумать о том, какое отношение они имеют к работе, которую вы выполняете. Это поможет глубже воспринять XP.

Я читал в интернете, что разработка через тестирование умерла. Это правда?

Разработка через тестирование — это инструмент. Говорить, что он умер, так же нелепо, как сообщить о смерти отверток. Многие команды используют разработку через тестирование каждый день и находят ее успешной. До тех пор, пока команды используют эту методику, она остается живой и активной.

Однако можно обсуждать этот вопрос на более глубоком уровне. Одна из причин, по которой разработчики (особенно имеющие большой опыт использования Agile и XP) поднимают этот вопрос, заключается в том, что существуют команды, воспринимающие TDD не только как инструмент, но и как разновидность верования. Они сосредотачиваются на тестах как на конечной цели, а не как на средстве для создания лучшего программного обеспечения. Такие команды попадают в платформенную ловушку, затрачивая много времени на создание обширных и сложных платформ для тестирования. Разработка через тестирование предполагает помощь команде в сохранении простого исходного кода. Но когда он становится сложным, это значит, что-то пошло не так*.

* Считаете ли вы, что создаете сложные модульные тесты? Добиваетесь ли вы 100%-ного тестового покрытия искусственно, придумывая тесты на все случаи жизни или очень сложные объекты? Получаете ли вы в итоге трудноизменяемые модульные тесты? Есть ли у вас тесты, закомментированные из-за того, что кому-то потребовалось внести изменения, «сломавшие» тест, а исправить его он не смог? Если вы ответили утвердительно на любой из этих вопросов, то, вероятно, создаете модульные тесты, которые усложняют ваш код, вместо того чтобы его упрощать.

Те, кто заявляет, что «TDD умерла», считают, будто XP-практики (особенно TDD и парное программирование) создают значительное эмоциональное сопротивление у команд, которые недостаточно знакомы с ними.

Команда, использующая XP, не считает своей целью написание модульных тестов, так же как водопадные команды не рассматривают в качестве цели создание спецификаций. Главное — собрать работающее программное обеспечение, а TDD и спецификации лишь средства для достижения этой цели.

Есть еще один аспект, который поможет вам лучше разобраться в этом споре.

В главах 6 и 7 мы использовали термины «разработка через тестирование» и «разработка с опережающим написанием тестов» как синонимы. В течение многих лет их именно так и рассматривали, и мы решили, что для понимания концепции это упрощение было допустимо. Но некоторые люди рассматривают разработку с опережающим написанием тестов лишь как создание сначала тестов, а потом кода, в то время как TDD подразумевает более широкий подход к проектированию.

Надо ли писать тесты первыми, чтобы получить хорошо спроектированный код? Конечно, нет — многие команды делают прекрасный код без использования TDD. Но является ли TDD ценным инструментом для сборки хорошо продуманного кода? Безусловно. Если вы еще не применяли TDD, то без труда придумаете *гипотетические* причины не делать этого («Мне нравится делать наброски того, что я создаю, и видеть, как оно скрепляется вместе. В противном случае мне незачем писать тесты, поэтому ясно, что при применении TDD мне придется идти на компромисс!»)

Но дело в том, что многие команды используют TDD в строгом соответствии с методикой*, а многие из этих гипотетических причин исчезают, когда вы делаете серьезные попытки применить эту практику.

Полезно заглянуть в статью, с которой началась большая дискуссия на тему, мертва ли TDD, и не пропустить ответ Кента Бека**, опубликованный в Facebook. В нем он подробно объясняет, как TDD помогает решать проблемы программирования и лучше проектировать архитектуру.

* В том числе и мы, поскольку выработали привычку всегда использовать TDD при написании кода.

** Очень трудно не заметить сарказма в сообщении Кента Бека, опубликованном в Facebook. Объяснение этой шутки делает ее несмешной. Но дабы быть правильно понятым, Кент рассказывает, что раз TDD мертва, то он будет искать ему замену, которая была бы не хуже.

Покойся с миром, TDD

29 апреля 2014 года, 11:10

DHH послал TDD на свалку истории <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>. Мне грустно прежде всего не потому, что я спас TDD от попадания на свалку истории, а потому, что теперь нужны новые методы, чтобы я мог решить многие проблемы текущего программирования.

- Чрезмерное проектирование. У меня есть склонность «забираться» в функциональность, о которой я «знаю», что она «понадобится». Превращения одного теста из красного в зеленый (параллельно со списком будущих тестов) мне достаточно, чтобы избавиться от этого «копания». Мне нужно найти новый способ, чтобы оставаться сосредоточенным.
- Обратная связь по API. Мне нужно найти новый способ, чтобы получать быструю обратную связь о моих решениях, принятых по API.
- Логические ошибки. Мне нужно найти новый способ, чтобы ловить эти надоедливые ошибки, которые я так часто делаю.
- Документация. Мне нужно найти новый способ сообщать коллегам, каким я предполагаю использование API, и записывать мои мысли во время разработки.
- Ощущение подавленности. Я действительно буду скучать по тем временам, когда, применяя TDD, я, написав тест, мог представить реализацию, даже если поначалу у меня не было никаких мыслей по этому поводу. Мне нужно найти новый способ, чтобы сделать следующий шаг к вершине.
- Размышление о том, как отделить интерфейс от реализации. У меня есть склонность загрязнять решения по дизайну API его предположительными реализациями. Мне нужно найти новый способ разделения двух уровней мышления, обеспечивая оперативную обратную связь между ними.
- Соглашение. Мне нужно найти новый путь, чтобы точно передавать свои мысли партнерам по программированию о тех проблемах, которые я решаю.
- Тревога. Скорее всего, я буду больше всего скучать по возможности, которую дает мне мгновенная кнопка TDD «Все хорошо?».

Я уверен, что найду другие способы решить эти проблемы. Со временем. Боль исчезнет. Прощай, TDD, старый друг.

(Facebook, страничка Кента Бека)

Так каков же будет вердикт? Одни команды пробовали использовать разработку через тестирование и нашли ее очень полезной. Другие считают, что ее трудно внедрить. Самый лучший способ решить это для себя — попробовать. Если вы попробуете — а мы очень рекомендуем это сделать, — то для получения хороших результатов, прежде всего, нужно быть очень осторожными, чтобы не попасть в платформенную ловушку во время создания модульных тестов.

Кент Бек отвечал на эссе Давида Хейнемейера Ханссона (известного в сообществе Ruby как DHH), создателя Ruby on Rails и одного из идеологов agile-сообщества, озаглавленное так: «TDD умерла. Да здравствует тестирование!»*. DHH смог идентифицировать некоторые важные признаки модульных тестов, которые попали в платформенную ловушку.

Первоочередное тестирование модулей приводит к чрезмерно сложной сети промежуточных объектов и косвенным обращениям в запросе, чтобы избежать их медленной работы. Как попадание в базу данных. Или файловый ввод-вывод. Или тестирование в браузере всей системы. Это породило некоторых архитектурных монстров. Густые джунгли сервисных объектов, паттернов команд и много чего еще.

Я редко разрабатываю модульные тесты в традиционном смысле этого слова, где все зависимости являются смешными и тысячи тестов можно выполнить в считанные секунды.

Вам это описание знакомо? Должно быть, поскольку оно очень похоже на примеры кода «с душком», о котором вы узнали в этой главе. Модульные тесты подвержены точно таким же проблемам сложности, какие могут повлиять на любой код. Если вы поймаете себя на создании очень сложных модульных тестов, то вполне возможно, что вы попали в такую же ловушку, которую так хорошо сформулировал DHH.

DHH также отмечает важную особенность TDD.

Это было не так. Когда я впервые открыл для себя TDD, это было похоже на вежливое приглашение в лучший мир написания программного обеспечения.

Разум заставляет вас применять практику тестирования там, где тестирования никогда не было. TDD открыла для меня спокойствие, вызываемое хорошо протестированным кодом,

* David Heinemeier Hansson. TDD is dead. Long live testing (23 апреля 2014 года).

и блаженство, с которым я мог вносить изменения в программное обеспечение.

Предваряющее тестирование было замечательным курсом тренировок, который научил меня думать о тестировании на более глубоком уровне, но который я быстро завершил.

Это отличный повод, чтобы попробовать TDD. Это поможет вам думать о тестировании на более глубоком уровне, как это делает Кент Бек в упомянутом сообщении на Facebook. Единственный ли это способ для реализации перечисленных в сообщении приемов? Нет. Но он очень эффективен и поэтому является очень важной частью XP.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с командой).

- Если вы разработчик, то попробуйте сделать рефакторинг своего кода. Возможно, вы уже используете IDE, которая имеет встроенные инструменты рефакторинга. Сможете ли вы найти простой способ изменить структуру кода, с которым вы работаете сейчас, упростив его, но не меняя поведения?
- Посетите сайт WikiWikiWeb* и прочитайте страницу о коде «с духом». Можете ли вы найти подобный код у себя? Если нет, то всмотритесь внимательнее.
- Легко ли проверить ваш исходный код из системы управления версиями и собрать его? Если вам необходимо выполнить много различных действий, то попробуйте найти способ автоматизировать этот процесс при помощи скрипта сборки. Каждый шаг в сторону упрощения проверки и сборки делает вашу жизнь проще в долгосрочной перспективе.
- Вы уже пробовали разработку через тестирование? Нет? Попробуйте. Возьмите пользовательскую историю или функцию, которую вас просили создать. Перед тем как приступить к написанию кода, придумайте один-два теста для того модуля, который вы собираетесь написать.

* <http://wiki.c2.com/?CodeSmell>

Вам не нужно увлекаться полным набором тестов. Достаточно одного или двух, а затем соберите код, который пройдет их.



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше о простоте, инкрементальной архитектуре и других целостных практиках в книге Кента Бека и Синтии Андрес *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 2004).
- Можно узнать больше о рефакторинге в книге Мартина Фаулера, Кента Бека, Джона Бранта и Уильяма Апдайка «Рефакторинг. Улучшение существующего кода» (М. : Символ-Плюс, 2008).
- Узнайте больше о потоке и о том, как создавать более энергичную рабочую среду, в книге Тома Демарко и Тимоти Листера «Человеческий фактор. Успешные проекты и команды» (М. : Символ-Плюс, 2014).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Попадает ли ваша команда в платформенную ловушку? Пробует ли она решать большие абстрактные проблемы вместо маленьких и конкретных? Помогите ей осознать, что это сложное мышление. Изменить его на простое — один из самых важных способов, ведущих команду к принятию XP.
- Поговорите с командой о YAGNI («Вам это не понадобится»). Помогите ей найти хотя бы один фрагмент кода, который на самом деле не нужен. Многие команды собирают целые функции, никогда никем не востребованные и не используемые. А у вашей команды есть такой опыт?

- Выясните, что вызывает чувство дискомфорта у членов команды во время применения практики разработки через тестирование или парного программирования, и помогите им устранить это ощущение. Избегайте длительных философских дебатов о достоинствах практики, вместо этого сосредоточьтесь на поиске конкретных проблем команды и практиках, которые помогут их решить. Опыт — лучший способ преодоления защитной реакции и дискомфорта. Иногда для этого сотруднику требуется час или два.
- Некоторые практики — особенно разработка через тестирование — кажутся привлекательными только одному члену команды. Он исполнен энтузиазма, но его благие намерения могут восприниматься коллегами как издевательство или фанатизм. Напомните ему, что для получения нового навыка требуются время и усилия, но пока никто в команде этим не занимался. Терпение — это то, чему научить труднее всего, поэтому будьте терпеливы.

Lean, ликвидация потерь и целостная картина

*Lean — это образ мышления, ментальная модель того,
как устроен мир.*

Том и Мэри Поппендик. The Lean mindset: ask the right questions

Из предыдущих глав вы узнали о Scrum и XP. Каждый из этих подходов имеет практики, которые вы можете внедрить у себя, а также ценности и принципы, способные помочь всем членам группы достичь эффективного мышления. Можете считать себя членом команды, использующей Scrum, если вы присутствуете на ежедневных Scrum-митингах, используете спринты и работаете с владельцем продукта и Scrum-мастером. То же самое касается XP: если вы занимаетесь беспощадным рефакторингом, ведете разработку через тестирование, исповедуете непрерывную интеграцию и создаете инкрементальную архитектуру, то ваша команда использует XP.

Но что объединяет XP и Scrum? Не понимая их ценностей и принципов, вы будете в итоге выполнять множество суетливых движений ради сомнительного результата «лучше-чем-ничего». Кен Швабер убежден, что, не проникнувшись коллективной ответственностью и волей к самоорганизации, невозможно получить Scrum, и ее ценности помогают вам это понять. Так же и в случае с XP: без понимания таких его ценностей, как простота и энергичная работа, вы станете рассматривать эту практику как список мероприятий — по сути ваша команда не примет изменений и вы останетесь со сложным программным обеспечением, которое трудно поддерживать.

Lean (бережливость) — это другое. В отличие от Scrum и XP, в него не входит набор практик. Этот тип мышления имеет свои ценности и принципы (в lean-терминологии они называются «инструменты мышления»), иногда

его называют «бережливое мышление». Термин «бережливое» применяется в производстве в течение многих десятилетий. Он был приспособлен для разработки программного обеспечения Томом и Мэри Поппендик в первом десятилетии двадцать первого века. Мы будем использовать термин Lean (на английском языке, с заглавной буквы), чтобы обозначить эту адаптацию lean-идей применительно к гибкой разработке программного обеспечения.

В этой главе вы узнаете о Lean, ценностях, которые помогут вам постичь бережливое мышление, и инструментах, способных помочь вашей команде выявлять потери и удалять их, а также увидеть целиком систему, которую можно использовать для создания программного обеспечения.

Бережливое мышление

Lean — это образ мыслей. Давать имя способу мышления — интересная и очень полезная идея.

Вы уже знаете, что для эффективного внедрения Scrum команда должна иметь определенный настрой, а также ценности — приверженность, сосредоточенность, открытость, уважение и мужество, которые помогают ей включить именно такое мышление. XP, в свою очередь, также требует определенного образа мыслей, и XP-команды тоже используют устойчивый набор ценностей: простоту, коммуникацию, обратную связь, уважение и мужество.

Неудивительно, что и Lean приходит со своим набором ценностей, а команда, стремящаяся принять бережливое мышление, начинает именно с них.

Итак, перечислим ценности Lean.

Ликвидируйте потери

Выявите работы, выполняемые вами, но не создающие ценность, и избавьтесь от них.

Усиьте обучение

Используйте обратную связь, чтобы улучшить свою работу.

Принимайте решения как можно позже

Принимайте все важные решения по проекту, обладая максимумом информации о нем, — то есть в последний ответственный момент.

Доставляйте ценность как можно раньше

Проанализируйте реальную стоимость задержек и минимизируйте ее при помощи систем вытягивания и очередей.

Раскрепостите вашу команду

Сформируйте целенаправленную и эффективную рабочую среду и объедините энергичных людей.

Добейтесь целостности

Создавайте программное обеспечение, интуитивно понятное для пользователей и работающее сообразно их ожиданиям.

Следите за общей картиной

Постарайтесь досконально разобраться в той работе, которая выполняется у вас в проекте. Примените правильные методы измерения, чтобы убедиться, что вы действительно видите даже мельчайшие детали.

Каждая ценность обладает инструментами мышления, способными помочь применить эти ценности в реальных ситуациях. Любой из этих инструментов — примерный аналог одного из принципов XP: они используются одинаковым образом. Объясняя эти ценности, мы покажем связанные с ними инструменты мышления и то, как их использовать, чтобы помочь команде принять бережливое мышление.

ВЫ УЖЕ ПОНИМАЕТЕ МНОГИЕ ИЗ ЭТИХ ЦЕННОСТЕЙ

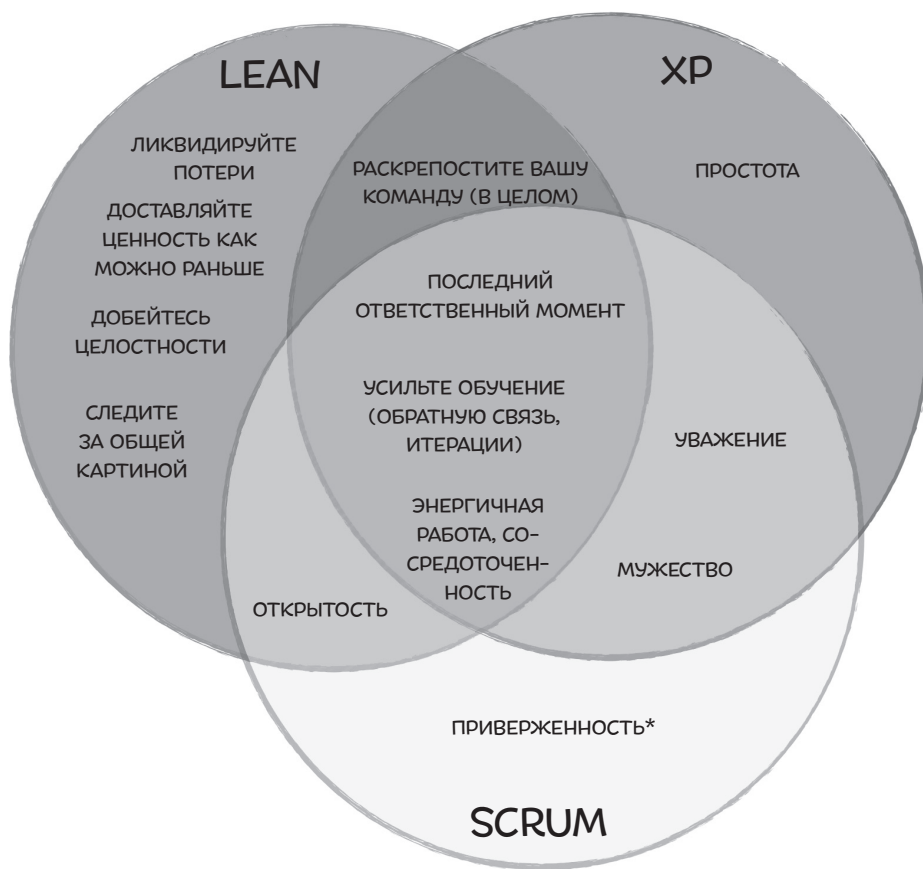
Компания получит то, что она ценит, и Agile-манифест помогает переключить восприятие ценности с процесса на людей, с документации на код, от договоров к сотрудничеству и от планов к действиям.

Том и Мэри Поппендик. Lean Software Development: An Agile Toolkit

Удивляет ли вас, что вы уже столкнулись со многими ценностями и инструментами мышления Lean? Надеемся, что нет. Lean — это важная часть Agile. Когда Том и Мэри Поппендик занимались адаптацией идей бережливого производства к разработке программного обеспечения, они активно заимствовали их из других частей Agile, включая XP. Но еще важнее то, что производственные идеи, которые они использовали, были важной частью

разработки ПО и управления качеством в течение десятилетий. Кен Швабер также опирался на множество подобных идей при создании Scrum. Напомним, в частности, что ежедневный Scrum-митинг — это аналог формализованной проверки состояния дел.

Поскольку данная книга описывает несколько гибких методологий, мы уже рассмотрели некоторые важные части Lean. Вновь коснемся их — и единственная причина, по которой они занимают относительно немного места в этой главе, состоит в том, что вы уже достаточно глубоко их изучили. Это бесспорно ключевые элементы бережливого мышления.



* ПРИВЕРЖЕННОСТЬ (ОТВЕТСТВЕННОСТЬ) — ВАЖНАЯ ЧАСТЬ XP И LEAN, НО SCRUM ПРЯМО НАЗЫВАЕТ ЭТО СВОЕЙ ЦЕННОСТЬЮ

Рис. 8.1. Вы уже немало знаете о Lean, потому что есть много совпадений между ценностями бережливого мышления, Scrum и XP

Еще раз взгляните на список ценностей Lean. Одна из них — принимать решения как можно позже — наверняка покажется вам очень знакомой. Scrum и XP в значительной степени опираются на похожую идею. Scrum-команды применяют ее к планированию. Команды XP — к проектированию систем и разработке кода. Lean-практики делают это тоже. По сути эта ценность использует ту же концепцию, называемую *последним ответственным моментом*.

Еще одна знакомая вам ценность — **«усильте обучение»**. Вам уже известны два основных инструмента ее мышления — **обратная связь** и **итерации**. С данными концепциями мы встречались при знакомстве со Scrum и XP. Эта ценность имеет также два других инструмента мышления: *синхронизацию* и *развитие на основе установок*. Синхронизация очень похожа на непрерывную интеграцию и коллективную собственность в XP, а развитие на основе установок мы рассмотрим чуть ниже.

Вы узнали также о ценности, называющейся **«раскрепощение команды»**, и используемых в ней приемах мышления: **самоопределении, мотивации, лидерстве** и **экспертизе**. Эти идеи почти идентичны тем, что лежат в основе XP-практик команды как целого и ее энергичной работы — в стремлении избежать работы допоздна, оказывающей столь серьезное отрицательное воздействие на программное обеспечение. В главе 4 говорилось, что команды также дорожат этим как частью такой scrum-ценности, как **сосредоточенность**. Разработчики, сами управляющие своей жизнью, создадут более качественное программное обеспечение. Позже вы узнаете, что важный аспект видения общей картины состоит в необходимости делиться информацией о проекте со всеми, включая руководителей. Именно поэтому scrum-команды так ценят **открытость**.

ОБЯЗАТЕЛЬСТВА, ВАРИАНТНОЕ МЫШЛЕНИЕ И РАЗРАБОТКА НА ОСНОВЕ УСТАНОВОК

Здесь написано то, о чем уже говорилось в главе 4, но нелишне напомнить.

Обязывают не планы, а наши обязательства. План — это просто удобный способ их фиксации. Обязательства даются людьми и документируются в планах проекта. А это не просто бумажка. Это зафиксированная информация, и все держат ее в голове.

Так что же это значит? Когда вы создаете план, то сначала принимаете окончательное решение и лишь потом записываете его.

Еще раз посмотрите на рисунок 8.1. В сноске к нему сказано, что приверженность (ответственность) — это важная часть XP и Lean, а Scrum прямо называет их своей ценностью. При этом Lean идет дальше, уточняя идею обязательств добавлением некоторых моментов. Это, во-первых, **вариантное мышление**, или понимание разницы между тем, за что вы отвечаете, и тем, что вы имеете право (но не обязаны) делать. А во-вторых, **развитие на основе установок**. Это такое выполнение проекта, при котором команда может прорабатывать сразу несколько вариантов его развития, чтобы сравнивать их и выбирать оптимальный.

SCRUM-КОМАНДЫ ОБЯЗУЮТСЯ ДОСТИГАТЬ ЦЕННЫХ РЕЗУЛЬТАТОВ, НО ОСТАВЛЯЮТ ЗА СОБОЙ ВЫБОР, КАК ЭТО СДЕЛАТЬ

Команды должны брать на себя обязательства, верно?*

Например, когда одного из членов scrum-команды просят о поставке конкретного компонента через считанные месяцы, он не может сказать пользователям и менеджерам: «Это Scrum, поэтому я не готов планировать что-либо за пределами текущего спринта. Поговорим об этом через пару месяцев». Вместо этого scrum-команда использует бэклог продукта и поэтому может работать с бизнесом, понимая, что является ценным для заказчиков. Она берет на себя обязательство поставлять самое ценное ПО в конце текущего спринта, в следующем и в каждом новом в ближайшие несколько месяцев.

Сравните это с административно-командным стилем управления, когда менеджер проекта пишет очень подробный план, который обязывает людей работать над конкретными задачами в течение следующего месяца. Если команда берет на себя обязательства поставить определенную функцию, для которой требуется большой объем детального планирования, то у каждого участника возникает иллюзия, будто все находится под контролем. Руководитель проекта может указать конкретные задачи, которые разработчик будет выполнять, скажем, через четыре недели во вторник в 10:30. У людей появляется ощущение, что команда уже продумала все варианты и стремится пойти наилучшим путем. Проблема состоит в том, что *в действительности* они ни за что не отвечают. Нет достаточного объема информации, чтобы сделать это. Есть только одна вещь, не вызывающая сомнений: план,

* Существует великолепный ресурс для изучения вариантного мышления. Это образный роман Олава Маасена, Криса Маттса и Криса Гири Commitment. Можете поблагодарить Дэвида Андерсена за то, что он порекомендовал его нам.

согласно которому программист будет делать что-то конкретное во вторник в 10:30 четыре недели спустя. Но это почти наверняка неверно*.

Поэтому вместо подробного планирования каждого задания и принятия на себя ответственности за эти детализированные задачи scrum-команды самоорганизуются и коллективно обязуются предоставлять ценность. То есть никто не требует, чтобы конкретный разработчик выполнил определенную задачу, скажем, во вторник в 10:30 утра через четыре недели. Принятие решений оставляют на более поздний (последний) ответственный момент (возможно, ежедневную scrum-планерку через четыре недели).

Однако scrum-команда не принимает на себя обязательство поставить определенные задачи бэклога спринта до его начала. И даже после начала спринта владелец продукта может изъять задачи из спринта, если они больше не имеют смысла. Формируя меньшее обязательство — поставить ценность — вместо завершения конкретных невыполненных работ, она оставляет эти решения и обязательства на последний ответственный момент.

Когда спринт стартует, элементы его бэклога могут восприниматься командой как обязательства, потому что они обсуждались в ходе планирования спринта и представлены на доске задач. Но это не обязательства, а **варианты**. Вы знаете это, так как владелец продукта может удалить их из спринта. И если команда узнает ближе к концу спринта, что они не будут выполнены в срок, то она вытолкнет их в следующий спринт. В этом одна из причин хорошей работы Scrum: мы отделяем истинное обязательство (поставку ценного работающего программного обеспечения в конце спринта) от опциональных вариантов (предоставления конкретного функционала к определенной дате).

Заставить людей думать о вариантах вместо обязательств сложно, потому что не всегда легко понять, что означает обязательство. Никто не любит неожиданно брать на себя обязательства. Большинство из нас бывали на таких собраниях, где руководитель требует от члена команды назвать дату и тот нервно ерзает в кресле и пытается уйти от ответа. Так бывает, когда руководство считает, что проект «погорел» по вине команды, не выполнившей взятое на себя обязательство. Нервная реакция заставляет его заняться микроменеджментом, требуя от всех членов команды по отдельности

* Если у вас есть опыт работы в сфере финансов, то вы найдете много общего с этим миром, когда дело доходит до вариантного мышления. Трейдеры и портфельные менеджеры продают и покупают опционы или инструменты, что дает одной стороне опцион на покупку или продажу базовой акции или товара (например, нефти или пшеницы) на определенную дату по определенной цене. Это дает им возможность придерживаться стратегии покупать акции или товары без необходимости полностью нести обязательства обладания ими. Другими словами, они могут держать свои опционы открытыми.

принятия обязательств в отношении множества краткосрочных задач. В такой обстановке разработчики не спешат брать на себя ответственность.

Как проекты доходят до точки, в которой боссы и менеджеры проекта теряют доверие к команде, неоднократно проваливавшей свои обязательства? Это происходит потому, что, хотя отдельные люди и могут уклоняться от ответственности (особенно под прессингом руководства), *команды имеют склонность к ее чрезмерной фиксации*. Иногда это связано с желанием погеройствовать — программист обещает больше, чем может сделать. А порой потому, что ответственность вознаграждается: это характерно для тех, кто привык обещать, а потом как ни в чем не бывало извиняться за непредвиденные обстоятельства, помешавшие проекту. («Мы никак не могли это предугадать!») В компаниях, где привыкли обвинять и спасать свою шкуру (CYA), подход «пообещать и извиниться», конечно, не обеспечивает эффективные поставки программного обеспечения, но зато дает возможность получать высокую зарплату и сохранять свою должность.

Повторим основную мысль: руководители склонны требовать принятия обязательств, а команды часто их завывают.

ИНКРЕМЕНТАЛЬНАЯ АРХИТЕКТУРА, РАЗРАБОТКА НА ОСНОВЕ УСТАНОВОК И ДРУГИЕ СПОСОБЫ ДАТЬ КОМАНДЕ ВОЗМОЖНОСТЬ ВЫБРАТЬ ИЗ НЕСКОЛЬКИХ ВАРИАНТОВ

Задача на scrum-доске проходит через три состояния: «сделать», «в процессе» и «сделано». Если задача находится в состоянии «сделать», то это все-таки опция, а не обязательство. И даже в ходе выполнения задания команда может решить на ежедневном scrum-митинге изменить направление работы, если это имеет смысл. В главе 4 мы говорили о том, что scrum-команда не тратит много времени на моделирование и отслеживание зависимостей между задачами, потому что они в основном полезны для составления проектного плана. Вместо этого команда может ежедневно добавлять и удалять задачи на доске задач во время scrum-митинга. Но эти задания — варианты, а не обязательства: они не имеют пока сроков и нет такого понятия, как «поздняя» задача, которая приводит оставшуюся часть проекта к срыву дедлайна. Это поощряет команду к альтернативному мышлению, потому что она привержена только цели и может в любое время изменить задачи, чтобы достичь ее более эффективным способом, основываясь на какой-либо вновь обнаруженной информации.

Приведем пример того, как scrum-команда использует вариантное мышление. Скажем, во время планирования спринта она разбила историю

на четыре задачи, основанные на предположениях, сделанных о хранении данных в базе, и включающие задачу проектирования базы данных, которая, вероятно, должна быть выполнена ее администратором (Database administrator, DBA). Спустя две недели разработчик обнаруживает, что нужные данные уже есть в базе в другом формате. И для него гораздо полезнее самому создать объект или сервис, чем собирать данные в прежнем формате, который может использоваться остальной системой. Для scrum-команды это хорошая новость! Теперь она удалит задачу с доски задач на следующем ежедневном scrum-митинге, освобождая время в спринте, чтобы добавить еще один пункт бэклога или прояснить некоторые технические вопросы.

В то же время для традиционной водопадной команды тот факт, что задача должна быть выполнена кем-то другим, способен вызвать осложнения: теперь менеджер проекта вынужден перераспределять ресурсы, потому что для этой задачи уже не нужен администратор, а это может разорвать множество зависимостей внутри проекта и затронуть другие проекты. Если план проекта содержал много обязательств, которые теперь должны быть пересмотрены, то налицо проблема. Менеджеру проекта захочется вернуться к команде и потребовать, чтобы она в любом случае использовала базу данных. Или технический архитектор предполагал, что команда собиралась использовать его архитектуру, зависящую от базы данных, а теперь это обязательство нарушается.

Многие разработчики ощущают, что их попросили пойти на ненужные или даже вредные технические компромиссы некие руководители, недостаточно разбирающиеся в проекте, — например, менеджер проекта или технический архитектор. С точки зрения программиста команду просят нарушить целостность программного обеспечения. А с позиции менеджера проекта или архитектора все выглядит так, будто разработчик нарушает обязательства. У каждого есть причина обвинить другого.

Проблема заключается в том, что разработка дизайна не должна стоять на первом месте — если его рассматривают как вариант, то команде легче создать лучшее ПО, а руководителю проекта и архитектору не придется менять свои планы и чувствовать себя ущемленными.

Но как добиться того, чтобы и руководство, и члены команды брали меньше обязательств и воспринимали большее количество «окончательных» решений в качестве вариантов?

XP дает ясный ответ на этот вопрос — применять инкрементальное (пошаговое) проектирование. Когда команда создает простые компоненты и каждый из них выполняет что-то одно, это позволяет комбинировать их множеством различных способов. Рефакторинг и разработка через

тестирование помогают поддерживать эти компоненты в максимально независимом состоянии.

Иными словами, отделенные, независимые компоненты создают варианты.

При возникновении новой идеи или обнаружении нового требования, если компоненты крупные и между ними много зависимостей, то команда потратит большую часть своих усилий на разделение этого кода путем «стрельбы дробью». Но команда XP, использующая инкрементальную архитектуру, сохраняет свои варианты открытыми. Она добавит только минимальный код, который должен удовлетворять новым требованиям, проведет рефакторинг и продолжит использовать разработку через тестирование. Команда прибавит минимум зависимостей, что позволит ей в дальнейшем сохранить максимум открытых вариантов.

Так XP и scrum-команды практикуют вариантное мышление в том, как они планируют свою работу и проектируют ПО. Но есть ли что-то такое, что команда может сделать, чтобы явно создавать и применять варианты в своем проекте?

Да. Когда команда практикует **разработку на основе установок**, она целенаправленно тратит время на обсуждение своих возможностей и меняет планы на сегодня, чтобы иметь больше возможностей в будущем. Дополнительная работа позволит команде предложить несколько вариантов, и люди надеются, что она окупит себя, давая им максимум информации, что позволит принять верное решение позже.

Допустим, в команде есть бизнес-проблема, которая имеет ряд возможных технических решений, — например, наша scrum-команда обнаружила во время спринта, что она не нуждается в DBA для внесения изменений в базу данных. Как быть, если команда не знает, какое решение лучше — объектно ориентированное или с использованием базы данных? Это очень распространенная ситуация в проектах разработки программного обеспечения. Часто неизвестно, какое из двух решений лучше, пока команда по крайней мере *не начнет* создавать оба.

Разработчики, занимаясь непростыми проблемами, не очень понимают, что участвуют в их решении, пока не измажут о них свои собственные руки. Каждый программист наверняка помнит случай, когда начинали работать над «простой» проблемой, а потом выяснялось, что она сложная. Это характерно для команд по разработке программного обеспечения. Порой, взяв на себя обязательство по решению простой проблемы, команда обнаруживает, что все гораздо сложнее, чем ожидалось. Тогда разработчики оказываются перед выбором: разорвать обязательство или поставить «костыль» (некачественный, неэлегантный код) и нарастить техническую задолженность.

Разработка на основе установок поможет scrum-команде справиться с ситуацией, когда непонятно, какое из двух решений сработает лучше. Вместо того чтобы создавать модель данных или объектно ориентированное решение, команда добавляет обе задачи на доску задач, чтобы отработать два варианта. Возможно, это выглядит расточительством, но на самом деле экономит много сил в долгосрочной перспективе. Если один из этих подходов приводит к лучшему решению, а другой — к «костылям» с большим количеством технических долгов, то для команды выгодно добиваться реализации обоих вариантов — по крайней мере в течение времени, требующегося разработчикам для продумывания решений по обоим возможным путям. Это даст максимум информации для принятия верного решения, и ответственный момент для него наступит после того, как будет потрачено время на отработку проблемы.

Другой пример разработки на основе установок — регулярно используемое командами А/Б-тестирование. Такая практика часто встречается при создании пользовательских интерфейсов и улучшении работы пользователей, ее с большим успехом реализовывали в Amazon, Microsoft и других компаниях. При А/Б-тестировании команда создает два или более решения — например, два различных макета или варианта решения для веб-интерфейса (А- и Б-макеты). Команда будет случайным образом передавать А- либо Б-вариант бета-тестерам — или, в некоторых проектах, реальным пользователям — и контролировать результаты тестирования и показатели успеха. Компании неоднократно убеждались: хотя такой подход требует больше времени и усилий, чтобы создать два комплексных решения, он хорошо окупается в долгосрочной перспективе, поскольку появляется возможность провести сравнительные замеры и доказать, что одно из решений успешнее другого. Более того, зачастую менее удачное решение тоже приносит пользу, например функции, которые разработчики позднее включают в конечный продукт.

Эти примеры показывают, как команды применяют разработку на основе установок и вариантное мышление. Но важнее другое: они демонстрируют, как идеи, уже знакомые вам по Scrum и XP, дают основу для изучения Lean и бережливого мышления.



Ключевые моменты

Lean (или бережливое мышление) — это название образа мыслей. Подобно другим agile-методам, оно имеет свои ценности, помогающие понять и принять его.

Lean — мышление, а не методика, поэтому в нем не представлены практики, которые помогут выполнять проекты.

Есть ценности, общие для бережливого мышления и более широкого круга agile-решений: как можно более позднее принятие решений (или принятие решений в последний ответственный момент) и усиление обучения (при помощи обратной связи и итерации).

Lean-ценность «расширение прав и возможностей команды» очень похожа на scrum-ценность «сосредоточенность» и XP-ценность «энергичная работа».

Вариантное мышление означает умение понимать разницу между вариантами и обязательствами, а также принимать такие решения по проекту, которые дадут множество вариантов в будущем.

Когда команда использует **разработку на основе установок**, она исследует более одного варианта решений за один раз и выбирает лучший — например, когда она выполняет А/Б-тестирование для проверки нескольких параметров пользовательского интерфейса на выборке пользователей, выбирает функции, работающие лучше всего.



Описание: команда, работающая над созданием приложения для камеры мобильного телефона в организации, купленной большой и многопрофильной интернет-компанией

Кэтрин — первый разработчик

Тимати — второй разработчик

Дэн — их руководитель

Акт I. И вот еще что...

— Ну, все. Встряхнитесь и работайте.

Кэтрин было неприятно слышать это от руководителя. Их встреча только что закончилась, и он был недоволен тем, как команда разрабатывает новую функцию, добавленную по его просьбе. Двое программистов столкнулись с нехваткой времени, но он не дал им его, тем самым поставив под удар сроки. Он не сказал, что их ждут неприятности, если они не сделают работу вовремя, но всем и так было ясно, что задержки — это плохо.

В прошлом году Кэтрин чувствовала себя гораздо счастливее. Она тогда работала в маленькой компании, имевшей всего одну команду программистов. Все вместе они трудились над созданием приложения для мобильного телефона, которое обеспечивало дополнительными функциями его камеру.

Команда была небольшой, но весьма инновационной, и все чувствовали, как им хорошо работается вместе.

Их труд высоко ценили коллеги по отрасли. Поэтому казалось естественным, что крупный интернет-гигант выразил желание купить компанию. Кэтрин и ее товарищи по команде были очень рады, потому что слышали об этой компании много хорошего: там царит творческая атмосфера, в холодильниках полно прохладительных напитков, у сотрудников гибкий график работы, есть также много других льгот. Когда сделка наконец была завершена, все они получили большие бонусы (им оплатили студенческие кредиты!) и команда переехала в новый красивый офис в центре города.

— Почему все так плохо закончилось, Тимати? — спросила Кэтрин, выходя из кабинета руководителя. — Ведь было так здорово. Что случилось?

Тимати был программистом и работал в команде почти столько же времени, сколько и Кэтрин.

— Я не знаю, — сказал он. — Похоже, все тянется вдвое дольше, чем следует.

— Я не против работать больше, — вздохнула Кэтрин. — Но возникает ощущение, что, сколько бы мы ни напрягались, мы все равно опаздываем.

— Да, — согласился Тимати. — И нам никогда не выбраться из этого замкнутого круга.

— Кэти, Тим! У вас есть минутка?

Кэтрин и Тимати переглянулись.

— Это звучит не слишком обнадеживающе, — сказала она. Дэн, руководитель, снова позвал их в свой кабинет.

— Я только что разговаривал по телефону с одним из топ-менеджеров социальной сети нашей компании, и у меня отличные новости. — Так уж повелось, что все новые запросы к команде преподносились как «отличные новости!» — У него была идея по интеграции данных социальной сети с нашим приложением для камеры. Я хочу, чтобы вы взялись за это немедленно. Я пришлю письмо со списком функций, которые мы должны добавить, и обязательно включите их в следующий спринт.

Тимати сказал:

— Хорошо. Мы на середине спринта. Что нужно удалить?

Кэтрин бросила на него быстрый взгляд. Она знала, что добром это не кончится.

Во время разговора Дэн смотрел в свой экран. Он медленно поднял глаза на Тимати.

— Вы думаете, у вас недостаточно ресурсов, чтобы сделать работу в этом цикле?

— Ну, э-э, у нас есть четыре другие функции, и одна из них уже закончена... — забормотал Тимати.

Дэн оборвал его:

— Это отговорки. Все эти дни я не видел никого, кто бы задерживался допоздна или слишком перенапрягался. Я пришел в шесть утра — офис был пуст. Ты говоришь, что мы не можем уделить этому чуть больше времени? Это не слишком большая работа. Я мог бы и сам потихоньку писать код.

Кэтрин поняла, к чему он клонит. Она уже слышала нечто подобное, и в прошлый раз все закончилось не слишком хорошо. Буквально только что она превратила в программное обеспечение точно такие же «хорошие новости» в виде запроса от Дэна, пропуская модульные тесты и накапливая технический долг. И пользователи это заметили — если прежде их отзывы были четырех- и пятизвездочными, то теперь они упали до двух-трех звезд.

— Попросите всех сконцентрироваться и выполнить задание. Это не займет много времени, зато мы заслужим признательность всей компании.

Создание героев и магическое мышление

Среди руководителей бытует ошибочное мнение, будто, установив высокую планку, можно мотивировать сотрудников трудиться с утроенной энергией для достижения заданной цели. Если дать каждому члену команды большие цели и сжатые сроки, то все будут стремиться оказаться на высоте. Есть ощущение, что такой «грубо индивидуалистический» подход к построению команд позволит устранить бюрократию. Каждый человек уполномочен решать свои проблемы, и в результате вся команда становится высокоэффективной.

Такой индивидуалист-менеджер поощряет практику «геройства». Программист, который работает допоздна, по выходным и создает для команды сложные решения, получает наивысшее признание и попадает в число лучших сотрудников. Такой «герой» поднимается на вершину не благодаря командной работе, не потому, что делает лучший продукт или совершенствует деятельность всей команды, а лишь из-за сидения в офисе допоздна и в выходные.

Это непродуктивный, хотя довольно традиционный подход. Но группы по-прежнему приходят к выводу, что такое «индивидуалистское» мышление способствует созданию плохого ПО. Почему?

Эффективное программное обеспечение получается тогда, когда команды работают вместе. Мы уже знакомы с подходами к командной работе

в Scrum (самоорганизация и коллективная ответственность) и XP (творческая и энергичная среда, где люди собираются вместе как целая команда). Многие agile-команды постоянно размышляют над тем, как улучшить взаимодействие и создать комфортные условия труда, помогающие разрабатывать качественное программное обеспечение.

Так почему же многие руководители собирают группы «сильных личностей», вместо того чтобы создавать реальные команды, в которых люди плодотворно сотрудничают?

Попробуйте поставить себя на место такого менеджера. Взгляните на команду как на своего рода «черный ящик», создающий программное обеспечение. Вы говорите программистам, что хотите получить, ждете несколько дней, и программное обеспечение появляется на свет. Если один из разработчиков готов трудиться допоздна и в выходные, то он сразу привлечет к себе ваше внимание своим энтузиазмом. Это подарок для вас как для руководителя: сваливайте всю работу на него, и она будет сделана. Затем воздайте этому человеку должное, и, возможно, другим придется начать работать таким же образом.

Похоже, чем сильнее ваше давление на команду, тем выше отдача от нее. Чем щедрее вы вознаграждаете «полуночников» и тех, кто может быстрее других продаться через хаос, тем больше программ они создают. А как вы оказываете влияние на людей? Вы убеждаетесь: они знают, что им всегда предстоит сделать еще больше работы, а то, что в настоящее время ими дорабатывается, должно быть как можно быстрее «вытолкнуто» к потребителю.

Важно проявлять терпимость к этому руководителю, даже если он создал абсолютно невыносимую атмосферу, где каждый член команды чувствует: некогда думать — надо действовать. Благодаря XP мы имеем возможность увидеть, как это заставляет команду разработчиков создавать плохо спроектированное программное обеспечение, которое трудно изменить. Но руководитель так не считает.

Такой менеджер использует *магическое мышление*.

Если есть такое мышление, то возможно все. Команда готова взять на себя проект любой величины. О каждом новом проекте говорится, что это «просто маленькая функция» или «ничего страшного, команда справится с этим», потому что группа умных и профессиональных разработчиков может добиться чего угодно. Неважно, сколько работы они выполнили на этой неделе. Наверняка на следующей они способны взвалить на себя еще больше. Менеджер всегда может возложить дополнительную задачу на плечи команды, и она обязательно будет сделана, не повлияв на какие-либо другие работы. Если считается вполне нормальным, когда команде

приходится пахать по 20 часов в сутки или несколько разработчиков проводят в офисе выходные, — то они стремятся выполнить свою работу. Действительно, как по волшебству.

Но дело тут не только в менеджерах с магическим мышлением. Здесь имеет место симбиоз с героем — герой готов работать сверхурочно, творить «чудеса» ради признания, лидерства и, возможно, более высокой зарплаты. Он становится эталоном, с которым сравнивают всех других членов команды. Руководитель не хочет разбираться в том, как фактически спроектировано и написано программное обеспечение или сколько плохо проработанных временных заплаток превратились в долгосрочные решения. Таким образом, главным мерилom того, кто является наиболее ценным членом команды, становится количество отработанных часов в неделю.

Магическое мышление воспринимается хорошо. Неважно, кто вы: менеджер, «мотивировавший» команду, или «герой», который очень много работал, чтобы получить готовое программное обеспечение, — вам кажется, что вы решили серьезную проблему. Но программа, разработанная под воздействием магического мышления, вызывает больше вопросов, чем решает их: технический долг накапливается, ПО никогда не достигает состояния «полностью завершено», а качество и тестирование не всегда являются «желательными». Слишком часто появляются ошибки, которые, как правило, обнаруживают пользователи уже после выпуска продукта. В итоге команда серьезно замедляется, потому что тратит основное время на исправление этих ошибок, поддержание плохого кода, а не на создание новых объектов.

Описывая Scrum и XP, мы уже говорили о том, насколько быстро может производиться лучшее программное обеспечение теми командами, которые уделяют достаточно времени выполнению работы (через ограниченные временные итерации), проявляют способность сосредоточиться на одной задаче и создают атмосферу, в которой члены команды помогают друг другу преодолевать препятствия. Разумеется, Agile не уживается с магическим мышлением или героями-разработчиками.

Так как же избежать магического мышления?

Это одна из главных целей Lean. Вместо того чтобы рассматривать коллектив как «черный ящик», бережливое мышление поможет точно понимать, чем именно занимается команда день за днем, чтобы создать программное обеспечение. Lean-мышление заставляет вас смотреть на команду, ясно видя, что происходит, в том числе прежде, чем команда начнет работать, и после поставки готового ПО — потому что магическое мышление происходит и там тоже. Бережливое мышление помогает отринуть ту «маленькую» неправду, которую руководство сообщает команде, менеджеры — друг

другу, а остальные участники — сами себе. Ведь именно эта ложь не дает создать лучшее программное обеспечение, на которое вы способны, и сделать это максимально быстро. Lean пытается избавить вас от этих фальшивок и помогает командам работать вместе, думая о том, как обеспечить реальную ценность, а не просто затратить усилия.

Ликвидируйте потери

Не всегда легко увидеть потери или признать, что вы и ваша команда тратите время на то, что не имеет значения для проекта. Lean-ценность под названием «**ликвидируйте потери**» говорит о выявлении деятельности, не добавляющей ценности, и о том, как ее устранить.

Большинство команд не думают над тем, как они создают программное обеспечение. У них есть некий стиль работы, который подхватывается новыми людьми, когда они присоединяются к команде. Мало кто из нас готов остановиться, чтобы проанализировать свою работу над ПО от концепции до выпуска. По сути, нам и в голову не приходит задумываться о том, как трудится команда. Все привыкли к тому, как это делается. Если вы обычно начинаете проект с большой спецификации, то попытка создать программное обеспечение без нее вызовет у людей удивление. Если все используют систему сборки, созданную три года назад, то вы примените ее и в следующем проекте.

Если вы работаете в команде, то ваша цель — получить готовое программное обеспечение, верно? Никому не приходит на ум остановиться и задуматься о смысле жизни или цели спецификаций.

При знакомстве с XP мы узнали, что, когда команды привыкают постоянно проводить рефакторинг своего кода, они получают гораздо более гибкие архитектуры. Та же идея применима к осмыслению деятельности команды: привыкнув постоянно проводить «рефакторинг» создания проекта, вы окажетесь в итоге с более гибкой командой с большими возможностями.

Итак, как вы проводите «рефакторинг» выполнения проекта?

Первый шаг рефакторинга кода — это поиск антипаттернов. В бережливом мышлении антипаттерны выполнения проектов называют **потерями**. В этом есть смысл: потерями называется все, что не помогает команде делать программное обеспечение лучше.

Вспомните свои недавние проекты. Было ли в вашей работе то, что не помогало им? Как насчет тех вещей, которых все ожидали, но не начинали делать и жалели об этом? Требовалось ли написать спецификации, которые никто никогда бы не прочел? Или вы получили абсолютно бесполезную

спецификацию? Возможно, вы собирались писать модульные тесты или делать обзоры кода, но почему-то они так и не были выполнены. Или вы сделали анализ кода, но только перед выходом — поэтому никто не задавал лишних вопросов, опасаясь задержать выпуск продукта, и все обнаруженные проблемы просто откладывались на потом?

Мы часто сталкиваемся с ситуацией, при которой значительный объем работ по проекту не имеет никакого отношения к улучшению программы. Например, менеджер проекта может потратить немало усилий на большую диаграмму Ганта или иную форму плана проекта, который не соответствует действительности, поскольку основан на информации, значительно изменившейся между этапом написания плана и моментом, когда началась работа над созданием ПО. Еще хуже, если руководитель проекта вложит много сил в актуализацию плана для периодических статус-митингов. Ясно, что это не поможет программному обеспечению: к тому времени, когда план будет готов, программа уже будет поставлена клиенту. Может быть, теперь этот менеджер проекта вовсе не станет вкладывать усилия в то, что не используется его командой. Тогда не исключено, что некий топ-менеджер засуетится, видя отличие плана от жесткой установки «всё всегда вовремя и в рамках бюджета». Всем, включая руководителя проекта, прекрасно известно, что эти волнения не помогут делу.

Типичный пример потерь — это работа менеджера проекта над планом, который не отражает действительности и фактически не используется командой разработки программного обеспечения. Но не с каждым планом происходит такая история — даже на водопадных проектах. Множество из них планируются эффективно (по крайней мере, настолько, насколько это возможно в неэффективной компании). Но если вы руководитель или разработчик проекта, подобного описанному, то уже узнали этот антипаттерн. Абсолютно ясно, что перед нами потери.

Когда вы объективно оцениваете то, что делает команда каждый день, вы определите все виды потерь. Может быть, у вас на полке давно пылится папка спецификаций? Силы, потраченные на создание этих документов, — потери. Если вы часами сидите над обзором кода, который полностью сосредоточен на поверхностных ошибках или личных предпочтениях, но не влияет на дизайн и выявление реальных проблем, — это тоже потери. Некоторые виды документов появляются до начала проекта, например подробное описание работ. На его изучение команда тратит целый день, но документ выбрасывают, когда начинается фактическая работа. Вы часами занимаетесь отладкой и проблемами развертывания, которые можно решить при помощи скриптов? Это явные потери. Бесполезные встречи, где участники по очереди сообщают свои индивидуальные статусы, чтобы координатор

проекта смог записать их в протокол, куда никто никогда не заглянет, — также типичный пример потерь. Scrum-команда, даже получающая результаты «лучше-чем-ничего», заменяет бесполезные совещания ежедневными scrum-митингами, тем самым исключая потери. В книге представлены и другие примеры потерь, которые также могут быть устранены.

Если что-то считается потерями с точки зрения разработки ПО, то это не обязательно бессмысленные действия. Просто они не помогают построению программного обеспечения. Возможно, старший менеджер действительно нуждается в плане проекта, потому что с его помощью хочет убедить акционеров продолжить финансирование. А протоколы совещаний бывают необходимы государственным регулирующим органам. Заявление о приеме на работу бесполезно для команды, но оно обязательная часть процесса заключения трудового договора. Все это может потребоваться, но не приносит пользы непосредственно проекту. Поэтому мы говорим о потерях.

Устранение потерь начинается с выявления. Умение видеть их — первый инструмент мышления для данной ценности.

Часто расточительная деятельность — это приоритет для кого-то (руководителя проекта, который не является частью команды, аккаунт-менеджера, кого-то из старших менеджеров), поэтому ее трудно распознать. Некоторые потери принимаются командой. Например, все знают, что процесс бюджетирования занимает много времени и не включает никакой деятельности, напрямую приводящей к созданию программного обеспечения. Есть вид потерь, настолько привычных, что они, по сути, невидимы. Так, члены одной команды могут располагаться в трех различных частях офиса, и, чтобы дойти до напарника и поговорить с ним, нужно дополнительно потратить пять минут.

Мэри и Том Поппендик решили определить семь потерь разработки программного обеспечения. Как и многое другое, связанное с Lean, эта идея позаимствована у компании Toyota в середине прошлого века. Поппендик обнаружили, что это поможет увидеть потери в вашем проекте.

Частично проделанная работа

Когда вы осуществляете итерации, вы лишь приближаете работу к состоянию «сделано», потому что если это не полностью работающий продукт, то он не создает ценности для пользователей. Любая деятельность, не предоставляющая ценность, — потери.

Дополнительные процессы

Пример дополнительного процесса — антипаттерны управления проектом (глава 7), где команда тратит 20% времени на отчет о состоянии проекта и оценки, которые используются исключительно для

обновления отчета о его состоянии. Все это требует дополнительных усилий в процессе отслеживания проекта и формирования отчетности, но не создает стоимости.

Лишние функции

Когда команда создает никем не заказанную функцию вместо той, которая на самом деле нужна пользователям, — это потери. Иногда причина в том, что кто-то в команде слишком увлекся новой технологией и хочет воспользоваться возможностью изучить ее. Возможно, это ценно для того, кто таким образом улучшает свои навыки, и даже для всей команды (в долгосрочной перспективе). Но это не помогает построению ценного программного обеспечения, так что является потерями.

Переключение между задачами

Некоторые команды работают в многозадачном режиме, и часто он выходит из-под контроля. Люди понимают, что полностью заняты (созданием программного обеспечения) и к тому же имеют дополнительные задачи, относящиеся к частичной занятости (поддержка, обучение и т. д.). И каждый кусочек работы важен и приоритетен. Scrum-ценность «сосредоточенность» демонстрирует нам, что переключение между проектами или не связанными между собой задачами в рамках одного проекта вызывает неожиданные задержки и усилия, потому что создает дополнительную нагрузку на способность к восприятию задач. Теперь мы можем назвать это иначе: потери.

Ожидание

Есть множество вещей, заставляющих профессиональных разработчиков ПО сидеть и ждать: кто-то должен закончить обзор спецификации, утвердить доступ к системе проекта, исправить проблемы с компьютером или получить лицензию... Все это — потери.

Движение

Когда члены команды располагаются в разных помещениях, людям приходится вставать и идти к коллегам, чтобы обсудить проблемы, связанные с проектом. Если сложить время, затраченное на такие прогулки, то потери составят несколько дней или даже недель.

Дефекты

Разработка через тестирование предотвращает множество дефектов. Каждый программист, «зараженный тестированием», хорошо знаком

с ситуацией, когда модульный тест обнаруживает ошибку, которую было бы очень сложно исправить позднее, и понимает: требуется гораздо меньше времени на написание всех тестов, чем на отслеживание одной ошибки, особенно если пользователи обнаружат ее после выпуска новой версии. Но если команде приходится задерживаться, чтобы исправлять ошибки, которые можно было предупредить, — это тоже потери.

Кажутся ли вам некоторые перечисленные нами потери полезными? Даже когда что-то теряется, это обычно полезно (или кажется полезным) для кого-то. Так, очевидно неэффективное размещение разработчиков по разным офисам, вероятно, помогает офис-менеджеру решать какие-то организационные проблемы. Если вы видите потери, то сможете понять эти мотивы и объективно оценить, насколько они важнее, нежели возможность сделать ваш проект качественно. Даже принося кому-то пользу, все эти вещи расточительны по отношению к созданию продукта, который обеспечивает ценность для пользователей и компании. Бережливое мышление предполагает четкое видение деятельности людей (внутри и за пределами команды), не добавляющей ценности для конкретных целей проекта.

Платформенная ловушка в главе 7 — хороший пример командных потерь. Если разработчики создают большую платформу для решения проблемы, которую можно устранить с гораздо меньшим количеством кода, то это означает, что эта платформа и есть потери — что довольно иронично, учитывая, что изначальной целью создания платформы было как раз предотвратить потери за счет автоматизации повторяющихся задач и устранения дублирования кода. Еще хуже, что эта платформа часто становится препятствием в будущем, потому что команде приходится либо расширять ее, либо выстраивать работу вокруг нее, когда необходимо добавить компоненты, которые она явно не поддерживает. Эти дополнительные усилия — тоже потери.

Команда, которая может разглядеть потери, ясно видит, что эта платформа предотвращает добавление ценности в проект. Люди понимают: потери влияют на то, как они выполняют свои задания, и ясно видят эти потери в своей ежедневной работе. Даже если их сочтут необходимыми для компании, они все-таки выглядят как мусор, потому что не добавляют ценности продукту.

ИСПОЛЬЗУЙТЕ КАРТУ ПОТОКА ЦЕННОСТИ, ЧТОБЫ ЯСНО УВИДЕТЬ ПОТЕРИ

В своей книге *Lean Software Development* Мэри и Том Поппендик рекомендуют запастись карандашом и бумагой и выполнить простое упражнение, чтобы помочь вам найти потери. Его называют «карта потока ценности», и вы можете реализовать его для любого процесса. Как и многие методы,

используемые в сочетании с Lean, он возник в производственной сфере, но актуален и для разработки программного обеспечения.

Чтобы создать карту потока ценности для проекта, требуется не более полу-часа. Вот как нужно действовать. Начните с небольшого рабочего элемента, несущего ценность клиенту, уже созданного командой и представленного пользователям. Попробуйте найти наименьшую единицу, такую, например, как **минимальная маркетинговая функция** (ММФ) или самый маленький «кусочек» продукта, приоритетный для потребителей. Заново продумайте все шаги, проводящие этот фрагмент от начала до поставки. Нарисуйте прямоугольник для каждого из шагов, используя стрелки, чтобы соединить этапы друг с другом. Поскольку вы рисуете актуальный путь, который проходит реальная функция, это будет прямая линия. Нет никаких точек принятия решения или развилок, потому что перед вами история реального объекта.

Концепция ММФ очень важна. К счастью, вам уже знакомо это понятие. Когда владелец продукта в scrum-команде управляет элементами в бэклоге, он имеет дело с типичными ММФ. Они часто принимают форму пользовательских историй, требований или запросов.

Затем оцените, сколько времени потребовалось для выполнения работы на первом шаге и сколько — до момента начала следующего. Повторите эту процедуру для каждого шага и нарисуйте линии под прямоугольниками, чтобы представить с их помощью время работы и время ожидания.

Рисунок 8.2 показывает пример карты потока ценности для реального объекта, проходящего через традиционный процесс водопада

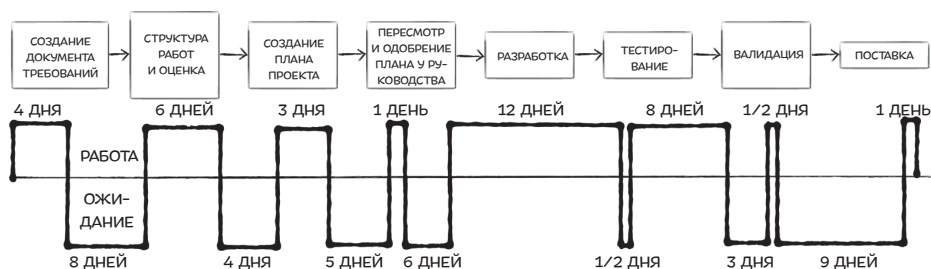


Рис. 8.2. Эта карта потока ценности показывает, как объект в проекте разработки программы проходит через традиционный цикл управления проектами. Команда может его использовать, чтобы отчетливее видеть, где время тратится впустую

Карта потока ценности наглядно показывает, сколько времени было потрачено на ожидание. В общей сложности с момента, когда коллектив начал

работать над программой, и до момента развертывания проекта прошел 71 день. Из этого количества 35,5 дня ушли на ожидание. На это может быть множеством причин: на формирование требований могло потребоваться много циклов ревью и утверждений, пришлось долго ждать встречи по оценке проекта, поскольку график каждого из ее участников был слишком плотный, планы всегда рассматриваются комитетом, собирающимся всего раз в неделю, и т. д. Карта потока ценности показывает совокупное влияние этих задержек на объект, не нагружая вас подробностями, почему так произошло. Видение общего воздействия помогает вникать в то, какие задержки — это потери, а какие необходимы для реализации проекта.

Для команды, возможно, это ожидание не казалось слишком долгим, потому что она наверняка занималась работой над другой функцией. Но клиент, который нуждается в этой функции, не всегда знает о других приоритетах команды. И, откровенно говоря, они ему неинтересны. Главное он знает точно: на создание нужной ему функции от момента запуска проекта до момента поставки ушел 71 день.

Представьте реакцию руководителя, когда он увидит, сколько времени было затрачено на ожидание, особенно если ему приходится иметь дело с нетерпеливым и требовательным клиентом. Если команда сможет найти способ снизить количество потерь и время ожидания, то она сумеет существенно сократить время поставки будущих функций. Это осчастливит клиента, который, в свою очередь, сделает счастливым руководителя. И вот как визуализация потерь позволяет проще убедить всех — особенно руководителя: что-то нужно менять.

Постарайтесь глубже понять продукт

Воспринимаемая целостность означает, что продукт в своей совокупности достигает баланса функциональности, практичности, надежности и экономичности, что и приводит в восторг клиентов. Концептуальная целостность означает, что центральные концепты системы работают вместе как сплоченное, единое целое.

Мэри и Том Поппендик. Lean Software Development: An Agile Toolkit

Принятое в вашей команде «бережливое мышление» — это больше, чем просто представлять себе, как выполняется работа, и видеть потери. Такая команда ясно понимает, какой программный продукт создает и то, как он обеспечивает ценность для пользователей. Думая о том, как этот продукт предоставляет ценность, члены команды размышляют о целостности. Итак, следующая ценность Lean — **целостность построения**. Команда,

исповедующая бережливое мышление, всегда думает о том, как придать целостность программному обеспечению.

Существуют два аспекта — внутренняя и внешняя целостность. Программное обеспечение должно иметь целостность с точки зрения пользователей (внешнюю) и с позиции разработчиков (внутреннюю). Lean включает в себя два инструмента, помогающие понять внутреннюю целостность: **рефакторинг** и **тестирование**. Это ровно то же самое, о чем вы узнали в главе 7. Очень эффективный способ построить систему с высокой степенью внутренней целостности заключается в использовании инструментов XP — особенно разработки через тестирование, рефакторинга и инкрементальной архитектуры.

В этой главе мы сосредоточимся на внешней целостности — том, что делает программное обеспечение более ценным для пользователей. Это касается понимания того, как рассуждают пользователи.

Данный предмет может показаться абстрактным. К счастью, Lean имеет два инструмента мышления, помогающие настроить мозг на восприятие целостности. Первый — это **воспринимаемая целостность**, или то, насколько товар отвечает потребностям человека, и как быстро пользователь понимает, что его запросы удовлетворены.

Каждый хороший продукт создан для того, чтобы решить проблему или удовлетворить потребность. Иногда речь идет о сугубо деловых аспектах: в бухгалтерской компании требуется автоматизировать налоговый и бухгалтерский учет, включающий изменения налогового кодекса, сделанные в этом году, чтобы показать, что удержания у клиентов законны. В других случаях ситуация сложнее: например, видеоигра должна быть очень веселой.

Если программа «глючит» и часто «падает», то налицо проблема целостности. Но как только вы создаете нормально работающее программное обеспечение, его воспринимаемая целостность может стать более коварной. Один крупный новостной сайт, например, на протяжении многих лет испытывал трудности с целостностью. В течение этого времени было трудно скопировать размещенные на нем тексты и вставить их в документы или электронные письма. Первая попытка выделить текст вызывала появление на сайте всплывающего определения слова, на которое нажал пользователь. В конце концов эта функция была отключена, и у пользователей появилась возможность копировать и вставлять. Но при редизайне сайта было заблокировано выделение текста целиком, а попытка сделать это приводила к всплыванию в новом окне соответствующей статьи.

Такое поведение сайта представлялось пользователю непоследовательным и непонятным. Возможно, новостное агентство делало все это умышленно.

Подобным СМИ бывает необходимо предотвратить копирование своей интеллектуальной собственности людьми, желающими вставить некий фрагмент в свою почту, и они предпочитают, чтобы пользователи, желая поделиться прочитанным, применяли функцию «переслать эту статью по почте». Но это не меняет того факта, что сайт работал не так, как от него ожидалось. Перед нами пример плохо воспринимаемой целостности.

Второй инструмент мышления, помогающий понять целостность, — это **концептуальная целостность**, или то, как функции программного обеспечения работают вместе, чтобы сформировать единый унифицированный продукт. Когда вы чувствуете воспринимаемую и концептуальную целостность ПО, вы можете сделать его более ценным для пользователей.

Приведем пример того, как концептуальная целостность оказала влияние на целую отрасль. Речь идет об эволюции видеоигр в первом десятилетии двадцать первого века. В конце 1990-х большинство любителей видеоигр составляли продвинутые пользователи. Случайных игроков было гораздо меньше, и многие из них испытывали разочарование, потому что, купив новую игру, обнаруживали, что она слишком трудна для них. Наряду с этим наиболее продвинутые хардкор-геймеры регулярно жаловались, что многие игры оказывались чересчур простыми.

В течение следующего десятилетия видеоигры становились все более популярными. Поэтому разработчики упорно искали способы спроектировать игры для обеих аудиторий. Как этого удалось добиться?

Первым делом следовало уразуметь, что как случайным, так и хардкорным игрокам нужна концептуальная целостность. Забавные казуальные игры типа Tetris, Angry Birds и Candy Crush предусматривали медленное развитие со все увеличивающейся сложностью на многих уровнях. Ценность для случайных геймеров — это постоянное увеличение сложности в сочетании с постоянным чувством удовлетворения от достижения цели. Если Angry Birds начинали с пяти легких уровней, а потом сталкивали игрока с уровнем, который оказывался для него крайне сложным, то люди переставали играть, потому что обнаруживался явный разрыв в концептуальной целостности. Такой разрыв называется *диссонансом*.

Хардкорные геймеры не любят игры с медленным, стабильным обучением или постоянным вознаграждением за успехи, которых они не «заработали». Они часто получают большее удовлетворение от «шлифовки» или необходимости вступать в повторяющиеся и часто неприятные задачи ради получения награды, соответствующей прогрессу.

Забавной видеоигре не следует иметь уровень, чреватый фрустрацией игрока, «шлифующая» видеоигра не должна иметь легкий уровень. Игры

типа Flappy Bird, Super Meat Boy и многие из игр Final Fantasy заслужили похвалы хардкорных геймеров благодаря своей сложности и тому количеству раз, которое следовало пройти многие уровни, прежде чем игра оказывалась освоена. Простой уровень в «шлифующих» видеоиграх мог вызвать такой же диссонанс в душе хардкорного геймера, какой сверхсложный уровень в относительно легкой игре — у случайных игроков.

Команды, создающие видеоигры, столкнулись с большим количеством проблем в отношении своей концептуальной целостности, в то время как индустрия видеоигр активно росла в начале двадцать первого века. Множество игр получили плохие отзывы, потому что были отмечены как слишком легкие для хардкорных геймеров или чересчур сложные для обычных игроков. Эти команды научились включать в свои игры функции, которые улучшали концептуальную целостность для обеих аудиторий. Например, большинство игр, направленных на оба рынка, теперь имеют настройку сложности. Если ваш персонаж постоянно умирает, игра предложит вам снизить сложность. Хардкорный геймер никогда бы не выбрал этот вариант — и игра с хорошей концептуальной целостностью не станет спрашивать его, хочет ли он легкой игры, потому что даже задавать этот вопрос несвойственно для сложной игры. Но в индустрии есть мнение, признающее казуальных и хардкорных геймеров различными рынками, и существует много игр, продающихся только в одной из этих двух групп.

Все эти разработки касаются повышения ценности, которую игра приносит игроку, за счет понимания того, как он играет, и разработки игр с концептуальной целостностью в их уровне сложности. Команды, создающие видеоигры, модифицировали способ своей работы, внося изменения в разработку игры, в своем стремлении улучшить концептуальную целостность. Сейчас принято в самом начале проекта решать, для кого команда создает игру: для случайных игроков, хардкорных геймеров или для обеих категорий. Программисты включают в свою работу тестовые задачи, ориентированные на целевую аудиторию, и сотрудничают со своими маркетинговыми отделами, чтобы выяснить, нацелены ли игры на нужную аудиторию. Именно так команда может изменить подход к работе, чтобы повысить концептуальную целостность.

РАССМАТРИВАЙТЕ СИТУАЦИЮ В ЦЕЛОМ

Когда вы работаете в команде по созданию программного обеспечения, вы не находитесь в вакууме. То, как организованы структуры ваших команд и компании, активно влияет на организацию рабочего процесса. Помимо этого, на проекты оказывают воздействие всевозможные барьеры

и препятствия, имеющиеся в любой компании. Например, вам может понадобиться полтора десятка согласований спецификации с менеджерами, прежде чем вы сможете начать создавать новую функцию программного обеспечения. Не исключено, что негативные комментарии пользователя вызовут у владельца продукта панику и желание спланировать ваши выходные за вас. Ваш руководитель может плениться чрезвычайно сложным процессом распределения заданий на работу, и теперь вам предстоит проводить каждое задание через восемь этапов оформления, прежде чем вы сможете начать работать над ним. Это лишь несколько примеров, и вы, вероятно, можете вспомнить немало случаев неэффективной деятельности, с которыми вам приходилось сталкиваться в процессе работы.

Мы видели, как такие действия становятся потерями, но иногда они служат целям, которые могут не принести пользу проекту, однако необходимы для него или для компании в целом. Например, команда может впустую тратить время (с точки зрения пользы для проекта), заполняя отчеты, не помогающие проекту. Но если они необходимы для регулятора, то нужны и компании. Как мы узнаем, какие виды деятельности действительно полезны?

Вот где проявляется следующая lean-ценность: рассматривайте ситуацию в целом. Чтобы понять, результативна ли ваша команда, оцените всю систему, чтобы иметь *объективный* взгляд на вещи: слишком легко в процессе решения полностью выложиться эмоционально. Например, менеджер проекта создал систему расписаний, которая требует, чтобы каждый разработчик ежедневно каждые 15 минут заполнял табель учета времени. Он, возможно, будет счастлив, обзаведясь постоянно обновляющимся статусом проекта, которого он так добивался, но при этом даже не догадывается, сколько сил уходит на это у команды. Поэтому намного проще убедить его отказаться от этих дополнительных обновлений, если удастся доказать, что они обходятся команде, скажем, в 5% от ее производительности.

Но хотя признание характера системы, в которой команда работает, может показаться вам простым, это не всегда легко сделать. Каждый участник команды ощущает себя настолько комфортно, насколько его удовлетворяет работа в проекте и то, как он видит свой индивидуальный вклад в нее. Например, программист может считать проект удачным, если ему приходится решать интересную проблему кодирования. Менеджер доволен, когда команда не срывает сроков. Но если разработчику пришлось отказаться от сна по ночам и выходных дней, чтобы уложиться в срок, то он может почувствовать себя так, как если бы этот проект оказался значительно менее успешным.

Вот почему lean-команды применяют **метрики** — еще один инструмент бережливого мышления, благодаря которому каждый может видеть проект

с одной и той же точки зрения. Есть множество аспектов, которые команда разработчиков способна измерить, и выбор правильной метрики помогает получить более полное представление о проекте. Различные измерения проводятся в зависимости от того, какую проблему требуется решить. Все участники проекта имеют различные точки зрения, а целевое использование метрик помогает команде воспринимать проект одинаково.

Для людей, которые не потратили много времени на измерения систем, особенно систем, которые команда использует для создания программного обеспечения, — эта идея может показаться очень абстрактной. Давайте рассмотрим конкретный пример.

Пользователи очень трепетно относятся к тому, насколько команды учитывают их потребности. Клиенты хотят, чтобы их требования быстро превращались в программное обеспечение. Они чувствуют себя намного счастливее, получая от команды одну новую функцию в месяц вместо трех — но за три месяца. Именно поэтому Scrum и XP используют итерации, а agile-команды — короткие циклы обратной связи.

Представьте, что вы владелец бизнеса, который платит за программный проект. Как понять, правильно ли вы расходуете свои деньги? Допустим, вы постоянно получаете доклады от руководителя проекта, где утверждается, что проект выполняется в соответствии с графиком, в рамках бюджета и вообще все прекрасно. В докладе много диаграмм, отчетов и завершенных задач по планам проекта, которые изобилуют зелеными точками, чтобы показать, насколько идеально совпадал с графиком ход выполнения проекта за последние четыре выпуска.

Это выглядит неопровержимым свидетельством того, что команда проекта успешно создает программное обеспечение, предоставляя те возможности, о которых просили пользователи. Более того, менеджер может указывать на буферы в расписании, чистый перечень рисков и систему учета работ, демонстрирующую, что проект полностью контролируется и все необходимые предосторожности были сделаны для известных и неизвестных рисков в будущем. Как руководителю вам приятно осознавать, что у вас есть контроль над проектом, представление о том, как он работает, и способ справиться с неожиданными проблемами.

Но если вы слышите от клиентов, использующих это программное обеспечение, что они сделали ряд простых запросов несколько месяцев назад и до сих пор не обнаружили их среди добавленных функций? Возможно, вы начинаете терять клиентов, которые переходят к конкуренту, потому что он более чутко реагирует на их потребности? И при этом вы хотите сказать, что проект удачен? Налицо серьезная проблема, и вам нужно работать с командой, чтобы это исправить.

Но как убедить команду побыстрее ответить на самые важные запросы? Можно пытаться противостоять ей, но разработчики указывают на положительные отчеты о состоянии дел и утверждают, что с их точки зрения проект идет очень хорошо. Как заставить их увидеть проблему?

В этом поможет объективная метрика. Многие команды будут измерять **время выполнения** для каждой функции. Это метрика среднего времени — между моментом запроса объекта и его поставкой.

Вот как его высчитывают. Каждый раз, когда пользователь делает запрос, записывайте дату. Когда версия ПО, включающая эту просьбу, будет выпущена, зафиксируйте дату окончания. Разница между этими датами — это и есть время выполнения для каждого запроса. Сложите время выполнения всех запросов в релизе и разделите сумму на количество функций, чтобы вычислить среднее время выполнения для данного релиза*.

Если вы спросите команду, каково среднее время выполнения запросов, то что они, по вашему мнению, скажут? Разработчики, выпускающие программное обеспечение ежемесячно, наверняка догадаются, что это примерно один-два месяца. Вероятно, это приемлемый срок — большая часть ваших пользователей будут довольны им, разве что вы слышали несколько ворчливых историй от особо недовольных пользователей.

Но что если время выполнения окажется дольше того, на которое согласны клиенты? Не исключено, что реализация простого пользовательского запроса занимает у вас шесть месяцев. Это неудача команды? Может быть, ошибку допустили вы? Или длительное время выполнения — неизбежный побочный эффект того, как организован ваш бизнес? Пока вы этого не знаете. Но зато вы выявили проблему и, поскольку провели измерения — сможете помочь команде понять, что и у нее не все гладко.

Теперь, когда речь пойдет о жалобах на работу команды, вы сможете что-то противопоставить менеджеру проекта, который ссылается на успешные отчеты, и указать на объективное измерение времени выполнения. Так вы докажете, что у команды действительно есть проблема. Это гораздо лучше, чем фраза «я здесь главный, поэтому делайте так, как я сказал», потому что есть четкая и объективная цель, над которой может работать каждый. Речь не идет о произвольном решении или магическом мышлении.

* Есть и другие способы, чтобы вычислить время выполнения. Например, можно рассчитать так, что большие функции имеют более тяжелый вес, чем мелкие. Мы в качестве примера выбрали простой способ расчета времени, чтобы было понятно, как это может помочь команде.

НАХОДИТЕ ПЕРВОПРИЧИНЫ ОБНАРУЖИВАЕМЫХ ПРОБЛЕМ

Проводить измерения и видеть объективную истину в отношении проекта и команды — это только первая часть видения целого. Вторая часть — понимать **первопричины** (или фактические причины) проблемы.

В конце главы 6 мы обещали вернуться к причинно-следственному анализу, потому что, будучи важной частью бережливого мышления, он также является одним из методов вывода для команд XP. Команды XP и lean-группы используют метод **пяти «почему»**, чтобы выяснить коренную причину проблемы. Как и многое в бережливом мышлении, эта техника зародилась в японской автомобильной промышленности, но отлично подошла agile-командам. Согласно этой простой практике, члены команды задают вопрос «почему» и делают это не менее пяти раз (хотя получают ответы), пока не обнаружат первопричину проблемы.

В нашем примере команда может использовать метод пяти «почему», задавая примерно такие вопросы:

- *Почему среднее время выполнения так велико?* Потому что выполнение пожеланий большинства пользователей требует более шести месяцев, чтобы превратить их в программное обеспечение.
- *Почему требуется более шести месяцев, чтобы превратить запросы в программное обеспечение?* Потому что эти запросы на функции почти всегда отодвигаются, чтобы освободить место в расписании для изменений, поступивших в последнюю минуту.
- *Почему существует так много подобных изменений?* Потому что, прежде чем команда сможет выпустить ПО для пользователей, она должна сделать обзор со старшими руководителями, которые практически всегда запрашивают некие базовые, фундаментальные изменения.
- *Почему топ-менеджеры почти всегда запрашивают базовые, фундаментальные изменения?* Потому что каждый из них имеет собственное, довольно специфическое мнение о том, как программа должна выглядеть и функционировать, какие технические средства следует использовать, чтобы ее создать, но команда не слышит этих мнений до тех пор, пока она не напишет весь код и не сделает презентацию для старших менеджеров.
- *Почему команда не может услышать эти мнения до тех пор, пока она не напишет весь код и не проведет презентацию?* Потому что топ-менеджеры слишком заняты, чтобы поговорить с командой в начале проекта, они появляются только на финальной презентации — и после ее завершения отправят команду обратно к чертежной доске.

Вот теперь мы наконец поняли, почему реализация запросов пользователей занимает так много времени. Проведение измерений и поиск первопричины помогли выяснить, что не во всем виновата команда. Она заканчивает разработку многих функций, но, когда проводит демонстрацию для старших менеджеров, они просят внести целый ряд изменений. Возможно, выяснится, что эти изменения необходимы, то есть идеи менеджеров правильные. Но даже полезные изменения требуют, чтобы этот конкретный менеджер провел анализ последствий, обновил план проекта и перенес измененные функции на более поздний выпуск. Что и привело в итоге к весьма длительному периоду выполнения, который был измерен. Более того — некоторые пожелания уже были запланированы на следующую версию, так что пришлось сдвинуть на *еще более позднюю* версию, чтобы освободить место для изменений, внесенных менеджерами. В результате время выполнения некоторых запросов настолько затянулось, что некоторые клиенты были вынуждены уйти к конкурентам.

Понимание первопричины долгого времени выполнения дает несколько вероятных решений. Команда может создавать ПО более итеративно и попросить менеджеров принимать участие в демонстрации в конце каждой итерации, а не только основного релиза. Менеджеры могут делегировать свое право утверждать продукт тому (например, владельцу продукта), кто принимает более активное участие в проекте и чаще встречается с командой, и доверить этому человеку право принимать полномочные решения. Кроме того, команды и менеджеры могут продолжать создавать программы таким же образом и оставить неизменными длительные сроки выполнения, но обзавестись аккаунт-менеджерами для работы с пользователями и клиентами, чтобы управлять их ожиданиями.

Подведем итоги: команда начала с проблемы — недостаточно быстрой реакции на пожелания пользователей. Путем измерений и поиска первопричины она смогла *увидеть ситуацию в целом*. Разработчики поняли, как их проект вписывается в работу компании, и смогли выявить несколько возможных способов, приводящих к долгосрочному решению. И самое главное, что команда и руководитель теперь имеют одинаково объективную информацию и принимают решения вместе.

Поставляйте как можно быстрее

Есть еще одна lean-ценность — **поставляйте как можно быстрее**.

Как вы это понимаете? Возможно, вам представляется, как грозный руководитель или менеджер проекта принуждает команду работать допоздна,

чтобы поскорее выпустить готовый код. Означает ли решение «поставлять как можно быстрее» отказ от тестов, низкоприоритетных функций и всего остального, что считается «посторонним» или низкоприоритетным? А может быть, вы вспоминаете о героических разработчиках, засиживающихся в офисе до поздней ночи и в выходные или стремящихся пропихнуть свои «быстрые-и-грязные» костыли, чтобы получить важную функцию на выходе. Как раз все это рождается в умах большинства менеджеров, когда они слышат фразу «поставляйте как можно быстрее». Многие разработчики, тестировщики и другие инженеры думают то же самое.

Agile-команды знают, что подобные вещи заставят команду поставлять программные продукты медленнее, а не быстрее. Это идея о том, почему есть agile-принцип содействия устойчивому развитию («спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп в течение неопределенного срока» — один из принципов, о которых вы узнали в главе 3). Срезание пути и углов и сверхурочная работа требуют больше времени и денег, чем экономят. Команды выполняют работу лучше и быстрее, когда у них есть время, чтобы сделать все правильно.

Хотя это верно, но звучит абстрактно и несколько возвышенно. Принцип сосредоточенности из Scrum и XP-практики энергичной работы помогает сделать это правило более конкретным. Scrum и XP дали понимание того, как добиться оптимальных темпов для поставки с использованием итераций и потока. Lean развивает эту идею, предлагая три инструмента мышления, способные помочь командам поставлять как можно быстрее: систему вытягивания, теорию массового обслуживания и стоимость задержки.

Цель теории массового обслуживания — это необходимость убедиться, что люди не перегружены работой и у них есть время делать вещи правильно. Очередь представляет собой список задач, функций или элементов для группы либо индивидуального разработчика. Очереди имеют порядок. Это, как правило, «первый вошел, первый вышел». Согласно данному правилу, никто не может изменять очередность и элемент, который попал в очередь первым, прежде других вытягивается следующим участником и берется в работу. Теория массового обслуживания является математическим исследованием очередей, и одно из ее направлений включает в себя предсказание тех последствий, которые вызываются добавлением очередей на входе системы. Lean говорит, что если сделать очередь как поток командной работы публичной и центральным местом в процессе принятия решений, то это поможет командам поставлять результаты работы значительно быстрее.



Ключевые моменты

Когда менеджеры считают, что команды могут сделать невозможное, и игнорируют реальные ограничения проекта, они используют **магическое мышление**.

Команды при помощи бережливого мышления ведут работу по **ликвидации потерь**, находя любые виды деятельности, которые напрямую не способствуют построению ценных продуктов, и устраняя их.

Любая деятельность, которая явно не добавляет ценности проекту, считается **потерями**. Lean-команды стремятся увидеть их и исключить из проектов, насколько это возможно.

Мэри и Том Поппендик зафиксировали **семь потерь разработки программного обеспечения**: частично проделанная работа, дополнительные процессы, лишние функции, переключение между задачами, ожидание, движение и дефекты.

Построение целостности — это lean-ценность, включающая в себя **воспринимаемую целостность**, или то, как хорошо продукт удовлетворяет потребности пользователей, и **концептуальную целостность**, или то, как слаженно работают функции продукта.

Бережливое мышление помогает командам **увидеть ситуацию в целом** или объективно понять то, как работает команда, в том числе ее недостатки. **Измерение** дает возможность сохранить объективное представление о проекте и команде.

Поставлять как можно быстрее значит избавиться от любых ненужных действий, приводящих к задержкам в работе и возникновению узких мест.

ИСПОЛЬЗУЙТЕ ГРАФИКИ ДЛЯ ВИЗУАЛИЗАЦИИ НЕЗАВЕРШЕННЫХ РАБОТ

Как узнать, сможете ли вы поставить программный продукт максимально быстро?

Бережливое мышление предлагает использовать для этого измерения. Эффективный способ определить, как команда поставляет ценные продукты, — применение **диаграммы незавершенных работ (work-in-progress, WIP-диаграмма)**. Это простая схема, которая показывает, как минимальные маркетинговые функции протекают через ваш поток создания ценности.

Если вы создали карту потока ценности, то можете построить WIP-диаграмму — плоскую диаграмму, показывающую, как объекты, продукты или другие меры величины потока ценности проходят через каждую часть потока создания ценности. Это работает нагляднее, если вы используете ММФ, поскольку они представляют собой минимально определенный размер того «куска» пользовательской ценности, который создается.

Рассмотрим карту потока ценности, которая показывает, как мастерская веб-разработки создает большую часть своих ММФ.

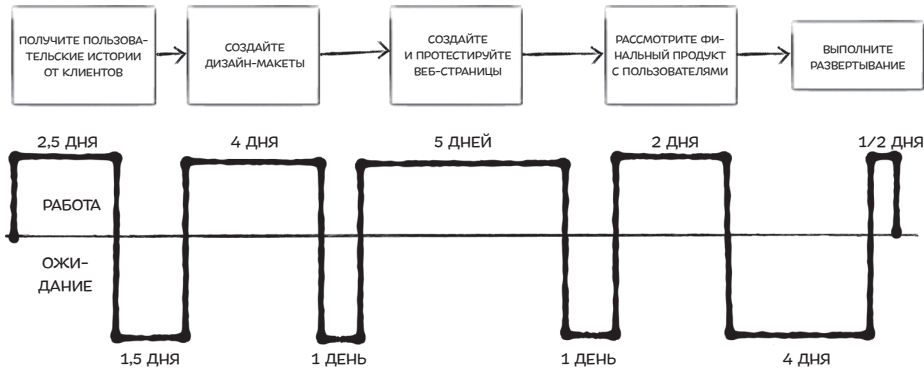


Рис. 8.3. Мы будем использовать карту потока ценности для построения WIP-диаграммы

Цель WIP-диаграммы состоит в том, чтобы показать полную историю **прогресса работ** и все те ценные характеристики, над которыми команда работает в настоящий момент. График демонстрирует, сколько ММФ находятся в работе в любой из дней и как они разбиваются на различные этапы потока создания ценности. Прогресс работ — это *измерение, касающееся функционала, несущего ценность заказчику, а не технических задач*. Другими словами, он показывает количество функционала или частей продукта, которые находятся в работе, а не конкретные задачи, выполняемые командой, чтобы произвести их. В Scrum мы называли их историями. Команда разбивала истории на задачи, которые перемещала на доске задач. Здесь, по аналогии, мы проводим измерение потока историй. Пользовательская история — хороший способ понять ММФ, потому что она представляет собой небольшой, самодостаточный кусок ценности, поставляемой клиенту. История могла бы появиться в WIP-диаграмме, но задачи, которые команда использует, чтобы реализовать историю, этого не могут.

Чтобы построить WIP-диаграмму, начните с оси *X*, показывающей дату, и оси *Y*, которая обозначает количество ММФ. Диаграмма содержит линию

для каждого из элементов на карте потока ценности. Линии делят диаграмму на области для элементов карты потока ценности.

Нет прогресса никаких ММФ до начала проекта, так что есть только одна точка при $X = 0$ в левой части диаграммы (день 0). Допустим, что при запуске проекта команда начинает работать с пользователями над девятью пользовательскими историями, и она применяет их как ММФ для своего проекта. Несколько дней спустя добавляются еще три истории. Вы сможете нарисовать точку на 9 в первый день, потом еще на $9 + 3 = 12$, и, когда эти новые ММФ добавятся, вы сможете соединить точки линиями.



Рис. 8.4. Начните строить WIP-графики, создавая линейную диаграмму ММФ (например, пользовательских историй) на первом этапе потока создания ценности, и затените область под ней

Несколько дней спустя программисты начинают работать над созданием дизайн-макетов для четырех историй, так что эти истории прогрессируют к следующему этапу потока создания ценности. Общее количество ММФ в системе — по-прежнему 12, но теперь они разделены на 8, еще находящихся в разработке (или **ожидающих**, поскольку карта потока ценности описывает и время работы, и время ожидания), — на первом этапе потока

создания ценности, и 4 — на втором этапе. Так что вы можете добавлять точки на 4 и 12 ММФ, чтобы представить это.

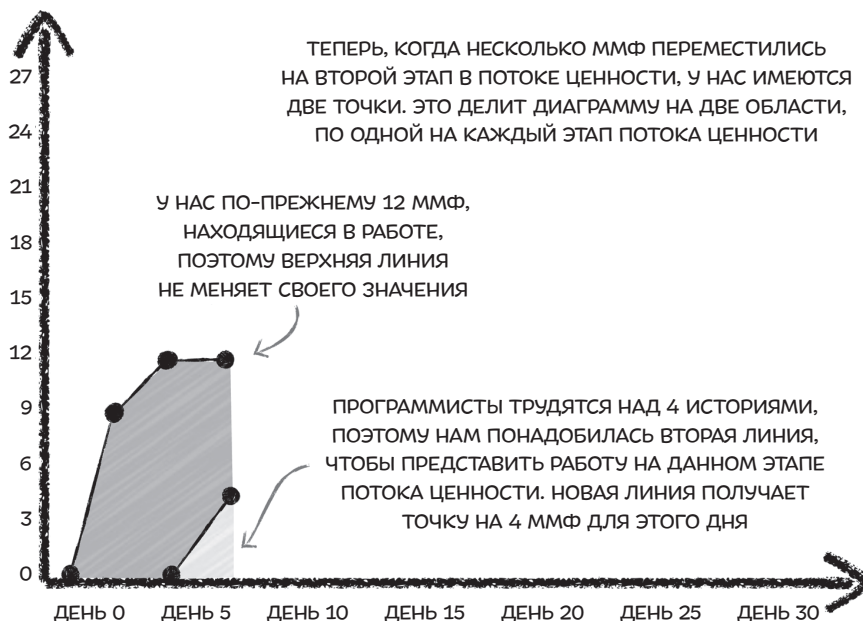


Рис. 8.5. Когда работа прогрессирует к следующему этапу в потоке ценности, на WIP-диаграмме добавляется новая линия, разделяя ее на две зоны. Верхняя линия по-прежнему представляет общее количество ММФ в прогрессе, а пространство между ней и новой линией представляет количество ММФ, находящихся на первом этапе

Поскольку ММФ перемещаются по потоку создания ценности, общее количество задач растет и с течением времени WIP-диаграмма приобретает вид полосок, соответствующих каждому из этапов карты потока ценности. Что происходит, когда создание всех ММФ завершено? Если вы продолжаете показывать их на графике, то в конце концов количество ММФ в стадии «сделано» разрастается до таких размеров, что мешает воспринимать все остальные столбцы. Это делает активные ММФ похожими на ленту на вершине горы.

Это показывает рост, а не поток. Безусловно придавая больший вес вашему отчету и помогая произвести впечатление на старшего менеджера (поскольку демонстрирует, что команда проделала огромный объем работы), это не слишком-то полезно для управления потоком. Удаление с диаграммы работ со статусом «сделано» дает более четкую визуализацию того, как со временем изменяется протекание потока ценности.

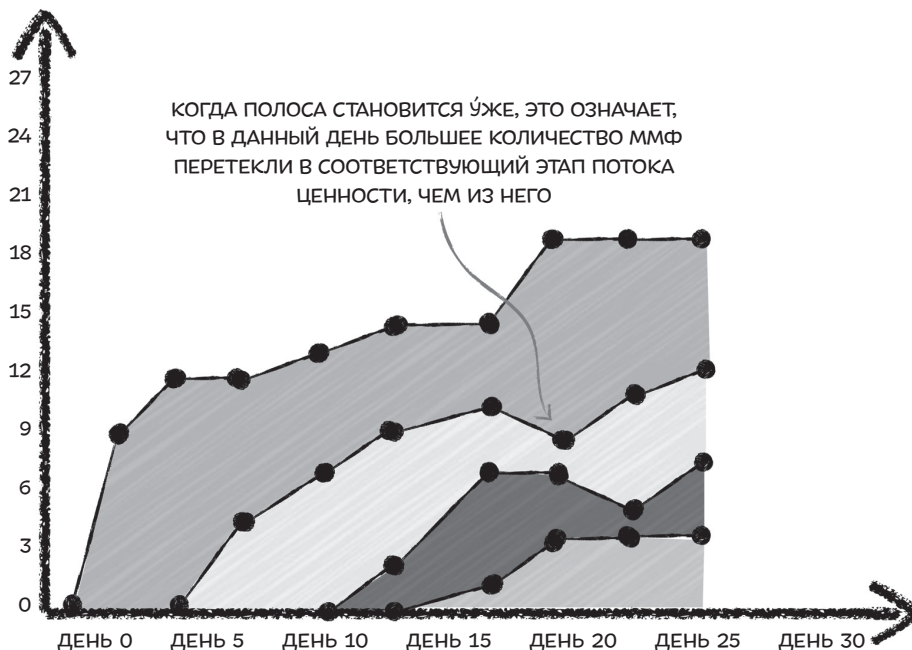


Рис. 8.6. WIP-диаграмма показывает, как работа прогрессирует со временем

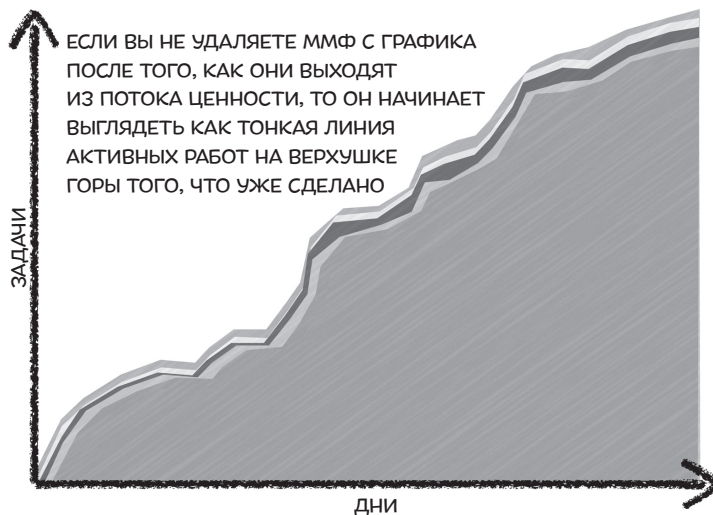


Рис. 8.7. Когда команда хочет показать руководителю, сколько работы выполнено, она оставляет задачи со статусом «сделано» на графике, потому что это выглядит впечатляюще. К сожалению, это делает его менее удобным для измерения прогресса работ

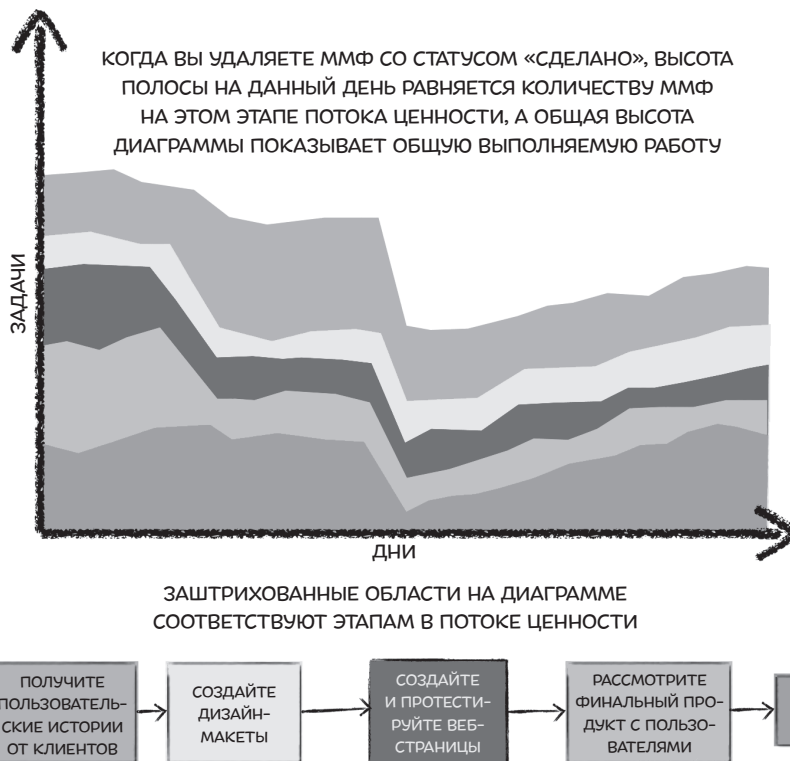


Рис. 8.8. Полезно удалять задачи со статусом «сделано» с графика и использовать разные оттенки для каждой полосы, чтобы было понятно, каким столбцам они соответствуют

Вот почему большинство WIP-диаграмм не включают в себя завершенные задачи. Таким образом, если проект стабилизируется с течением времени, то WIP-диаграмма также выглядит стабильной*.

Когда ММФ переходит из одного этапа потока создания ценности в следующий, полоса старого этапа становится тоньше, а полоса нового — толще.

Это позволяет легко обнаруживать тенденции — как, например, когда много ММФ переходят из одного столбца в другой (или удаляются из потока ценности полностью) в одно и то же время.

* Глядя на этот график, вы, возможно, удивляетесь, почему мы не называем его диаграммой совокупного потока или CFD? В главе 9 мы объясним разницу между WIP-диаграммой и CFD.

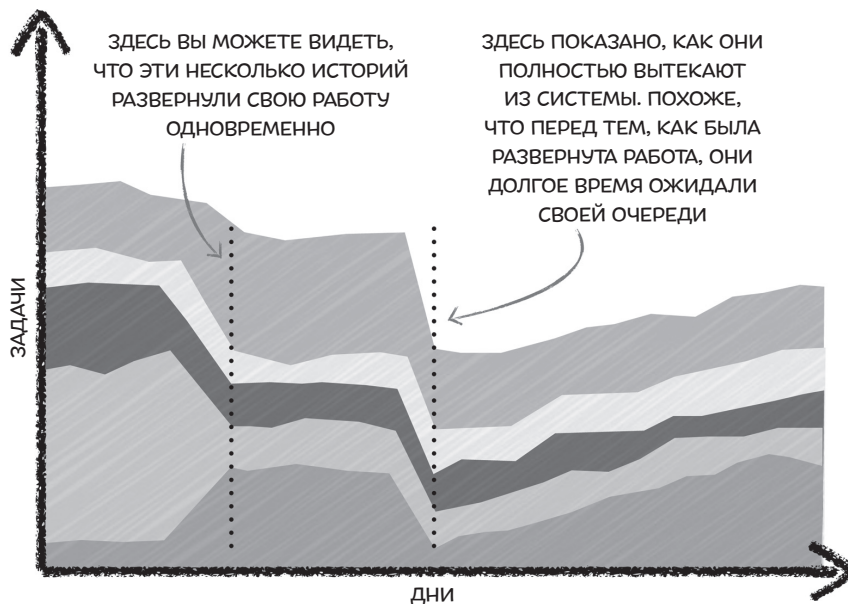


Рис. 8.9. WIP-диаграмма помогает видеть, как работает поток создания ценности, а также легче обнаруживать задержки и другие потенциальные потери — например, те истории, которые слишком долго ожидают своего развертывания в производство

КОНТРОЛИРУЙТЕ УЗКИЕ МЕСТА, ОГРАНИЧИВАЯ ВЫПОЛНЯЕМУЮ РАБОТУ

Есть важная идея, которую использует теория массового обслуживания. Это теория ограничений, созданная физиком и гуром менеджмента Элияху Голдраттом. Одна из главных идей этой теории заключается в том, что особое ограничение (например, работа, которая копится перегруженной командой) устанавливает предел общего объема работ, который можно пропустить через систему. Когда данное ограничение устранено, другое ограничение становится критическим. Теория ограничений утверждает: каждый перегруженный рабочий процесс имеет по крайней мере одно ограничение.

Когда ограничение накапливается в определенной точке рабочего процесса, люди часто называют это **узким местом** или бутылочным горлышком. Если убрать одно узкое место в системе (за счет изменения процесса или добавления людей), то можно добиться, чтобы работа протекала более гладко. Теория ограничений говорит нам, что обнаружится, однако же, какое-нибудь иное ограничение или узкое место где-то еще в системе.

Между тем можно уменьшить общий объем потерь путем систематического отслеживания критических ограничений и их ликвидации.

Каково это — работать в команде, которая ежедневно имеет дело с одним из этих ограничений? Другими словами, как быть, если вы и ваша команда — это и есть узкое место?

Оказавшись узким местом в системе, всегда ожидаешь работы в многозадачном режиме с постоянным переключением между нормальной работой с полной занятостью и более редкими эпизодами разовых задач. Имейте в виду: множество команд, как правило, нагружены массой задач, возложенных на них в то же самое время, но не называйте это «многозадачностью». Люди обычно используют термин «многозадачность», чтобы замаскировать факт перегруженности команды. Разделение работы и подталкивание команды к многозадачному режиму часто удерживает вас от признания, что работы больше, чем времени, особенно если учесть дополнительное время и когнитивные усилия, необходимые для переключения между задачами.

Например, команда, которая 100% своего времени посвящает разработке, может иметь руководителя с магическим мышлением, который просит ее потрудиться несколько часов в неделю в режиме многозадачности, в связи с чем люди уже не имеют ни поддержки, ни обучения, ни ремонта, ни совещаний, ни других дел. Им трудно однозначно определить, что работы оказалось больше, чем времени, особенно если лишние задачи добавляются понемногу, а не за один раз. Разработчики начинают чувствовать себя перегруженными, и, поскольку все это носит название «многозадачность», они не всегда догадываются, почему им так тяжело. Появится чувство, будто есть много работ на неполный рабочий день, за которыми невозможно угнаться. Можно помочь команде, применяя теорию массового обслуживания, чтобы разобраться в проблеме. Теперь мы знаем, что работа накапливается в узком месте где-то в рабочем процессе и растет взваленный на разработчиков ее объем.

СИСТЕМА ВЫТЯГИВАНИЯ ПОМОГАЕТ КОМАНДАМ УСТРАНИТЬ ОГРАНИЧЕНИЯ

И у меня есть философия, которой я живу. Все, кто работает со мной, знают об этом, вот она — на стене: «Если глупость входит в комнату, то у вас есть моральный долг застрелить ее независимо от того, кто ее сопровождает».

Кеоки Андрус. Beautiful Teams (глава 6)

Система вытягивания — способ выполнения проекта или процесса, который использует очереди или буферы для устранения ограничений. Это еще одна концепция из тех, что возникли в японской автомобильной

промышленности, но впоследствии нашли свое место в области разработки программного обеспечения. Производители автомобилей, в частности Toyota, в 1950-х и 1960-х годах оценили свои склады комплектующих и попыталась найти способы уменьшить количество деталей, которые должны там лежать. После длительных экспериментов они обнаружили, что даже если в наличии почти все детали, необходимые для сборки автомобилей, то дефицит нескольких деталей может задержать всю линию. Потому что в итоге все монтажные бригады ждут поставки недостающих запчастей. **Стоимость задержки** становится важна: если деталь в дефиците, то задержка ее поставки на конвейер оказывается очень дорогой, а если часть имеется в изобилии, то стоимость задержки низкая.

Команды Toyota обнаружили, что можно сократить расходы и поставлять автомобили гораздо быстрее, если знать, какие запчасти нужны прямо сейчас, и поставлять на линию только их. Чтобы реализовать это, они придумали производственную систему Toyota (TPS). Это предшественница бережливого производства, которые Том и Мэри Поппендик адаптировали для создания бережливой разработки программного обеспечения.

В основе TPS лежит идея существования трех типов потерь, которые создают трудности в рабочем процессе и должны быть удалены.

- Мۇда (無駄), что означает «бесполезность, бесперспективность, праздность, избыточность, потери, расточительность».
- Мۇра (斑), что означает «неравномерность, неритмичность, отсутствие единообразия, неоднородность, неравенство».
- Мۇри (無理), что означает «неразумность, невозможность, перегруженность, излишняя сложность, пересиливание, принуждение, превышение, чрезмерность».

Любой, кто участвовал в плохо управляемых, буксующих проектах программного обеспечения — особенно тех, что используют неэффективный процесс, — не понаслышке знаком с идеями неравномерности, бесполезности и неразумности. Это верно для неэффективных водопадных процессов, но любой работавший, скажем, в команде с неправильным мышлением, способной достигать только результата «лучше-чем-ничего» (или еще худшего) с применением Scrum или XP-практики, также может распознать эти случаи.

Не кажутся ли вам знакомыми какие-либо из перечисленных ниже действий?

- Заставить всех подписать спецификацию занимает много времени, а разработчики между тем сидят и ждут старта проекта. Как только он начнется, они уже опоздали.

- Вечная забота менеджмента — обеспечить бюджет. К тому времени, когда проект получает зеленый свет, думать об этом уже поздно.
- На середине разработки группа признает, что важные элементы дизайна или архитектуры программного обеспечения должны быть изменены, но это вызовет большие проблемы, потому что зависят от них многие другие вещи.
- QA-команда (Quality Assurance) не начинает тестирования программного обеспечения, пока каждая функция не будет завершена. Позднее они найдут серьезную ошибку или проблемы с производительностью, и команде разработки придется отвлекаться от текущей работы, чтобы это исправить.
- Анализ и дизайн продолжают так долго, что, когда кодирование все-таки начнется, каждый программист вынужден работать по ночам и выходным, чтобы уложиться в срок.
- Архитектор программного обеспечения проектирует большие красивые сложные системы, которые внедрять нецелесообразно.
- Даже мельчайшие изменения в спецификации, документе или плане проекта должны пройти трудоемкий процесс управления изменениями. Вместе с тем люди находят способы, чтобы обойти его, внося радикальные, крупномасштабные изменения в систему заявок на техподдержку или отслеживание ошибок.
- Проект не успевает завершиться в срок, поэтому руководитель добавляет людей в команду в последние несколько недель. Вместо ускорения выполнения проекта этот шаг создает путаницу и хаос*.

Вспомните ваши собственные проекты, которые столкнулись с проблемами из-за заведомо глупых действий. Возможно, это правила, которые пришлось выполнять (или обходить), установленные вашим руководителем или являющиеся частью культуры компании. А может быть — ненужные дедлайны, произвольно назначенные, чтобы мотивировать вас.

Эти вещи неслучайны. Найдите время поразмыслить над определениями «муда», «мура» и «мури». Обдумайте список знакомых проблем проекта, а также тех, с которыми вы сталкивались сами. Сможете ли вы сопоставить каждую из них с одной из этих трех категорий. Были ли действия, которые пришлось выполнять, тщетными, бесполезными или лишними? Это **муда** — работа, которую вы должны делать, хотя она не добавляет

* В книге *Mythical Man Month* Фред Брукс предложил нам так называемый закон Брукса: «Добавление людских ресурсов в конце проекта затягивает его выполнение». Подумайте об этом, читая о муда, мура и мури.

ценности. Случалось ли вам сидеть, ожидая кого-то, чтобы получить работу? Это **мúra** — неравномерность, или работа, происходящая урывками. А как насчет сверхурочных или работы по выходным, потому что вам надо было сделать больше, чем позволяют человеческие силы? Это **мúри** — перегрузка, или ожидание необоснованных либо невозможных вещей.

Хотя есть много различий между разработкой программного обеспечения и производством автомобилей, Поппендик признали, что идеи бережливого производства имеют место в программных проектах. Поэтому логично предположить, что если команды разработки ПО сталкиваются с проблемами, похожими на те, что встречаются в промышленном производстве, то и решения, подходящие для производителей автомобилей, будут полезны и для команд программного обеспечения. В промышленности таким решением является вытягивающая система (также известная под названием «точно-в-срок»).

В 1950-е годы производственный коллектив Toyota возглавлял Тайити Оно. Он признавал, что трудно предугадать заранее, какие запчасти будут в дефиците, — часто оказывалось, что это самые разные детали. В этом заключалась первопричина потерь (**мúда**, **мúra** и **мúри**). Поэтому они придумали систему, при которой на каждой сборочной линии специальные станции сигнализировали о готовности к большому количеству деталей группе, созданной для их поставки на основе этих сигналов. Каждая станция должна иметь небольшие очереди из запчастей, в которых нуждается производство. Вместо того склада, который выталкивал запчасти на линию, появилась линия, вытягивающая детали и бравшая их только тогда, когда запас в очереди сильно снижался.

Это называется *вытягивающая система*, потому что она состоит из отдельных независимых групп или членов команды, которые вытягивают лишь нужные им запчасти (вместо большого регулярно пополняемого запаса деталей, проталкиваемых к ним). Toyota и другие автопроизводители обнаружили, что процесс сборки стал значительно быстрее и дешевле, потому что избавился от огромных потерь и времени на ожидание. Действительно, каждый раз, когда определенный элемент потерь выявлялся, им удавалось сделать небольшое изменение в процессе с целью его ликвидации.

Системы вытягивания очень полезны для создания программного обеспечения — именно по той же причине, по которой они нужны на производстве. Вместо того чтобы пользователи, менеджеры либо владельцы продуктов проталкивали задачи, функции или запросы в направлении команды разработки, они добавляют их в очередь, из которой команда сама их вытягивает. Когда работа возвращается, вызывая неравномерность в середине проекта, они могут создать буфер, чтобы сгладить ее. Команда способна поддерживать несколько различных очередей и буферов в течение проекта. И как оказалось, это очень эффективный способ сокращения времени

ожидания, удаления потерь и оказания помощи пользователям, менеджерам, владельцам продуктов и командам разработки в принятии решения о том, какое программное обеспечение уже готово.

Вот очень простой пример того, как система вытягивания может решить знакомую проблему: команде разработчиков ПО приходится ждать, пока каждая функция будет описана в большой спецификации, которая затем должна пройти через сложный процесс обзора и согласования. Может быть, этот процесс — способ получения всех перспективных мнений от каждого человека либо попытка спасти собственную шкуру, когда руководители и заинтересованные стороны боятся взять на себя ответственность. Это также может быть частью культуры компании, которая всегда так поступала, не думая о расточительности подобного поведения. Как бы выглядел этот процесс, если бы мы заменили его простой системой вытягивания?

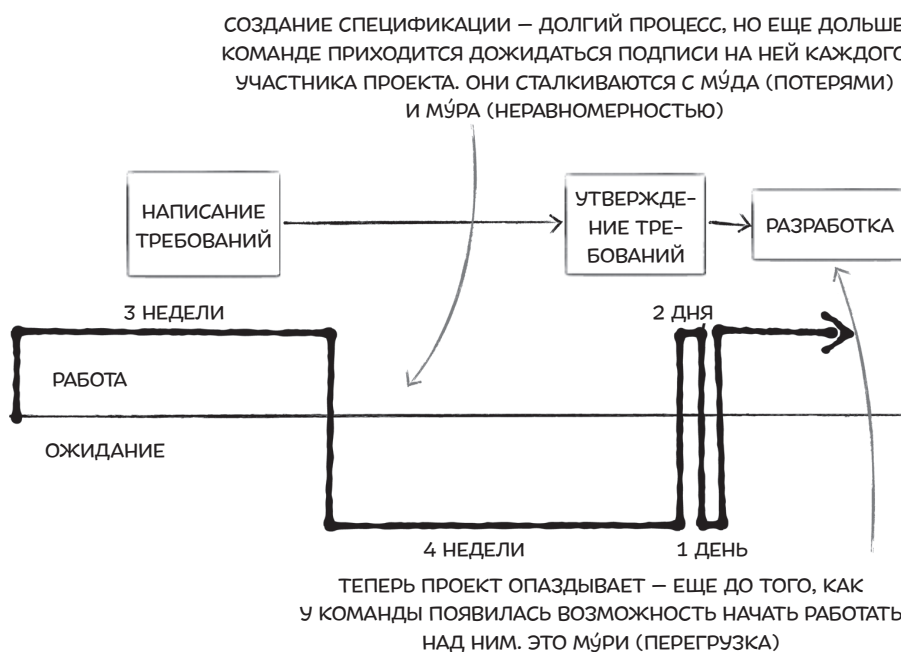


Рис. 8.10. Эта карта потока ценности показывает потери, которые происходят, когда вся команда ждет большой спецификации, которая должна быть создана и утверждена

Это очень знакомая проблема, и в реальном мире многие команды нашли способы обойти ее. Например, дизайнеры и архитекторы могут сделать предпроект, основанный на раннем черновике спецификации и их лучших предположениях о том, что они будут создавать. Но время, которое команда

тратит на поиск способов устранения потерь, она могла бы использовать гораздо продуктивнее. Она по-прежнему будет нести потери, часто и много, особенно если ее догадки окажутся неверными и ей придется отменить некоторые из предпроектных решений.

Система вытягивания — это лучший способ удалить неравномерности и предотвратить перегрузки.

Первым шагом в создании системы вытягивания должно стать разделение работы на маленькие вытягиваемые куски. Таким образом, вместо создания большой спецификации команда может разбить ее на минимальные маркетинговые функции — скажем, отдельные пользовательские истории и, возможно, небольшие фрагменты документации, сопровождающей каждую историю. Затем эти истории будут утверждены по отдельности. Как правило, когда процесс просмотра и согласования спецификации затягивается, причина в том, что люди имеют проблемы с некоторыми функциями. (Способны ли вы понять, как разделение работы на более мелкие ММФ дает команде больше возможностей? Это вариантное мышление.) Утверждение индивидуальных ММФ должно привести к быстрому получению одобрения как минимум для нескольких функций. Как только первая ММФ утверждена, команда может начать работать над ней. Теперь не нужно строить предположений. Вместо этого начинается реальное обсуждение той работы, которую требуется выполнить. Процесс утверждения может иметь под собой реальные основания (например, нормативное требование регулятора или подлинную необходимость узнать точку зрения каждого), теперь команда может получить реальный сигнал начать работу.

Вот так все принципы бережливого мышления — видение целого, выявление потерь и создание системы вытягивания для устранения неравномерности и чрезмерных нагрузок — могут собраться вместе, чтобы помочь команде усовершенствовать работу. Но это только начало. В следующей главе вы узнаете о том, как применить *lean*-мышление, чтобы улучшить способ, при помощи которого команда создает программное обеспечение.



Ключевые моменты

Минимальная маркетинговая функция (ММФ) — самый маленький «кусочек» ценности или функциональности, который команда способна поставить, например пользовательская история или запрос пользователя — то, что может закрыть пункт в бэклоге владельца продукта.

Карта потока ценности — это визуальный инструмент, помогающий *lean*-команде реально увидеть, как работает проект, показывая

весь жизненный цикл ММФ, в том числе время работы и время ожидания на каждом этапе.

Понимание **первопричин** проблемы поможет вам увидеть ее в целом. Метод **пяти «почему»** — эффективный способ сделать это.

Полезный инструмент, помогающий команде **поставлять как можно быстрее**, — **WIP-диаграмма**, визуальный инструмент, который показывает, как ММФ следуют через поток создания ценности проекта.

Есть три важных вида потерь, которые ограничивают рабочий процесс: **мұда** (потери), **мұра** (неравномерность) и **мұри** (необоснованность и невозможность).



Часто задаваемые вопросы

Информация о том, как выполнять проекты, кажется полезной, но мне до сих пор трудно понять, как это влияет на мою повседневную работу. Как может Lean помочь в моей работе?

Обычно нам не нравится слушать теоретические рассуждения — когда говорят нечто подобное, это обычно признак того, что не видят здесь возможности немедленного практического применения. Однако во многом это на самом деле правда о Lean, так как Lean — это мышление. Оно не включает практик, которые ваша команда могла бы применять ежедневно, как Agile-манифест или Scrum и XP-ценности. Но, как и Agile-манифест, Scrum и XP-ценности, Lean является правильным мышлением, очень ценным для команды, если вы хотите *лучше писать* программное обеспечение.

В главе 2 мы ввели идею раздробленного видения. На протяжении всей книги вы видели много примеров, как это приводит к тому, что команда либо получает результат «лучше-чем-ничего», либо (в худшем случае) приходит к полному провалу проекта. Бережливое мышление помогает вам взглянуть с высоты птичьего полета не только на проект, но и на всю команду, компанию, ее правила, политику и культуру, которые вызывают у вас серьезные проектные проблемы.

Вернемся в начало этой главы и посмотрим на lean-ценности еще раз. Эти ценности помогут взглянуть на ту работу, которую вы делаете прямо сейчас.

Как только вы начинаете искать потери, вы замечаете их повсюду («устраняйте потери»). Вы перестаете видеть программу как набор отдельных задач и начинаете воспринимать ее как систему («видеть целое»). Вы будете использовать такие инструменты, как «пять “почему”», чтобы выйти за пределы фиксации отдельных проблем в работе, которую сделала команда, и распознать системные проблемы, влияющие на проект снова и снова («усиление обучения»). Каждая итерация изменяет способ, которым вы смотрите на проект, команду и компанию. Это поможет вам увидеть более широкую перспективу.

В главе 9 вы узнаете, как обзавестись этой новой точкой зрения и использовать ее для реальных, постоянных изменений, чтобы улучшить создаваемое вами программное обеспечение.

Вы сказали, что «мүри» может переводиться как «невозможно», но разве это не отрицание? Ведь считается, что все возможно при наличии мотивированной команды, верно?

Если бы только это было правдой. Проведем мысленный эксперимент, чтобы показать вам, что некоторые вещи действительно невозможны.

Представьте себе, что СЕО вашего небольшого стартапа заходит в офис и говорит, что ваш самый крупный клиент заиклен на Бруклине. Руководитель кладет пакет зубочисток и палочек для мороженого на стол и сообщает: если ваша команда сможет сделать идеальный макет Бруклинского моста из зубочисток и палочек для мороженого к моменту появления клиента, то тот утроит бизнес компании и сделает вас богатыми. Если же работа не будет выполнена на отлично, то он уйдет к конкуренту, и тогда вашей компании конец.

Клиент будет здесь менее чем через час. Все за дело! Нет ничего невозможного! Вы сможете сохранить вашу компанию, верно?

Если вы не инженер-строитель, который к тому же виртуозно управляет с клеем, то вы потерпите неудачу. Некоторые задачи невозможно выполнить, несмотря ни на какую мотивацию.

Есть люди, умудряющиеся убедить свои команды, что мотивация — это все, что нужно для достижения цели. Управление такого типа (то, что мы называем магическим мышлением) опасно. Когда вы работаете в команде, которая действует так, словно вы и впрямь на многое способны, вы будете регулярно ожидать того, чего невозможно достичь. Часто это происходит из-за нереалистичных или неразумных сроков. Но иногда проблема в технической неосуществимости (или, по крайней мере, невозможности обойтись без очень большого объема

работы — как в том случае, когда весь ваш код находится в ужасной форме и «с душком»).

Это *мүри*. Еще раз прочтем определение, которое мы дали ранее: «неразумность, невозможность, перегруженность, излишняя сложность, пересиливание, принуждение, превышение, чрезмерность».

Бережливое мышление помогает определить *мүри* и понять, что любое усилие, которое вы тратите, пытаясь сделать невозможное, — это потери. Самый эффективный способ устранить их — это в первую очередь исключить магическое мышление, которое побуждает стремиться к невозможному.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с вашей командой).

- Определите все ММФ в вашем проекте. Как они достигаются? Вы пишете истории пользователей на карточках или стикерах и помещаете их на доску задач? Есть ли у вас индивидуальные требования в спецификации? Найдите большую функцию или историю и решите, сможете ли вы разбить ее на более мелкие куски.
- Ищите потери в вашем проекте. Запишите примеры *мүда*, *мүра* и *мүри*.
- Найдите любые ММФ, которые ваша команда уже завершила, и создайте карту потока ценности для них. Хорошо, если вы сможете собрать всю команду, чтобы сделать это вместе. Затем создайте карту потока ценности для другой ММФ. Каковы сходства и различия между двумя потоками ценности?
- Найдите узкое место, на которое ваша команда регулярно натывается. (Карты потока ценности могут оказаться для этого очень полезными.) Организуйте дискуссию о том, как вы можете облегчить эту проблему в будущем.
- Посмотрите на даты, за которые вы и ваша команда сейчас отвечаете. Вы действительно взяли на себя те обязательства, про которые думаете, что вы их взяли? Есть ли варианты поставить что-то другое, одновременно выполнив эти обязательства?



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше о Lean, ценностях бережливого мышления и картах потока ценности в книге Мэри и Тома Поппендик *Lean Software Development: An Agile Toolkit* (Addison-Wesley, 2003).
- Узнайте больше о Lean и картах потоков создания ценности в книге Алана Шэллоувея, Гая Бивера и Джеймса Тротта *Lean-Agile Software Development: Achieving Enterprise Agility* (Addison-Wesley, 2009).
- Вы сможете узнать больше о ММФ, о том, как разбить или разложить пользовательские истории, в книге Майка Кона «Пользовательские истории. Гибкая разработка программного обеспечения» (М. : Вильямс, 2012).
- Узнайте больше о возможностях вариантного мышления в романе об управлении рисками проекта Олава Маасена, Криса Матса и Криса Хиппаху *Commitment* (Hathaway te Brake Publications, 2013).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Большинство людей, работающих в командах разработки программного обеспечения, никогда не сталкивались с мышлением, имеющим собственное название. Помогая им понять, что Lean — это мышление, а не методология, вы сделаете первый шаг к его принятию командой.
- Привычка говорить о потерях — одна из самых трудных частей бережливого мышления. В то время как большинство коучей соглашались, что команды должны оставаться позитивными, «выяснение отношений» может оказаться полезным, когда дело доходит до помощи команде научиться определять потери (особенно когда речь идет о неразумности и невозможности).

- Вы можете вести дискуссию о потерях в положительном ключе, помогая найти примеры того, что можно считать потерями с точки зрения *проекта*, но не *компании*, для которой это жизненно важные вещи. Это поможет вести вашу дискуссию эмоционально верно, не говоря о потерях в негативном смысле.
- Часть вашей работы как коуча состоит в том, чтобы избегать трений между командой и компанией. Если имеются случаи, когда даже обсуждение проблем с топ-менеджерами может привести к серьезным последствиям для команды, то внедрение Agile способно вызвать более серьезные проблемы. Максимально *аккуратно* работайте с менеджерами, чтобы помочь им признать, что они используют магическое мышление.
- Еще один способ оставаться позитивным — это помогать обеим сторонам (и командам, и менеджерам) отделять процессы или системы от людей, работающих в них. Потери, неэффективность, обратная связь — это про то, что нужно сделать с процессом, а вовсе не суждения о людях, выполняющих свою работу.

Канбан, поток и постоянное совершенствование

Канбан — это не методология разработки программного обеспечения, прием описания жизненных циклов или подход к управлению проектами. Вам необходимо наличие какого-то базового процесса, к которому можно применить Канбан, для постепенного изменения и улучшения этого процесса.

Дэвид Андерсон. Канбан. Альтернативный путь в Agile*

Канбан — это метод улучшения процессов, используемых гибкими командами. Команды, применяющие его, начинают понимать, как они создают программное обеспечение, и постепенно улучшают его. Канбан, так же как Scrum и XP, затрагивает образ мышления человека. В частности, он требует бережливого мышления. Мы уже узнали, что Lean — это образ мышления, ценности и принципы. Команды, использующие Канбан, начали с применения мировоззрения Lean. Это обеспечивает прочный фундамент, который в сочетании с Канбаном дает возможность улучшить процессы. Когда команды используют Канбан для усовершенствования, они сосредоточены на устранении потерь из процессов (в том числе муда, мура и мури — потери неровности, перегрузки и бесполезности, о которых мы узнали в главе 8).

Канбан — это технологический термин, адаптированный Дэвидом Андерсоном для разработки программного обеспечения. Вот как он описывает взаимоотношения с Lean в своей книге «Канбан. Альтернативный путь в Agile»: «Канбан-метод представляет собой сложную адаптивную систему, предназначенную для активации lean-решений в рамках организации». Есть команды, которые применяют Lean и бережливое мышление для разработки программ без использования Канбана, но на сегодняшний день это наиболее распространенный метод (и эффективный для многих agile-практиков) внедрения бережливого мышления в организацию.

* Андерсон Д. Канбан. Альтернативный путь в Agile. М. : Манн, Иванов и Фербер, 2017.

Канбан отличается от гибких методологий, таких как Scrum и XP. Scrum преимущественно ориентирован на управление проектами: объем работы, который должен быть проделан, чтобы эта работа была выполнена, и результат, необходимый пользователям и стейкхолдерам. XP ориентирована на разработку программного обеспечения, ее ценности и практики строятся вокруг создания благоприятных условий для развития и формирования привычек, помогающих разработчику писать простой и легко изменяемый код.

Канбан *помогает команде улучшить способы разработки программного обеспечения*. Команда, использующая этот метод, имеет четкое представление о том, какие действия совершает при создании программного продукта, как взаимодействует с остальной компанией, как образуются потери, вызванные неэффективностью и неравномерностью, и каким образом со временем исправить ситуацию, устранив основную причину этих потерь. Когда команда улучшает способ создания ПО, это традиционно называется *процессом усовершенствования*. Канбан — хороший пример применения гибких идей (таких как последний ответственный момент) для создания метода усовершенствования процесса, который команды могут легко принять.

Практики Канбана позволяют стабилизировать и улучшить систему разработки программного обеспечения. Актуальный список основных практик можно найти в группе Kanban Yahoo!

Во-первых, следуйте основополагающим принципам:

- Начните с того, что вы делаете сейчас, уважайте имеющиеся роли, обязанности и должностные инструкции.
- Договоритесь об эволюционном развитии.
- Поощряйте лидерство на всех уровнях.

Затем принимайтесь за основные практики:

- визуализацию;
- ограничение числа задач в работе (WIP);
- управление потоком;
- сделайте правила явными;
- введите петли обратной связи;
- развивайтесь совместно и экспериментируйте (используя моделирование или научный подход).

На начальном этапе не требуется внедрять все шесть практик. Частичная реализация признается поверхностной и предполагает постепенное

углубление, поскольку большинство практик принимаются и выполняются с большим потенциалом.

В этой главе вы узнаете о Канбане, его принципах, взаимоотношениях с Lean и практиках. Мы расскажем, как сосредоточение на потоке и теории массового обслуживания поможет вашей команде внедрить бережливое мышление в практику и создать культуру постоянного совершенствования.



Описание: команда, работающая над созданием приложения для камеры мобильного телефона в организации, купленной большой и многопрофильной интернет-компанией

Кэтрин — первый разработчик

Тимати — второй разработчик

Дэн — их руководитель

Акт II. Игра в догонялки

Дэн раздражал Кэтрин и Тимати. Они понимали, что есть сроки и выполняемая работа важна. Они даже не возражали против давления, которое он оказывал. Но казалось, что каждый, даже самый маленький проект завершался «в режиме интенсивной терапии», как называл это Дэн, когда начал заниматься микроменеджментом.

Текущий проект не отличался от остальных. Они работали над новой функцией приложения для камеры телефона, которая превращает лица друзей в старые плакаты «Разыскивается». Предполагалось, что это будет простой проект, который интегрирует их приложение с системой социальных сетей материнской компании.

Как обычно, во время тестирования обнаружилось множество ошибок, что заставило их пропустить дедлайн.

— Я уверен, что у нас не было бы проблем со сроками, если бы он не мешал нам работать, — сказал Тимати.

— Я знаю, что ты имеешь в виду, — добавила Кэтрин. — Это его вмешательство в каждую мелочь. Мы говорим о вмешательстве после семи вечера, когда на улице уже темнеет. Мы опаздываем на вечерний статус-митинг.

Это был «решающий момент», как называл это Дэн, имея в виду, что есть дедлайн, в который они не могут уложиться, но это означало, что аврал был всегда.

Кэтрин и Тимати вошли в кабинет Дэна и сели. Другие члены команды были уже там, и никто не выглядел счастливым. Дэн настроился на микроменеджмент.

— Итак, последние три проекта протекают в режиме «интенсивной терапии». Тим, Кэтрин, начнем с вас.

— Мы делаем успехи... — начала было Кэтрин.

Дэн прервал ее:

— У вас нет прогресса. Этот проект разваливается. Я дал вам достаточно времени для выбора собственного метода работы, а теперь мы должны действовать *правильно*.

Тимати возразил:

— Но там была ошибка, которую требовалось исправить.

— Вам всегда кажется, что есть ошибка. Просто вы, ребята, неспособны выполнить проект в срок. Мне придется погрузиться в процесс. Я подозреваю, что вы не понимаете всей его важности.

— Пойдите! — воскликнула Кэтрин. — Ведь нет ничего удивительного в том, что в конце всегда возникают проблемы. В ходе работ накапливается много потерь.

— Что ты понимаешь под словом «потери»? — поинтересовался Дэн. — Это звучит негативно. Если мы хотим добиться успеха, то нужно оставаться позитивными. (Дэн всегда говорил о позитивном настрое, даже когда ругал своих сотрудников.)

— Ну, например, согласование любого изменения в пользовательском интерфейсе занимает недели. В середине обсуждения вы отдаете распоряжение приступить к разработке, и, похоже, мы тратим больше времени на изменение кода, чем на его написание.

— Правильно. И мы всегда удивляемся, когда команда тестировщиков находит ошибки. Почему-то это происходит постоянно, но мы никогда не находим времени, чтобы их исправить, — добавил Тимати.

Дэн начал злиться:

— Смотри, это такой стиль управления программными проектами. Перестань указывать пальцем на людей и обвинять их. Виноват я или команда тестировщиков.

— Слушай, Дэн! Хватит... — не сдержалась Кэтрин. Все удивленно взглянули на нее: она почти никогда не повышала голоса. — Мы никого не виним.

Но есть проблемы, которые происходят снова и снова. Мы встречаемся два раза в день, и наши разговоры звучат как заезженная пластинка. Приходится бесконечно обсуждать одни и те же проблемы, которые почему-то всякий раз вызывают удивление.

Дэн немного опешил от такого напора. Он встал, посмотрел на Кэтрин, затем снова сел в свое кресло.

— Знаете, все это очень важно! Прямо сейчас найдем время и займемся этой проблемой. Это решающий момент.

Принципы Канбана

Давайте подробнее рассмотрим основополагающие принципы Канбана.

- Начните с того, что вы делаете сейчас, уважайте имеющиеся роли, обязанности и должностные инструкции.
- Договоритесь об эволюционном развитии.
- Поощряйте лидерство на всех уровнях.

Первый принцип — начните с того, что вы делаете сейчас, — отличается от всего того, о чем вы читали в этой книге.

Мы потратили много времени на сравнение гибких методологий с традиционными водопадными проектами. Например, Scrum дает полную систему для управления и реализации проектов. Если вы хотите внедрить Scrum, то нужно создавать новые роли (scrum-мастер и владелец продукта) и новые виды деятельности (планирование спринта, ежедневные scrum-митинги, доски задач). Это необходимо, поскольку система Scrum предназначена для управления проектами и поставки программного обеспечения.

Канбан *не является* системой управления проектами. Это метод улучшения процесса: шаги, которые команда принимает для создания и поставки ПО. Прежде чем вы сможете что-нибудь улучшить, нужна отправная точка, а для Канбана это как раз то, что вы делаете сегодня.

НАЙДИТЕ ОТПРАВНУЮ ТОЧКУ И НАЧНИТЕ ЭКСПЕРИМЕНТАЛЬНОЕ РАЗВИТИЕ

Привычное дело — думать о типичных проблемах.

Когда проектная команда делает то, что в итоге приведет к ошибкам и срыву сроков, это не похоже на ошибку. Позднее вы можете заняться анализом первопричин, ведь, столкнувшись с точно таким же выбором, команда, скорее всего, примет то же самое решение. Таковы люди.

Предположим, что команда всегда предоставляет программное обеспечение клиентам только после многократных и непростых встреч с ними, во время которых пользователи таки и не могут найти обещанных функций. Конечно, не исключено, что эти разработчики невероятно рассеянны и всегда забывают об одной-двух функциях, которые обсуждали с клиентами. Но более вероятно, что их преследуют одни и те же проблемы в процессе выяснения требований пользователей.

Главная цель процесса улучшения состоит в поиске повторяющихся проблем, выяснении их общности и создании инструмента исправления.

Ключевая здесь вторая часть предложения: выявление того, что общего у этих проблем. Если вы предполагаете, что разработчик не может вспомнить все то, о чем просили пользователи, или они постоянно меняют свои требования, то вы придете к выводу, что эти проблемы невозможно решить. Но если предположить, что существует реальная и к тому же повторяющаяся причина, то есть шанс справиться с ситуацией.

Вот с чего начинается Канбан: взгляните на то, как вы работаете, и представьте свою текущую деятельность в виде совокупности заменяемых, повторяющихся действий. Канбан-команды называют правила, которым они всегда следуют, **стратегией**. По существу, это сводится к признанию привычек, тех шагов, которые делаются при создании программного обеспечения и их фиксации.

Иногда сложно записывать правила, которым следует команда, потому что легко попасть в ловушку, если судить о результатах проекта по команде в целом или отдельному ее участнику: если проект успешен, то все члены команды должны быть профессионалами, а если нет — то они явно некомпетентны. Это несправедливо, потому что предполагается, что в проекте все зависит от команды. Бережливое мышление помогает преодолеть это заблуждение, рассказывая, как увидеть картину в целом, что в данном случае означает *увидеть большую систему целиком*.

Стоит повторить, что *каждая команда имеет производственную систему для создания программного обеспечения*. Она бывает хаотичной и часто меняется. В основном она живет в головах членов команды, которые никогда не обсуждали, насколько она велика и что они придерживаются именно ее. Для тех, кто использует Scrum, система кодифицирована и понятна. Но многие команды следуют системе, существующей как некое «врожденное» знание: мы всегда начинаем проект беседой с представителями заказчика или составления графика, создания карт с пользовательскими историями либо сразу «погружаемся» в код после встречи с менеджером.

Именно с такой системой Канбан начинает работать. У команды уже есть способ запустить свой проект, а Канбан нужен, чтобы лучше понять систему. То есть начните с того, что вы делаете сейчас. Цель Канбана — сделать небольшие улучшения в этой системе. Вот что значит добиваться постепенных, эволюционных изменений. И именно поэтому Канбан имеет практики **совместного улучшения и экспериментального развития**. В бережливом мышлении необходимо видеть все целиком, чтобы понять, что требует измерений. А они составляют сердцевину эксперимента и научного подхода. Канбан-команда начнет с исследования того, как производится программное обеспечение, и получит объективное понимание ситуации. Затем она проведет определенные изменения *как команда* (в этой главе вы узнаете, как именно эти изменения работают) и проверит свои измерения на предмет соответствия желаемому эффекту.

Lean-ценность усиливает обучение и также является важной частью развивающейся системы, которую команда использует для сборки программного обеспечения. В этой книге вы узнали о петлях обратной связи. Когда вы сотрудничаете для измерения системы и экспериментального развития, петли обратной связи становятся очень важным инструментом сбора информации и подачи ее обратно в систему. Канбан-практика **внедрения циклов обратной связи** должна помочь вам увидеть тесную связь между Канбаном и Lean.

Ускоренное обучение — это также фактор канбан-принципа, который требует уважения текущих ролей и обязанностей участников. Предположим, что каждый проект начинается встречей руководителя проекта, бизнес-аналитика и программиста. Может быть, у них нет запротоколированных правил проведения таких бесед, но, скорее всего, эти правила легко понять, ознакомившись с названиями должностей.

Канбан уважает текущие роли и обязанности, потому что все это важная часть системы.

Объединяет все эти принципы то, что Канбан работает только для тех команд, которые не жалеют времени, чтобы понять свою собственную систему сборки программного обеспечения.

Если бы существовал один правильный способ сборки ПО, то все бы просто использовали его. Но мы в главе 2 этой книги утверждали: серебряной пули нет — не существует набора «лучших» практик, гарантирующих сборку программы в каждом конкретном случае. Даже та команда, которая уже имеет опыт успешного применения практики в проекте, может с треском провалиться в следующем. Поэтому работа с Канбаном начинается с понимания настоящей системы запуска проекта: как только вы увидите систему целиком, Канбан предлагает соответствующую практику по ее улучшению.

ТАК, ВЫХОДИТ, КАНБАН НЕ ПОДСКАЖЕТ, КАК МНЕ РЕАЛИЗОВАТЬ ПРОЕКТ?

Не подскажет*. Канбан предлагает начать с понимания того, как вы в настоящее время реализуете свои проекты. Это может быть Scrum, XP, «лучше-чем-ничего», эффективный водопадный процесс, неэффективный водопадный процесс или даже беспорядочный метод научного тыка или кодирования «с наскока». Как только вы выясните, каким образом ваша команда создает программное обеспечение, Канбан предложит практики для улучшения.

Если у вас уже есть процесс разработки программного обеспечения, то зачем вам Канбан?

Большинству команд удастся *что-нибудь* поставлять. Вы знаете, сколько усилий тратит команда? Какой ценностью обладает результат их работы? Или он вызывает проблемы и затрудняет поставку ценного программного продукта?

Если есть система — неважно какая, — то большинству такие вопросы могут даже не приходиться в голову. Даже если вы используете Scrum или XP, вы можете терять впустую много времени, не подозревая об этом. Привычные проблемы очень трудно обнаружить. Каждый может следовать правилам и делать все верно. Но так же как поведение формируется самой системой, потери складываются из совместной работы многих людей.

Такой пример приведен в главе 8: команда невольно увеличивает производственный цикл, даже если все активно трудятся и никто намеренно не затягивает работу. Но несмотря на это, заказчику кажется, что проект сильно задерживается. Хотя в команде даже не заметили этого, потому что делают все возможное, чтобы выполнить задачу как можно скорее.

Это и есть та проблема, которую решает Канбан.

ИСТОРИИ ЗАХОДЯТ В СИСТЕМУ, А НА ВЫХОДЕ ПОЛУЧАЕМ КОД

Системное мышление воспринимает организацию как систему: анализирует то, как ее части взаимосвязаны и как организация работает в целом.

Том и Мэри Поппендик. Lean Software Development: An Agile Toolkit

Первый шаг к улучшению системы — признание того факта, что она существует. Идея *видеть целое* лежит в основе принципа Lean. Когда вы видите цельную картину, то перестаете думать о команде как об отдельном,

* Канбан — это не способ управления проектом, что, конечно, не означает, будто Канбан не для менеджеров проектов! Дэвид Андерсон выпустил серию публикаций, которые объясняют, как менеджеры проектов могут применять Канбан в своей практике. Мы рекомендуем почитать его публикации, чтобы узнать об этом больше.

разобленном решении и начинаете воспринимать ее как единую систему. В Lean это называется **системное мышление**.

Каждая система принимает входные сигналы и превращает их в готовую продукцию. Как выглядят scrum-команды с точки зрения системного мышления? Вы можете думать, что Scrum — это система, которая берет задачи бэклога проекта на входе, а на выходе выпускает код. Многие scrum-команды имеют бэклог проекта, который состоит исключительно из историй. Такие команды считают себя «машинами», которые превращают истории в код.

Но очевидно, что они остаются командами, и мы предпочитаем рассматривать их участников как людей, а не роботов. Однако полезно думать о задаче, которую вы выполняете, как о части большой системы. Если применить системное мышление к scrum-команде, то становится ясно: выполняемая работа напрямую (или даже косвенно) способствует превращению историй в код. Признавая, что все в Scrum — система, вы поймете, как начать работать лучше и вносить усовершенствования. Такое мышление приводит к улучшениям, таким как дополнительные вопросы Джеффа Сазерленда для ежедневных scrum-митингов, о которых вы узнали в главе 5. Это хороший пример системного мышления, применяемого к Scrum, что приводит к эволюционному усовершенствованию.

Канбан призывает начать с понимания системы, которую использует ваша команда. И даже если вы не будете следовать какой-то определенной методологии, вы все равно сможете применять системное мышление в своей команде для выяснения способа работы.

КАЖДАЯ КОМАНДА РАЗРАБОТЧИКОВ СЛЕДУЕТ СИСТЕМЕ, ДАЖЕ ЕСЛИ ОНИ ОБ ЭТОМ НЕ ЗНАЮТ

Легко понять, что команда, исповедующая Scrum, использует систему. Но что если ваша команда не применяет подобные методы? Может быть, вы просто погружаетесь в работу над программным обеспечением? Выглядит все так, будто вы каждый раз запускаете проекты по-разному. Поэтому вы не имеете системы, правда?

Но самое забавное, что люди — особенно команды — *всегда* стремятся следовать правилам. Они могут быть не записаны, и мы нередко придумываем их, если они не существуют. Но правила существуют на интуитивном уровне, и если они засели в голове, то от них трудно избавиться. Даже когда вам кажется, что вы действуете спонтанно, появление нового человека в команде все расставит на свои места: вы очень быстро обнаружите, что он нарушает привычные для вас правила.

Lean предлагает инструмент, чтобы принять неписанные правила и превратить их в систему: систематизирование потока создания ценности.

Когда вы берете ММФ (минимальную маркетинговую функцию, о которой мы узнали в главе 8, — например, историю, требование или пользовательский запрос) и выявляете систематизированный поток ценности, ведущий к написанию кода, вы должны записать путь через вашу систему.

Если вы работаете в команде, которая придерживается неписанных правил, то вполне вероятно, что одна ММФ сильно отличается от другой. Но поскольку люди интуитивно следуют правилам, довольно вероятно, что вы можете наметить небольшое количество потоков создания ценности, охватывающих основные ММФ, которые ваша команда превращает в код. Если вам удастся сделать это, то вы сможете создать очень точное описание системы, которой следует ваша команда. Прежде всего система решает, какой поток создания ценности ММФ будет из нее вытекать.

Полезно иметь систему, которая работает одинаково для всех, даже если есть много различных путей для ММФ. Как только вы поймете, как работает система (то есть увидите целое), вы сможете принимать решения о том, какие пути расточительные, и заниматься постепенными изменениями подхода к разработке.

КАНБАН — ЭТО НЕ МЕТОДОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И НЕ СИСТЕМА УПРАВЛЕНИЯ ПРОЕКТАМИ

Одна из самых распространенных ошибок в начале изучения Канбана — это попытка использовать его как методологию для разработки программного обеспечения. Это неправильно. Канбан — это способ улучшения процессов. Он поможет вам понять методику, которую вы используете, и найти способы усовершенствовать ее.

Выделите немного времени и пролистайте книгу на несколько страниц вперед. Вы увидите рисунки, похожие на доски задач. Но на самом деле это канбан-доски. На них нет задач, зато есть **рабочие элементы**. Так называется самостоятельная единица работы, которую нужно отследить через всю систему. Как правило, она больше, чем ММФ, требования, пользовательские истории или подобный фрагмент общего пула работ.

Существует разница между доской задач и канбан-доской: в то время как задачи перемещаются по доске задач, рабочие элементы не являются задачами. Задачи — это то, что делают люди, чтобы переместить рабочие элементы в рамках системы, то есть это «винтики» механизма, который

толкает рабочие элементы. Именно поэтому вы можете использовать системное мышление для понимания рабочего процесса при создании программного обеспечения, не унижая человеческое достоинство разработчиков мнением, будто они части некой машины. Задачи — это «винтики», а люди — уникальные личности с собственным характером, желаниями и мотивацией.

Столбцы на канбан-доске могут показаться похожими на этапы в потоке ценности. Однако многие канбан-эксперты разделяют понятия «систематизация потока ценности» и «канбан-доска». Они отражают на доске состояние рабочих элементов в рабочем процессе независимо от потока ценности и называют это **картой жизненного цикла**. Разница в том, что поток создания ценности — инструмент бережливого мышления, помогающий понять работу системы, а карта жизненного цикла — это то, как канбан-метод определяет конкретные этапы, которые проходит каждый рабочий элемент. (Дальше в этой главе вы узнаете, как построить канбан-доску.)

Вот пример того, как доска задач делает одно, а канбан-доска другое. Scrum помогает командам в самоорганизации и выполнении коллективных обязательств. К моменту начала использования доски задач команда уже выбрала элементы бэклога (рабочие элементы), которые включены в спринт, и разбила их на задачи. Так как текущие задачи перемещаются по доске задач, рабочие элементы начинают перемещаться из колонки «сделать» в колонку «сделано». Команда воспринимает это как прогресс.

Типичная канбан-доска показывает крупные рабочие элементы, а не отдельные задачи. И хотя на доске задач видны только рабочие элементы с разными статусами («сделать», «в процессе» или «сделано»), канбан-доска будет иметь широкую картину. Откуда берутся рабочие элементы? Как владелец продукта узнает, что рабочий элемент размещен в бэклоге проекта, и как расставляются приоритеты? После того как команда закрывает рабочий элемент, как она может отследить правильность его разворачивания? Рабочий элемент имеет большой жизненный цикл, выходящий за пределы создающей его команды. Канбан-доска будет иметь колонки для этапов, которые проходит рабочий элемент до того, как scrum-команда получит его, и после завершения работы над ним.

Более того, канбан-доска поможет показать проблемы, которые никогда не появляются на доске задач. Многие scrum-команды очень сильны в создании программного обеспечения, о котором их просят пользователи, но разочарованные клиенты все-таки остаются. Возможно, они создают рабочие элементы, не удовлетворяющие потребностям пользователей. Или между созданием рабочих элементов и процессом обзора результатов

либо развертывания наблюдается слишком большой временной промежуток. Хотя эти проблемы вне контроля команды, часто обвиняют именно ее. Канбан-доска поможет решить этот вопрос.

Таким образом, хотя Канбан — это не система для управления проектами, он имеет очень важную связь со способом управления проектом, который применяет команда. Канбан направлен на улучшение и изменение процесса, используемого в проекте, и на то, как это может повлиять на управление им. Как правило, Канбан применяют для повышения предсказуемости потока, и это влияет на планирование и план проекта. Широкое использование Канбана и его показателей, скорее всего, окажет существенное воздействие на метод проектного управления*.

Дальше вы узнаете о практиках Канбана и о том, как создавать и использовать канбан-доску, чтобы постепенно усовершенствовать систему *в целом*.

Совершенствование процесса с Канбаном

Мы уже знаем, что Канбан — это метод, который направлен на улучшение процесса, базируется на ценностях Lean и бережливого мышления. Он был разработан Дэвидом Андерсоном, который первым начал экспериментировать с идеями Lean во время работы в Microsoft и Corbis. Так же как Lean, название «канбан» связано с идеями разработки на автомобильных производствах в Японии. Но что делает Канбан гибким? Чем он отличается от традиционного процесса улучшения?

Команды разработки программного обеспечения занимаются совершенствованием процессов с тех пор, как люди начали создавать ПО. В идеале совершенствование процессов работает очень хорошо: команда получает поддержку со стороны руководства, проводит замеры, определяет проблемы, улучшает орудия труда, а затем снова начинает выявлять то, что можно усовершенствовать. В конце концов улучшение всей организации в первую очередь позволяет создавать повторяемые процессы, затем управлять ими и наконец взять под статистический контроль. Уже многие компании заявили о больших успехах в этой области.

Если вы разработчик, уже переживший типичную попытку усовершенствования процессов, то, скорее всего, поспешите отложить книгу в сторону, прочитав это описание.

Термин «усовершенствование процесса» часто вызывает в воображении образ героя комиксов по имени Дилберт. Он ассоциируется

* Мы выражаем благодарность Дэвиду Андерсону за помощь в формулировке этого раздела.

с бесконечными совещаниями и процессами, полными технологической документации, потому что типичное улучшение процесса сильно отличается от идеального. В случае типичного процесса совершенствования крупная компания решает, что их программисты производят программное обеспечение недостаточно эффективно (или что они нуждаются в сертификации процесса для заключения контрактов либо для маркетинговых целей), поэтому нанимают консалтинговые компании, тратят много времени (и денег) на создание блок-схем существующих и желаемых процессов развития и обучают команды использовать новые процессы. Затем команды тратят около 10 минут, чтобы опробовать один из новых процессов, выясняют, что он им не подходит, и отказываются от него. Но поскольку топ-менеджеры спонсировали весь процесс усовершенствования усилий, команда вынуждена делать вид, что использует нововведения, поэтому усердно заполняет любые документы по новым требованиям (например, область применения документа и техническое задание) и создает обязательные артефакты (протоколы собраний для просмотра кода и отчеты о тестировании, которые, как правило, пусты и шаблонны). Для каждого успешного процесса совершенствования усилий (их несколько) есть много ничего не дающих или вовсе неудачных попыток приложения усилий, побочный продукт которых — глубокая неприязнь к термину «улучшение процесса».

Существует одно большое отличие между Канбаном и традиционным процессом улучшения. В последнем случае решения, как правило, спонсируются высшим менеджментом, готовятся комитетом (например, группами процессов разработки программного обеспечения) и доводятся до сведения команд через их непосредственных руководителей. В Канбане улучшение остается в руках команды. Именно поэтому agile-команды добились успеха в применении этого метода. Члены команды самостоятельно находят проблемы в рабочем процессе, предлагают свои варианты улучшения, оценивают результаты и берут на себя ответственность за собственные стандарты.

Как Канбан помогает команде улучшить собственный процесс?

ВИЗУАЛИЗАЦИЯ РАБОЧЕГО ПРОЦЕССА

Первый этап в улучшении процесса — это понимание того, как в настоящее время работает команда, и практика **визуализации** в Канбане позволяет это сделать. Звучит просто, но все гораздо сложнее, чем кажется, — многие традиционные процессы совершенствования идут неправильно.

Представьте, что вы программист, а ваш руководитель приходит к вам и спрашивает: «Как ты ведешь разработку программного обеспечения?» Ваша задача — написать, как вы делаете свою работу. Поэтому вы запускаете Visio (Omnigraffie или другое приложение диаграмм) и начинаете создавать блок-схему, показывающую все то, что вы делаете каждый день. Иногда будет обнаруживаться, что замечательные практики, такие как обзор кода (или тестирование кода, прежде чем его выпустить, и т. д.), вы действительно применяете, но не каждый раз. Полагая, что это надо делать всегда, и вы наверняка иногда так делаете, вы добавляете этот элемент в свою схему. Такова человеческая природа. Легко найти оправдание собственным поступкам — если это хорошая идея, значит, можно создать уверенность в том, что все это время вы применяете такие практики.

Это уничтожает процесс совершенствования, потому что скрывает реальную проблему.

Если этап, который вы добавили к вашей схеме, — хорошая идея, то теперь это выглядит так, будто вы уже это делаете. Никто и не подумает спросить: «Почему мы этого не делаем?» Зачем? Ты уже так работаешь! Что если есть причина, по которой вы не делаете это каждый раз, — скажем, обзор кода всегда отменялся, потому что его всегда проводят менеджеры команды, а они всегда заняты? Вы никогда не обнаружите и не попытаетесь исправить основную проблему, потому что каждый будет смотреть на диаграмму, видеть блок, касающийся обзора кода, и искать точки улучшений процесса где-нибудь в другом месте.

В Канбане визуализация означает запись именно того, что делает команда. Недостатки и тому подобные вещи не приукрашиваются. Это часть бережливого мышления: канбан-команда принимает lean-принцип. Чтобы видеть картину *целиком*. Правильное мышление не позволяет команде возиться с визуализацией рабочего процесса, потому что это помешало бы увидеть картину целиком. Ценность принимать *решения как можно позже* также важна: у вас еще нет всей информации о том, как разрабатывать программное обеспечение, поэтому остается последний ответственный момент, чтобы принять решение, как вы измените его.

Наряду с другими гибкими методологиями канбан-практики помогают вам заполучить lean-мировоззрение и принять бережливое мышление. Чем точнее и объективнее вы визуализируете рабочий процесс, тем быстрее примете такие ценности, как *видеть целое* и *принимать решения как можно позже*.

Так как же команды визуализируют рабочий процесс?

ИСПОЛЬЗУЙТЕ КАНБАН-ДОСКУ ДЛЯ ВИЗУАЛИЗАЦИИ РАБОЧЕГО ПРОЦЕССА

Канбан-доска — это инструмент, который команды используют для визуализации рабочего процесса. (Заглавная буква «К» используется при упоминании метода, со строчной буквы «к» пишется название доски.) Канбан-доска похожа на доску задач Scrum: обычно там есть колонки и стикеры. (Чаще всего на канбан-доске встречаются стикеры, а не учетные карточки.)

Существует три очень важных различия между доской задач и канбан-доской. Вы уже знаете о первом из них: канбан-доски содержат только истории и не показывают задачи. Еще одно состоит в том, что столбцы в канбан-досках обычно могут быть разными у различных команд. Наконец, на канбан-досках могут устанавливаться лимиты на объем работ в колонке. В дальнейшем мы поговорим об этих ограничениях, а сейчас сосредоточимся на самих колонках и том, как команды, использующие Канбан, часто имеют разные столбцы на канбан-досках. Одна доска команды может иметь колонки «сделать», «в процессе» и «сделано». А другая — совершенно иные колонки.

Когда команда хочет внедрить Канбан, она прежде всего визуализирует рабочий процесс на канбан-доске. Например, одна из первых канбан-досок в книге Дэвида Андерсона «Канбан. Альтернативный путь в Agile» имеет следующие столбцы: «входящая очередь», «анализ» (в процессе, готово), «готово к разработке», «разработка» (в процессе, готово), «готово к сборке», «тестирование» и «готово к релизу». Эта доска будет использоваться командой, которая следует за процессом, где каждая функция проходит через анализ, развитие, сборку и тестирование. Поэтому она может начать с такого варианта канбан-доски, как показано на рисунке 9.1, на которой есть колонки с текущими рабочими элементами.

Команда будет использовать канбан-доску, так же как Scrum-команды — доски задач. Канбан-команда проводит встречи (обычно ежедневные), которые называются «прогулка вдоль канбан-доски» (walking the board). На них обсуждается состояние каждого элемента на доске. Она должна отражать текущее состояние системы: каждый элемент, завершающий текущий этап, должен быть перемещен в следующую колонку. Но если этого еще не сделано, то команда знает, что во время встречи доска будет обновлена.

Важно понимать, что канбан-доска визуализирует основной рабочий процесс и процесс использования. Любые примеры, приведенные здесь и в других текстах (в книге «Канбан. Альтернативный путь в Agile» Дэвида Андерсона), — это лишь примеры из реальных контекстов. Вообще не стоит копировать канбан-доску, лучше разработать свою, изучая собственный рабочий процесс и визуализируя его. Копирование существующего процесса

без привязки к конкретному контексту противоречит эволюционному подходу Канбана. Если метод требует начать с того, что сейчас делаете именно вы, то не стоит копировать действия других людей*.

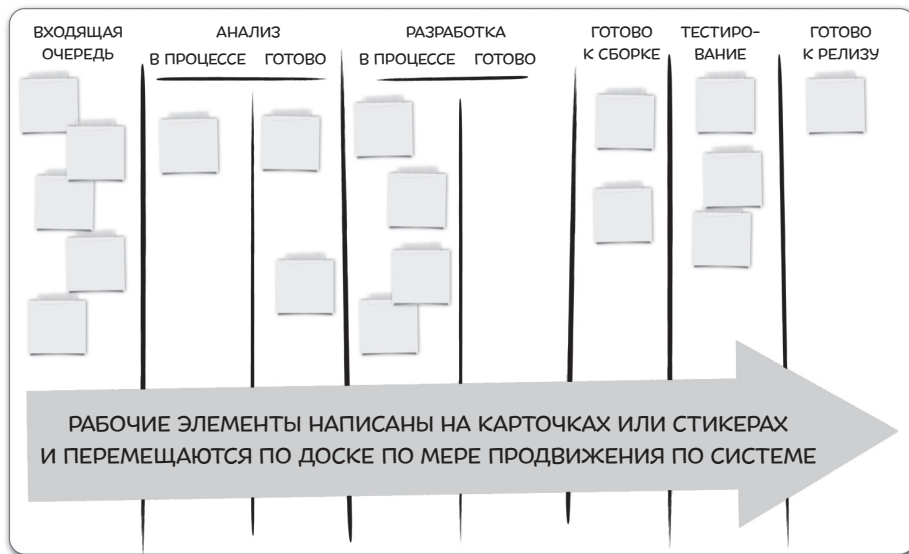


Рис. 9.1. Пример того, как рабочие элементы, написанные на бумажках, перемещаются по канбан-доске. Дэвид Андерсон в своей книге «Канбан. Альтернативный путь в Agile» использовал именно такой вариант доски, но названия колонок могут отличаться в зависимости от того, как члены команды выполняют свою работу

Вернемся к примеру из главы 8, в котором команда пыталась справиться с очень длительными сроками и разочарованными клиентами. Если вернуться к изначальному описанию команды, то изложение их рабочего процесса, которое менеджер проекта показывал руководству, будет таким:

1. Команда получает запрос от пользователя.
2. Руководитель проекта намечает функционал для следующего релиза.
3. Команда разрабатывает функционал.
4. Команда тестирует функционал.
5. Менеджер проекта проводит приемку.
6. Функционал реализован и включен в следующий релиз.

* Выражаем благодарность Дэвиду Андерсону за помощь в формулировке.

Пункты в главе 8 — это один из способов поддержания связи в рабочем процессе. И этот пронумерованный список тоже, но визуализация более эффективный инструмент для выполнения этой задачи.

Рисунок 9.2 показывает, что вариант рабочего процесса менеджера проекта, который можно назвать «счастливый путь», выглядит так, как показано на канбан-доске.

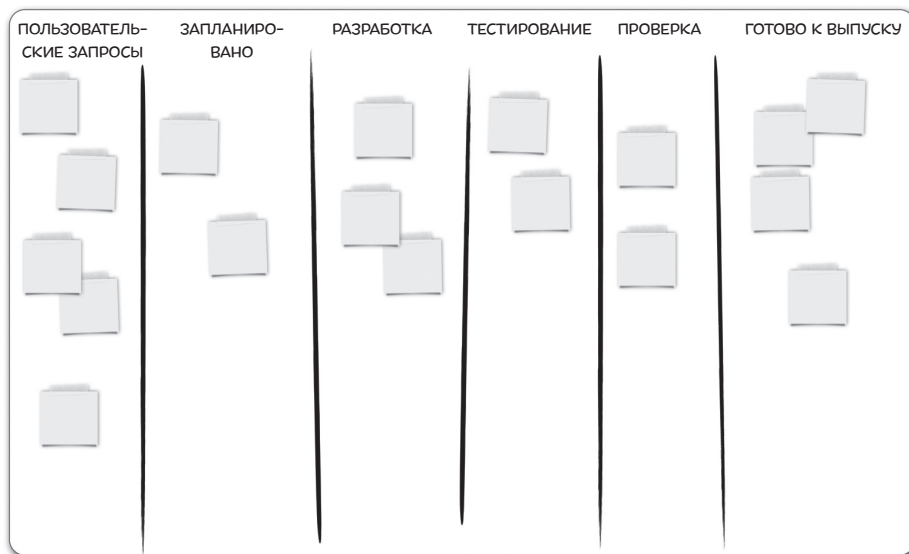


Рис. 9.2. Здесь показано, как люди воспринимают Канбан в ходе работы над проектом

Но это не то, что происходит в реальной жизни. В главе 8 команда использовала пять «почему», чтобы узнать больше о рабочих процессах. В дальнейшем список выглядел так:

1. Команда получает запрос от пользователя.
2. Менеджер проекта намечает функции на следующий трехмесячный релиз.
3. Команда собирает функцию.
4. Команда тестирует функцию.
5. Менеджер проекта проверяет прохождение тестирования.
6. Менеджер проекта планирует показ демоверсии высшему руководству.

7. Если топ-менеджер хочет, чтобы команда внесла изменения, то руководитель проекта проводит анализ влияния изменений, функция возвращается в пункт 1 и по порядку двигается к пункту 8.
8. Функция выполнена и включена в следующий релиз.

Теперь мы знаем, что есть дополнительный шаг, когда старшие менеджеры при необходимости могут вносить изменения в функции и переносить их в будущие релизы, хотя команда считает функции готовыми.

Мы будем менять канбан-доску, чтобы представить это нагляднее, добавляя колонки «комментарии менеджера» для тех функций, которые ожидают старшие менеджеры в демоверсии.

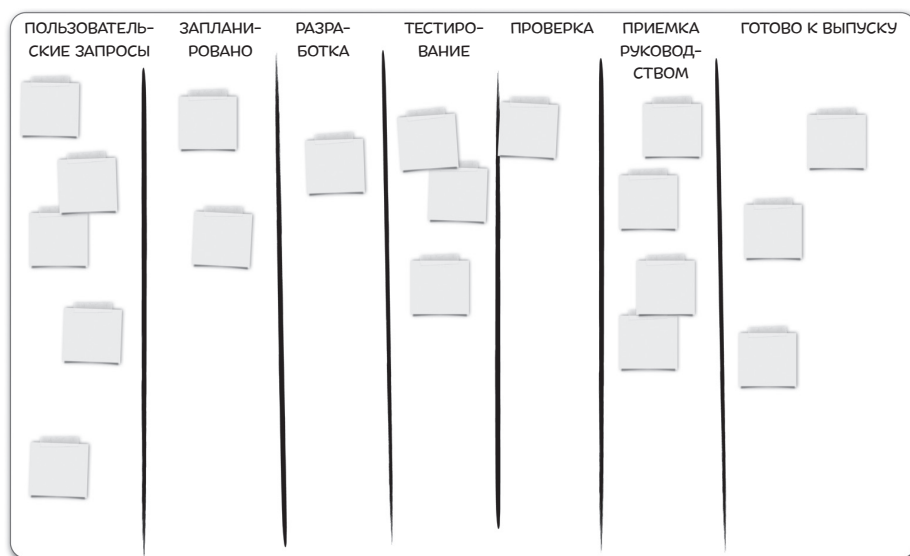


Рис. 9.3. Эта канбан-доска представляет более точную картину того, как протекает проект

Теперь у нас есть более точная визуализация рабочего процесса в команде. Если на канбан-доске видно все течение релиза, то проблема становится очевидной. Рабочие элементы накапливаются в столбце «приемка руководством» и хранятся там до окончания релиза, как показано на рисунке 9.4.

А как насчет рабочих элементов, которые были отнесены к будущему релизу в связи с доработками, которые инициированы менеджерами? Мы особенно заботимся о таких элементах, потому что из-за них некоторые пользователи ушли к конкурентам.

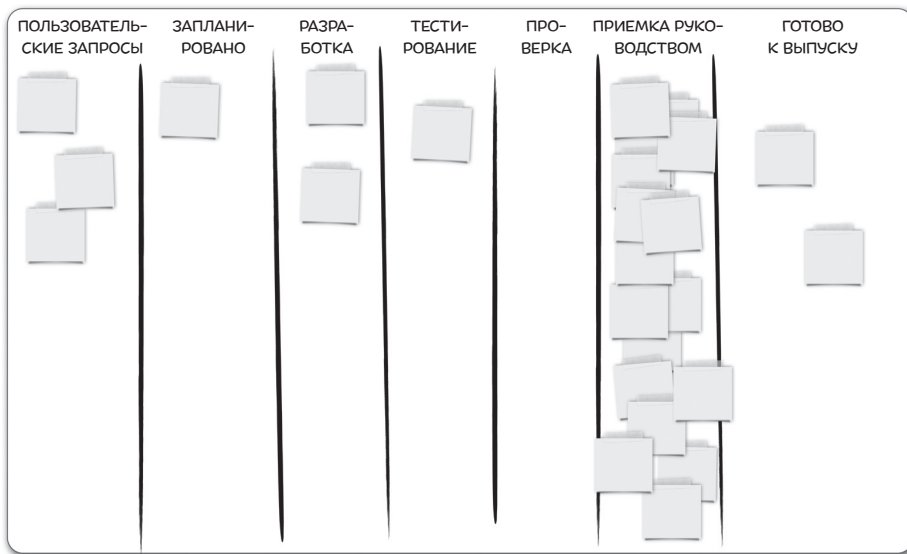


Рис. 9.4. Когда вы используете канбан-доску для визуализации рабочего процесса, проблемы, возникающие из-за неравномерности, легче обнаружить

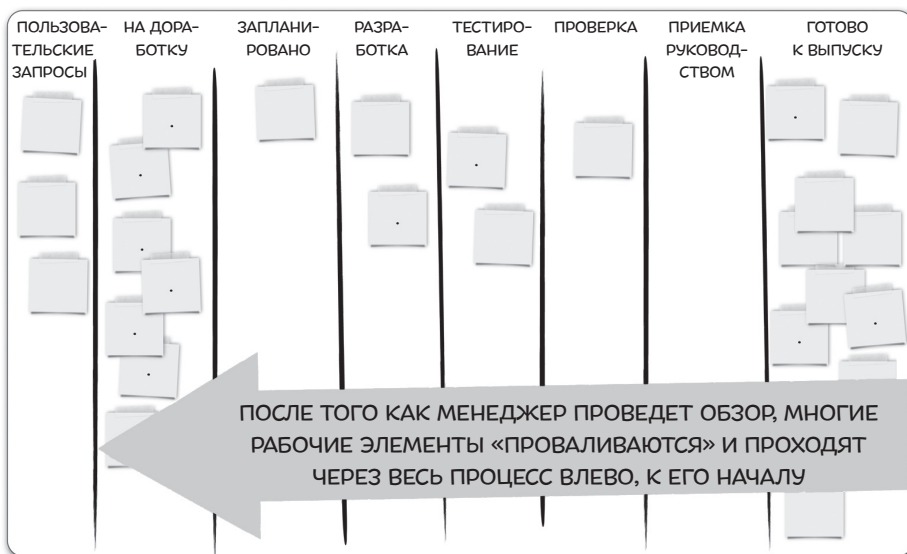


Рис. 9.5. Канбан-доска делает потери более очевидными, когда вы видите, что в течение рабочего процесса они встречаются несколько раз

По некоторым из элементов работа уже началась и должна продолжаться. Когда рабочие элементы требуют доработки уже после проверки менеджером, они возвращаются в начало процесса. Давайте убедимся, что они представлены на канбан-доске, — мы добавим столбец в начале пути, назовем его «На доработку» и передвинем стикеры влево (ставим маленькую точку на каждом стикере, чтобы сразу обнаружить, если они попадутся во второй раз).

Это довольно хорошая визуализация процесса, которому следует команда. Теперь мы можем точно видеть, что не так с проектом и почему ситуация ухудшалась. Некоторые члены команды, наверное, догадывались, что происходит, но теперь это ясно любому, кто смотрит на доску. И еще важнее, что это объективное свидетельство того, что способ, которым старшие менеджеры проводят обзор функций, — это основная причина проблем со сроками.

ОГРАНИЧЕНИЕ НА ВЫПОЛНЕНИЕ НЕЗАВЕРШЕННЫХ РАБОТ

Мы никогда не стали бы запускать серверы в компьютерных залах на полную мощность — почему мы не усвоили этот урок при разработке программного обеспечения?

Мэри и Том Поппендик. *Lean Software Development: An Agile Toolkit*

Только команда может делать так много работы. Мы узнали об этом при изучении Scrum и XP, и это важная часть бережливого мышления. Когда команда соглашается сделать больше того, на что способна, случаются неприятности. Она либо игнорирует некоторые виды работ, некачественно создавая продукт, либо работает в неустойчивом темпе, что в будущих релизах обойдется очень дорого. Иногда не сразу становится очевидно, что команда взяла на себя больше обязательств, чем может выполнить: каждый отдельный дедлайн выглядит разумным, но если предполагается, что все работают в многозадачном режиме в нескольких проектах или с несколькими элементами одновременно, то команда постепенно начинает испытывать перегрузки и требуются дополнительные усилия для переключения с одной задачи на другую. А это приводит к потере до 50% производительности.

Визуализация рабочего процесса позволяет команде распознать перегрузку. Это первый шаг к устранению проблемы. Неравномерность и перегрузки, о которых мы узнали в главе 8, проявляются на канбан-доске, когда стикеры собираются в одном столбце. К счастью, теория массового обслуживания не только предупреждает нас о проблеме, но и предлагает способ ее исправить. После выявления неравномерностей в рабочем процессе мы можем управлять объемом работ во всей системе, поставив жесткое

ограничение на выполнение незавершенной задачи. Такая канбан-практика называется **ограничение на выполнение незавершенных работ** (limit work in progress, WIP-лимит).

Ограничение на выполнение незавершенных работ означает установление ограничения на количество рабочих элементов, которые могут находиться на разных стадиях реализации проекта. Как правило, большинство людей сосредоточены на том, чтобы как можно скорее переместить рабочие элементы внутри процесса. Для одного элемента этот рабочий процесс — линейный: если вы программист и закончили написание кода для функционала, а ваш рабочий процесс требует, чтобы вы еще и протестировали его, то легко начать ненужную спешку, потому что это последний этап вашей работы.

Но что делать, если команда тестировщиков уже работает над несколькими функциями и не может одновременно проверить их? Не имеет смысла немедленно браться за новую функцию, если потом придется ждать, чтобы кто-нибудь ее протестировал. Это приведет к перегрузке команды тестировщиков. Так что же делать?

Для этого Канбан возвращается к бережливому мышлению — в частности, к принципу вариантного мышления, о котором говорилось в главе 8. Одна из причин, почему канбан-команда использует канбан-доску, — это возможность увидеть на ней все варианты. Если вы как разработчик только что закончили создавать архитектуру нового функционала, то легко предположить, что теперь вы возьметесь за код. Но работа над кодом для данного функционала — это возможность, а не обязательство. Глядя на канбан-доску, вы видите много стикеров, с которыми можно дальше работать. Вероятно, есть другие стикеры, сообщающие о необходимости разработки нового функционала, или столбец с ошибками, найденными во время тестирования функционала и ждущими исправления. Как правило, у вас большой выбор. Какой вариант вы предпочтете?

Установление WIP-лимитов на этапы в рабочем процессе подразумевает ограничение количества элементов работы, допустимых на этом этапе. Это облегчает команде выбор элементов работы, позволяет избежать перегрузки и способствует равномерному и максимально эффективному протеканию рабочего процесса в ходе разработки функционала. После ее завершения вы видите, что взять в разработку новый функционал невозможно из-за ограничений лимита. Тогда вы смотрите на другие возможности применения своих навыков, имеющиеся на доске, и работаете с ними — и команда тестировщиков не будет перегружена. (Представляете себе, как это может сократить среднее время разработки функционала? Если да, то вы начали приобретать навык системного мышления!)

Давайте вернемся к команде, которая использует пять «почему», чтобы найти причину проблемы во времени, отведенном на выполнение заказа. После того как мы создали для них канбан-доску, определилась перегрузка: стикеры начали накапливаться в колонке «приемка руководством». Поэтому, чтобы ограничить объем выполняемых работ для этой команды, мы просто нашли способ ограничить количество скапливающихся элементов до того, как менеджеры проведут сессию приемки.

В Канбан, узнав о такой проблеме рабочего процесса, как эта, вы **устанавливаете WIP-лимит** (или работаете с ограничением на выполнение незавершенных работ). Для этого команда должна встретиться с руководством, чтобы убедить всех принять новые правила. Новая политика устанавливает WIP-лимит, подразумевающий, что в столбце «приемка руководством» может накапливаться только определенное количество элементов работы. Канбан-доска и подсчет времени на выполнение заказа дают команде и руководителю проекта достаточно объективных доказательств, чтобы убедить менеджера согласиться с этим.

Когда мы устанавливаем WIP-лимит на столбец, он перестает быть местом, где скапливается масса рабочих элементов. Вместо этого образуется очередь, или стадия в рабочем процессе, где управление идет плавным и упорядоченным образом.

Не существует жесткого правила, устанавливающего размер WIP-лимита. Команда использует эволюционный подход к этому элементу. Обычно исходя из здравого смысла экспериментальным путем находят оптимальный вариант. В нашей команде каждый релиз, как правило, имеет 30 элементов функционала. Чтобы старшие менеджеры чувствовали себя комфортно, они могут встретиться три раза на протяжении релиза, поэтому мы выберем WIP-лимит, ограниченный десятью элементами в колонке «приемка руководством», как это показано на рисунке 9.6.

Что происходит, когда в колонку «приемка руководством» добавляется десятый элемент работы? Достигнув лимита, команда больше ничего не помещает в этот столбец. Вероятно, для нее найдется и другая работа. Если разработка закончена, то она поможет тестировщикам. Не исключено, что есть технический долг, который можно устранить, — если на доске нет стикеров с соответствующими элементами, то таких элементов работы не должно быть на самом деле. Единственное, чего они не делают, — не «засовывают» в колонку «Приемка руководством» дополнительные элементы работы. Для этого существует соглашение с топ-менеджерами: как только колонка достигнет WIP-лимита, менеджеры будут проводить обзоры и работа будет накапливаться до тех пор, пока это не произойдет.

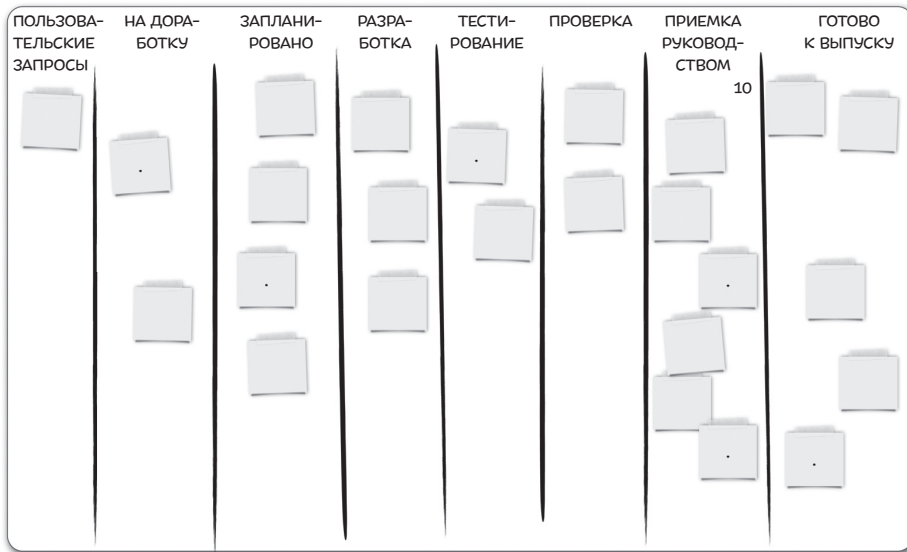


Рис. 9.6. Число 10 в столбце «Приемка руководством» — это его WIP-лимит

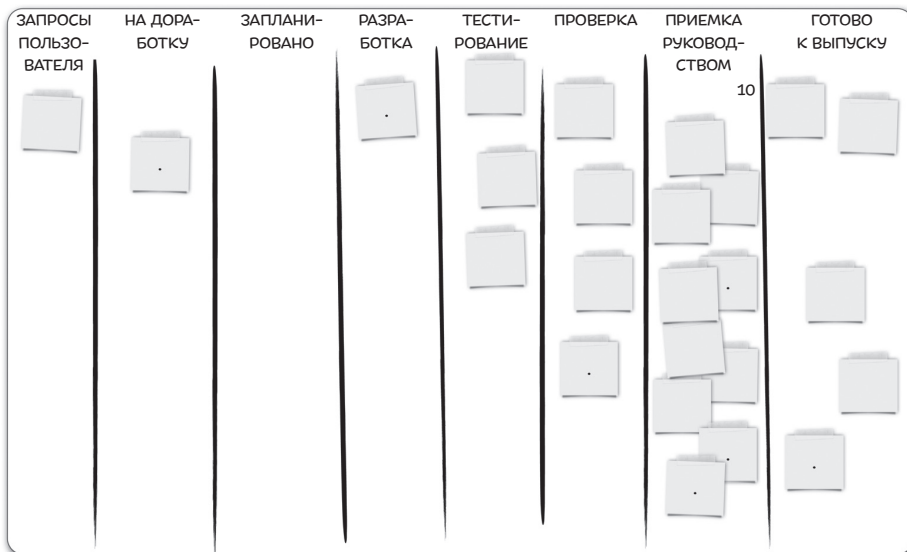


Рис. 9.7. Когда колонка достигает своего WIP-лимита, в нее больше не добавляются стикеры, даже если это выполненная работа. Команда переносит внимание на рабочие элементы в других столбцах, которые еще не достигли WIP-лимита

Как только менеджеры выполняют свое обещание, проводят обзор и рассчитывают «завалы», работа движется вперед. WIP-лимит заставляет их давать обратную связь гораздо раньше, поэтому команда может немедленно внести необходимые изменения. Если рабочий элемент натывается на препятствия, то менеджеры узнают об этом в начале проекта, а не в конце. Это наносит меньший вред команде, потому что ей не приходится выполнять много ненужной работы, которая затем может оказаться в колонке «На доработку». Сделанная работа немедленно приобретает ценность, в отличие от той, что находится в колонке «На доработку» до тех пор, пока ею не заинтересуется компания (или пока разозленные клиенты не уйдут к конкуренту).

Это эффективно, потому что *цикл приемки менеджментом и изменения, внесенные командой, замыкаются в петле обратной связи*. При этом, если петля обратной связи слишком длинная — например, равняется длине всего релиза, — то информация, попадающая назад в проект, становится разрушительной, потому что команде приходится пересматривать массу уже проделанной работы. Это приводит к потерям.

Добавление WIP-лимита устраняет причины перегрузки, среднее время рабочего элемента в системе получается намного короче. Теперь в течение проекта менеджеры несколько раз проведут приемку, что позволит команде раньше внести коррективы. И менеджеры смогут использовать эту информацию для более раннего изменения приоритетов, поэтому ценная работа не пойдет в потери.

Более того, теперь команда и менеджеры могут контролировать длину петли обратной связи. Например, если руководители считают, что информация является ценной, и они могут согласиться уменьшить WIP-лимит до шести элементов, это приведет к более частым встречам и даст команде более раннюю обратную связь.

ПОЧЕМУ БЫ НЕ СДЕЛАТЬ WIP-ЛИМИТ РАВНЫМ ЕДИНИЦЕ И НЕ ПОТРАТИТЬ ВСЕ ВРЕМЯ ТОЛЬКО НА ЭТО? ДЕЙСТВИТЕЛЬНО ЛИ КОРОТКАЯ ПЕТЛЯ ОБРАТНОЙ СВЯЗИ ЛУЧШЕ?

Нет, если команда тратит больше времени на проведение совещаний и обратную связь, чем на выполнение самой работы. Как и любым инструментом, WIP-лимитами и петлями обратной связи можно злоупотреблять. Когда система имеет слишком короткую петлю обратной связи, она может попасть в состояние, которое называется **пробуксовкой**. Это случается, когда слишком много информации попадает обратно в систему и не хватает времени для ее обработки, а тем временем поступает следующая партия данных.

Визуализация рабочего процесса при помощи канбан-доски позволяет команде увидеть петли обратной связи и экспериментировать с WIP-лимитами, чтобы найти оптимальную длину обратной связи. Это обеспечивает регулярную обратную связь, причем команда успевает ответить на замечания прежде, чем придет следующая партия.

Нужно избегать при этом многократной отправки одних и тех же элементов через цикл обратной связи, потому что это засоряет систему. Но Канбан помогает заметить и это. Например, когда менеджеры дают обратную связь команде и просят переделать, работа отправляется назад в бэклог. Члену команды приходится перемещать стикер с задачей в колонку с более ранним этапом. Команда может отслеживать стикеры, которые были смещены назад, если поставит на них точку или любой другой знак — явный показатель того, что петля обратной связи для этой функции будет повторяться. Когда функция возвращается через рабочий процесс, она в итоге снова окажется в столбце «Приемка руководством» и заполнит собой одно место среди WIP-лимитов. Это приводит к эффекту засорения петли обратной связи.

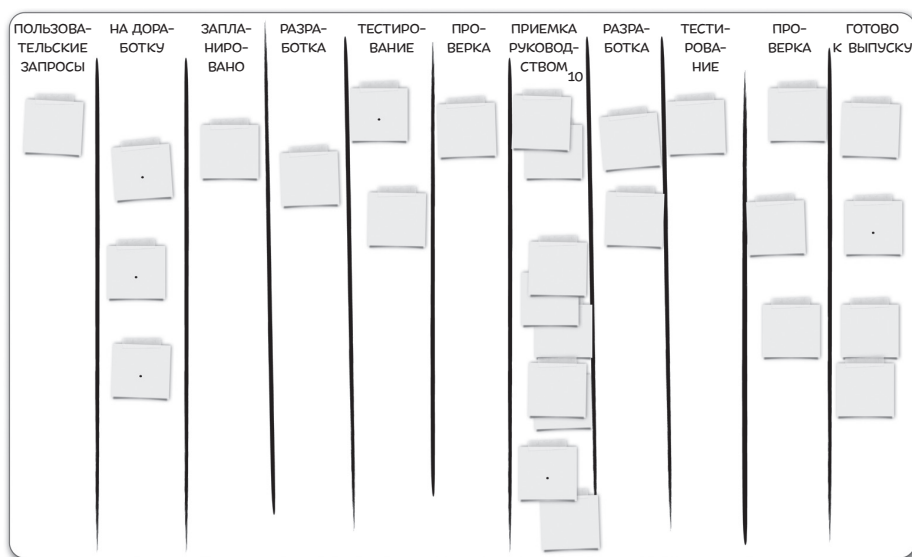


Рис. 9.8. Добавление дополнительных столбцов на канбан-доску — и предотвращение попадания множества стикеров обратно в предыдущие колонки из-за их пробуксовки — дает команде больший контроль над процессом

Команда может избежать данной проблемы, удалив дополнительную петлю из рабочего процесса, сделав ее более линейной для большинства элементов. Что если команда сможет убедить менеджеров согласиться на единственный

этап приемки руководством с выбором тех из элементов, которые надо пере-
делать? Раз менеджеры доверяют команде принимать их обратную связь для
большинства элементов и не требуют их повторной проверки руководством,
прежде чем функционал будет выпущен, команда сможет добавлять допол-
нительную разработку и этап тестирования после приемки.

Этот процесс на канбан-доске выглядит длиннее. Но поскольку мы исполь-
зовали объективные данные, полученные в результате измерения времени
на освоение новой продукции и экспериментальные данные с WIP-лимита-
ми, чтобы развивать рабочий процесс, как в нашем примере, то мы можем
с уверенностью утверждать, что так действительно быстрее. И мы продол-
жим проводить измерение времени на освоение новой продукции и визу-
ализировать рабочий процесс, чтобы доказать команде и менеджерам: не-
смотря на добавление нескольких этапов, производство каждой функции
и включение ее в релиз протекают быстрее. Но эти измерения полезны
для дальнейшего совершенствования проекта и убеждения менеджеров
в том, что он улучшается, а команде все это, скорее всего, не потребуется.
Ее убедят результаты, потому что после устранения неравномерной рабо-
ты и перегрузок проект явно станет *продвигаться быстрее*.



Ключевые моменты

*Канбан — это метод, улучшающий процесс разработки програм-
мно-го обеспечения и командной работы, основанный на lean-
мировоззрении. Канбан-команды начинают с текущей работы. Они
оценивают целиком реальную действительность и со временем про-
водят постепенные эволюционные изменения системы.*

*Канбан-практика развивать совместно и улучшать эксперименталь-
но подразумевает проведение измерений, постепенное улучшение,
а затем подтверждение того, что работа включала эти измерения.*

*Каждая команда имеет систему для производства программного
обеспечения (независимо от того, осознает ли она это), и идея сис-
темного мышления, принадлежащая Lean, подразумевает оказание
помощи команде в понимании этой системы.*

***Канбан-доска** — это способ визуализации рабочего процесса команды.*

*Канбан-доска разделена на столбцы, которые представляют каж-
дый этап рабочего процесса. Рабочие элементы написаны на стике-
рах и распределены по колонкам. По мере продвижения по системе
они перемещаются из столбца в столбец.*

Канбан-доска использует рабочие элементы, а не задачи, потому что Канбан — это не система управления проектами.

*Когда канбан-команды вводят **лимит на выполнение незавершенных работ** (WIP-лимит), они добавляют цифру в колонку на доске, которая указывает на максимальное число рабочих элементов, разрешенных на этой стадии рабочего процесса.*

Канбан-команды работают со своими пользователями, менеджерами и другими заинтересованными сторонами, чтобы убедиться: все согласны с тем, чтобы команда сосредоточила свои усилия на другой части проекта, когда количество рабочих элементов в колонке достигает лимита, и не пытаться ничего вносить в нее сверх того.

Измерение и управление потоком

Команды продолжают поставлять функционал, они выявляют проблемы рабочего процесса и корректируют WIP-лимит так, чтобы петли обратной связи обеспечивали достаточный объем информации, не вызывая пробуксовки. **Поток** — это скорость, с которой рабочие элементы перемещаются по системе. Когда команда находит оптимальный темп для поставки в сочетании с удобной обратной связью, она *максимизирует поток*. Сокращение неравномерной работы и перегрузок, возможность заканчивать одно задание и только затем переходить к следующему увеличивает поток. Когда из-за неравномерной работы накапливаются задачи, это приводит к прерыванию рабочего процесса и сокращению потока. Канбан-команда использует практику **управления потоком** путем его измерения и предпринимает активные шаги по его улучшению.

Вы уже знаете, как это вдохновляет, когда работа продвигается. Вы чувствуете, что успеваете многое выполнить, не тратите время и не ждете, когда кто-то другой завершит свою часть работы и передаст ее вам. Когда вы работаете в команде с большим потоком, вы постоянно чувствуете, что выполняете нечто ценное. К этому стремятся все.

Вы также знаете, каково это, когда работа буксует. Такое чувство, будто вы увязли в жиже и едва можете двигаться. Кажется, что вы вечно кого-то ждете, чтобы закончить создание необходимого, или принимаете решение, влияющее на работу, согласовываете карточку, либо находится еще что-нибудь, препятствующее работе, — даже если вы знаете, что никто намеренно не мешает. Вы ощущаете несогласованность и непоследовательность и тратите много времени на объяснение, почему вы ждете. Это не означает, что команда недостаточно загружена, возможно, люди выкладываются

на 100%, а может даже и перегружены. Но в то время, как план проекта говорит о том, что работа выполнена на 90%, вам кажется, что, наоборот, еще нужно сделать 90% работы.

И ваши пользователи знакомы с ситуацией, когда работа не продвигается, потому что время на выполнение нового заказа непрерывно нарастает. Кажется, что команде требуется все больше времени, чтобы ответить на их запросы, и даже разработка простых функций длится вечность.

Весь смысл Канбана заключается в увеличении потока, следовательно, вся команда должна участвовать в этом. Как только увеличивается поток, разочарование, связанное с неравномерной работой, уменьшается.

ИСПОЛЬЗОВАНИЕ CFD И WIP-ДИАГРАММ ДЛЯ ИЗМЕРЕНИЯ И УПРАВЛЕНИЯ ПОТОКОМ

Канбан-доска — важный инструмент управления потоком, потому что на ней виден источник проблемы, что позволяет сосредоточить работу на том месте, где она будет наиболее эффективной. Когда вы ищете завалы в работе и устанавливаете WIP-лимит, чтобы их расчистить, вы принимаете меры по увеличению потока. WIP-лимит работает, потому что вы помогаете команде сосредоточить усилия на конкретной части проекта, которая блокирует поток. По сути, WIP-лимит меняет приоритеты рабочих элементов, заставляя команду действовать так, чтобы не было неравномерной работы, и предотвращает появление проблем.

Но откуда известно, что добавление WIP-лимита действительно увеличивает поток?

Вернемся к бережливому мышлению, которое подсказывает, что мы должны проводить измерения, а эффективный инструмент измерения потока — **кумулятивная диаграмма потока*** (cumulative flow diagram, CFD). CFD похожа на WIP-диаграмму, но имеет одно важное отличие: вместо того чтобы исчезать, выполненные рабочие элементы накапливаются в нижней полосе диаграммы. И если на WIP-диаграмме (из главы 8) полосы соответствуют состояниям в карте потока создания ценности, то CFD (и WIP-диаграмма) в этой главе имеет полосы, соответствующие столбцам на канбан-доске.

CFD в этой главе имеет дополнительные линии, показывающие среднюю частоту **поступления** (число рабочих элементов, ежедневно добавляемых

* Cumulative Flow Diagram (CFD) также иногда называют накопительной диаграммой потока. *Прим. ред.*

в рабочий процесс) и **среднюю емкость** (общее количество рабочих элементов в рабочем процессе). CFD также может показывать среднее **время производства** (или время, на протяжении которого рабочий элемент остается в системе, как мы говорили о рабочих элементах в главе 8). Не все CFD имеют эти линии, однако они очень полезны для понимания потока рабочих элементов в рамках системы.

Ключ к управлению потоками с CFD нужно искать в шаблонах, которые указывают на проблему. Канбан-доска поможет понять, где находятся неравномерности, петли и другие проблемы рабочего процесса сегодня, и поможет управлять базовым ежедневным потоком при помощи WIP-лимитов. CFD позволяет взглянуть на течение процесса целиком, поэтому можно предпринять шаги для обнаружения и устранения коренных причин проблем в долгосрочной перспективе.

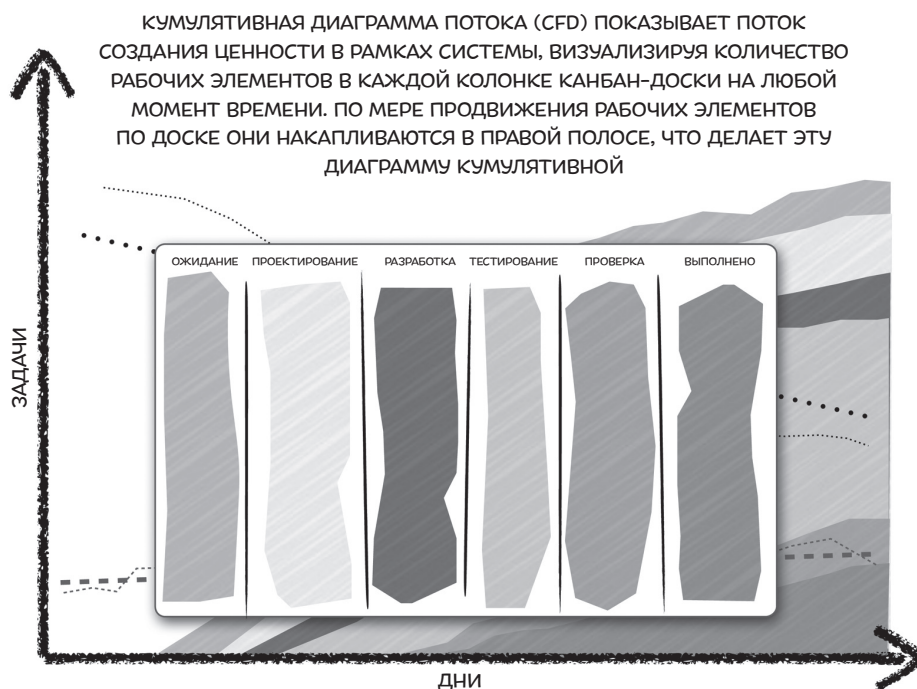


Рис. 9.9. Канбан-команды используют кумулятивные диаграммы потока (CFD) с полосами, которые соответствуют колонкам на канбан-доске. CFD похожа на WIP-диаграмму, за исключением того, что со временем рабочие элементы не перемещаются к концу графика, а накапливаются в полосах или линиях

КАК ПОСТРОИТЬ КУМУЛЯТИВНУЮ ДИАГРАММУ ПОТОКА И ИСПОЛЬЗОВАТЬ ЕЕ ДЛЯ РАСЧЕТА СРЕДНЕГО ВРЕМЕНИ НА РАЗРАБОТКУ ПРОДУКТА

Чтобы построить CFD, начните с создания WIP-диаграммы. Но вместо того, чтобы собирать данные из карты потоков создания ценности, ищите информацию о числе рабочих элементов в каждой колонке на канбан-доске.

Еще потребуются два дополнительных элемента данных, которые нужно ежедневно добавлять в диаграмму: это скорость поступления и общее количество элементов. Чтобы найти частоту ежедневного поступления, посчитайте число рабочих элементов, которые были добавлены в первый столбец. Для составления графика общего количества элементов подсчитайте общее количество рабочих элементов в каждом столбце. Каждый день добавляйте точку в CFD, соответствующую скорости поступления и общему количеству элементов, чтобы прочертить две линии в верхней части WIP-диаграммы.

Большинство команд, использующих CFD, не рисуют их поэтапно на доске. Они используют Excel или другую программу, позволяющую создавать графики. Помимо простоты управления данными электронная таблица может автоматически добавлять линию тренда к скорости поступления и общему количеству элементов. Эти линии очень полезны, потому что могут продемонстрировать, действительно ли система стабильна. Если они плоские и горизонтальные, то система устойчива. Если одна из них имеет наклон, то значение меняется с течением времени. Вам необходимо добавить WIP-лимиты для стабилизации системы, и можно будет сказать, что система устойчива, когда эти линии станут ровными.

Если посмотреть на рукописные заметки на рисунке 9.10, то видно, что мы обозначили эти ценности следующим образом: буква L используется для среднего значения общего количества элементов, греческая буква λ — для среднего значения скорости поступления (или количества рабочих элементов, добавляемых ежедневно), а W — для среднего времени производства (или среднего времени ожидания пользователями окончания работы по запросу).

Взгляните на линию списка элементов в CFD (толстая пунктирная линия наверху). Тренд снижается; это говорит нам, что общий объем элементов со временем уменьшается, то есть много рабочих элементов вышли за пределы системы (сделаны) и никогда не возвращались. Но, посмотрев вниз, вы увидите, что скорость поступления увеличивается.

Что будет, если мы продолжим отслеживать этот проект? Заполнится ли список снова? Появится ли еще один релиз, который уберет рабочие

элементы из системы? Когда запросы на разработку прибывают в большем объеме, чем убывают, то в долгосрочной перспективе список имеет тенденцию расти вверх — и команда почувствует это. Люди будут постепенно заваливать себя работой и осознавать, что времени для ее выполнения не хватает. Ощущение, что ты «увяз в трясине», происходит из-за того, что в системе нет течения.

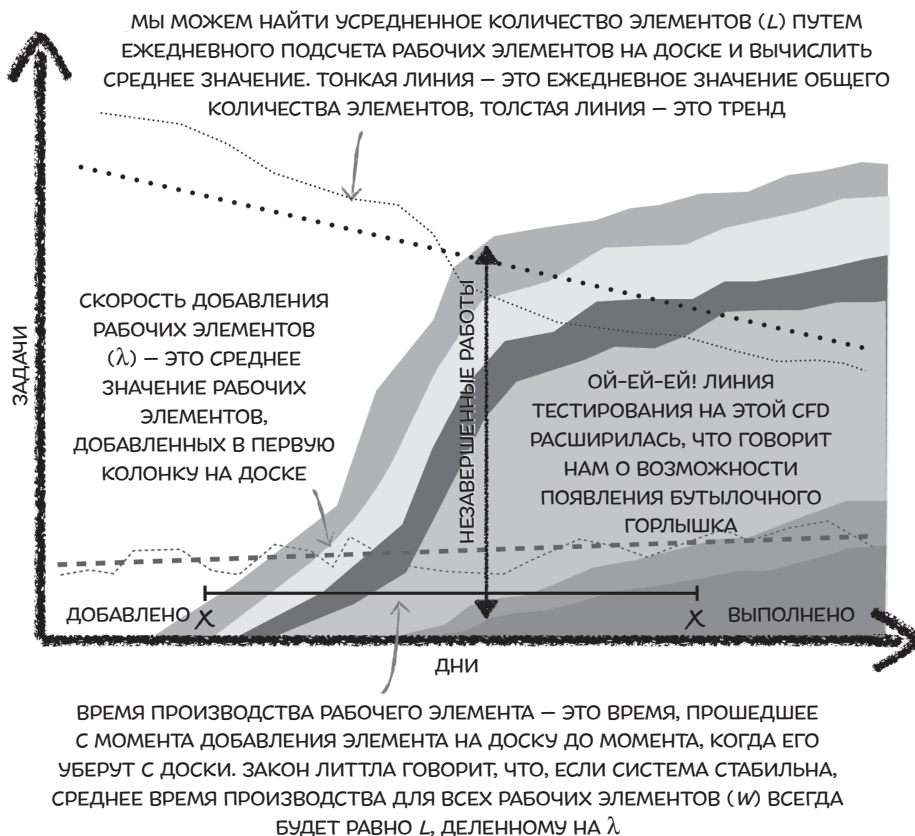


Рис. 9.10. Этот пример CFD также показывает частоту поступления и усредненное количество элементов. Общий размер незавершенной работы на любой момент времени можно найти, измерив разницу между верхней частью диаграммы и верхней частью линии «Выполнено». Горизонтальная сплошная черная линия показывает время выполнения для конкретного рабочего элемента в системе. Мы пока не можем вычислить среднее время выполнения нового элемента, потому что система нестабильна — тренды общего количества элементов и скорости их поступления имеют вид наклонной линии. Это означает нестабильность

К счастью, есть решение этой проблемы: добавить WIP-лимит. Команда может экспериментировать и использовать обратную связь, чтобы найти ценности WIP-лимита, которые работают для их системы, и если им удастся получить их, то в конце концов установится баланс между скоростью поступления и выполнения. Тенденция долгосрочного списка приобретет вид ровной линии, линия скорости поступления — тоже. И как только это произойдет, система станет **стабильной**.

Когда система *стабильна*, существует простая связь между этими величинами, так называемый **закон Литтла** (время, затрачиваемое на выполнение полезной работы).

Эта теорема — часть теории массового обслуживания, названная в честь Джона Литтла, который впервые предложил ее в 1950-х годах и считается «отцом маркетинга». И хотя греческая буква L означает **закон Литтла**, не нужно быть математиком, чтобы использовать эту формулу:

$$L = W \times \lambda.$$

В переводе на обычный язык это означает, что если у вас есть стабильный рабочий процесс, то средний список *всегда* равен средней скорости поступления, умноженной на среднее время, необходимое для выполнения нового заказа. Это математический закон: если доказано и если система стабильна, это всегда верно.

Верно также и обратное:

$$L = W \div \lambda.$$

Если вы знаете среднее количество и среднюю скорость поступления, то *можете рассчитать среднее время выполнения нового заказа*. Вычислить среднее количество и скорость поступления довольно просто: каждый день записывайте на канбан-доску общее количество рабочих элементов, а также число, которое было добавлено в первую колонку в тот же день. На нашей CFD они указаны тонкими прерывистыми или пунктирными линиями.

Если ваша система работает стабильно, то через некоторое время вы увидите, что средний ежедневный список и ежедневная скорость поступления выглядят как толстые прямые линии. Разделите средний список на среднюю скорость поступления, и вы получите время на выполнение нового заказа.

Задумайтесь над этим. Время — один из лучших способов измерения уровня разочарования пользователей: быстрая доставка обрадует их, а затягивание сроков поставки вызовет рост недовольства.

Продолжительное время выполнения нового заказа — хороший индикатор качества проблем: Дэвид Андерсон в уже упоминавшейся книге «Канбан. Альтернативный путь в Agile» так говорит об этом:

Похоже, что увеличение времени выполнения оборачивается существенно худшим качеством. В нашем случае увеличение среднего времени выполнения в 6,5 раза повлекло за собой более чем тридцатикратное увеличение первичных ошибок. Более долгое время выполнения связано с увеличением количества незавершенных задач.

Ваше время на выполнение нового заказа полностью определяется скоростью поступления рабочих элементов в систему и протекания по ней — и WIP-лимиты позволяют контролировать скорость потока.

Так что это значит? Когда система работает стабильно, вы можете сократить время выполнения нового заказа, *отказавшись от разработки новых функций*.

ИСПОЛЬЗОВАНИЕ CFD ДЛЯ ЭКСПЕРИМЕНТИРОВАНИЯ С WIP-ЛИМИТАМИ И УПРАВЛЕНИЯ ПОТОКОМ

Одна из основных идей Канбана заключается в том, что визуализация процесса позволяет измерить поток, делает вашу систему стабильной и фактически берет под контроль время выполнения проекта путем управления скоростью начала работы с новыми элементами.

Это может показаться немного абстрактным. Полезно представить, как применялась бы CFD в случае наиболее знакомой и простой системы — медицинского центра. Допустим, вам нужно несколько раз посетить врача, чтобы сдать ряд анализов и обсудить результаты. Вы заметили следующее: если встреча с доктором назначена на утро, когда прием только начинается, то вам не приходится долго ждать. Но если визит предполагается в более позднее время, то ожидание затягивается — и чем позднее время встречи, тем дольше вы ждете приема. Очевидно, что эта система нестабильна. Как использовать CFD для ее стабилизации?

Прежде всего нужно визуализировать рабочий процесс.

Допустим, каждый пришедший начинает свой визит в приемном покое. В конце концов медсестра вызывает его в один из кабинетов, где взвешивает, измеряет давление и температуру. Затем пациенты ждут приема врача. В данном медицинском центре пять процедурных и два врачебных кабинета, и они почти всегда заняты. Важно понять: в процедурных

не может быть одновременно более пяти больных, а в кабинетах врача — более двух. Это WIP-лимит, введенный реальными ограничениями системы.

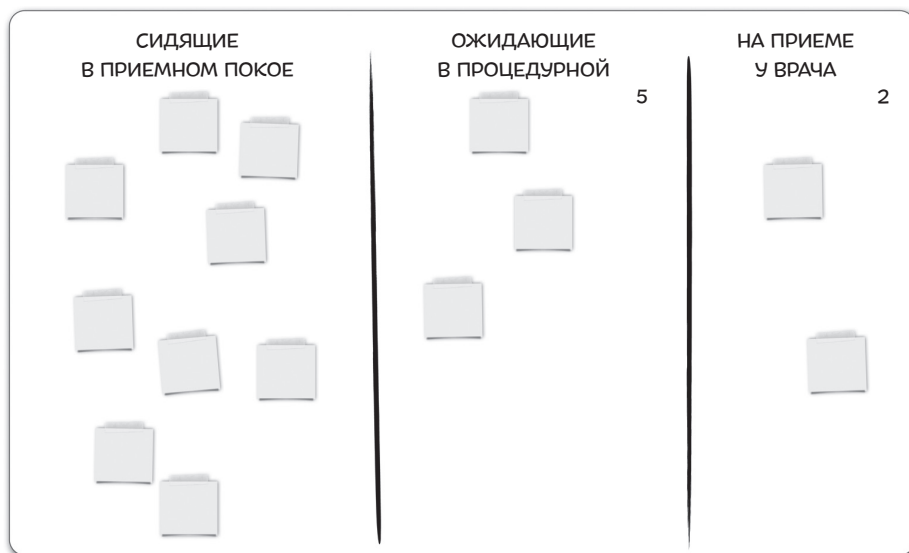


Рис. 9.11. Канбан-доска для медицинского центра. Первый столбец показывает пациентов, которые находятся в приемном покое, второй — пациентов в процедурных и третий столбец — тех, кто находится у врача. Есть пять процедурных и два врачебных кабинета — это естественный WIP-лимит, поэтому визуализируем его на доске

Врачам не нравится, что пациенты, записанные на поздние часы, жалуются на длительное ожидание. Еще хуже то, что в конце дня они начинают поторапливать пациентов, чтобы те быстрее покинули кабинет. Врачи сами обеспокоены тем, что не всегда способны принять наилучшие решения, потому что больше думают о том, как бы поскорее спровадить больного. Может ли Канбан помочь сократить время ожидания и обеспечить лучшее лечение?

Давайте выясним. Начнем с построения CFD для типичного дня. Попросим персонал центра посчитать количество пациентов, входящих в медицинский центр каждые 15 минут. Это покажет скорость прибытия — в буквальном смысле количество прибывших в центр пациентов. И мы можем рассчитать список для каждого 15-минутного интервала путем подсчета общего количества пациентов в приемном покое и пяти процедурных кабинетах.

Каждый раз, когда кто-то приходит, добавляется стикер в первую колонку на канбан-доске. Когда пациент переходит из приемного покоя в процедурную, стикер перемещают во вторую колонку. А если пациент уже зашел к врачу, то стикер сдвигается в третью колонку. После того как врач закончил работу с пациентом, стикер удаляют с доски. Сотрудники центра могут записать количество стикеров в колонках для каждого 15-минутного интервала.

Теперь у них есть все данные, которые нужны для построения CFD.

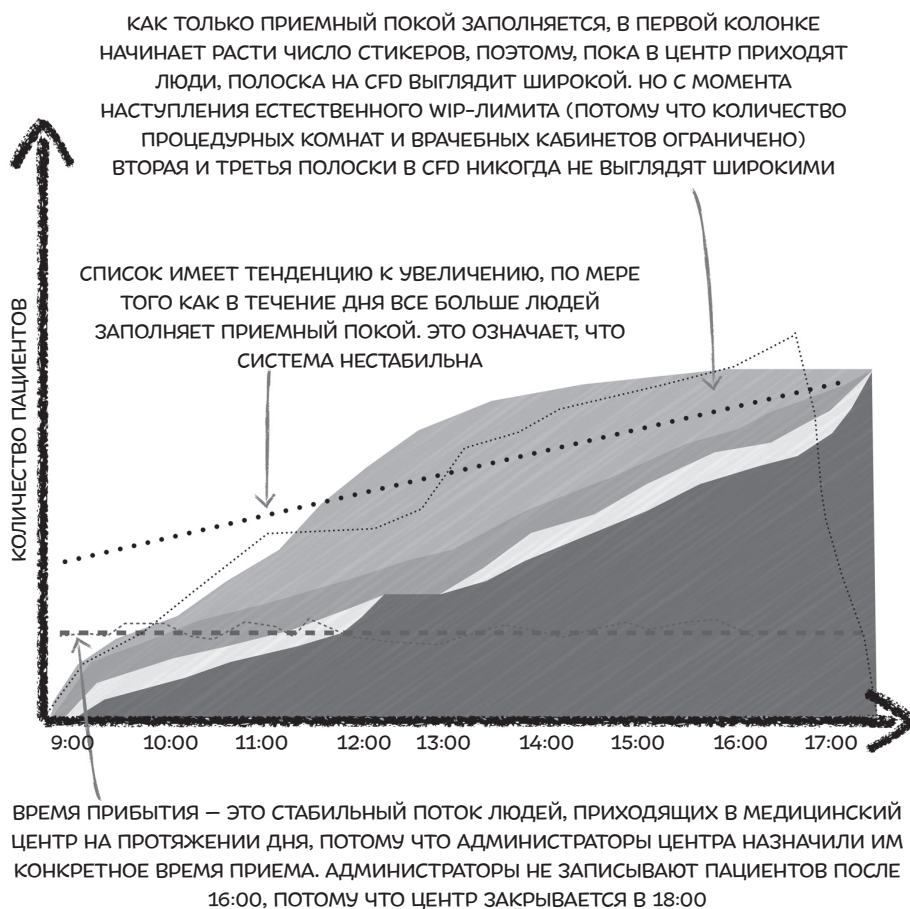


Рис. 9.12. Эта CFD показывает поток пациентов через медицинский центр. В течение дня приемный покой все больше заполняется людьми. Догадываетесь ли вы, почему полоска для третьего столбца исчезает между 12:15 и 13:00?

Эта система нестабильна. Скорость прибытия пациентов стабильна, потому что администраторы сами назначают время приема пациентов, а те, в свою очередь, приходят с постоянной скоростью. Администраторы не хотят задерживаться на работе. Поэтому прекращают запись после 16:00. Люди попадают к врачу позже назначенного времени, но тенденция скорости прибытия плоская, поэтому она должна быть довольно постоянной на протяжении всего дня.

Тенденция очереди, однако, не является плоской. Она стремится вверх, потому что очередь растет. В этом есть смысл — количество пациентов в приемном покое также увеличивается на протяжении дня. Каким образом сотрудники могут использовать эту новую информацию для улучшения обслуживания пациентов и снижения времени ожидания?

Первое, что нужно сделать, — стабилизировать систему. Инструментом для этого послужит WIP-лимит. Сотрудники будут использовать канбан-практику эволюционного развития и экспериментальное улучшение путем определения WIP-лимита в качестве отправной точки. Посмотрев на данные, каждый решает установить для приемного покоя WIP-лимит, равный 6. Но это возможно при одном условии: врачи должны согласиться, что если там собирается шесть пациентов, то администраторы должны позвонить тем, кому назначено позднее, и попросить их перенести встречу (стараясь найти такое время, чтобы не нанести ущерба здоровью пациента). Кроме того, администраторы предложат ожидающим добровольно перенести визит, обещая при этом выбрать приоритетное для них время. Этой *новой политики* нужно четко придерживаться. Они будут делать это, добавляя WIP-лимит на канбан-доске, а также размещая уведомления на информационной стойке, давая понять, что такова теперь политика центра.

Потребуется приложить немного усилий, но через несколько дней сотрудники клиники привыкают к новой политике. Они обнаруживают, что должны записывать пациентов с учетом их жалоб. Делается это посредством определения **класса обслуживания**: розовые стикеры будут использоваться для обозначения пациентов в более тяжелом состоянии, поэтому становится понятно, что перенести их визит невозможно. Желтые стикеры обозначают пациентов с незначительными проблемами, чей визит можно перенести. Это позволит обеспечить максимально быстрое обслуживание пациентов, которые нуждаются в этом больше всего.

И это работает! Персонал обнаружил, что, как только был введен WIP-лимит, появилась возможность записывать пациентов и после 16:00. Теперь их успеют обслужить до 18:00. Они могут планировать запись на 17:40,

а пациентов с более серьезными проблемами — после 16:40 (и они придерживаются этой политики). Безусловно, если кто-то обратится в клинику с серьезной проблемой, то врач осмотрит этого пациента (или отправит в отделение неотложной помощи). Но это редкое исключение, потому что персонал клиники — умные и ответственные люди, способные принимать решения в каждом конкретном случае. Пациенты теперь выглядят намного счастливее, потому что им не приходится долго ждать.

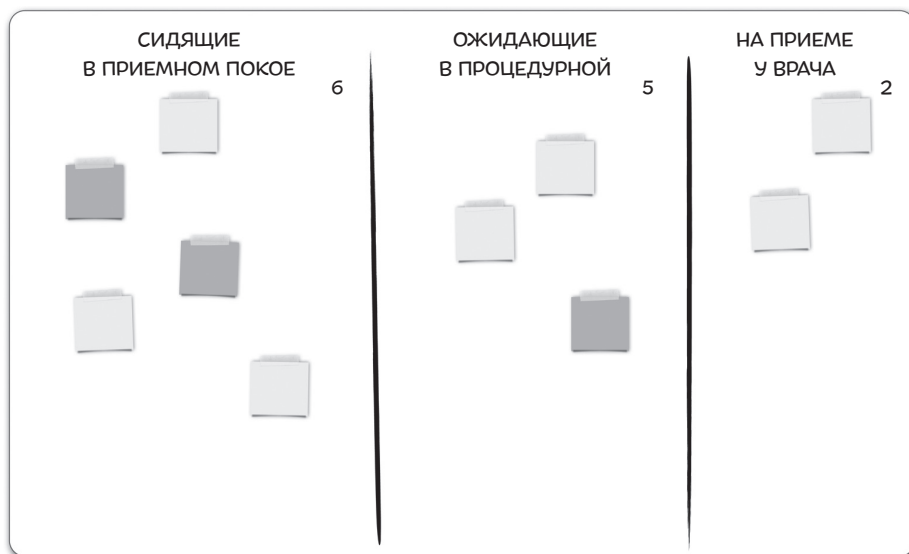


Рис. 9.13. Персонал устанавливает политику, ограничивающую WIP-лимит шестью пациентами в приемном покое. Они придерживаются этой политики, обзванивая пациентов и перенося записи, как только число ожидающих достигает лимита. Для обозначения пациентов с серьезными проблемами, чье время приема нельзя перенести, используются розовые стикеры

ЗАКОН ЛИТТЛА ПОЗВОЛЯЕТ КОНТРОЛИРОВАТЬ ДВИЖЕНИЕ ПОТОКА ЧЕРЕЗ СИСТЕМУ

Сотрудники офиса отнеслись к канбан-практике улучшения очень серьезно. И экспериментальным путем пришли к интересному выводу: они обнаружили, что после введения WIP-лимитов появилась возможность контролировать время ожидания пациентом врачебного приема. Если на каждый час планировалось много визитов, то четырем-пяти пациентам пришлось бы долго ждать. А если запись сокращалась, то перед кабинетом врача сидели бы только два-три пациента, и ждать им пришлось бы меньше. Впервые

они почувствовали, что действительно контролируют систему, которая в прошлом вызывала головную боль.

Так что же происходит?

Сотрудники клиники обнаружили существование в стабильной системе связи между списком приема (количеством), временем обслуживания и скоростью поступления. Например, если персонал каждый час записывает 11 пациентов (так как скорость поступления λ равна 1 часу) и среднее количество в течение дня превышает семь пациентов (так как L равно 7), то закон Литтла говорит нам, что среднее время ожидания пациентами своего приема составит:

$$W = L \div \lambda = 7 \text{ пациентов} \div (11 \text{ часов}) = 0,63 \text{ часа} = 37 \text{ минут.}$$

Но вдруг после ряда экспериментов они обнаружат, что планирование 10 пациентов, прибывающих каждый час, снижает список до четырех пациентов? В час пик все процедурные переполнены, но большую часть времени один пациент находится в приемном покое, один ожидает приема врача в процедурной комнате и двое — на приеме у врача:

$$W = 4 \text{ пациента} \div (10 \text{ пациентов каждый час}) = 0,4 \text{ часа} = 24 \text{ минуты.}$$

При помощи канбан-доски, CFD и экспериментирования с WIP-лимитом сотрудники клиники обнаружили, что могли бы уменьшить время ожидания пациента почти на 15 минут только за счет сокращения записи на одного человека в час.

Это работает, потому что закон Литтла говорит нам, что в стабильной системе на время обслуживания пациента влияет две вещи: очередь и скорость прибытия. А WIP-лимиты позволяют контролировать одну из них. При добавлении WIP-лимита на канбан-доску вы можете уменьшить неравномерность, которая приводит к нагромождению в очереди. Это дает возможность сократить время обслуживания за счет снижения скорости поступления (например, удерживая работы в бэклоге до тех пор, пока команда занята выполнением текущего задания, как делают scrum-команды, или уменьшая число пациентов, записанных в течение каждого часа.

Именно так сотрудники клиники могут использовать закон Литтла для расчета среднего времени, затраченного на прием одного пациента. Но даже если они не рассчитают это время, *они по-прежнему зависят от него*. Причина в том, что закон Литтла действует в любой стабильной системе. Этот доказанный математический закон применяется к любой системе, имеющей долгосрочную очередь.

Теперь рассмотрим пример, приближенный к реальной жизни. Допустим, каждые три недели ваша команда вязнет в работе по выпуску релиза.

К сожалению, дела идут не очень хорошо. Сначала все было отлично, но со временем возникли проблемы. Несмотря на все усилия, потраченные на выполнение работы, команда чувствует, что не имеет достаточно времени на ее завершение. И каждый испытывает более сильный стресс, чем в предыдущем месяце. К тому же все знают: если они не выполнят работу максимально быстро, то в следующем месяце будет еще хуже.

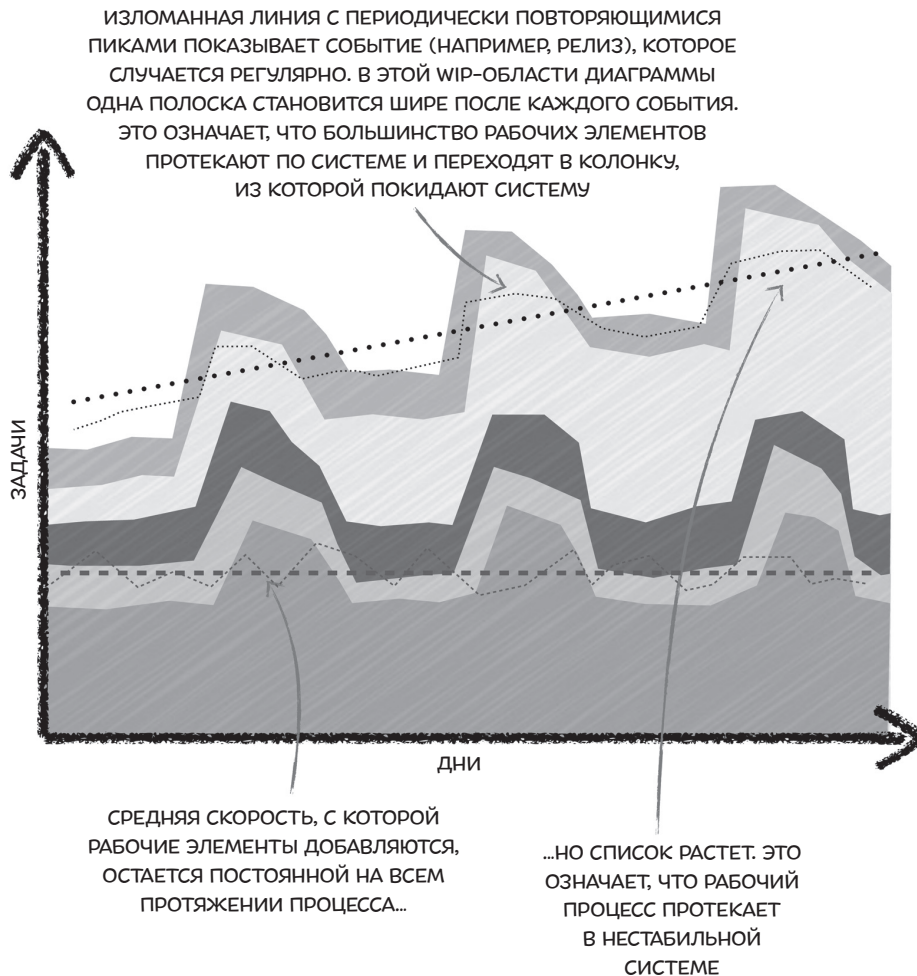


Рис. 9.14. Мы вернулись к области WIP на диаграмме, чтобы лучше рассмотреть стабильность системы. Эта диаграмма показывает: скорость прибытия постоянна, но среднее общее количество запросов со временем увеличивается, что означает нестабильность системы

Это знакомо многим командам. Они чувствуют, как медленно вязнут в проблемах, и в конце концов все становится настолько плохо, что некогда думать о работе. В главе 7 рассказывалось, как нарушение рабочей атмосферы, подобно этому примеру, может влиять на код: в итоге разработчики собирают плохо спроектированное программное обеспечение и создают такой исходный код, который имеет технический долг и с которым сложно работать.

Как можно решить эту проблему? Она не всегда становится очевидной даже при ежедневном осмотре канбан-доски, потому что большую часть времени ситуация на ней выглядит вполне приемлемой. На ней бывают дополнительные стикеры, пока команда занимается релизом, но это ожидаемо. В конце концов эти стикеры перемещаются по доске, и кажется, что ситуация вернется в норму. Но команда это нормой не считает (хуже, если она воспринимает это как норму). Могут ли инструменты, о которых мы узнали, помочь выявить и исправить проблемы в команде? Давайте посмотрим.

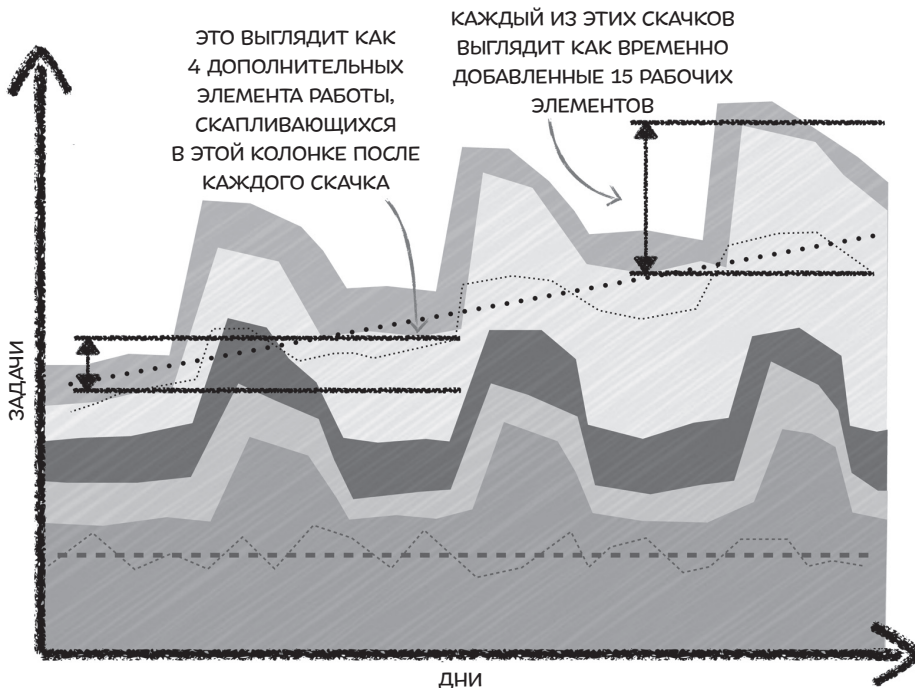


Рис. 9.15. Эта WIP-область диаграммы также имеет подсказки, которые помогут выяснить способы стабилизации системы. Общее количество запросов увеличивается после периодических всплесков в работе. Если мы выясним количество скопившихся запросов, то это поможет выбрать хорошую отправную точку для экспериментов с WIP-лимитом

Дополнительная нагрузка во время работы над каждым релизом приводит к заметному скачку на CFD. Помните, что каждая полоса соответствует одной из колонок на доске, а толщина полосы показывает количество стикеров в соответствующей колонке в этот день. На данной диаграмме одна из полос становится толще после каждого релиза, и со временем это приводит к общему росту диаграммы.

WIP-область диаграммы проясняет причину этой проблемы: работа **накапливается** в этой колонке, потому что количество работ, поступающих в систему, превышает количество работ, выходящих из нее. Если команда не изменит ситуацию, то с каждым новым релизом эти работы будут проталкиваться назад, а полоса — расширяться после каждого релиза.

Мы видим, что общее количество запросов растет, а это означает нестабильность системы и остановку потока работы через систему. Неудивительно, что команда чувствует, как вязнет.

Команда может не осознавать, что работа «застревает» в одном столбце на канбан-доске. WIP-диаграмма визуализирует это, потому что полоса, соответствующая этому столбцу, со временем утолщается. К счастью, у нас есть инструмент, чтобы сгладить эту неравномерность, — добавление WIP-лимита. Измерение количества «застывших» элементов работы поможет найти хорошую отправную точку.

Пики еще есть, но *уже нет увеличения общего числа рабочих элементов*. Путем добавления ограничений мы остановили поток дополнительной работы, впадающий в систему, который увеличивал общий поток проекта в целом. Работа распределяется более равномерно по всем столбцам. Когда перегруженный столбец достигает WIP-лимита, команда переносит свое внимание на элементы работы в предыдущих колонках. Вы можете видеть это на WIP-диаграмме: подъемы кривой на нижней полосе меньше, чем они были до введения WIP-лимита.

Канбан-команды не просто устанавливают WIP-лимит. Они реализуют петлю обратной связи: поскольку проект продолжается, они постоянно регулируют WIP-лимит на основе новой информации. Если по-прежнему происходит накопление задач, то команда экспериментирует с WIP-лимитом до тех пор, пока не найдет значение, определяющее оптимальный поток. Канбан-команды серьезно относятся к эксперименту: они не просто случайным образом устанавливают эти WIP-лимиты, но проводят эксперименты, создавая гипотезы о том, как WIP-лимиты будут влиять на систему, и тщательно доказывают их, проводя измерения. Это пример того, как команда может улучшить совместную работу и эволюционировать экспериментально.

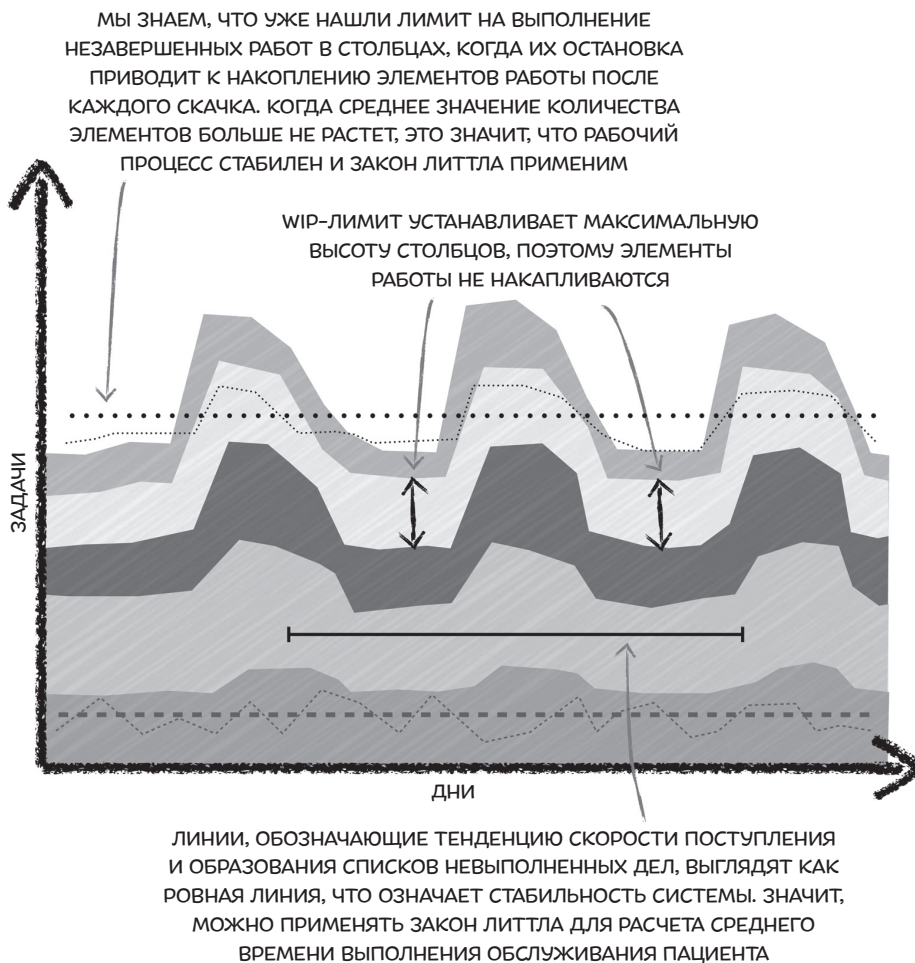


Рис. 9.16. Когда вы находите удачное значение для WIP-лимита, в столбце больше не хранится работа, а полосы WIP-области на диаграмме не накапливают работу со временем. Как это выглядит на диаграмме? Как вы думаете, будет ли диаграмма CFD визуализировать эту конкретную проблему лучше, чем WIP-область на диаграмме?

По мере того как команда совместными усилиями развивает свою систему, она чувствует увеличение потока в повседневной работе. Вспомогательные задачи не вызывают ощущения, будто команда откладывает важную работу по созданию нового функционала, потому что работа по поддержанию так же значима, как и сама разработка функций. Помещая рабочие элементы по сопровождению на доску, команда эффективно продвигает

их в число приоритетных работ проекта, может сосредоточить на них свое внимание и лучше выполнять работу (вместо попыток втиснуть эти задачи в свои планы и пытаться спешно их преодолеть, создавая себе лишние трудности).

Существует дополнительное долгосрочное преимущество. Многие проблемы поддержки вызваны технической задолженностью, образовавшейся из-за того, что команда старалась не отставать и была вынуждена накапливать долг. Теперь, когда у нее есть время правильно выполнять работу, она обнаруживает, что будет иметь меньше проблем с поддержкой. Тем временем команда может получать удовлетворение от повышения сосредоточенности на задаче и более стимулирующей рабочей среды, которая позволит ей лучше создавать программное обеспечение.

ЕСЛИ КОМАНДА ПО-ПРЕЖНЕМУ ДОЛЖНА ВЫПОЛНЯТЬ РАБОТУ, СВЯЗАННУЮ С ПОДДЕРЖКОЙ, ТО ЧТО СЛУЧИТСЯ С ДРУГИМИ ЕЕ ЗАДАЧАМИ? НЕ БУДУТ ЛИ ОНИ НАКАПЛИВАТЬСЯ ГДЕ-НИБУДЬ ЕЩЕ?

Нет. Дополнительная работа больше не накапливается, потому что она *не добавляется в проект по принципу первоочередности*. Эта команда испытывала стресс, потому что тратила все усилия на разработку программного обеспечения, но одновременно вынуждена была каждые несколько недель останавливаться и сосредоточиваться на поддержке без ущерба для разработки. Таков был выбор руководства. Точнее, руководитель не сделал правильного выбора. Его магическое мышление позволяло ему делать вид, что команда готова справляться с нагрузкой по разработке и с дополнительной работой по поддержке каждые несколько недель. Объем работ вынудит руководителей выбирать, что именно из них команда будет выполнять.

Но что если работа по поддержанию занимает все места в очереди? Как тогда будет совершаться разработка?

Если работа по поддержанию превалирует в очереди, значит, руководитель уделяет приоритетное внимание не разработке. Это главный приоритет команды, независимо от того, признает это руководство или нет. Размещение рабочих элементов для поддержки на канбан-доске помогает команде сосредоточиться на заслуживающих внимания первоочередных элементах. Программисты не виноваты, что больше не занимаются новой разработкой. Вероятно, многие из них вовсе не готовы выступать в роли круглосуточной службы поддержки. Но это лучше, чем отвечать за нее и одновременно заниматься разработкой.

Теперь руководство имеет гораздо более точную информацию об успехах команды. Это один из принципов agile-разработки программного обеспечения, описанный в главе 3: *работающее программное обеспечение — это главная мера прогресса*. Раньше казалось, что перегруженная команда способна производить программное обеспечение и поддерживать его. Не всем было очевидно, что дополнительная нагрузка вынуждает ее снижать качество ПО и приводит к тому, что его трудно поддерживать, а также потенциально является причиной некоторых случаев, когда поддержка необходима. Теперь, когда команда сократила поставку программного обеспечения, руководитель может точнее оценить прогресс. Это мешает ему делать вид, будто люди могут выполнить гораздо больше того, что в человеческих силах. Если он хочет получить больше ПО, то должен либо поставить это в приоритет над поддержкой, либо нанять больше сотрудников. Но намного легче просто обвинять команду.

УПРАВЛЕНИЕ ПОТОКОМ С WIP-ЛИМИТОМ ЕСТЕСТВЕННЫМ ПУТЕМ СОЗДАЕТ ВРЕМЕННОЙ РЕЗЕРВ

Разработчикам нужен временной резерв, или «пространство для маневра», в расписании. Это дает уверенность, что они успеют хорошо и в срок выполнить работу. В главах, посвященных XP, мы узнали: при дефиците времени команды «срезают углы» и создают технический долг. Им хочется сделать текущую работу как можно быстрее, потому что впереди всегда больше дел, чем позади.

Атмосфера постоянного отставания оставляет мало места для творчества и душит инновации на каждом шагу. А также убивает качество практик, потому что команды, испытывающие непрерывную гонку, часто не успевают создать ничего, кроме кода. Именно поэтому XP-команды включают временной запас в свои основные практики.

Команды, использующие Канбан, также ценят временной запас и осознают его влияние на способность каждого своего члена хорошо выполнять свою работу. Поэтому они устанавливают лимит количества текущих работ.

Когда команды вместо следования строгому расписанию используют Канбан для улучшения процесса, они внедряют **ритмичную поставку**. Для этого они обязуются поставлять программное обеспечение на регулярной основе (например, каждые шесть недель), но не фиксируют определенный список рабочих элементов, которые будут включены в релиз. Вместо этого они доверяют своей системе поставки рабочих элементов. Если они уже избавились от перегрузок и неравномерной работы, то

будут получать список незавершенных рабочих элементов в каждой поставке.

Так что же заставляет программиста или другого члена команды ощущать нехватку времени? Это чувство вызвано осознанием объема задач, которые нужно выполнить, и пониманием, что текущая работа блокирует остальной проект. Когда кто-то чувствует, что текущее задание — это преграда, мешающая выполнению работы, он старается сделать его как можно быстрее.

Именно поэтому многие люди с удивлением реагируют на то, что их руководитель согласился на WIP-лимит: они *испытывают облегчение*.

До введения WIP-лимита всем казалось, что дополнительная работа «просачивается» в спринт, и команда вынуждена была полагаться на временной резерв, чтобы ее выполнить. Она всегда начинала спринт при условии, что определено только 70% работы, потому что нетерпеливые руководители и пользователи будут искать возможность втиснуть в последнюю минуту еще 30% (а то и все 40% или даже больше) изменений и срочных просьб.

После введения WIP-лимита (и, что не менее важно, согласия на его поддержание) дополнительные просьбы по-прежнему будут появляться, но теперь команда не обязана выполнять всю незапланированную работу. Вместо этого из новых задач формируется очередь. Давление на команду уменьшается, потому что объем работ будет ограничен и не будет накапливаться. Она может управлять потоком (возможно, используя CFD), так как знает, что WIP-лимит гарантирует необходимое время.

Именно поэтому многие канбан-команды вводят WIP-лимиты в каждом столбце канбан-доски.

Это помогает команде контролировать поток на каждом шаге разработки. Даже для столбца «Готово к выпуску» тоже есть WIP-лимит. Если выполненной работы накапливается слишком много, то команда может заниматься другими делами, чтобы подготовиться к следующему релизу, — и теперь она располагает большим объемом информации, чтобы отрегулировать ритмичность поставок в будущем за счет сокращения времени между релизами.

Команде не всегда удастся получить общее согласие по всем WIP-лимитам с первого раза, поэтому канбан-команды придерживаются стратегии совместного улучшения и эволюционного развития рабочего цикла. После того как менеджеры увидели, что WIP-лимиты помогают команде быстрее разрабатывать программное обеспечение и сокращать время выполнения нового заказа после первого раунда улучшения, они охотнее соглашались на введение WIP-лимитов в последующих раундах.

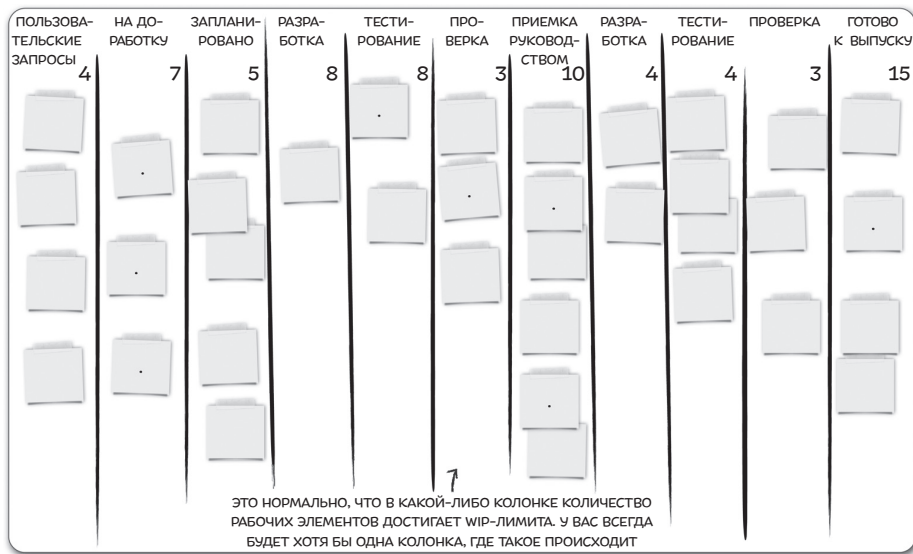


Рис. 9.17. Введение WIP-лимитов в каждой столбце канбан-доски помогает команде максимизировать поток на протяжении всего проекта

Управление потоком на протяжении всего проекта помогает каждому члену команды сосредоточиться на выполнении текущей работы, не беспокоясь, что задачи накапливаются. Они могут доверять WIP-лимитах защите от хаоса. И знают, что, как только в системе появляется беспорядок, он становится заметным, когда измеряется рабочий процесс. Поэтому можно регулировать пределы WIP-лимитов, чтобы сохранить целенаправленность работы и ее поток.

СДЕЛАЙТЕ ПРАВИЛА ЯВНЫМИ, ЧТОБЫ ВСЕ ПОНИМАЛИ ИХ ОДИНАКОВО

Что бы произошло, если бы вы попросили всех членов эффективной scrum-команды написать подробную инструкцию, как они разрабатывают программное обеспечение? Вероятность того, что их описания будут совпадать, очень высока. Потому что они знакомы с правилами Scrum и постоянно с ними работают. Каждый отдельный участник имеет целостное понимание того, как команда разрабатывает ПО, а правила scrum-проекта просты и любой способен их понять.

А что если вы попросите об этом же неэффективную команду с типичным водопадным процессом? Велика вероятность того, что у них раздробленное

видение (мы писали об этом в главе 2). Наверняка каждый расскажет, как выполняет свою ежедневную работу: разработчики будут писать о кодировании, тестировщики — о тестировании, бизнес-аналитики — о сборе требований и т. д. Руководитель проекта *может* иметь более широкую перспективу, потому что должен знать роль каждого в создании плана проекта, поэтому, вероятно, включит в описание работу всей команды. Но также не исключено, что он просто опишет шаги, которые необходимы для планирования и отслеживания проекта.

Иногда сложный процесс важен и полезен, а в других случаях — расточителен и бюрократичен. Команде может быть привычно рассылать массу писем, как только обновляется спецификация, и она не может (или не хочет) ничего менять, пока достаточное количество людей не примет в этом участия.

Это отдельный процесс, и даже если он не прописан, то является частью знаний, принадлежащих «племени». Как вы узнаете, в какую из категорий попадет эта конкретная спецификация обновленных процессов? Ответ дает канбан-практика **визуализации правил** — описания работы команды и ознакомления с ним всех, кого это затрагивает. Возможно, здесь необходим сложный процесс. Например, менеджер проекта может указать на то, что наличие спецификации процесса контроля изменений обязательно для соблюдения нормативных требований. Но чаще всего неписаная стратегия может заставить людей покачать головами и согласиться, что это глупо.

Канбан-командам не нужно писать длинные документы или создавать огромные вики-проекты, добиваясь того, чтобы правила стали явными. Они могут быть простыми, как и WIP-лимиты на вершинах столбцов. Команды также описывают их, добавляя заметки «сделано» или «критерии выхода» в нижней части каждого столбца на канбан-доске. Это позволит всем членам команды точно знать, в каких случаях можно передвигать рабочие элементы во время рабочего процесса. Это особенно эффективно, когда правила создавались совместно и развивались экспериментально всей командой, так как это означает, что все понимают их смысл.

Сложные процессы и неписанные правила обычно возникают со временем и особенно распространены в командах с раздробленным видением. Усложнение процессов контроля изменений может произойти, потому что для бизнес-аналитика проблематично поддерживать большое количество изменений в последнюю минуту и он громко и часто возмущается при всей команде, что «наше программное обеспечение не удовлетворяет потребностям конкретного пользователя». Он может усложнить процесс, чтобы контролировать границы изменений и убедиться, что существуют бумажные документы, благодаря которым можно спасти свою шкуру, если

в дальнейшем кто-нибудь решит изменить мнение. Трудно винить бизнес-аналитика в создании оборонительной бюрократии. Это может оказаться единственно возможным способом контролировать спецификацию, за которую он несет ответственность.

Установление WIP-лимитов — это стратегия выбора, и работает она только потому, что каждый соблюдает соглашение не проталкивать дополнительную работу, если очередь достигла своего предела. Фиксация договоренностей (особенно если они находятся на видном месте, как канбан-доска) помогает убедиться в том, что каждый согласен с ними, после того как они записаны. Команда может указывать на записанные правила каждый раз, когда ретивый менеджер или пользователь пытается протолкнуть дополнительную работу в очередь. Это дает прочную основу, когда нужно убрать какой-нибудь элемент из очереди, чтобы освободить место для срочного запроса. Команде гораздо проще обсуждать это, если менеджер или пользователь уже согласны со стратегией, которая дает ясное понимание вещей.

Эмерджентное поведение с Канбаном

*Чем сильнее вы сожмете кулак, Таркин, тем больше звездных систем
ускользнет сквозь ваши пальцы.*

Принцесса Лея

Вспомним пример с медицинским центром, приведенный выше. Если бы вы попросили команду, которой руководит менеджер с командно-административным стилем управления, помочь персоналу улучшить процессы, то что бы произошло? Первым делом оценили бы, сколько времени врач тратит на каждого пациента. Это потребовало бы вовлечения врачей, медсестер и сотрудников клиники, но позволило бы руководителям проекта взять под контроль систему, создать подробное описание всех ресурсов (процедурных комнат, врачей и медсестер) и на микроуровне управлять каждым аспектом практики.

Такой способ успокоил бы всех, но, к сожалению, он неэффективен. Врачи учились лечить, а не оценивать. Руководители проекта могли бы придумать идеальное расписание, но более вероятно, что они создадут нереальное расписание, потому что врачи дадут им скудные данные. Именно поэтому Канбан смотрит на всю систему *в совокупности*. Вместо попыток заниматься микроменеджментом любой небольшой деятельности команда использует системы мышления Канбан, чтобы понять, измерить и постепенно совершенствовать систему в целом. Принимается тот факт, что отдельные элементы работы

могут меняться, но система в целом действует предсказуемо, когда потери, неравномерность и излишек (мúда, мúра и мúри) снижаются.

Когда команда использует Канбан, чтобы постепенно улучшить процесс сборки программного обеспечения, зачастую начинает меняться и вся компания. Рассмотрим пример компании, пытающейся сократить время выполнения задачи. До попытки команды использовать Канбан менеджеры и пользователи ругали ее за отсутствие немедленной реакции на запросы пользователей. В ответ команда создала планы проекта, которые «доказали», что ничего не произошло. Это привело к конфликту и негативным эмоциям.

Канбан помог решить основную проблему. Команда использовала метод пяти «почему», чтобы найти причину удлинения времени выполнения задач, и ввела WIP-лимиты для исправления ошибки.

Но как это решило проблему? Давайте вернемся к первым двум канбан-практикам, о которых говорилось в этой главе, — **совместному улучшению, экспериментальному развитию и реализации петли обратной связи**. Они помогают понять, что происходит. Во-первых, *в буквальном смысле* изменилось восприятие процессов не только всеми членами команды, но и руководителем. Визуализация рабочего процесса при помощи канбан-доски выявила раздробленное видение. Менеджеры смогли понять, что работы скапливались, и это убедило их в необходимости введения WIP-лимитов, а также уменьшения управляющих воздействий на команду. Вот пример того, как лимитирование незавершенных работ меняет поведение.

Рассмотрим ситуацию, когда команда постоянно получает запросы от разных менеджеров и ей приходится соблюдать баланс чужих интересов. Каждый руководитель считает свою просьбу самой важной. Если команда работает по запросу одного менеджера, то другой чувствует себя оскорбленным. Все испытывают большое давление, потому что столкнулись с неразрешимой проблемой.

Если эта команда использует Канбан для визуализации рабочего процесса, то все запросы менеджеров по написанию функций будут в итоге находиться в качестве рабочих элементов в первом столбце. Если запросов больше, чем команда способна выполнить, то стикеры накопятся в столбце и каждый сможет это увидеть. Теперь разработчикам понятно, что делать, — нужно получить согласие всех менеджеров на введение WIP-лимита в этом столбце.

Допустим, они договорились и ввели лимит на выполнение незавершенных работ. Менеджеры продолжают добавлять новые функции до тех пор, пока этот WIP-лимит не будет исчерпан. Но как только первый менеджер достигает лимита, он больше не может просто так добавить новый стикер

на доску, для этого придется удалить какой-то другой. Если он не хочет удалять один из своих стикеров, нужно будет договариваться с другими менеджерами.

Повторимся: когда менеджер уперся в WIP-лимит, он не обвинил команду. Он *признал это как ограничение системы* и занялся поиском решения с коллегами-менеджерами. Перед нами важная часть системного мышления. Когда все, кто имеет дело с системой, осознают это, они работают в ее рамках, чтобы решать свои проблемы. И так как все понимают, как работает система, потери и неравномерная нагрузка больше не являются проблемой только одной команды. Это общая проблема, в том числе и для менеджеров.

Так *новое поведение проявляется и вне команды*. Вместо того чтобы просто обвинять разработчиков за неравномерную нагрузку, в чем необязательно они виноваты, всех начинают волновать стикеры в первом столбце, потому что именно ими будет заниматься команда. Можно организовывать встречи для определения приоритетов, устраивать «торги» между собой или искать любые другие пути, чтобы решить, какие задачи выполнять в первую очередь. Но, главное, команде больше не нужно брать на себя вину, потому что она не должна работать на пределе возможностей.

Другими словами, до внедрения Канбана разработчикам приходилось иметь дело с руководителями, считающими, что команда может выполнить неограниченное количество задач. И они всегда были разочарованы, когда результаты работы не соответствовали обещаниям. С введением WIP-лимита менеджеры с магическим мышлением перестали так думать. Они пересмотрели свое поведение не из желания что-то изменить, а потому что работали в системе, которая побуждала их действовать по-другому. В результате у команды появился временной запас, необходимый для отдыха и качественной работы. Все силы, которые раньше тратились на отдельные переговоры с менеджерами по удовлетворению их запросов, теперь полностью отдаются работе. Это гораздо более эффективный способ управления командой.



Ключевые моменты

*Одна из важных целей, которую ставит перед собой канбан-команда, — максимизация **потока** или скорости, с которой рабочие элементы перемещаются по системе.*

*Канбан-практика **измерения и управления потоком** подразумевает измерение потока и корректировку процесса для его максимизации.*

Кумулятивная диаграмма потока (CFD) показывает лимит на выполнение незавершенных работ (*WIP-лимит*), число рабочих элементов, добавляемых ежедневно (*скорость поступления*), общее количество рабочих элементов в рабочем процессе (*список*) и среднее время нахождения рабочих элементов в системе (*время на выполнение заказа*).

Когда скорость поступления и список со временем не меняются, система **стабильна**, канбан-команды устанавливают *WIP-лимит* для ее стабилизации.

Когда система стабильна, к ней можно применять **закон Литтла**, который означает, что среднее время на выполнение заказа всегда равно долгосрочной скорости поступления, поделенной на долгосрочный список.

Если ваша команда может стабилизировать рабочий процесс при помощи *WIP-лимитов*, то вы сумеете уменьшить время выполнения заказа для пользователей при условии, что они не будут добавлять новые рабочие элементы, сокращающие скорость поступления.

Канбан-команды часто прибегают к **процессу явной стратегии**, добавляя определение понятия «сделано» или дополнительный критерий внизу каждого столбца на канбан-доске.

Когда канбан-команды постепенно совершенствуют систему, добавляя *WIP-лимиты*, управляя потоком и используя явную стратегию, они тем самым улучшают управление остальной компанией.



Часто задаваемые вопросы

Вы говорите о *WIP-лимитах* и явной стратегии как об инструментах, способных изменить работу команды. Но этот аргумент кажется надуманным. Или это все-таки правда?

Удивительно, но это правда. Чтобы понять почему, вернемся к истокам Lean и Канбана.

WIP-лимиты — простой, но эффективный инструмент для сглаживания потока, и его эффективность известна уже давно. Канбан базируется на сигнальной системе для создания вытягивающей производственной системы, которая возникла в компании Toyota в 1950-е годы

(как и многие идеи и инструменты Lean). Название «канбан» происходит от японского слова, которое означает «сигнальная карточка» (буквально «вывеска» или «рекламный щит»); для обозначения сигнальной карточки принято писать слово «канбан» с маленькой буквы. На станции сборочной линии завода-изготовителя использовалось определенное количество канбан-карточек, на которых печатался номер каждой детали или ее название, требующейся на сборочной линии.

По мере окончания комплектующих на этапе сборочной линии канбан-карточка вставлялась в пустой слот на тележке для деталей; это сигнализировало, что их количество надо увеличить. Так выглядел цикл пополнения сборочной линии, а производственная команда использовала только необходимые в данный момент комплектующие, что позволило всей компании уменьшить общее количество деталей, которые необходимо хранить на складе. Ограничение количества канбан для каждой детали позволило команде установить WIP-лимиты.

Программистам иногда трудно понять, как систему с использованием канбан-карточек можно применить к разработке программного обеспечения. Им не нравится сравнивать себя с персоналом сборочного конвейера — и это справедливо, потому что разработка скорее похожа на то, что делают автомобильные инженеры и конструкторы, а не рабочие. Но в Канбане не принято относиться к людям как к винтикам механизма. Канбан и конвейер объединяет лишь то, что они используют канбан-карточки на контейнерах с запчастями или стикеры на доске, чтобы сигнализировать: еще не все готово.

Иногда легче понять это при помощи примера, не имеющего ничего общего ни с конвейером, ни с программным обеспечением. Каждое лето на большой выставочной площадке в Сент-Поле проводится ярмарка, которую посещает более миллиона жителей штата. На территории выставочного комплекса располагается множество поставщиков продуктов питания. Один из самых популярных — стенд компании — производителя кукурузы гриль. Другие известные поставщики имеют длинные очереди у своих стендов — например, не менее двух десятков людей выстраиваются, чтобы купить жареные оливки на палочке*. Но там, где посетителям предлагают кукурузу гриль, как правило, огромное количество людей обслуживается практически без очередей. Как они этого добились?

* Поставщик, продающий жаренные во фритюре оливки на палочке, использует типичную push-систему — длинную очередь людей; так было в 2014 году. Они могли перейти на Канбан с тех пор, как была опубликована эта книга.

В отличие от поставщика жареных оливок, который имеет два одинаковых по назначению окна с длинными очередями, на стенде кукурузы гриль созданы два отдельных окна. В первом вы платите деньги и получаете чек, а во втором отдаете чек в обмен на початок кукурузы.

Поначалу участникам ярмарки это казалось излишеством: зачем отдавать деньги одному человеку, брать у него чек, а затем передавать его кому-то другому? Но теперь все вынуждены согласиться с необходимостью этого дополнительного этапа: чек — это канбан-карточка, и поставщик кукурузы гриль создает WIP-лимит и вытягивающую систему.

Люди, готовящие кукурузу во втором окне, знают, сколько чеков выбито, и будут отправлять в первое окно только нужное количество чеков, поэтому очередь ожидающих небольшая. Это позволяет регулировать количество готовящейся кукурузы. Поэтому в периоды замедления спроса они не держат кукурузу на гриле так долго, что она превращается в уголь, а в пиковые моменты увеличивают объемы, чтобы свести очередь к минимуму. Это также означает, что они могут нанять дополнительных людей для трудоемких действий и жарить кукурузу, не отвлекаясь на денежные расчеты. В тех редких случаях, когда количество людей, желающих купить кукурузу, превышает возможности людей внутри киоска, образуется очередь в первом окне. Те, кто готовит кукурузу, защищены от хаоса, связанного с покупателями, пытающимися всучить им деньги. Они могут полностью сконцентрироваться на процессе жарки, что позволяет очереди двигаться быстрее или не появляться вовсе.

Благодаря тому, что их работа течет плавно, процесс приготовления может стать ритмичным и позволяет сотрудникам чувствовать себя более комфортно и хорошо выполнять свою работу.

Регулирование WIP-лимита предоставляет поставщику кукурузы гриль свободу действий. Увеличение чеков приводит к росту количества жарящейся кукурузы в периоды с пиковой нагрузкой (например, после окончания больших шоу на ближайших трибунах). Они могут быстро и плавно нарастить производство, а затем так же плавно его уменьшить после спада спроса. Знают ли продавцы кукурузы, что они применяют закон Литтла? Ограничивая число прибывающих в систему людей, они могут уменьшить время обработки заказа, чтобы клиенты, заплатившие за кукурузу, не ожидали ее. (Вы можете представить, как выглядит кумулятивная диаграмма потока?)

Этот пример помогает проиллюстрировать разницу между выталкивающими и вытягивающими системами. Производители жареных оливок используют выталкивающую систему — длинную очередь. Люди «проталкивают» ее, потому что она привлекает покупателей и создает спрос. Поставщики оливок не имеют контроля над спросом, они могут продавать оливки только следующему человеку из очереди. Продавцы кукурузы гриль «втягивают» очередь, потому что выдают чеки заплатившему клиенту. Они используют их для притягивания людей к своему стенду и могут ограничить количество покупателей, если оно достигает лимита. Так они устанавливают WIP-лимит, позволяющий им уравнивать производственные возможности и спрос. Регулирование WIP-лимита дает возможность мгновенно приспосабливаться к меняющейся ситуации.

У меня нет обилия деталей для сборки автомобиля или кукурузы, которую нужно обжарить на гриле. Я создаю программное обеспечение, поэтому постоянно решаю различные проблемы. Какое отношение ко мне имеют эти примеры?

Канбан имеет к вам самое прямое отношение, потому что это система, позволяющая мгновенно реагировать на изменения в проекте. Вводя лимит на выполнение незавершенных работ и контролируя размер очереди, вы уменьшаете вариабельность. Выделяя источники задержек и применяя такие методы, как кластеризация блокеров (или анализ коренных причин блокировок и усовершенствования, направленные на снижение вероятности и влияния блокеров), мы сокращаем вариабельность. Атака на ее источники, негативно влияющие на экономический результат системы, является одной из основных техник Канбана*.

Командам разработчиков нужно этим заниматься — и много. Можно утверждать, что разработчики гораздо чаще имеют дело с изменчивостью, чем производители автомобилей или поставщики кукурузы. Программное обеспечение отличается от любого другого инжинирингового продукта тем, что оно *изменчиво*. Так как ПО не является физическим объектом, программисты имеют гораздо больше возможностей, чем другие инженеры, изменять его и на более поздних этапах проекта. Но каждое изменение несет риск. И если ваш проект бросало из стороны в сторону вследствие изменившихся требований, то появляются неравномерности (мұра), потому что эти изменения могут стать очень разрушительными.

* Выражаем благодарность Дэвиду Андерсону за то, что он помог нам сформулировать пример того, как Канбан взаимодействует с вариативностью.

Так что же вы можете сделать, чтобы убедиться, что эти неравномерности не сорвут проект?

Большинство водопадных команд мыслят традиционно, считая, что работа выполняется через процесс управления изменениями. Изменения контролируются, замедляются, и, если возможно, их избегают. Изменения в плане должны быть предложены, рассмотрены командой, проанализированы с точки зрения их влияния на проект, а затем согласованы со всеми сторонами, прежде чем они будут реализованы. Это эффективный способ защиты проекта от изменчивости, уменьшающий его риски, — но при этом он гарантирует, что ваша команда создаст именно запланированное изначально независимо от того, на самом ли деле это ценно для пользователей.

Мир традиционного управления проектами выглядит не таким мрачным, как это может показаться. В главах 6 и 7 мы видели, что команда имеет больше возможностей для разработки легко изменяемого программного обеспечения, если ей предоставить спокойную обстановку, в которой удастся работать разумное количество часов, и такую эмоциональную атмосферу, чтобы было проще сосредоточиться на решении проблем. Хорошие руководители проектов признают это. Поэтому они будут добавлять временной запас в график работы, так же как в XP-командах. А управление процессами изменений означает, что они не имеют XP-возможности добавлять задачи с низким приоритетом, которые команда может сместить на более поздние итерации, потому что это потребует изменения плана, а значит, его рассмотрения и утверждения. Вместо этого они будут добавлять *буферы*, или дополнительные задачи, не имеющие реальной значимости, просто заполняющие график, часто используя формулы для определения того, сколько еще можно втиснуть таких «пустых» задач. Теперь они могут следовать первоначальному плану и дать команде достаточно времени для работы — и в то же время избавляются от дополнительной изменчивости, поэтому проект завершается вовремя.

Другими словами, когда команда испытывает трудности с традиционным проектным управлением, в ответ она обычно добавляет в график работ «пустые» задачи — независимо от того, что является причиной проблемы.

Канбан-команды не создают буферы и «пустые» задачи в графике, потому что это скрывает изменчивость от тех, кто старается создавать правильные решения. Ценности бережливого мышления в том, что они позволяют видеть картину целиком и препятствуют сокрытию

информации. Кроме того, бережливое мышление говорит нам, что, когда есть потери, вызванные неравномерностью, мы должны найти причину проблемы и устранить ее. Вместо того чтобы скрывать потери за «пустыми» элементами, канбан-команды показывают их, визуализируя рабочий процесс, вводя WIP-лимиты и другие стратегии, чтобы их исправить. Они делают изменчивость и ее основные причины явными.

Что делать, если мой руководитель не согласен с введением лимитов на выполнение незавершенных работ?

Тогда вы будете иметь проблемы с внедрением Канбана. Когда вы читаете о том, как действует Канбан, кажется, что создание очередей и ограничение потока поможет существенно улучшить работу команды. Но все начинается с введения WIP-лимитов, которые нужны, чтобы ваше руководство согласилось не добавлять дополнительную работу, когда команда достигает WIP-лимита.

Это может стать ахиллесовой пятой при попытке совершенствования, особенно когда руководитель убежден, что мышление приравнивается к действию. Допустим, вы потратили несколько недель или даже месяцев, помогая команде разобраться с Канбаном, работая вместе над визуализацией рабочего процесса, создавая канбан-доску и проводя тщательные измерения для определения WIP-лимита. Теперь вы вместе с другими менеджерами с гордостью приносите свое предложение. Ваш руководитель понимает, что это заслуживает внимания. Он говорит: «Мне нравится канбан-доска, я думаю, что эти измерения замечательны, и полностью вас поддерживаю. Но мне нужно сделать всего одно небольшое изменение. Стереть этот номер в верхней части столбца на доске».

Ему кажется, что это абсолютно несущественно — стереть один маленький номер. Но WIP-лимит — это ноу-хау Канбана. Без него команда не имеет возможности ограничить поток и Канбан не работает.

На интуитивном уровне не всегда понятно, что сокращение работы всего на один шаг позволяет ускорить весь проект. И даже если вы визуализируете рабочий процесс, проводите измерения и выявляете потери, вызванные неравномерной работой, перегрузками и необоснованной или невозможной работой, это может оказаться недостаточно убедительным для руководителя. Когда это не так, он будет ориентироваться на WIP-лимит и видеть в нем компромисс. Может быть, он чувствует, что, удалив его, команда сможет оптимизировать свою работу и предотвратить неполную занятость. Или он считает, что WIP-лимит дает

команде временной запас, и боится, что она будет злоупотреблять им, замедляя темпы работы и устраивая себе отпуск. Либо эта идея просто кажется ему неудачной. А возможно, он чересчур рационален и совершенно лишен магического мышления, поэтому считает так: поставка по готовности приведет к тому, что функция будет переходить из одного релиза в другой и пользователи, увидев это, одуреют и начнут настаивать на увольнении разработчиков и выгонять людей. Но независимо от того, что он думает, удаление WIP-лимита выбивает опору из-под всех действий в стиле Канбана.

Так что же делать, если вы не можете получить согласия руководителей на введение WIP-лимита?

Это отличная возможность, чтобы попробовать Scrum, в котором есть предсказуемые временные ограничения и заранее оговоренный объем, а также руководство со стороны специально выделенного для этого владельца продукта.

Причина, по которой Scrum эффективен, когда менеджеры сталкиваются с ограничениями, в том, что бэклог находится *внутри* проекта. Scrum-команды полагаются на владельца продукта, имеющего полномочия принимать решения, какие рабочие задачи следует добавлять в бэклог и включать в каждый спринт, и вводят его в состав команды. Теперь команда контролирует объем, а руководители и пользователи могут работать с владельцем продукта, чтобы вносить любые изменения, в том числе и в объем работы. Это дает гораздо больше гибкости, чем традиционный процесс управления изменениями, но все равно ограничивает изменчивость, которой подвержена компания. Команда даже использует доску для визуализации рабочего процесса — но это доска задач, а не канбан-доска, потому что содержит задачи, а не рабочие элементы и в ней нет WIP-лимитов или других явных стратегий.

В то же время Канбан имеет очереди и буфера — *внешние* объекты по отношению к проекту.

Соглашаясь с WIP-лимитом, клиенты (руководители, пользователи, владельцы продукта и т. д.) решают, что помещать в очередь. Они должны согласовывать WIP-лимиты и не защищаться от изменчивости, но взамен получают гораздо больше контроля над тем, какие функции создаются и в каком порядке. Используя производственные термины, мы называем это поставкой «точно в срок», потому что команда сама решает, какая задача выполняется следующей, основываясь на рабочих элементах, ожидающих на различных этапах рабочего процесса (стикеры в столбцах на канбан-доске), и эти решения могут

приниматься очень поздно — даже за минуту до начала работы над рабочим элементом.

Если попытка освоить Канбан заканчивается внедрением Scrum — это очень хороший результат. Как только команда заработает последовательно внутри стабильной системы, вы можете ей узнать, что такое процесс усовершенствования. Обнаружив, что команда получает результат «лучше-чем-ничего», считайте это шагом в правильном направлении. Теперь вы можете продолжать двигаться вперед. Использование Канбана улучшает работу команды, создающей программное обеспечение, и является прекрасным способом для прогресса.

Считается, что Канбан — это не система управления проектами, но, когда вы перемещаете рабочие элементы по канбан-доске, это похоже на управление. Вы уверены, что это не система управления проектами?

Да, это вовсе не система управления проектами и не методология создания программного обеспечения. Канбан — это метод улучшения процесса.

Прежде чем вы сможете улучшить процесс, вы должны понимать это. Канбан-доска — прекрасный инструмент, помогающий создавать ПО, потому что это очень эффективный способ визуализации рабочего процесса.

Это также очень эффективный способ исправлять серьезные ошибки, которые часто встречаются в процессе улучшения: команда не всегда начинает работу, действительно хорошо понимая, как сегодня нужно разрабатывать программное обеспечение. Процесс усовершенствования начинается с того, что группа менеджеров и руководитель проекта собираются вместе и создают схемы или описания текущего процесса. Обычно они понимают, что для процесса усовершенствования необходима точка отсчета, поэтому занимаются рассылкой опросов, интервьюируют команды разработчиков и ищут другие способы, чтобы собрать информацию. Затем они объединяют все полученные сведения в документированный процесс и используют его в качестве отправной точки для совершенствования.

Проблема в том, что сбор такой информации — очень сложное дело. Он редко дает реальное представление о том, как на самом деле команды собирают создавать ПО. Представьте, что вы член команды, которого комитет старших руководителей спросил, как работает ваш проект. Вы будете указывать на неудачи? Или постараетесь нарисовать радужную картину? Даже если вы стремитесь быть максимально правдивым, вам

не избежать обычной слабости, свойственной любому человеку: хорошо помнить успехи и забывать о проблемах.

Вот где лучше всего работает Канбан. Вначале команда визуализирует рабочий процесс на канбан-доске. Но не останавливается на достигнутом — по мере развития проекта она обновляет рабочие элементы на доске. Все члены команды понимают, что делают это не для управления проектом, а чтобы создать точную картину системы, которую используют для производства программного обеспечения. Это действительно реальная картина, потому что команда постоянно обновляет ее. И это очень полезно: когда программисты хотят выяснить, над какой пользовательской историей или функцией работать дальше, они четко видят, как работа протекает через систему. Но канбан-доску не используют для того, чтобы выяснить, какие задания следует выполнять. На ней присутствует информация об уровне рабочих элементов, но не задач. Если система включает доску задач, диаграмму Ганта или какой-то другой способ управления проектом, то они по-прежнему будут его применять.

Из-за того, что Канбан имеет такой эффективный способ визуализации рабочего процесса (канбан-доска) и измерения потока через систему (при помощи CFD), команда способна достичь наиболее эффективного улучшения. Но это не единственная причина, по которой команды, использующие Канбан, добиваются успеха в процессе совершенствования. Канбан позволяет получить отличные результаты, когда команда внедряет бережливое мышление. Как мы видели в примере со Scrum и XP, самый эффективный способ помочь команде принять бережливое мышление — использовать канбан-практики. Как только команда начинает визуализировать рабочий процесс и измерять поток, она понимает, где происходят потери в системе, и использует варианты мышления, чтобы создавать для себя новые возможности и учиться видеть работу системы в целом.



Что вы можете сделать сегодня

Предлагаем несколько вариантов действий, которые вы можете предпринять уже сегодня (самостоятельно или вместе с вашей командой).

- Если бы вам предложили создать канбан-доску сегодня, то какие столбцы вы бы в нее включили? Если вы еще не создавали карты потока

ценности, то сейчас подходящий случай сделать это — они помогут выяснить, как может выглядеть канбан-доска.

- Можно ли рассматривать один из этих столбцов в качестве удачного кандидата для WIP-лимита? Создайте стикеры с рабочими элементами только для этого столбца. Сколько из них задействовано прямо сейчас? Что было бы, если бы вы ввели WIP-лимит?
- Выясните, к кому вы можете обратиться, чтобы ввести WIP-лимит.
- Создайте простую канбан-доску и еженедельно отслеживайте проект. Постройте CFD на основе ваших данных. Стабилен ли ваш список незавершенных дел? А как насчет скорости поступления?



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше о канбан-методе в книге Дэвида Андерсона «Канбан. Альтернативный путь в Agile» (М. : Манн, Иванов и Фербер, 2017).
- Узнайте больше о системах мышления в книге Мэри и Тома Поппендик *Lean Software Development: An Agile Toolkit* (Addison-Wesley 2003).



Подсказки

Здесь мы предлагаем советы для agile-коучей, помогающих своей команде разрабатывать идеи этой главы.

- Канбан начинается с бережливого мышления, поэтому первым делом помогите вашей команде освоить это понятие. Вам помогут подсказки для коучей в главе 8.
- Самый большой барьер для команд, стремящихся использовать Канбан как средство усовершенствования их процесса, — понимание того, что этот метод — не система управления проектами. В качестве

коуча вы подскажете, что это означает для некоторых систем управления проектами и как Канбан помогает командам лучше понять собственную систему, а не управлять проектами.

- Помогите команде научиться видеть преждевременность некоторых ее поступков, что становятся причиной потерь в проекте. Научите людей определять параметры и принимать решения в последний ответственный момент. Некоторые нервные менеджеры или менеджеры с магическим мышлением часто требуют, чтобы команды определились со сроками. Помогите команде научиться выполнять только те обязательства, которые необходимы для проекта.
- Канбан также нуждается в командной работе, чтобы принять системное мышление. Это важная часть Lean и одна из основных идей, которая делает Канбан работающим методом. Помощь команде в визуализации рабочего процесса при помощи канбан-доски — это зачастую удачный первый шаг на пути психологического освобождения людей от роли винтиков в механизме, выполняющем работу.

ГЛАВА 10

Agile-коуч

Вы уже познакомились со Scrum, XP, Lean и Канбаном, знаете, что у них общего, и понимаете, какие задачи они решают. Если вы работаете над разработкой программного обеспечения, то обратили внимание по крайней мере на несколько вещей (практики, идеи, изменения в отношениях), которые могут помочь вашей команде.

А теперь идите и сделайте это. Подтолкните вашу команду к Agile прямо сейчас!

Это кажется почти невыполнимой задачей, не так ли? Есть разница между чтением о ценностях, принципах, мировоззрении и практиках и их внедрением.

Некоторые команды, прочитав книги о Scrum или XP, берут эти практики на вооружение и сразу получают отличные результаты. Все предыдущие девять глав посвящены тому, чтобы вы поняли, почему это возможно: такие команды уже обладают мышлением, совместимым с ценностями и принципами Agile-манифеста и методологией. Они с легкостью внедряют Agile, потому что им не приходится менять мировоззрение. Если дела в вашей команде обстоят точно так же, то вы имеете гораздо больше шансов на успех.

Но что если образ ваших мыслей несовместим со Scrum, XP или другими гибкими методологиями? А атмосфера, в которой вы работаете, не позволяет добиться успеха в применении agile-ценностей? Как быть, если вклад каждого отдельного участника ценится выше командной работы, а за ошибки полагается суровое наказание? Что если среда душит инновации или ваша команда не имеет доступа к клиентам и другим людям, имеющим возможность помочь вам понять, какое программное обеспечение вы производите? Все это — барьеры на пути внедрения Agile.

Вот когда нужен **agile-коуч** — человек, который помогает команде внедрять Agile. Благодаря ему каждый член команды узнает о новом отношении,

мировоззрении и преодолевает психологические, эмоциональные и технические преграды, мешающие внедрению Agile. Коучи работают с каждым членом команды, чтобы те смогли понять не только «как» применять новую практику, но и «почему» ее нужно использовать. Они помогают команде преодолевать естественную неприязнь и даже страх перед изменением, что происходит с теми, кого просят попробовать что-то новое в работе.

В этой книге есть множество примеров, когда люди получали результаты «лучше-чем-ничего»: команда внедряет практики agile-методологий, но ее члены получают лишь незначительные улучшения, потому что по-настоящему не меняют свои взгляды или отношение к командной работе над созданием программного обеспечения. Другими словами, *команда нуждается в agile-мировоззрении, чтобы получить хорошие результаты от методологии Agile*. Agile-ценности и принципы, описанные в манифесте, помогают команде приобрести правильное мировоззрение, и по этой же причине каждая методология предлагает свои ценности и принципы. Команда получает наилучшие результаты от внедрения Agile, когда мышление каждого ее участника совместимо с ценностями и принципами гибкой разработки и конкретной методологией, которую они внедряют.

Цель agile-коуча — *дать возможность команде получить более гибкое мировоззрение*. Хороший коуч помогает выбрать методологию, наиболее подходящую для уже существующего менталитета, и знакомит команду с ценностями, принципами и практиками методологии так, чтобы она работала на этих людей. Вместе с коучем команда начнет внедрять практики, а затем использовать их, чтобы учиться и усваивать ценности и принципы, постепенно менять свое отношение и приобретать правильное мировоззрение, что поможет не ограничиваться получением результата «лучше-чем-ничего».

В этой главе вы узнаете об agile-коучинге: как команды учатся, а agile-коучи помогают им изменить мировоззрение, чтобы было легче внедрить методологию Agile, и как коуч может сделать вашу команду более гибкой.



Описание: команда, работающая над созданием приложения для камеры мобильного телефона в организации, купленной большой и многопрофильной интернет-компанией

Кэтрин — первый разработчик

Тимати — второй разработчик

Дэн — их руководитель

Акт III. И вот еще что (опять?!)...

— Кэти, у тебя есть минутка? У меня отличные новости.

Кэтрин поморщилась, услышав, что Дэн зовет ее в свой кабинет. Она подумала: «Я никогда не привыкну к этому». Но все-таки зашла, села в кресло и приготовилась слышать «отличные новости».

— Вы проделали замечательную работу, используя гибкие методологии. Просто фантастическую.

Кэтрин припомнила все те восемь месяцев, которые они с Тимати потратили, чтобы при помощи Канбана улучшить процесс работы.

Она выявила несколько узких мест в процессе (одно особенно серьезное), заключающихся в том, что Дэн и другие менеджеры практически наугад спускали вниз задания на разработку новых функций.

После того как она установила очередь на выполнение работ, произошла удивительная вещь: менеджеры начали соперничать за место в очереди и постепенно перестали требовать от нее и Тима выполнения непосильного объема заданий.

Впервые за месяц Дэн вызвал ее, чтобы поделиться действительно «отличной новостью».

Раньше это словосочетание обычно означало, что в последний момент он решил внести изменения, которые пускали под откос весь проект.

— Я рада слышать, что ты счастлив, Дэн, — с усмешкой сказала Кэтрин.

Дэн продолжал:

— В головной компании тоже заметили наше продвижение. Они уже читали про Agile и теперь, когда у нас есть кое-какие успехи, хотят, чтобы мы научили этому другие команды.

— Подожди. Так ты просишь, чтобы я... стала их коучем? — воскликнула Кэтрин.

— Точно, — сказал Дэн.

— Я понятия не имею, как это делается, — ответила Кэтрин.

Дэн посмотрел ей в глаза.

— Это не обсуждается, Кэти. Встряхнись — и за дело.

Коучи понимают, почему люди не всегда хотят меняться

Большинство людей в вашей компании стараются хорошо выполнять свою работу. Они хотят, чтобы их коллеги и руководители видели, что они хорошо выполняют возложенные на них задачи. Когда кто-то достигает определенного уровня комфорта в работе и знает ее досконально, последнее, чего бы он или она хотели, — это чтобы кто-то пришел и заставил их работать по-другому.

Эндрю Стеллман и Дженнифер Грин. Applied Software Project Management

Большую часть своего времени agile-коучи тратят на то, чтобы помочь людям изменить способ их работы. Это непростая задача, потому что только коуч видит всю картину целиком. Люди, которых просят работать по новым правилам, могут не понимать, зачем им это.

Существует много способов такого внедрения практики, когда команда, имеющая самые благие намерения, умудряется сделать так, что практика не работает. Например, в главе 5 описано, что команды часто превращают ежедневный scrum-митинг в статус-митинг. Важная задача scrum-митинга — замена командно-административной формы управления на самоорганизующуюся. Три вопроса, которые каждый задает в ходе встречи, позволяют команде самостоятельно контролировать план проекта. Но многие команды, пытающиеся внедрить Scrum, в итоге используют его как простое ежедневное совещание, где члены команды озвучивают состояние своей работы. Scrum-мастер фактически выступает на нем в роли руководителя проекта и распределяет задания между сотрудниками.

Точно так же некоторые команды добавляют пользовательские истории в документированные бизнес-требования, но рассматривают их как обычные спецификации, характерные для каскадной модели разработки.

Эти команды по-прежнему создают громоздкие спецификации до начала разработки продукта (BRUF, big requirements up front development) и просто добавляют к ним пользовательские истории. А вместо того, чтобы использовать разработку через тестирование, некоторые при попытке внедрить XP просто хотят убедиться, что их код полностью покрыт тестами, которые они разрабатывают после сборки кода, что означает, что тесты никаким образом не влияют на архитектуру, потому что она полностью завершена к моменту, когда пишутся тесты.

Во всех этих случаях люди искренне пытаются внедрять Agile. Но они не понимают, как эти практики вписываются в более широкую систему методологии. Поэтому вместо того, чтобы попытаться изменить способ

работы, они *сосредотачиваются на той части практики, которая кажется им знакомой*. Так стоит ли ожидать каких-то отличий? Команда, знакомая только с командно-административным способом управления, не имеет опыта самоорганизации, в ней не сформирована среда, стимулирующая глубже изучать суть Scrum.

Так давайте воздадим должное командам, когда это необходимо. Большинство команд внедряют Agile уже в ходе разработки программного обеспечения и добиваются некоторого успеха. (Совершенно недееспособные команды редко имеют менеджера, мировоззрение которого позволило бы им первым делом попробовать Agile.) Поэтому они ищут возможность вносить незначительные изменения, потому что не хотят ломать то, что реально работает.

Это ведет к одному из основных препятствий для внедрения Agile — убеждению каждого члена команды наподобие «я знаком с Agile, внедрил практики, которые мне понятны и близки, моя команда работает лучше, чем раньше, и мне этого достаточно». Такой результат называется «лучше-чем-ничего», и именно это заставляет многих людей отказаться от целостного внедрения Agile, сведя роль методологии к конкретным, несущественным улучшениям и, в итоге, к разочарованию после первоначальной эйфории.

Почему члены команды так часто настаивают на принятии только таких практик, которые кажутся им знакомыми, и отвергают любые другие, не известные им на данный момент?

Потому что каждая новая практика — это изменение, и всегда есть риск ошибиться. А в подобных случаях людей нередко увольняют.

Это то, о чем каждый agile-коуч должен постоянно помнить. Задача коучинга — помочь командам измениться, а это, в свою очередь, может стать причиной нестандартного (но *вполне разумного*) эмоционального отклика людей, которых принуждают меняться. Почему? Потому что *они держатся за свои рабочие места, чтобы жить и кормить семью*.

Вспомните: когда нас просят выполнить незнакомые задачи, мы ощущаем эмоциональный стресс, если не уверены, что сможем легко овладеть новыми навыками. Глубоко в нашем подсознании сидит охотник-собиратель, который рассуждает так: «Вчера я не сомневался, что выполню свою работу и принесу домой еду для всей семьи, но сегодня я в этом уже не уверен». Это важная причина, объясняющая беспокойство сотрудников, когда от них ждут изучения чего-то нового, и их, казалось бы, иррациональную реакцию на изменения (которая на самом деле вполне ожидаема).

Другая причина, почему люди отвергают перемены и тяготеют к тому, в чем хорошо разбираются, заключается в том, что они чувствуют недостаток

времени, не позволяющий им спокойно обдумать все приводимые доводы. Например, обычно команды внедряют Agile после опробования ее на пилотном проекте. Часто это начинается с чтения книги об Agile. После этого человек пытается внедрить какую-либо методологию в команде, одновременно занимаясь дедлайнами, ошибками, конфликтами, изменениями требований и прочими проблемами, которые присущи любому проекту. Все это не самая идеальная атмосфера для внедрения совершенно нового способа думать о работе. Люди будут стараться изо всех сил, но если столкнутся с чем-нибудь непонятным, скрывающимся за общей вывеской Agile, названиями методологий и практик, то фактически в их работе мало что изменится. Контрольные точки в плане проекта теперь будут называться «спринты», или кто-нибудь решит использовать доску задач, которая не повлияет на выполнение работы. В конце концов команда вернется к прежнему стилю работы, потому что она так привыкла и к тому же существуют дедлайны.

Когда команда использует названия известных agile-практик, но не меняет при этом способа работы, нетрудно понять, почему люди разочаровываются в новых идеях. Они приходят к выводу, что Agile — это просто другое название того, чем они давно занимаются. Думая, что внедрили Agile, они не изменили способа работы и продолжают получать прежние результаты.

Такая команда решит, что Agile не работает, даже не понимая, что в действительности и близко не подошла к настоящей методологии. Подобная реакция — следствие того, что людей просят изменить способ работы, но они не понимают зачем. И рядом нет никого, кто помог бы им пройти через эти изменения без ущерба для целостности новых практик и идей.

Именно поэтому командам необходимы agile-коучи. Их важнейшая задача — определить момент, когда люди сталкиваются с чем-то новым, и помочь им принять эти изменения. Коуч должен объяснить каждому члену команды суть изменения, чтобы тот понимал «зачем» (а не только «что» конкретно нужно). Это поможет команде изменить способ работы, а не просто взять название какой-нибудь гибкой методологии и присвоить его привычной практике.

Хороший коуч старается максимально понятно объяснить, как ежедневные scrum-митинги помогают самоорганизоваться, а разработка через тестирование — мыслить функционально и двигаться в сторону инкрементального проектирования. Как пользовательские истории дают возможность каждому члену команды понять позицию людей, применяющих программное обеспечение. Коуч помогает выйти за рамки простого принятия новых правил, и люди начинают видеть правильное направление движения и то, что оно способно им дать.

КОУЧИ ПРИСЛУШИВАЮТСЯ К СИГНАЛАМ О ТОМ, ЧТО У КОМАНДЫ ТРУДНОСТИ С ИЗМЕНЕНИЯМИ

Если вы обучаете много команд, то слышите одинаковые мнения от разных людей. Приведем несколько высказываний членов команды, которые могут свидетельствовать об их дискомфорте в связи с изменением, а также о том, что можно предпринять в такой ситуации. На все это полезно взглянуть с точки зрения коуча, даже если вы им не являетесь.

«Мы и так успешно разрабатываем программное обеспечение.

Почему ты призываешь меня делать что-то по-другому?»

Трудно спорить с успехом. Если вы работаете с командой, которая имеет свою историю разработки ПО для клиентов, то у нее есть право знать, почему нужно меняться. Недостаточно просто сказать, что это решение руководства, — такое заявление ослабит моральный дух. Как коуч вы должны позитивно относиться к работе, которую проделала команда. Но не нужно стесняться указывать на недостатки. Практики в каждой методологии Agile направлены на решение проблем команды. Если вы сможете помочь понять, почему эти проблемы возникают, и предложить команде решения, то шансы, что они будут одобрены, возрастают.

«Это слишком рискованно».

Это очень распространенная реакция на Agile, особенно среди тех, кто привык к командно-административному стилю управления проектом. Где все эти буферы, реестр рисков, лишняя бюрократия, которая тормозит проект и дает возможность в случае чего спасти свою шкуру? План проекта может быть непрозрачным, что порой удобно для команды: им не нужно делиться деталями, кроме как при завершении отдельных этапов; они могут вставлять буферы в промежутках, чтобы уменьшить и даже ликвидировать неопределенность. При использовании отчетов о статусе в качестве основного показателя прогресса разработки вам удастся контролировать информацию, которая поступает к пользователям и клиентам. Команды, работающие по таким принципам, чувствуют в Agile угрозу. В главе 3 упомянут agile-принцип, в котором говорится, что работающее программное обеспечение — основной показатель прогресса проекта. Если в agile-команде что-то идет не так, то все об этом знают. Задача agile-коуча — помочь менеджерам, пользователям и клиентам освоиться с этими неприкрашенными критериями достигнутого прогресса и предоставить команде безопасную среду, в которой она получит возможность ошибиться, чтобы впоследствии иметь возможность учиться на собственном опыте. Но если пока это нереально, то ваша

задача подсказать всем, а особенно руководителю, какую поставить перед собой цель и как изменить климат и настрой команды в будущем.

«Парное программирование (разработка через тестирование или другие практики) не подходит мне».

Программист, привыкший работать в одиночку, может интуитивно чувствовать, что парное программирование будет тормозить его работу. Особенно понятна такая позиция, когда человек работает в атмосфере, не принимающей ошибок. Вполне объяснимо, что он чувствует себя неуютно в присутствии другого человека, наблюдающего за тем, как он пишет код. И каждому, кто когда-либо обучал юного новичка водить автомобиль, знакомо неприятное чувство, которое испытывает старший член команды, передавая клавиатуру неопытному сотруднику при работе в паре. Существует много причин, по которым люди чувствуют себя некомфортно при использовании этих практик, особенно когда мировоззрение команды им не соответствует. Хороший agile-коуч сначала поможет команде выбрать практики, совместимые с ее мышлением. И в процессе их применения члены команды увидят преимущества и поймут, как и почему эти практики работают. Под руководством коуча все начинают активнее воспринимать гибкое мышление.

«Agile не подходит для нашего бизнеса».

Часто люди говорят так, потому что привыкли создавать подробные планы или описания проектов, прежде чем начать реальную работу. Они не могут представить процесс иным. Руководители проектов и те, кто привык к командно-административному стилю мышления, чувствуют себя комфортнее, имея оформленные в виде бумажных документов описания задач, требования и план проекта до начала любых работ. Точно так же архитекторы и разработчики успокаиваются, когда весь дизайн системы отражен на бумаге до того, как написана первая строка кода. И вдруг их просят доверить команде принятие решений — вплоть до последнего ответственного момента, а это означает потерю контроля. Поэтому они будут повторять: «У нас очень сложный бизнес. Люди в других компаниях, может быть, и в состоянии с ходу включиться в проект, но из-за нашей специфики необходимо планировать и проектировать все заранее». Действительно, любой бизнес сложный и каждый проект требует анализа и планирования. Agile-коуч поможет менеджерам, представителям бизнес-подразделений и руководителям разработки понять, что команды намного лучше справляются с этими сложностями, если позволить им разделить проект на более мелкие части. И у них должна быть возможность принимать решения в последний ответственный момент.

«Именно это мы и делаем, только называется по-другому».

Это одна из наиболее распространенных причин, по которым люди отвергают методологии Agile. Они будут искать способы продолжать работать в привычном стиле, используя при этом новые названия гибких практик, о которых читали или слышали. В таком случае Agile станет выглядеть обыденной — и даже неправильной, — если они решат присвоить это имя своей неудавшейся практике.

Попробуйте ввести в строку поиска «Agile — ерунда» (Agile sucks), и обнаружите высказывания людей, которые сделали именно это. Они называют Agile «очковтирательством», считая, что это просто практики каскадной модели, но под новыми названиями. Agile даже будут называть способом «запутать» принципы разработки программного обеспечения, диктуемые здравым смыслом.

Как коуч вы должны признать, что это делается не по злому умыслу. Это естественная реакция человека, который никогда не видел, что такое настоящая Agile (но думает, что видел). Задача коуча — помочь команде понять, что Agile действительно отличается от всего того, что они до сих пор делали, а вовсе не «ерунда».

Коучи понимают, как люди учатся

Хорошая модель освоения чего-либо (если освоение в принципе возможно) пришла из боевых искусств. Ученик боевых искусств проходит через три этапа мастерства, которые называются «сю», «ха» и «ри». «Сю» означает «следуй правилу». «Ха» — «ломай правила». «Ри» — «будь правилом».

Лисса Адкинс. Coaching Agile Teams: A Companion for ScrumMasters

Кент Бек, автор книги Extreme Programming Explained, описывал применение экстремального программирования (XP), используя аналогичные уровни. Когда его спросили об XP и пяти уровнях «модели зрелости возможностей», разработанных Институтом программной инженерии, он ответил, что XP имеет три уровня зрелости:

- 1. Делайте все, как написано.*
- 2. После того как работа выполнена, экспериментируйте с вариациями в правилах.*
- 3. В итоге не беспокойтесь о том, соответствуете вы XP или нет.*

Алистер Кокберн. «Быстрая разработка программного обеспечения».

Вернемся к главе 2, в которой мы узнали, как по-разному люди воспринимают Agile. Так же как слепой, встретивший слона, каждый человек по-своему ответит на вопрос «что такое Agile?».

Разработчик, знающий о таких XP-практиках программирования, как разработка через тестирование и инкрементальный дизайн, решит, что Agile — это о разработке ПО, его проектировании и архитектуре. Менеджер проекта, прочитав о scrum-спринтах, планировании, бэклоге и ретроспективах, подумает, что Agile — это об улучшении практик управления проектами.

Чтобы объяснить суть Scrum, XP, Lean и Канбана, потребовались сотни страниц. Мы говорили о мировоззрении, ценностях, принципах и практиках. И показали вам, как использовать их все вместе, чтобы помочь команде поставлять большую ценность для клиентов. Мы объяснили, что вы можете сделать сегодня, чтобы помочь команде. Мы предложили инструменты (например, неоднократное повторение вопроса «считаете ли вы это нормальным?»), чтобы помочь вашей команде изменить мировоззрение и для изучения Agile, понимания Scrum, XP, Lean и Канбана.

Это *не тот способ*, которым большинство людей внедряют Agile.

Как было бы хорошо, если бы все разработчики мира купили и прочли нашу книгу. Но, к сожалению, большинство людей изучают Agile не при помощи книг. Они учатся на практике. И проще всего выполнять задачу, если она разбита на небольшие части.

Другими словами, когда люди пытаются освоить Agile, они хотят получить список простых правил, который поможет им быстрее и качественнее создавать программное обеспечение.

Если вы новичок в agile-коучинге, то это может вызвать у вас разочарование. Представьте, что вы пытаетесь научить команду внедрять Scrum. Из главы 5 вам известно, что если кто-то не понимает принципа самоорганизации и коллективной ответственности, то у него не получится Scrum. Поэтому вы тратите время, объясняя такие scrum-ценности, как мужество, приверженность, уважение, сосредоточенность и открытость, и то, как работает самоорганизация. Но команда разочарована, потому что это очень абстрактные понятия. Люди хотят поговорить о доске задач и пользовательских историях. Вы стремитесь научить их Scrum, а они, по всей видимости, полны решимости удовлетвориться результатом «лучше-чем-ничего». И ничего с этим не поделаешь. Так что же происходит?

Есть старая поговорка об учениках: *встречайте их там, где они находятся, а не там, где бы вы хотели, чтобы они вас ждали*. Хороший agile-коуч понимает, как происходит процесс обучения, и предлагает людям информацию, поддержку и примеры, помогающие подняться на следующую ступень знаний. Коучи знают, что люди не меняются в одночасье.

В уже упоминавшейся книге «Быстрая разработка программного обеспечения» Алистер Кокберн рассказывает об основной идее обучения, которая называется «сюхари». Это очень ценный инструмент коуча, старающегося понять, где команда находится в текущий момент. «Сюхари» взята из боевых искусств, но это хороший способ подумать о том, как вообще происходит процесс обучения.

В «сюхари» различают три этапа обучения. Первый этап, «сю» (守, «повиновение» или «наблюдение»), описывает мышление человека, впервые сталкивающегося с новой идеей или методикой. Ученик хочет получить простые правила, которым может следовать. Именно поэтому люди чаще всего держатся за практики: команда может создать правило внедрения спринтов, заниматься парным программированием, использовать доску задач и т. д. И каждый может сказать, соблюдается или нет правило, о котором идет речь.

Правила особенно важны для взрослых, потому что, в отличие от детей, они не привыкли постоянно познавать новое. Предлагая тому, кто изучает новую систему — например, Agile, — простые правила, вы даете возможность начать их применять. Несложные практики, являющиеся частью таких методологий, как Agile, Scrum, XP и Канбан, хорошо подходят тем, кто находится на этапе «сю», потому что благодаря своей простоте дают возможность людям сосредоточиться на оттачивании мастерства. Установив для команды правило (например, «ежедневно в 10:30 проходит scrum-митинг, где каждый отвечает на три вопроса»), вы можете направить ее по верному пути для изучения принципа («мы используем принцип самоорганизации для постоянного анализа и коррекции плана»).

Цель коуча — не в навязывании слепой веры в чудо Agile. Фанатик Agile — это тот, кто решил, что каждая команда имеет только один способ применения этой методологии. И он назойливо навязывает его всем и каждому. Адепты Agile, как правило, сосредотачиваются только на тех правилах, которые поняли, и останавливаются на уровне «сю». Они считают, что все проблемы можно решить при помощи усвоенного и проповедуемого ими правила. Из agile-фанатиков не получается хороших коучей, потому что они не тратят время на попытки понять, где находятся люди, которых они обучают.

Чтобы привести нас к этапу «сюхари» — «ха», agile-коуч должен подняться выше понимания простых правил (破, «обособление» или «перелом»). Это этап, где люди уже овладели правилами и могут усваивать принципы, которые управляют ими. На протяжении всей книги мы повторяли, насколько эффективно для изменения мировоззрения команды внедрение практик

и как через них (и многочисленные командные обсуждения) приходит понимание ценностей и принципов методологии. Когда команда достигает уровня «ха», она начинает двигаться от результата «лучше-чем-ничего» к реальному приросту производительности. А он возможен, когда все разработчики действительно приняли мировоззрение Agile.

Мы еще ничего не сказали об этапе «ри» (離, «освобождение от правил»). Когда вы достигаете этого уровня обучения, вы в совершенстве владеете идеями, ценностями и принципами и меньше заботитесь о методологии, потому что продвинулись гораздо дальше. Если появляется проблема, которую можно решить при помощи определенной практики, то команда просто выполняет ее. Вас действительно не волнует, Scrum ли это, XP, Канбан или каскадная модель. Потому что ваш процесс не нуждается в имени. Но это не тот хаос, который характерен для команд, где никогда не пытались использовать Agile. Команда, в которой каждый достиг этапа «ри», — это отлаженная, хорошо функционирующая производственная единица, где каждый делает то, что ему положено.

Кокберн отмечает, насколько «удручающим дзеном» это может прозвучать. В книге «Быстрая разработка программного обеспечения» он рассказывает, как люди в стадии обучения «ри» говорят то, что трудно понять ученикам, находящимся на *этапе «сю»*:

«Делайте все, что работает».

«Когда вы действительно делаете это, вы не отдаете себе отчет, что вы это делаете».

«Используйте методологию до тех пор, пока она приносит хороший результат».

Для того, кто находится на уровне освобождения от правил, это полностью верно. А тех, кто еще только пытается разобраться, это сбивает с толку. Тем же, кто ищет порядка, это не принесет пользы.

Хороший коуч первым делом выясняет у каждого члена команды, каков текущий этап обучения. Если люди находятся на уровне «ри», то с ними легко работать, потому что каждый член команды знает, как учиться, и готов адаптироваться к новым практикам.

И наоборот, на уровне «сю» все хотят иметь однозначные правила. Вы используете стикеры или карточки на доске задач? Как коучу вам известно, что на самом деле это не имеет значения. Но тот, кто никогда не использовал доску задач, понятия не имеет, важно это или нет. Поэтому ваша цель — дать ему правило. Позже с вашей помощью он поймет, почему вы выбрали стикеры, и узнает, что карточки также подойдут. Просто они немного

разные — например, на карточке можно писать с обеих сторон, а на стикере только с одной. И вы будете использовать сходства и различия, чтобы члены команды разобрались, как стикеры и карточки реализуют одни и те же принципы.

ИСПОЛЬЗУЙТЕ «СЮХАРИ», ЧТОБЫ ПОМОЧЬ КОМАНДЕ ПОНЯТЬ ЦЕННОСТИ МЕТОДОЛОГИИ

Вновь обратимся к ситуации, когда разочарованный коуч старается «внедрить» Scrum. «Сюхари» поможет понять, почему команда разочаровывается, пытаясь осознать ценности, и почему хочет говорить только о наиболее значимых практиках: досках задач и пользовательских историях. Доску задач или пользовательскую историю нетрудно представить в качестве этапа «сю» данной концепции: простое правило, которое можно начать применять прямо сегодня. Scrum полон подобных правил («встречайтесь каждый день и отвечайте на три вопроса»). В этом причина, что команды успешно внедряют scrum-практики.

Такие ценности, как открытость и приверженность, — это уже идеи уровня «ха»: абстрактные понятия, управляющие тем, как вы используете правила в системе. Самоорганизация и коллективные обязательства возможны только тогда, когда каждый член команды действительно понял и усвоил эти ценности. «Сюхари» помогает понять, почему команда должна пройти через применение практик, прежде чем сможет отказаться от них и осознать, что на самом деле понимается под открытостью и приверженностью.

Именно поэтому внедрение практик Scrum — очень эффективный первый шаг в оказании помощи команде, пытающейся постичь ценности этой методологии. Хороший коуч терпелив и во время каждого спринта ждет подходящего момента, чтобы рассказать команде о ценностях. Часто такая возможность возникает тогда, когда два человека имеют разные точки зрения на аспекты «слона» Scrum.

Предположим, разработчик обнаруживает в середине спринта, что функциональность не будет закончена. На ближайшем scrum-митинге он просит владельца продукта перенести ее разработку на следующий спринт. Владелец продукта рассержена. Она боится гнева клиента или менеджера по этому поводу и требует, чтобы разработчик «сделал все возможное» для реализации функционала — даже за счет «срезания углов» и отсутствия законченного кода. На эту проблему есть ответ на уровне «сю»: правила Scrum диктуют, что команда должна демонстрировать только полностью

готовый программный продукт, а незаконченная версия передвигается в следующий спринт. Agile-коуч знает правила Scrum и может использовать их для разрешения конфликта.

Кроме того, agile-коуч также видит, что есть возможность для обучения уровню «ха». Признав существование раздробленного видения (об этом мы говорили в главе 2), которое может стать целостным, коуч поможет владельцу продукта взглянуть на проблему с точки зрения разработчика: некачественное выполнение работы увеличит технический долг, а это приведет к сложностям в поддержании всей кодовой базы. Технический долг, в свою очередь, станет причиной удлинения спринтов. Таким образом, хотя владелец продукта и может быстрее поставить функционал клиенту, игнорируя правила Scrum, в итоге команда окажется в худшем положении, потому что дальнейшая разработка будет протекать более медленно. Это важный урок, который поможет владельцу продукта узнать больше о такой scrum-ценности, как *сосредоточенность*, и о том, как команда, сфокусировавшая свое внимание на завершении работы, быстрее получит более качественный продукт.

Коуч также может помочь разработчику понять, насколько важна эта функциональность и как она обеспечивает ценность для клиентов. Кроме того, команда узнает, как в дальнейшем лучшее понимание ценности поможет ей установить более тесный контакт с владельцем продукта, чтобы найти способ разбить функционал на мелкие части и поставлять клиентам больше ценности максимально быстро. Этот важный урок даст возможность разработчику подробнее узнать о такой scrum-ценности, как *обязательство*, а также о том, как команда становится эффективнее, если понимает, что действительно важно для клиентов.

Это пример того, как коуч с глубоким пониманием методологии выполняет важную роль в команде. Он помогает ей понять и принять методы и правила методологии. И через них может помочь каждому члену команды усвоить ценности Agile. А используя эти ценности, он поможет каждому человеку и команде в целом развить в себе гибкое мировоззрение.

Коучи понимают, что делает методологию работающей

Конфликты и проблемы возникают в проектах ежедневно. Хороший scrum-мастер, например, посвящает большую часть своего времени их разрешению. Но далеко не все эти проблемы можно использовать в качестве обучающих примеров при изучении Scrum.

В частности, как коуч смог в конкретной конфликтной ситуации, о которой мы рассказали выше, увидеть возможность объяснить команде о таких scrum-ценностях, как *сосредоточенность* и *обязательство*? Относились ли эти ценности к данной конкретной проблеме?

Все дело в том, что коуч понимает не только как scrum-команды используют итерации, но и *зачем* они это делают. Он знает, что ограниченные по времени итерации могут быть полностью неэффективными, если команде разрешается поставлять *не выполненную до конца* работу. При таком развитии событий итерации превращаются в простые этапы проекта: объем каждой из них становится все более изменчивым, и команда уже не обязана поставлять работающее программное обеспечение в конце каждой итерации.

Это противоречит нескольким важным ценностям и принципам, благодаря которым Agile работает. Например, коуч знает, что работающее программное обеспечение — это основной показатель прогресса, а поставка неготового функционала в конце спринта дает клиентам ложное представление о достигнутом прогрессе. Коучу также известно: agile-команды ценят сотрудничество с пользователями, но это вовсе не означает, что клиент всегда прав. Иными словами, если команда видит реальную проблему с планом, она готова взаимодействовать с заказчиком, чтобы вместе придумать решение, обеспечивающее наибольшую ценность. Коуч понимает: если клиент и владелец продукта могут оказывать давление на разработчиков, чтобы включить не до конца собранное ПО в обзор спринта, то реальный прогресс команды будет преувеличен и сотрудничество с пользователями превратится в переговоры по контракту между командой и заказчиком.

Все это известно коучу, потому что он понимает ценности и принципы Agile и Scrum и знает, как они управляют практиками, а также то, из-за чего эти методологии могут стать формальными. Scrum-коуч хорошо знаком с понятиями «коллективная ответственность» и «самоорганизация». XP-коуч отлично понимает, что такое приветствие изменений и инкрементальная архитектура. Канбан-коуч знает, как улучшить командные процессы при помощи введения WIP-лимитов, и использует их для контроля рабочего потока — и когда кто-то пытается нарушить выставленные лимиты и включить в список как можно больше задач, то вся методология разваливается.

В своей книге *Coaching Agile Teams* Лисса Адкинс советует коучам, как правильно подходить к проблемам, связанным с командой. Об этом мы уже упоминали в главе 5, но стоит повторить.

Позвольте команде неудачу. Конечно, это не значит хладнокровно наблюдать, как команда приближается к краю пропасти. Но используйте десятки возможностей, которые возникают в каждом спринте, чтобы позволить команде провалить одно из заданий. Команды, попадавшие в трудные ситуации и выходившие из них благодаря в том числе сплоченности, становятся крепче и работают быстрее по сравнению с теми, кого оберегали от неудач. Такая команда может удивить вас. То, что должно было причинить ей вред, сделало ее сильнее. Наблюдайте и ждите.

Это превосходный совет для любого agile-коуча, особенно того, кто склонен к командно-административному стилю управления или считает необходимым контролировать каждый аспект обучения и прогресса команды. Хороший коуч понимает: есть моменты, когда команда может и должна потерпеть неудачу, потому что это самый действенный и эффективный путь к успеху.

Это тот случай, когда ретроспективы приобретают чрезвычайную ценность. Мы узнали о том, как scrum- и XP-команды используют ретроспективы, чтобы в ходе проекта провести ретроспективный анализ и найти пути для улучшения. Для agile-коуча это отличная возможность помочь команде извлечь урок из провала, именно для этого его и позволяют. Если команда ошибается, потому что недостаточно точно применяет практику, то коуч поможет понять, какие из текущих навыков требуют улучшения. Но иногда сбой происходит из-за устаревшего образа мыслей. Зная принципы работы методологий и практик, agile-коуч использует провал, чтобы помочь команде узнать больше о конкретной ценности или принципе. Благодаря этому она примет правильное решение и избежит неудачи. Благодаря этому команды прогрессируют.

Но хороший тренер отлично знает, когда провал недопустим. Как и любая сложная конструкция, каждая методология имеет свои «несущие балки», которые нельзя трогать, чтобы не нарушить целостность проекта. Коучу нужен уровень «ха» понимания методологии: он глубоко изучил ее как систему и знает, почему она работает.

В главе 8 мы прочли о руководителе, который непреднамеренно «подрезает крылья» всем канбан-усилиям, когда просит команду убрать WIP-лимит. Кто-то с уровнем понимания «сю» может позволить изменить WIP-лимит, полагая, что сможет получить пользу, оставив хотя бы часть Канбана. Хороший канбан-коуч осознает, что некоторые вещи (например, конкретные столбцы на канбан-доске) могут быть изменены, но WIP-лимит должен оставаться без изменений, потому что это стержень, на котором держится методология Канбан.

Одна из наиболее трудных задач коуча — понять, какую именно информацию необходимо объяснить команде, какую — руководителю, а какую — клиентам. Разъяснения правил методологии на уровне «сю» достаточно, чтобы работа была выполнена, но мало, если нужно помочь команде получить правильное мышление и достичь результата «лучше-чем-ничего».

Иногда команде хватает элементарного набора правил, а порой она чувствует, что просто подменяет случайный набор правил каскадной модели столь же случайным набором agile-правил. Уровень «ха» помогает увидеть, почему следование этим правилам позволит создавать лучшее программное обеспечение. Но это также может звучать как «удручающий дзен», как выразился Кокберн. Agile-коуч нужен, чтобы помочь команде двигаться вперед на уровень понимания «ха» в таком темпе, при котором можно справляться с работой без чрезмерно абстрактных объяснений, которые ей не помогают.

Принципы коучинга

Основная идея, которую мы продвигаем на протяжении всей книги, — это понимание того факта, что для внедрения Agile необходимо правильное мировоззрение и команда получает его через ценности и принципы.

Вот почему каждая методология Agile имеет свой набор ценностей. И команда приближается к ее реальному потенциалу только тогда, когда усваивает их.

Поэтому неудивительно, что коучинг имеет свой собственный набор ценностей. Джон Вуден, тренер мужской баскетбольной команды Калифорнийского университета в 1960–1970-х годах, считался одним из величайших тренеров в истории спорта. В своей книге «Современный баскетбол»^{*} он выделяет пять основных принципов коучинга: **трудолюбие, энтузиазм, состояние, основные принципы и развитие командного духа**. В agile-коучинге эти принципы значат так же много, как и в баскетболе.

Трудолюбие

Изменение стиля разработки программного обеспечения командой означает интенсивную работу над тем, чем она никогда раньше не занималась. Разработчики должны думать о планировании и тестировании, а не просто писать код. Владельцы продукта должны понимать, что делает команда, а не закидывать ее новыми функциональными

^{*} Вуден Дж. Современный баскетбол. М. : Физкультура и спорт, 1987.

требованиями. Scrum-мастера должны научиться передавать управление команде, оставаясь по-прежнему в проекте. Все эти навыки новые для команды и требуют работы над ними.

Энтузиазм

Когда вы полностью принадлежите работе и открывающиеся новые возможности вас вдохновляют, все остальное не имеет значения. У вас действительно множество причин быть в восторге: вы решаете проблемы, которые в прошлом вызывали головную боль. Когда каждый с энтузиазмом воспринимает Agile, вы получаете нечто большее, чем отличное ПО, — у вас формируется команда, любой участник которой более склонен к инновациям, счастлив и воодушевлен совместной работой.

Состояние

Agile работает тогда, когда каждый член команды хорош на своем месте. Известно, что каждый человек гордится своим мастерством. Если люди стараются разработать более качественное программное обеспечение, трудятся, чтобы добиться в этом успеха, то тогда они испытывают гордость за свое дело и максимально мотивированы. Вот почему члены agile-команды продолжают усердно работать над совершенствованием своих навыков, чтобы привнести в проект только самое лучшее.

Основные принципы

Вуден пишет: «Самая лучшая система не может преодолеть некачественного исполнения основных принципов. Коуч должен быть уверен, что никогда не позволит себе увлечься сложной системой». И это особенно актуально для agile-коучей. Agile работает, потому что ценности этой методологии просты и понятны и она содержит несложные практики. Это основы Agile, и хороший коуч стремится, чтобы команда сосредоточилась на них, и помогает ей поддерживать простоту в работе.

Развитие командного духа

Мы уже говорили о самоорганизации, целостности команды, энергичной работе и расширении прав и возможностей команды — это способы, при помощи которых agile-команды помогают развитию друг друга и совместно создают среду сотрудничества и инноваций. Одновременно с этим agile-коуч должен вести наблюдение за теми членами команды, которые в первую очередь ориентированы на собственную производительность, карьеру, результат или получают повышение

по службе, и помочь им изменить свое отношение к приоритетам. Вуден точно знает, что делать в таких случаях: «Коуч должен использовать каждую частицу психологии в управлении и все доступные методы, чтобы развить командный дух. При любой возможности следует поощрять коллективную работу и бескорыстие, и каждый [член команды] должен быть готов, а не просто желать пожертвовать личной славой ради благополучия команды. Эгоизм, зависть, самовлюбленность и критика друг друга могут разрушить командный дух и потенциал любой команды. Коуч должен знать это и быть постоянно начеку, чтобы не дать проблеме развиться».

Эти пять принципов — прочный фундамент мировоззрения agile-коуча. Чтобы стать хорошим коучем, необходимо усвоить и применять их точно так же, как вы используете ценности Agile-манифеста и другие гибкие методологии.



Ключевые моменты

***Agile-коуч** — это тот, кто помогает команде внедрить Agile, а каждому отдельному ее члену — освоить новое мышление.*

Эффективные agile-коучи помогают командам преодолеть сложности, связанные с внедрением новой практики, сосредоточив внимание на той ее части, которая им близка.

Коучи распознают предупреждающие сигналы о том, что команда испытывает дискомфорт в связи с изменениями.

Изучая новые системы или осваивая новые принципы мышления, люди проходят через три этапа, которые называются «слюхари».

Тот, кто впервые сталкивается с этой идеей, находится на этапе «сю», когда изучает правила и зачастую лучше реагирует на конкретные указания.

На этапе «ха» обучающиеся начинают принимать во внимание более крупные системы и адаптировать свое мировоззрение, чтобы соответствовать им.

На стадии «ри» люди владеют идеями системы уверенно и свободно — например, они знают, когда можно не следовать правилам.

Эффективный agile-коуч знает, почему система работает и зачем нужны правила.



Где вы можете узнать больше

Ниже перечислены ресурсы, которые помогут вам узнать больше об идеях, описанных в этой главе.

- Вы можете узнать больше об agile-коучинге в книге Лиссы Адкинс *Coaching Agile Teams* (Addison-Wesley 2010).
- Узнайте больше о концепции «сюхари» и о том, как люди учатся и адаптируются к новым системам, в книге Алистера Кокберна «Быстрая разработка программного обеспечения» (М. : Лори, 2013).
- Можно узнать больше об основах коучинга в книге Джона Вудена «Современный баскетбол» (М. : Физкультура и спорт, 1987).
- Вы можете узнать больше о том, почему команды сопротивляются переменам и как помочь им преодолеть это сопротивление, в книге Эндрю Стеллмана и Дженнифер Грин *Applied Software Project Management* (O'Reilly, 2005).

Об авторах

Эндрю Стеллман — разработчик, архитектор, лектор, agile-коуч, руководитель проектов, эксперт в разработке лучшего программного обеспечения. Имеет более чем двадцатилетний опыт профессионального создания программного обеспечения, в его активе — проектирование крупномасштабных серверных систем в режиме реального времени, руководство большими международными командами разработчиков, работа вице-президентом крупного инвестиционного банка, а также консультирование компаний, школ и корпораций, включая Microsoft, National Bureau of Economic Research, Bank of America, Notre Dame и MIT. За это время ему посчастливилось поработать с рядом замечательных программистов и узнать от них много нового.

Дженнифер Грин — agile-коуч, менеджер по развитию, бизнес-аналитик, менеджер проекта, тестировщик, лектор, имеет большой авторитет в области практик и принципов программной инженерии. Уже более двадцати лет занимается разработкой программного обеспечения в различных областях, включая СМИ, финансы и ИТ-консалтинг. Работала с замечательными командами разработчиков и тестировщиков над решением сложных технических вопросов и целенаправленно занималась поиском и преодолением возникающих на этом пути традиционных проблем с организацией процессов.

Несколько слов об обложке

Животное на обложке книги «Постигая Agile» — это гёльдиевая мармозетка, или гёльдиевая обезьянка (*Callimico goeldii*).

Это единственный вид рода *Callimico*, таких приматов также называют «калимико». Иногда их классифицируют отдельно от мармозеток из-за таких характеристик, как третий ряд коренных зубов, вынашивание одного детеныша и наличия когтей вместо ногтей.

Названа в честь швейцарского натуралиста Эмиля Августа Гёльди, который открыл этот вид в начале XX века.

Эти обезьянки обитают в верхнем бассейне Амазонки, Бразилии, Колумбии, Перу, Боливии и Эквадоре. Как правило, они живут в лесу с подлеском или в низкорослых зарослях рядом с лесом, расположенных вблизи ручьев, и в бамбуковых зарослях. Предпочитают питаться насекомыми, грибами (в сухой сезон) и фруктами (в сезон дождей). Живут небольшими стаями по шесть особей и поддерживают связь друг с другом, издавая пронзительные крики. Перемещаются сквозь заросли по стволам деревьев, но большую часть времени спят в гуще листвы.

Их тело всего около 20–23 сантиметров в длину, это немного больше белки, но хвост может достигать 30 сантиметров. Мордочка темного цвета, шерсть черная или темно-коричневая, часто с более светлыми бликами. Самки достигают половой зрелости в 8,5 месяца, а самцы в 16,5. Численность самок превосходит численность самцов, самки могут рожать два раза в год. Примерно через три недели после рождения главными воспитателями детенышей становятся самцы.

Продолжительность жизни гёльдиевых обезьянок — около 20 лет, даже в неволе. Природоохранный статус — «уязвимые», однако в меньшей степени, чем мармозетки в целом. Деятельность человека и другие изменения среды обитания угрожают их существованию. Их трудно поймать и сложно наблюдать за ними в дикой природе.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Все они важны для сохранения окружающего нас мира. Чтобы узнать больше о том, как вы можете им помочь, зайдите на сайт www.animals.oreilly.com.

Максимально полезные книги

Заходите в гости:

<http://www.mann-ivanov-ferber.ru/>

Наш блог:

<http://blog.mann-ivanov-ferber.ru/>

Мы в Facebook:

<http://www.facebook.com/mifbooks>

Мы ВКонтакте:

<http://vk.com/mifbooks>

Предложите нам книгу:

<http://www.mann-ivanov-ferber.ru/about/predlojite-nam-knigu/>

Ищем правильных коллег:

<http://www.mann-ivanov-ferber.ru/about/job/>

Научно-популярное издание

Эндрю Стеллман

Дженнифер Грин

ПОСТИГАЯ AGILE

Ценности, принципы, методологии

Главный редактор *Артем Степанов*

Ответственный редактор *Татьяна Рапопорт*

Литературный редактор *Игорь Лейко*

Арт-директор *Алексей Богомолов*

Верстка обложки *Наталья Майкова*

Верстка *Екатерина Матусовская*

Корректоры *Надежда Болотина, Юлия Молокова*

Почему agile-методологии завоевывают все большую популярность? Потому что перешедшие на Agile команды показывают отличные результаты: они создают качественный продукт, успешнее работают вместе, удовлетворяют запросы своих пользователей и добиваются всего этого в спокойной рабочей обстановке.

Эта книга рассказывает о самых популярных agile-подходах — Scrum, XP (экстремальное программирование), Lean (бережливое программирование) и Канбан. Она познакомит вас с методами, работающими в повседневной деятельности, а также с базовыми ценностями и принципами, которые помогут вашей команде полностью изменить свой подход к работе над проектами.

Что произойдет после того, как вы прочитаете эту книгу?

- Вы начнете лучше разбираться в конкретных agile-подходах и сможете сразу внедрить их в работу над проектами.
- Вы узнаете, как сделать переход максимально эффективным.
- Вы сможете выбрать тот agile-подход, который соответствует вашему мировоззрению (или максимально близок к нему).
- Вы узнаете, чем руководствуются эффективные agile-команды, а также какие ценности и принципы их объединяют.
- Вы поймете, для чего работать с agile-коучем и как он может помочь вам и вашей команде внедрить гибкие методологии.

А главное, вы сможете превратить группу сотрудников, добавляющих в свою работу Agile, в настоящую команду, которая действительно улучшает способ создания продукта и добивается выдающихся результатов.

“Эта книга о новой идеологии разработки ПО. Принципы и методы, описанные здесь, помогут улучшить не только рабочие процессы в команде, но и отношения между людьми. А люди — это самое главное”.

— Кирилл Семенихин,
директор Университета Иннополис

При поддержке

INNOPOLIS
UNIVERSITY

ISBN 978-5-00100-614-5



9 785001 006145 >

Максимально полезные
книги для детей на сайте
mann-ivanov-ferber.ru

издательство
МАНН, ИВАНОВ И ФЕРБЕР



facebook.com/**mifbooks**



vk.com/**mifbooks**



instagram.com/**mifbooks**