



ЧИСТЫЙ AGILE

ОСНОВЫ ГИБКОСТИ



РОБЕРТ МАРТИН

Clean Agile

Back to Basics

Robert C. Martin



Pearson

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

РОБЕРТ МАРТИН

ЧИСТЫЙ AGILE

ОСНОВЫ
ГИБКОСТИ



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2024

ББК 32.973.2-018
УДК 004.3
М29

Мартин Роберт

- М29 Чистый Agile. Основы гибкости. — СПб.: Питер, 2024. — 352 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-1552-5

Прошло почти двадцать лет с тех пор, как появился Манифест Agile. Легендарный Роберт Мартин (Дядя Боб) понял, что пора стряхнуть пыль с принципов Agile и заново рассказать о гибком подходе не только новому поколению программистов, но и специалистам из других отраслей. Автор полюбившихся айтишникам книг «Чистый код», «Идеальный программист», «Чистая архитектура» стоял у истоков Agile. «Чистый Agile» устраниет недопонимание и путаницу, которые за годы существования Agile усложнили его применение по сравнению с изначальным замыслом.

По сути Agile — это всего лишь небольшая подборка методов и инструментов, помогающая небольшим командам программистов управлять небольшими проектами, но приводящая к большим результатам, потому что каждый крупный проект состоит из огромного количества кирпичиков. Пять десятков лет работы с проектами всех мыслимых видов и размеров позволяют Дяде Бобу показать, как на самом деле должен работать Agile.

Если вы хотите понять преимущества Agile, не ищите легких путей — нужно правильно применять Agile. «Чистый Agile» расскажет, как это делать разработчикам, тестировщикам, руководителям, менеджерам проектов и клиентам.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.3

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-0135781869 англ.
ISBN 978-5-4461-1552-5

© 2020 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2024
© Серия «Библиотека программиста», 2024

ОГЛАВЛЕНИЕ

https://t.me/it_boooks/2

Отзывы на книгу «Чистый Agile»	8
Предисловие	11
Введение	14
Благодарности	19
Об авторе	22
От издательства	24
Глава 1. Введение в Agile	25
История Agile	27
Сноуберд	38
Краткий обзор Agile	44
Жизненный цикл	67
Заключение	71
Глава 2. Почему же Agile?	72
Профессионализм	73
Разумные ожидания	80

Билль о правах	98
Заключение	105
Глава 3. Методы взаимодействия с клиентами	106
Планирование	107
Небольшие частые релизы	131
Приемочное тестирование	138
Одна команда	146
Заключение	150
Глава 4. Методы взаимодействия внутри команды	151
40-часовая рабочая неделя	156
Коллективное владение	161
Непрерывная интеграция	164
Стендап-митинг	168
Заключение	170
Глава 5. Технические методы	171
Разработка через тестирование	172
Простота проектирования	187
Парное программирование	189
Заключение	194
Глава 6. Внедрение Agile	195
Ценности Agile	196
Методологический бестиарий	199
Преобразование	200
Коучинг	207
Сертификация	209

Agile в крупных масштабах	210
Инструменты Agile	214
Коучинг – альтернативный взгляд	225
Заключение (снова Боб)	240
Глава 7. Мастерство высшего уровня	241
Похмелье от Agile	243
Ожидание и реальность	246
Все дальше друг от друга	247
Высшее мастерство разработки	249
Идеология против методологии	251
Есть ли в мастерстве разработки методы?	253
Сосредоточьтесь на ценностях, а не на методе	254
Обсуждение методов	255
Влияние мастерства на личность разработчика	257
Влияние мастерства на отрасль разработки	258
Влияние мастерства на компании	259
Высшее мастерство и Agile	261
Заключение	262
Глава 8. Заключение	263
Послесловие	265

ОТЗЫВЫ НА КНИГУ «ЧИСТЫЙ AGILE»

Проделав огромный путь, Agile, не без помощи того самого Дяди Боба, достиг вершин. Безусловно, эта история знает свои взлеты и падения. Восхитительная книга, которую вы сейчас держите в руках, сочетает в себе историческую хронику и мемуары. Вся мудрость Agile собрана здесь. Если вам интересно, что такое Agile и как он появился, — эта книга для вас.

Гради Буч

Разочарования Дяди Боба пронизывают каждую строку «Чистого Agile», но, поверьте, эти разочарования оправданы. Все, что создано в мире Agile, — просто капля в море по сравнению с тем, что еще можно из него сотворить. Эта книга — взгляд Дяди Боба на перспективу. Его вклад велик. Его стоит послушать.

Кент Бек

Полезно почитать о том, что Дядя Боб думает об Agile. Может быть, вы новичок, а может, уже матерый спец — в любом случае вы найдете в этой книге что-то для себя. Я подписываюсь почти

под каждым словом в ней. В некоторых главах я вижу свои собственные недостатки, что греха таить. Теперь я проверяю наш код дважды, а это 85,09 %.

Джон Керн

В этой книге, словно под увеличительным стеклом, можно в подробностях разглядеть методологию Agile. Дядя Боб, несомненно, один из умнейших людей, которых я знаю. Его рвение к программированию бесконечно. Он как никто другой может развеять мистический туман, сгустившийся над Agile.

Из предисловия Джерри Фицпатрика

*Всем программистам,
кто хоть раз боролся
с ветряными мельницами
или водопадами*

ПРЕДИСЛОВИЕ

Так что же такое методология гибкой разработки Agile? Как появилась на свет? Как эволюционировала?

В этой книге Дядя Боб дает глубокомысленные ответы на эти вопросы. А еще рассказывает о разных способах неправильного или искаженного понимания методологии Agile. Его взгляды очень важны, поскольку он авторитет в этой области. Ведь именно с его именем связано появление Agile.

Мы дружим с Бобом уже не один год. Впервые я встретил его в 1979-м, когда устроился на работу в отдел телекоммуникаций компании Teradyne. Я был инженером-электриком, моя работа заключалась в том, что я помогал устанавливать и обслуживать нашу продукцию. Позже я вырос до разработчика аппаратных средств.

Где-то спустя год моей работы в компании стали искать новые идеи для своей продукции. В 1981-м мы с Бобом выдвинули идею создания электронного телефонного администратора — по сути дела, он представлял собой службу голосовой почты с функцией переадресации вызовов. В компании нашу концепцию приняли с одобрением, и вскоре мы приступили к разработке E. R. — электронного администратора (The Electronic Receptionist). Прототип получился

шедевральным. Он работал под управлением операционной системы MP/M на базе процессора Intel 8086. Голосовые сообщения хранились на пятимегабайтовом винчестере Seagate ST-506. Я занимался разработкой порта для передачи голоса, а Боб писал приложение. Закончив разработку, я тоже написал часть кода приложения. Поэтому с тех пор я еще и разработчик.

То ли в 1985-м, то ли в 1986-м Teradyne неожиданно остановила разработку Е. Р. и без нашего ведома отозвала заявку на патент. Компания вскоре пожалела о принятом решении, а мы с Бобом жалеем и по сей день.

В конце концов мы ушли из Teradyne в поисках лучшей доли. Боб занялся консалтинговым бизнесом в районе Чикаго. Я же стал преподавателем и ударился в разработку программ. Мы умудрялись не терять друг друга из виду даже после моего переезда в другой штат.

К 2000 году я преподавал объектно-ориентированный анализ и проектирование в Learning Tree International. Курс включал в себя преподавание UML и унифицированный процесс разработки (USDP). Тогда я уже поднаторел в этих технологиях, однако ничего не понимал в Scrum, экстремальном программировании и прочих методиках.

А в феврале 2001 года появился Манифест Agile. Моя реакция была примерно как у всех: «Что еще за Agile?» Единственный манифест, о котором я знал, — это Манифест Коммунистической партии, составленный Карлом Марксом и Фридрихом Энгельсом. Этот Agile призывал взяться за оружие? Чертовы айтишные радикалы!

Манифест породил массы бунтарей. Он вдохновил программистов на создание лаконичного чистого кода посредством совместной адаптивной работы с отложенной обратной связью. Они выгля-

дели заманчивой альтернативой тяжеловесным процессам вроде каскадной модели и USDP.

Прошло уже 18 лет, как был обнародован Манифест. Для большинства современных разработчиков все это история древнего мира. Именно поэтому ваше понимание Agile может отличаться от представлений его основателей.

Эта книга ставит своей целью точно передать посыл идеи. В этой книге, словно под увеличительным стеклом, можно в подробностях разглядеть методологию Agile. Дядя Боб, несомненно, один из умнейших людей, которых я знаю. Его рвение к программированию бесконечно. Он как никто другой может развеять мистический туман, сгустившийся над Agile.

Джерри Фицпатрик
Software Renovation Corporation
Март 2019 года

ВВЕДЕНИЕ



Книга, которую вы держите в руках, — не научное исследование. Я не старался провести тщательный обзор литературы. То, что вы собираетесь прочесть, — мои личные воспоминания, наблюдения и мнения. Как-никак, я около двадцати лет связан с Agile.

Книга написана в неформально-разговорном стиле. Поэтому иногда я могу выбирать не очень корректные выражения. Сам я далеко не любитель крепкого словца, правда одно (несколько искаженное) бранное слово попало на эти страницы, но лишь потому, что иначе выразиться было никак!

Я бы не сказал, что книга гневлива. Когда вдруг меня прошибало, что нужно как-то обосновать написанное, я приводил источники, на которые ссылался. Я сверялся с мнениями других ребят из сообщества Agile, которые в деле столько же времени, сколько и я. Иногда даже нарочно просил некоторых дополнить книгу своим мнением или выразить свое несогласие, чему посвящены отдельные главы и разделы. Как бы то ни было, не стоит воспринимать эту книгу как научный труд. Наверное, правильнее бы воспринимать эту книгу как мемуары — старческое брюзжание, адресованное новичкам в Agile, которые только делают свои первые шаги.

Книга подойдет как программистам, так и простым любителям. Это не техническая литература.

Тут нет кода. В книге дается обзор первоначальной цели разработки Agile без углубления в какие-либо технические нюансы программирования, тестирования и управления.

Книга невелика. Просто потому, что много писать и не надо. Agile — небольшая идея, предназначенная для решения небольших задач, поставленных небольшими командами программистов, которые

выполняют небольшую работу. Agile не рассчитан на решение крупных задач больших команд программистов, которые занимаются крупными проектами. Есть даже что-то ироничное в том, что такое решение для таких мелких задач вообще имеет название. В конце концов, мелкие задачи, о которых тут говорится, решили еще в 1950-е и 60-е, почти сразу, как изобрели программное обеспечение в принципе. Даже в те далекие времена небольшие команды научились неплохо справляться с небольшим объемом работ. Однако все испортилось в 1970-х. Тогда маленькие команды разработчиков, выполняющие небольшие объемы работ, запутались в идеях, пропагандирующих выполнение крупных работ в крупных командах.

А разве не так? Боже, да не так! Крупная работа не выполняется большими командами. На самом деле крупная работа выполняется большим количеством маленьких команд, которые в свою очередь выполняют много небольших задач. В 1950-х и 60-х годах программисты понимали это на уровне инстинкта. Но в 1970-е годы про это просто-напросто забыли.

А почему забыли? Полагаю, что так произошло из-за неравномерности. Количество программистов в мире стало резко расти в 1970-х годах. До этого в мире было всего несколько тысяч программистов. Но потом их количество резко выросло до сотен тысяч. Сейчас их число достигает сотни миллионов.

Те самые программисты 1950-х и 60-х годов были взрослыми. Они стали программистами лет в 30, 40 или даже в 50. К 1970-м годам, когда программистов вдруг стало тьма-тьмущая, эти «старички» ушли на пенсию. Поэтому некому было обучать новое поколение программистов. Молодые программисты от 20 лет и старше начали работать как раз тогда, когда более опытные ребята уже начали уходить, поэтому их опыт не передавался эффективно.

Некоторые скажут, что с этого события в программировании началось смутное время. На протяжении 30 лет мы боролись за идею, которая гласила, что следует выполнять большую работу в больших командах, совершенно не осознавая, что секрет успеха — это выполнение большого количества мелких задач множеством небольших команд.

Затем, в середине 1990-х, наконец было переосмыслено то, что было упущено. Идея работы в небольших командах получила новую жизнь. Эта идея распространилась в сообществе разработчиков программного обеспечения и набирала обороты. В 2000-х мы поняли наконец, что нужно перезагрузить всю отрасль целиком.

Теперь нам нужно вспомнить то, что знали те, кто были до нас, на уровне инстинкта. Нам еще раз потребовалось понять, что крупные задачи выполняются небольшими командами, которые сотрудничают между собой в решении небольших задач. Мы подумали, что идея, у которой есть имя, больше привлечет внимания. И мы назвали ее *Agile*.

Я написал это введение в самом начале 2019-го. Прошло уже около двух десятилетий с перезагрузки 2000-х годов, и, кажется, пришло время для еще одной.

Почему? Да потому, что простое и маленькое послание Agile за эти годы потеряло свою суть. Его перемешали с концепциями Lean, Kanban, LeSS, SAFe, Modern, Skilled и многими другими. Перечисленные идеи тоже по-своему хороши, но это все равно не Agile.

Вот и пришло время, чтобы напомнить нам то, о чем знали наши предки в 1950-х и 60-х годах, и о том, что мы вновь усвоили в начале 2000-х. Пора вспомнить, что такое Agile на самом деле.

В этой книге вы не найдете ничего особенного, поражающего или изумляющего. Никаких революций, ломающих привычные шаблоны. То, что вы узнаете отсюда об Agile, — это то, о чем уже говорилось в 2000-х. Хотя нет. Тут говорится с другой точки зрения. Ведь за 20 лет, которые прошли за это время, мы усвоили что-то новое, и это включено в книгу. Но в целом посыл этой книги тот же, что и в 2001-м, и в 1950-м.

Посыл стар как мир. Но тем не менее это истина. Этот посыл предлагает нам небольшую идею для решения небольших задач, поставленных небольшими командами программистов, которые выполняют небольшую работу.

БЛАГОДАРНОСТИ

Первая моя благодарность — двум отважным программистам, которые не без удовольствия открыли (а может, и заново открыли) методы, изложенные в этой книге, — Уорду Каннингему и Кенту Беку.

Следующую благодарность выражаю Мартину Фаулеру. Без его твердой руки революция, произведенная Agile, могла так и не увидеть свет.

Кен Швабер заслуживает особого упоминания за его неукротимую энергию в продвижении и внедрении Agile.

Мэри Поппенник также заслуживает отдельного упоминания за самоотверженность и неиссякаемую энергию, которую она вкладывала в движение Agile, и ее заботу об Agile Alliance.

На мой взгляд, Рон Джейфрис благодаря своим выступлениям, статьям, блогам и теплоте своего характера может считать себя совестью в начале движения Agile.

Майк Бидл отлично отстаивал честь Agile, однако погиб ни за что от рук бездомного на улицах Чикаго.

Прочим авторам оригинального Манифеста Agile здесь также отводится отдельное место. Перечислю их: Ари ван Беннекум, Алистер

Кокберн, Джеймс Греннинг, Джим Хайсмит, Эндрю Хант, Джон Керн, Брайан Марик, Стив Меллор, Джефф Сазерленд и Дэйв Томас.

Джим Ньюкирк, мой друг и деловой партнер, в то время без устали работал в поддержку Agile вопреки личным трудностям, которые большинство из нас (и я в том числе, безусловно) не могут даже представить.

Также я хочу упомянуть людей, работавших в корпорации Object Mentor Inc. Они приняли на себя основные риски по внедрению и продвижению Agile. Многие из них присутствуют ниже на фото, которое было сделано в начале первых уроков курса XP Immersion.



Задний ряд: Рон Джеффрис, я (автор), Брайан Баттон, Лоуэлл Линдстрём, Кент Бек, Мика Мартин, Анжелика Мартин, Сьюзен Рocco, Джеймс Греннинг. Передний ряд: Дэвид Фарбер, Эрик Мид, Майк Хилл, Крис Бигей, Аллан Фрэнсис, Дженинифер Конке, Талиша Джеффферсон, Паскаль Рой. Не присутствуют: Тим Оттингер, Джефф Лэнгр, Боб Косс, Джим Ньюкирк, Майкл Фезерс, Дин Уэмплер и Дэвид Хелимски

Также я хочу упомянуть ребят, которые смогли собраться и сформировать альянс Agile Alliance. Некоторых из них можно увидеть ниже на фото — оно было сделано в начале заседания альянса в нынешнем августе.



Слева направо: Мэри Поппенник, Кен Швабер, автор, Майк Бидл, Джим Хайсмит (не присутствует: Рон Крокер)

Наконец, спасибо всем ребятам из Pearson, в особенности моему издателю Джули Файфер.

ОБ АВТОРЕ



Роберт С. Мартин (Дядя Боб) является практикующим программистом с 1970 года. Он является также соучредителем cleancoders.com, где представлены различные видеоуроки для разработчиков программного обеспечения, учредителем компании Uncle Bob Consulting LLC, оказывающей услуги по консультированию, подготовке и развитию навыков крупным корпорациям по всему миру. Был высококлассным специалистом в консалтинговой компании, занимающейся отраслью программного обеспечения, 8th Light Inc., расположенной в Чикаго.

Боб Мартин написал десятки статей для различных профессиональных журналов и систематически выступает на международных конференциях и выставках. Он также является создателем известных образовательных видео на cleancoders.com. Боб Мартин является автором и редактором многих книг, в том числе:

«Разработка объектно-ориентированных приложений на C++ по методу Буча» (*Designing Object-Oriented C++ Applications Using the Booch Method*)

«Языки паттернов в процессе создания программ 3» (*Patterns Languages of Program Design 3*)

«Еще больше сокровищ C++» (*More C++ Gems*)

«Экстремальное программирование на практике» (*Extreme Programming in Practice*)

«Быстрая разработка программ. Принципы, примеры, практика» (*Agile Software Development: Principles, Patterns, and Practices*)

«UML для программистов на Java» (*UML for Java Programmers*)

«Чистый код»¹

«Идеальный программист»²

«Чистая архитектура»³

Лидер в отрасли разработки программного обеспечения, г-н Мартин три года был главным редактором журнала C++ Report, а также первым председателем Agile Alliance.

¹ Мартин Р. Чистый код: создание, анализ и рефакторинг. — СПб.: Питер, 2018. — 464 с.: ил.

² Мартин Р. Идеальный программист. Как стать профессионалом разработки ПО. — СПб.: Питер, 2019. — 224 с.: ил.

³ Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2020. — 352 с.: ил. .

ОТ ИЗДАТЕЛЬСТВА

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1

ВВЕДЕНИЕ В AGILE



https://t.me/it_boooks/2

В феврале 2001-го группа из семнадцати экспертов в области разработки программного обеспечения собралась в городке Сноуберд, штат Юта. На собрании обсуждалось плачевное состояние отрасли. Тогда большинство программ создавалось посредством неэффективных тяжеловесных фреймворков с большим количеством ритуалов, наподобие каскадной модели и раздутых реализаций Rational Unified Process (RUP). Целью этих экспертов было создание манифеста, который провозглашал бы более эффективный и легковесный подход.

Нельзя сказать, что царilo единодушиe. У всех семнадцати был разный опыт и, соответственно, сильные расхождения во мнениях. Рассчитывать, что такое собрание быстро придет к общему мнению, было бы слишком наивно. И все же, несмотря на все трудности, согласие было достигнуто, и Манифест Agile увидел свет — так родилось одно из самых мощных и живучих движений в отрасли программного обеспечения.

В этой отрасли практически все движения происходят по одному и тому же пути. Поначалу есть меньшинство, восторженно выступающее в поддержку чего-либо, другое меньшинство заряженных критиков и подавляющее большинство — те, кому, мягко говоря, нет дела.

Многие из этих движений хиреют или никогда не проходят этот этап. Можно вспомнить аспектно-ориентированное программирование, логическое программирование или CRC-карты. Некоторые, однако, способны преодолеть такую пропасть, становясь чрезвычайно популярными и разносторонними. Некоторым удается оставить позади противоречия и занять господствующее положение в современной мысли. Объектно-ориентированное мышление можно считать примером последнего случая. И Agile тоже.

К сожалению, как только движение набирает последователей и начинает распространяться, то сталкивается с искажением и незаконным присваиванием. Продукция и методики, не имеющие ничего общего с оригинальным движением, присваивают чужое имя, чтобы нажиться на его славе и значимости. Так случилось и с Agile.

Эта книга, написанная спустя почти два десятка лет после событий в Сноуберде, ставит своей целью точно передать посыл идеи. Эта книга — попытка в высшей мере кратко и точно описать Agile без брехни и обиняков.

В ней представлены основные принципы Agile. В приукрашивании и расширении этих идей нет ничего плохого. Тем не менее производные от Agile — это уже не сам Agile. Это дополненный Agile со своими опциями. А то, о чем вы узнаете из этой книги, — это и есть тот самый чистый Agile, который всегда был и непременно будет.

ИСТОРИЯ AGILE

Когда зародился Agile? Вероятно, более 50 тысяч лет назад, когда люди впервые решили работать совместно ради общей цели. Идея постановки небольших промежуточных целей и измерения продвижения после их достижения у человека проявляется на подсознательном уровне, поэтому вряд ли это какая-то настоящая революция.

Когда впервые появился Agile в современном мире? Трудно сказать. В моем представлении, первый паровой двигатель, первая мельница, первый двигатель внутреннего сгорания, первый самолет были созданы с помощью методик, которые сейчас можно от-

нести к Agile. Я считаю так, потому что предпринимать небольшие измеряемые шаги очень естественно для человека, сложно представить, что это происходит иначе.

Так когда же Agile появился среди программистов? Хотел бы я быть мухой на стене у Алана Тьюринга, когда тот писал свою книгу в 1936 году¹. По моим догадкам, многие свои «программы» он написал, разбивая работу на мелкие этапы с изобилием отладки по исходному тексту вручную.

Я также хорошо представляю, что первый код, который он написал для автоматической вычислительной машины (Automatic Computing Engine, ACE) в 1946 году, был написан постепенно, маленькими этапами, с частым проведением ручной отладки по исходному тексту и даже с небольшим тестированием в действии.

В первые дни существования программного обеспечения можно найти много примеров решения задач, которые сейчас бы отнесли к Agile. Например, программисты, писавшие код для управления пилотируемым космическим кораблем «Меркурий», работали по этапам с интервалом в полдня с перерывами на модульное тестирование.

Об этом периоде есть много материалов. Крэг Ларман и Вик Базили написали историю, которая кратко изложена в «вики» Уорда

¹ Turing A. M. 1936. On computable numbers, with an application to the Entscheidungsproblem [доказательство]. Proceedings of the London Mathematical Society (Труды Лондонского математического общества), 2 (изд. 1937), 42(1):230–65. — Лучший способ ознакомиться с этой работой — прочитать произведение Чарльза Петцольда: Petzold C. The Annotated Turing: A Guided Tour through Alan Turing’s Historic Paper on Computability and the Turing Machine. Indianapolis, Indiana: Wiley, 2008.

Каннингема¹, а также в книге Лармана *Agile & Iterative Development: A Manager's Guide*².

Однако существовал не только Agile. Действительно, есть конкурирующая методология, которая пользовалась значительным успехом в производстве и промышленности в целом: научная организация труда.

Научная организация труда — это командно-административный подход с иерархической структурой. Менеджеры применяют научную организацию труда, чтобы определять наилучший набор процедур для достижения цели и отдавать распоряжения всем подчиненным для выполнения плана с точностью до буквы. Другими словами, сначала планируется крупная задача, затем проводится тщательная и подробная проработка плана.

Научная организация труда, вероятно, такая же древняя, как пирамиды в Египте, Стоунхендж или прочие подобные работы древних времен: с трудом верится, что можно работать по-другому. Еще раз замечу, идея повторять успешный опыт настолько глубоко подсознательно заложена в человеке, что ее сложно охарактеризовать как революционную.

Научная организация труда получила свое название из работ Фредерика Уинслоу Тейлора, написанных в 1880-х. Тейлор сформировал этот подход, придав ему коммерческую ценность и сделав состояние на консультировании по управлению. Метод получил широкий успех и привел к значительному повышению эффективности и производительности в последующие десятилетия.

¹ «Вики» Уорда, c2.com — сайт оригинальной «Википедии», первый день появления в сети Интернет. Пусть поддержка длится как можно дольше.

² Larman C. Agile & Iterative Development: A Manager's Guide. Boston, Massachusetts: Addison-Wesley, 2004.

И так случилось в 1970-е, что мир разработчиков программного обеспечения разделился на два лагеря — сторонников того или иного метода. Прото-Agile (Agile, еще не получивший название «Agile») предлагал предпринимать в зависимости от обстоятельств небольшие случайные шаги, которые можно измерить и выделить, для того чтобы идти в правильном направлении к наилучшему исходу. Научная организация труда призывала откладывать действие до тщательного анализа и проработки заранее подготовленного плана. Прото-Agile прекрасно подходил для проектов с низкой стоимостью внесения изменений, которые решали частично определенные задачи с произвольно поставленными целями.

Научная организация труда лучше всего подходила для проектов, которые решали четко определенные задачи с весьма определенными целями. И в них стоимость изменений уже возрастила. Какими были проекты в отрасли разработки программного обеспечения? Была ли в этих проектах стоимость изменений высока? Были ли задачи определены четко? Или стоимость изменений была низкой, а цели были поставлены произвольно?

Не вчитывайтесь слишком внимательно в предыдущий абзац. Насколько я знаю, никто не задавался такими вопросами. Иронично и то, что путь, выбранный в 1970-х годах, кажется, был обусловлен, скорее, волей случая.

В 1970 году Уинстон Ройс в своей работе¹ описал идеи для управления крупномасштабными проектами по разработке программного обеспечения. В этой работе присутствовала схема (рис. 1.1), которая наглядно изображала его план. Ройс не был автором этой

¹ Royce W. W. 1970. Managing the development of large software systems. Proceedings, IEEE WESCON, August: 1–9. URL: <http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf>.

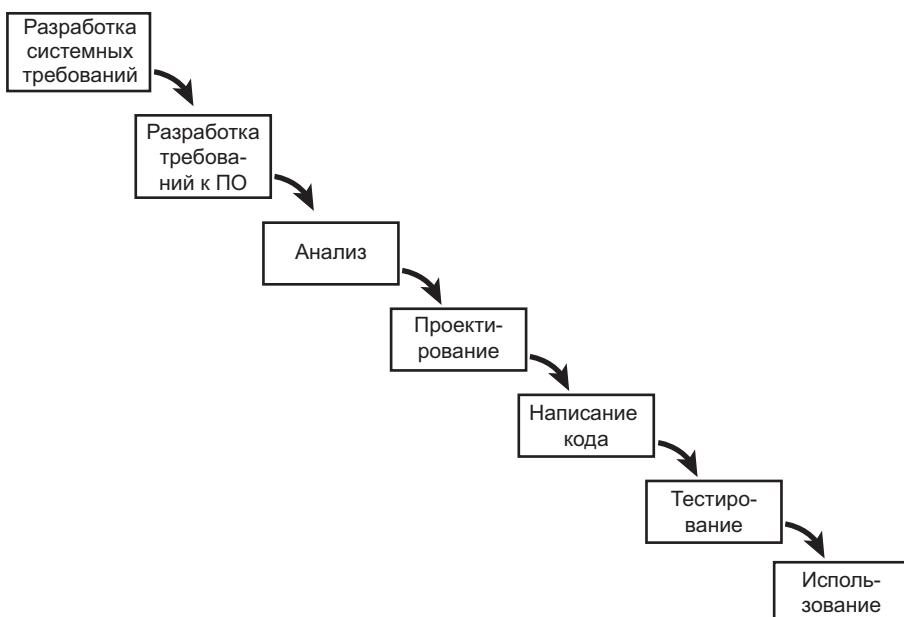


Рис. 1.1. Схема Уинстона Ройса, которая стала источником вдохновения для разработки каскадной модели

схемы и не продвигал ее в качестве плана. В действительности график представлял собой изображение соломенного человечка и помогал Ройсу ориентироваться в последующих страницах своего труда.

Схема была расположена на видном месте. А учитывая, что люди делают логические выводы о содержании статьи, посмотрев схему на первой или второй странице, это привело к резкому сдвигу в отрасли программного обеспечения.

Схема, первоначально составленная Ройсом, гораздо больше напоминала ручей, стекающий вниз со скалистого хребта, чем ныне известную каскадную модель.

Каскадная модель стала логическим продолжением научной организации труда. При использовании этой модели на первый план ставится тщательный анализ, составление подробного плана, а затем уже доведение этого плана до завершения. Хотя Ройс и не рекомендовал такой подход, но именно эту концепцию вынесли из его работы. А потом эта концепция главенствовала в отрасли более трех десятков лет¹.

Как раз тогда и начинается моя история. В 1970-м мне было 18 лет, я работал программистом в компании A. S. C. Tabulating, расположенной в Лейк Блафф, Иллинойс.

У компании был компьютер IBM 360/30 с памятью на магнитных сердечниках 16 килобайт, IBM 360/40 с памятью 64 килобайта и микрокомпьютер Varian 620/f с памятью 6 килобайт. Я писал программы для семейства 360 на COBOL, PL/1, Fortran и ассемблере. Для 620/f я писал только на ассемблере.

Важно помнить, каково в то время было программистам. Мы писали код в программных формулярах с помощью карандашей. У нас были операторы, работающие за перфоратором, которые наносили программы на карты. Мы передавали тщательно выверенные перфокарты операторам ЭВМ, которые проводили компиляцию и тестирование в третью смену, поскольку днем, когда работа кипела, компьютеры были постоянно заняты. От начала написания кода до первой компиляции зачастую проходило несколько дней, каждый цикл разработки вследствие этого занимал, как правило, сутки.

¹ Стоит отметить, что мое толкование этой временной шкалы подвергли сомнениям. См.: *Bossavit L. The Leprechauns of Software Engineering: How Folklore Turns into Fact and What to Do About It*, Ch. 7. Leanpub, 2012.

Для меня 620/f выглядел несколько иначе. Эту машину выделили нашей команде, поэтому мы могли работать на ней столько, сколько вздумается. Мы проводили два, три, иногда даже четыре цикла разработки и тестирования за сутки. Вместе со мной в команде были люди, которые, в отличие от большинства программистов того времени, умели печатать. Поэтому мы могли штамповывать свои собственные колоды перфокарт, а не зависеть от капризов операторов, работающих за перфоратором.

Какую методологию мы использовали на протяжении того времени? Это, конечно, была не каскадная модель. У нас не было концепций или подробных планов. Мы просто писали код каждый день, компилировали, тестировали и устранили ошибки. Это был бесконечный цикл без структуры. Также это был не Agile и даже неproto-Agile. В ходе работ мы не придерживались каких-либо правил организации. Тогда не было каких-либо пакетов программ для тестирования и измеряемых временных интервалов. Просто надо было писать код и фиксить баги. День за днем, месяц за месяцем.

Впервые я узнал о каскадной модели из профессиональных журналов около 1972 года. Мне она казалась даром свыше. Неужели мы могли бы проанализировать задачу, потом предложить ее решение, а затем реализовать замысел? Реально ли было на самом деле разработать график, основанный на трех перечисленных этапах?

Неужели, когда выполнен анализ, проект продвинется вперед на треть? Я почувствовал силу этой концепции. Я хотел в это верить. Если идея сработает, то мечта воплотится.

Судя по всему, я был не один, потому что многие другие программисты и центры программирования тоже вошли в кураж. И, как я уже писал, в нашем мышлении начала преобладать каскадная модель.

Она преобладала, но не работала. В течение последующих тридцати лет мои коллеги, я, братья и сестры по программированию по всему миру неустанно старались получить право на проведение анализа и проектирование. Но каждый раз, когда мы думали, что получали желаемое, оно ускользало из наших рук на этапе реализации. Месяцы тщательного планирования пошли прахом из-за необходимости сделать безумный рывок, в итоге мы сорвали сроки под свирепыми взглядами менеджеров и заказчиков.

Несмотря на практически нескончаемый поток неудач, мы все равно настаивали на состоятельности каскадной модели. В конце концов, почему возникали неудачи? Почему тщательный анализ задачи, внимательное проектирование решения и последующая реализация нескончаемо терпят зрелищный крах? Никто даже подумать не мог, что дело было в самой стратегии. Задача должна была лечь на наши плечи. Как бы то ни было, что-то мы делали не так.

Чтобы увидеть, насколько каскадная модель захватила наши умы, посмотрите на программные языки того времени. Когда Дейкстра в 1968 году представил структурное программирование, структурный анализ¹ и структурный дизайн² не сильно отставали. В 1988 году, когда объектно-ориентированное программирование (ООП) набрало популярность, объектно-ориентированный анализ³ и объектно-ориентированное проектирование⁴ также не сильно отстали.

¹ DeMarco T. Structured Analysis and System Specification. Upper Saddle River, New Jersey: Yourdon Press, 1979.

² Page-Jones M. The Practical Guide to Structured Systems Design. Englewood Cliffs, New Jersey: Yourdon Press, 1980.

³ Coad P., Yourdon E. Object-Oriented Analysis. Englewood Cliffs, New Jersey: Yourdon Press, 1990.

⁴ Booch G. Object Oriented Design with Applications. Redwood City, California: Benjamin-Cummings Publishing Co., 1991.

вали. Эта тройка идей, эти три этапа словно держали нас в плену. Мы просто не могли представить, что можно работать как-то по-другому.

А потом оказалось, что можно.

Зародыши преобразований, связанных с Agile, появились в конце 1980-х или в начале 90-х. В сообществе Smalltalk их признаки начали проявляться в 1980-х. В книге Буча по объектно-ориентированному проектированию, вышедшей в 1991-м, были намеки на них¹⁰. Прочие решения возникли в 1991 г. в книге Кокберна *Crystal Methods*. Сообщество Design Patterns начало обсуждать это в 1994-м под влиянием статьи, написанной Джеймсом Копlienом¹.

К 1995 году Бидл², Девос, Шэррон, Швобер и Сазерленд написали свои знаменитые труды о Scrum (Скрам)³. И затворы открылись. На бастионе каскадной модели образовалась брешь, и пути назад не было.

И здесь я снова возвращаюсь к нашей истории. То, что я расскажу дальше, — мои личные воспоминания, я не сверял их ни с кем из современников, участников событий. Поэтому следует предположить, что в моих воспоминаниях много опущений, недостоверно-

¹ Coplien J. O. A generative development-process pattern language. Pattern Languages of Program Design. Reading, Massachusetts: Addison-Wesley, 1995. P. 183.

² Майк Бидл был убит 23 марта 2018 года в Чикаго психически незддоровым человеком, до этого арестованным и отпущенными 99 раз, который должен был находиться в психбольнице. Майк Бидл был моим другом.

³ Beedle M., Devos M., Sharon Y., Schwaber K., Sutherland J. SCRUM: An extension pattern language for hyperproductive software development. Ссылка: http://jeffsutherland.org/scrum/scrum_plop.pdf.

стей или изложены они ужасно беспорядочно. Но не переживайте, я по крайней мере постарался рассказать все занимательно.

В первый раз мы встретились с Кентом Беком в 1994 году на той самой конференции PLoP¹, когда Коплин представил свою работу. Это была неформальная встреча, которая толком ничего не принесла. В следующий раз я встретил его в феврале 1999-го в Мюнхене на конференции, посвященной ООП. Но к тому времени я уже знал о нем намного больше.

В то время я занимался консультированием по C++ и объектно-ориентированному проектированию, летал с места на место, помогал разрабатывать и реализовывать приложения на C++ с помощью методик объектно-ориентированного проектирования.

Клиенты стали расспрашивать меня о процессе. Они слышали, что каскадная модель не применяется в объектно-ориентированном проектировании, и хотели услышать от меня совет. Я согласился с ними² и стал дальше думать об этом, мысли захватывали меня все сильнее.

Я даже подумывал написать свою собственную объектно-ориентированную методологию. К счастью, я скоро прекратил эти попытки, поскольку мне в руки попали труды Кента Бека по экстремальному программированию (XP).

Чем больше я читал об экстремальном программировании, тем больше я увлекался им. Идеи были революционны (по крайней

¹ Pattern Languages of programming — конференция, которую проводили в 1990-х неподалеку от университета штата Иллинойс.

² Это одно из тех странных совпадений, которые происходят время от времени. Нет ничего такого особенного в объектно-ориентированном программировании, что не дает применять в нем каскадную модель, тем не менее эта идея набирала в те дни популярность.

мере, я тогда так думал). Они казались разумными, особенно в контексте объектно-ориентированного мышления (опять же на тот момент я думал именно так). Мне не терпелось узнать больше.

К моему удивлению, на той самой конференции в Мюнхене, посвященной объектно-ориентированному программированию, я заметил, что через зал от меня читает лекцию сам Кент Бек. Как-то раз во время перерыва я натолкнулся на него и предложил встретиться как-нибудь за завтраком, чтобы обсудить экстремальное программирование. На том завтраке был заложен фундамент для плодотворного партнерства. Наши обсуждения побудили меня полететь к нему в Медфорд, штат Орегон, чтобы совместно разрабатывать курс по экстремальному программированию.

В ходе этого визита я впервые попробовал поучаствовать в разработке через тестирование, и это меня увлекло.

В то время под моим управлением была компания Object Mentor. В сотрудничестве с Кентом Беком мы хотели предложить пятидневный учебный курс по экстремальному программированию, который назывался XP Immersion. С конца 1999-го по 11 сентября 2001 года¹ он производил настоящий фурор! Мы обучили сотни человек.

Летом 2000 года Кент Бек созвал кворум из сообщества по экстремальному программированию и паттернам. Встреча проходила недалеко от его дома. Он назвал ее встречей ведущих специалистов в области экстремального программирования. Мы каталась на лодках и прогуливались по берегу реки Рог. И заодно решали, что делать дальше с экстремальным программированием.

¹ Этот день очень значим, поэтому его стоило упомянуть.

Была идея создать некоммерческую организацию. Я ее горячо продвигал, но многие не разделяли моего энтузиазма. Видимо, у них был неблагоприятный опыт со схожей организацией в поддержку паттернов проектирования. Я был расстроен тем, как прошло заседание. Но Мартин Фаулер поддержал меня и предложил встретиться позже в Чикаго, чтобы все обсудить и выговориться. Я согласился.

С Мартином мы встретились осенью 2000 года в кафе неподалеку от офиса Thought Works, где он работал. Я описал ему свою идею собрать сторонников всех конкурирующих легковесных методологий и составить манифест, провозглашающий единство. Мартин сделал несколько рекомендаций касательно пригласительного списка. Мы вместе начали составлять приглашение. В тот же день, немного позже, я отправил это письмо. Темой письма было проведение встречи по обсуждению легковесных методологий.

Одним из приглашенных был Алистер Кокберн. Он позвонил мне и сказал, что тоже подумывал провести подобную встречу, однако наш список ему понравился больше, чем его собственный. Он предложил объединить наши пригласительные списки и договориться о встрече, если мы согласимся провести ее на горнолыжном курорте Сноуберд неподалеку от Солт-Лейк-Сити.

Итак, встреча намечалась в Сноуберде.

СНОУБЕРД

Я был немало удивлен тем, что так много людей решило посетить мероприятие. Неужели кому-то действительно была интересна встреча, темой которой были легковесные методологии?

Однако мы все собрались в Сноуберде в гостиничном номере с прекрасным видом из окна.

Пришли 17 человек. С тех пор нас не раз критиковали за то, что все собравшиеся были белыми мужчинами среднего возраста. Критика была бы вполне справедлива, если бы не одно «но». Дело в том, что в списке приглашенных фигурировала одна женщина — Агнета Якобсон, но она не смогла приехать.

И, в конце концов, в то время во всем мире подавляющее большинство квалифицированных программистов было белыми мужчинами среднего возраста. А вот почему так сложилось — это отдельная история для совершенно другой книги.

У всех 17 из нас были довольно разные взгляды на пять различных легковесных методологий. Сторонников экстремального программирования было больше всех. Это были Кент Бек, Джеймс Греннинг, Уорд Каннингем, Рон Джейфрис и я.

Сторонников Scrum было немного меньше — Кен Швабер, Майк Бидл и Джейф Сазерленд.

Джон Керн высказывался в поддержку разработки, управляемой функциональностью, а Ариван Беннекум был сторонником метода разработки динамических систем. Наконец, Алистер Кокберн выступал за семейство методик, являвшихся его собственной разработкой — Crystal.

Остальные участники были относительно самостоятельны. Например, Энди Хант и Дэйв Томас пропагандировали прагматизм в программировании. Они даже написали работу на эту тему. Брайан Мариk был консультантом по тестированию. Джим Хайсмит был консультантом по управлению разработкой и сопровождению

программного обеспечения. Стив Меллор следил за честностью каждого, потому что был сторонником подходов, управляемых моделями, к которым многие из нас относились с недоверием. И, наконец, присутствовал Мартин Фаулер. У него были личные взаимоотношения с командой, занимавшейся экстремальным программированием, однако к каким-либо «фирменным» методикам он относился довольно скептически. Мнения всех присутствующих он воспринимал благожелательно.

Я почти ничего не помню из того, что произошло за два дня нашей встречи. Другие участники событий видят картину по-своему, не так, как я¹. Поэтому просто расскажу вам то, что сам помню. Расценивайте мои слова как воспоминания пожилого человека. Мне уже 65, а с того времени прошло почти два десятка лет. Возможно, я упустил несколько подробностей, но суть, думаю, передал правильно.

Каким-то образом мы решили, что я буду открывать встречу. Я поблагодарил всех за присутствие и высказал мнение, что наша цель состоит в составлении манифеста, в котором бы говорилось о разработке программного обеспечения в целом и описывались общие черты всех легковесных методологий, подмеченные нами. Закончив, я сел.

Я считаю, что это был мой единственный вклад в проведение встречи.

¹ Не так давно увидела свет история события, изложенная в литературном журнале The Atlantic за авторством Кэролайн Мимбс Найс: Caroline Mimbs Nyce. The winter getaway that turned the software world upside down // The Atlantic. 08.12.2017. URL: <https://www.theatlantic.com/technology/archive/2017/12/agile-manifesto-a-history/547715/>. Когда я это все писал, то еще не ознакомился с той статьей, поскольку мне не хотелось путать свои воспоминания, которые я изложил здесь.

Мы не занимались ничем необычным, когда записывали различные проблемы на карточках, а затем сортировали их на полу в группы по сходству. На самом деле я понятия не имею, что это нам дало. Просто помню, что мы это делали.

Затрудняюсь сказать, на какой день произошло чудо, — на первый или второй. Как мне кажется, это произошло к концу первого дня.

Возможно, именно группирование по сходству помогло нам выделить четыре ценности: личности и взаимодействие, рабочее программное обеспечение, взаимодействие с клиентами и реагирование на изменения. Кто-то написал это на магнитно-маркерной доске, находившейся в передней части комнаты. Затем ему в голову пришла блестящая мысль о том, что эти ценности приоритетны, но не заменяют остальные взаимодополняющие ценности методов, инструментов, документации, договоров и планов.

Это ключевая идея Манифеста Agile, и, кажется, никто отчетливо не помнит, кто первый обозначил ее на доске. Как мне помнится, это был Уорд Каннингем. Но сам Уорд приписывает это авторство Мартину Фаулеру.

Посмотрите на фотографию на сайте agilemanifesto.org. Уорд говорит, что сделал снимок, чтобы запечатлеть тот самый момент. На фото можно разглядеть Мартина Фаулера у доски и прочих участников встречи, которые собрались вокруг него¹. Это придает

¹ Слева направо, полукругом около Мартина Фаулера, на фотографии представлены: Дейв Томас, Энди Хант (или, возможно, Джон Керн), я (меня можно узнать по синим джинсам и мультитулу на ремне), Джим Хайсмит, кто-то, Рон Джеффрис и Джеймс Греннинг. Кто-то, уже не помню кто, сидел позади Рона. Возле его ботинка на полу, похоже, находится одна из карточек, которые мы использовали при группировке по сходству.

правдоподобность высказыванию Уорда о том, что идея на самом деле принадлежала Мартину.

С другой стороны, возможно, и хорошо, что мы никогда уже этого не узнаем точно.

Как только произошло чудо, все собравшиеся объединились под его знаменем. Мы проявляли таланты ораторского мастерства, как могли. Насколько мне помнится, именно Уорд написал преамбулу к Манифесту, которая гласила: «Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим». Некоторые из нас внесли крошечные изменения и предложения, однако было ясно, что мы достигли согласия. В номере ощущалась изоляция от всего мира. Никаких разногласий. Никаких споров. Никаких обсуждений или альтернатив. В этих четырех строках была вся суть.

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

Я говорил, что это все? Тогда так и казалось. Но, конечно, предстояло прояснить многое частностей. Первая из них — как назвать то, что нам удалось определить?

Название Agile не сулило легкого успеха. Было много разных претендентов. Мне понравилось что-то вроде «легковесный» (lightweight), но кроме меня — никому. Остальные считали, что

в таком названии подразумевалась несущественность. Им полюбилось слово «адаптивный» (adaptive). Кто-то вспомнил слово agile¹, а кто-то заметил, что в то время это слово было в ходу в армии. В результате, хотя никому особо не нравилось слово agile, его выбрали в качестве названия как наименьшее из зол.

Когда второй день подходил к концу, Уорд вызвался самостоятельно сделать сайт agilemanifesto.org и опубликовать манифест. Полагаю, что выставить манифест на всеобщее обозрение — его идея.

После событий в Сноуберде

Следующие две недели были не столь насыщены и романтичны, как те два дня в Сноуберде. В основном это время было занято трудомкой работой над составлением документа, провозглашавшего наши ценности, который Уорд в итоге выложил на сайт.

Мы все соглашались в необходимости написать такой документ, чтобы показать и объяснить четыре ценности. Все же те четыре ценности являются своего рода утверждениями, с которыми каждый может согласиться, не внося при этом в их понимание никаких изменений. Принципы дают понимание того, что действие этих ценностей выходит за пределы второстепенного значения так называемых прописных истин.

У меня мало отчетливых воспоминаний того времени, но помню, как мы пересыпали по электронной почте документ с принципами друг другу туда-сюда, неустанно пытаясь дополнить его. Было сложно, но мы все понимали, что это стоит наших усилий. Когда все было сделано, каждый из нас вернулся к своей обыденной жиз-

¹ С англ. «живой», «проворный», «гибкий». — Примеч. пер.

ни. Предполагаю, что многие из нас считали, что на этом история и закончится.

Никто и подумать не мог, что нас так поддержат. Никто не ожидал, насколько судьбоносными окажутся те два дня. Но чтобы не задирать нос от своей важности и причастности, я непрестанно напоминаю себе о том, что Алистер Кокберн тоже был близок к проведению подобной встречи. И поэтому задаюсь вопросом, сколько еще было таких же, как я. Поэтому успокаиваю себя мыслью, что просто настало время, и если бы не мы, 17 человек, собравшихся в горах Юты, то собралась бы другая группа, которая пришла бы примерно к тому же самому.

КРАТКИЙ ОБЗОР AGILE

Как вести управление проектом по разработке и сопровождению программного обеспечения? На протяжении многих лет существовало много подходов — и большинство из них, мягко говоря, далеки от идеала. Надежды и молитвы распространены среди менеджеров, верующих в то, что судьба их проекта зависит от воли божьей. А те, кто в это не верит, частенько полагаются на мотивационные методики: жесткие сроки с наказаниями плетками, цепями, раскаленным маслом, фотографии людей, покоряющих скалы, и чаек, парящих над морем.

Подобные подходы почти повсеместно приводят к характерным признакам отвратительного управления проектами — команды разработчиков постоянно задерживают проект, несмотря на то что много работают сверхурочно. Команды, которые пишут программы явно низкого качества, не соответствующие потребностям клиентов.

Правило креста

Причина, по которой эти методы терпят крах, заключается в том, что менеджеры не понимают элементарную сущность программных проектов. Сама сущность любого проекта накладывает на него ограничения. Есть такое понятие в управлении проектами, как «правило креста». Хорошо, быстро, дешево, готово. Выбирайте три любых пункта. Но четвертый будет не под силу. Проект может быть одновременно хорошим, дешевым, быстро выполняться. Но он никогда не будет завершен. Проект может быть завершен, и быть при этом быстрым и дешевым. Но вот хорошим не выйдет.

Реальность диктует свои правила, и умелый менеджер понимает, что у всех четырех параметров есть свои коэффициенты. В руках грамотного менеджера проект будет достаточно хорошим и дешевым, достаточно быстро выполняться и при этом будет завершен на требуемом этапе. Умелый менеджер распределяет эти коэффициенты по нужным параметрам, а не выдвигает требования к проекту по всем параметрам на 100 %. Такой способ управления проектами старается внедрить Agile.

Сейчас я хотел бы убедиться в вашем понимании того, что Agile – это набор методов, *помогающий* разработчикам и менеджерам проявлять необходимый прагматизм в управлении проектами. Однако такое управление не достигается автоматически. Нет никаких гарантий, что менеджер примет уместное решение. Действительно, вполне можно работать в рамках набора методов Agile, но несмотря на это управление проектом будет неграмотным и проект провалится.

Графики на стенах

Но как Agile способствует управлению проектами? Agile предоставляет данные. Когда применяется методология Agile, команда

разработчиков передает менеджерам именно те сведения, которые позволяют принимать верные решения.

Присмотримся к рис. 1.2. Представим, что такой график висит на стене кабинета, где ведется разработка проекта. Разве это не было бы потрясающее?

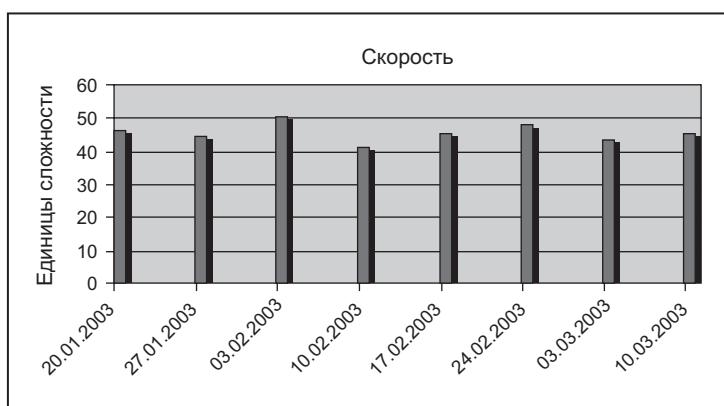


Рис. 1.2. Скорость работы команды

Этот график отражает производительность команды разработчиков за каждую неделю. Единицы измерения — единицы сложности (story point). Мы поговорим о них позже. Просто посмотрите на этот график. Каждый может сделать вывод, взглянув на график, насколько быстро продвигается работа команды. Менее чем за десять секунд можно понять, что средняя скорость работы составляет 45 единиц в неделю.

Кто угодно, даже сам менеджер, поймет, что на следующей неделе команда выполнит около 45 единиц работы. Получается, что через десяток недель команда выполнит уже примерно 450 единиц. Вот это мощь! Особенно это хорошо помогает, когда менеджеры

и команда хорошо осознают, сколько всего единиц насчитывает проект. На самом деле опытные команды, практикующие Agile, черпают эти сведения из еще одного графика.

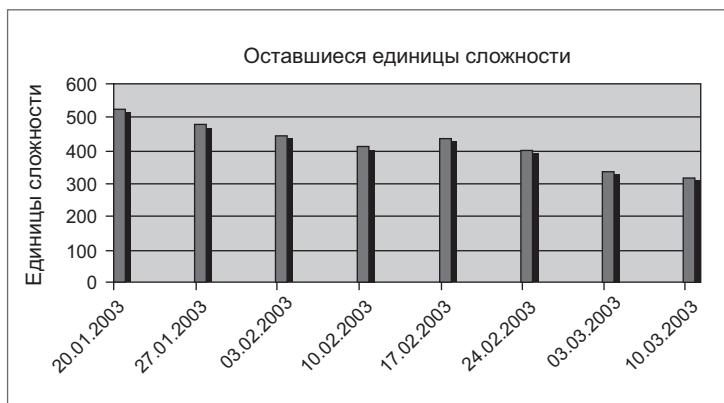


Рис. 1.3. Диаграмма сгорания задач

На рис. 1.3 изображена диаграмма сгорания задач. По ней можно судить, сколько единиц остается до следующей крупной вехи. Обратите внимание на то, как уменьшаются столбики с каждой неделей. Это связано с тем, что в процессе разработки постоянно появляются новые требования и проблемы.

Обратите внимание и на то, что у диаграммы сгорания задач есть угол наклона, который позволяет предположить, когда примерно будет достигнута нужная веха. Буквально любой может взглянуть на оба графика и вычислить, что следующий этап начнется в июне при скорости работ 45 единиц в неделю.

Присмотритесь внимательно к диаграмме, в ней есть странность. Столбец, обозначенный 17 февраля, почему-то выбивается из ряда. Это может быть связано с добавлением новой функции или

некоторыми другими существенными изменениями в требованиях. Или так получилось в результате переоценки разработчиками оставшегося объема задач. В любом случае мы хотим знать, как ход работ отражается на графике, чтобы организовать правильное управление проектом.

При использовании Agile чрезвычайно важно, чтобы эти два графика были на виду. Одна из движущих сил при использовании Agile во время разработки программного обеспечения — представление данных, необходимых менеджерам для распределения коэффициентов по параметрам согласно правилу креста и наиболее благополучного завершения проекта.

Многие не согласятся с этим. В конце концов, эти графики не упоминаются в Манифесте Agile, поэтому не все команды, его практикующие, их применяют. И если говорить начистоту, сами графики не так уж и важны. Важны как раз-таки данные.

Agile — это в первую очередь подход, который срабатывает только тогда, когда есть обратная связь. Каждая неделя, день, час и даже минута проходят в зависимости от результатов предыдущей недели, дня, часа или минуты. Соответствующие поправки вносятся уже после. Это относится как к управлению отдельными программистами, так и командами программистов целиком. Без необходимых данных не получится эффективного управления¹.

Поэтому даже если у вас на стене нет этих графиков, убедитесь в том, что есть данные, необходимые для управления. Убедитесь

¹ Этот принцип тесно связан с циклом НОРД — петлей Джона Бойда, кратко изложенным здесь: https://ru.wikipedia.org/wiki/Цикл_НОРД. Boyd J. R. A Discourse on Winning and Losing. Air Force Base Maxwell, Alabama: Aeronautics University Library, 1987. Document No. M-U 43947.

в том, что менеджеры знают, насколько быстро продвигается команда и сколько работы осталось сделать для завершения проекта. И предоставьте эти сведения в прозрачной, легкодоступной и очевидной форме — в виде двух графиков.

Почему же эти данные настолько важны? Разве можно эффективно вести управление проектом без таких данных? Мы пытались. Три десятка лет. И все получилось так, как получилось...

Первое, о чем нужно знать

Что в первую очередь нужно знать о проекте? Прежде чем узнать название проекта или требования к нему, прежде чем делать вообще какие-то движения, нужно получить еще некоторые сведения. Конечно же, это сроки. Уже после того, как выбраны сроки, их нужно зафиксировать. В обсуждении сроков нет смысла, поскольку их устанавливают в связи с объективными деловыми причинами. Если сроком стоит сентябрь, это не просто так. Возможно, в сентябре намечается какая-то выставка или собрание акционеров, а может, просто-напросто закончатся средства. Какой бы ни была причина, она имеет какую-то важную подоплеку. И причина не изменится просто оттого, что кому-то из разработчиков объем задач покажется непосильным.

В то же время требования могут изменяться в непрерывном потоке, который нельзя зафиксировать.

И на это тоже есть причина — клиенты зачастую не знают, чего именно они хотят. Они вроде и знают, какую проблему им нужно решить, но перевести такие знания в требования к проекту всегда затруднительно. Поэтому происходит постоянная переоценка и переосмысливание требований. Добавляются новые функции.

Какие-то старые исчезают. Пользовательский интерфейс изменяется быстро — за недели, если не за дни.

Так выглядит мир разработки программного обеспечения. В этом мире сроки фиксированы, а требования постоянно меняются. И каким-то образом в контексте всего этого разработчикам нужно благополучно завершить проект.

Собрание

Каскадная модель пророчила нам способ пойти в обход этой задачи. Чтобы объяснить, насколько это было соблазнительно и неэффективно одновременно, я приведу в пример одно собрание.

Было первое мая. Большой босс созвал подчиненных в конференц-зал.

Босс начал: «У нас новый проект. Нужно его закончить к первому ноября. Никаких требований у нас пока нет. Нам их огласят в ближайшие пару недель. Сколько времени понадобится на анализ проекта?»

Мы вопросительно стали коситься друг на друга. Все молчали, боясь сказать лишнего. Никто понятия не имел, что на это ответить. Кто-то промямлил: «Так у нас же нет требований, от чего отталкиваться?»

«Представьте, что они есть! — завопил босс. — Вы прекрасно знаете, как все работает. Вы ж специалисты! Мне не нужны точные сроки. Мне просто нужно как-то заполнить график. Имейте в виду, что если это займет более двух месяцев, о проекте можно уверенно забыть».

Кто-то вопросительно пробормотал: «Два месяца?» Начальник воспринял это как согласие на условия: «Отлично! Как раз то, что я думал. Теперь скажите мне, сколько займет проектирование?»

И снова все застыли в недоумении, комнату наполнила мертвая тишина. Считаем. И осознаём, что до первого ноября всего полгода. Вывод напрашивается сам собой. «Два месяца?» — спросите вы.

«Совершенно верно! — большой босс лучезарно заулыбался. — Как я и думал. И на реализацию у нас остается два месяца. Всем спасибо, все свободны!»

Многие читатели наверняка вспомнили, что что-то такое с ними уже было. У кого такого не было, что ж сказать, вы счастливчики!

Этап анализа

Итак, предположим, что мы ушли из конференц-зала и разбрелись по кабинетам. Что делать дальше? Начинается этап анализа — значит, нужно что-то анализировать. Но что именно мы называем *анализом*?

Если почитать книги на тему анализа в разработке программного обеспечения, можно обнаружить, что каждый автор дает собственное определение. Нет единого мнения, что такое анализ. Он может представлять собой создание структурной декомпозиции требований. А может — обнаружение и уточнение требований. Может представлять собой создание основополагающей модели данных или объекта и так далее... Лучшее определение анализа таково: это то, чем занимаются аналитики.

Конечно, есть очевидные вещи. Нам нужно оценить размер проекта, спрогнозировать показатели основных технико-экономических и человеческих ресурсов. Нужно убедиться, что график работ выполним. Это самое малое, чего будет ожидать от нас компания. Что бы ни называлось анализом, это как раз то, чем мы собирались заниматься ближайшие два месяца.

Это своего рода благоприятный этап проекта. Все спокойно про-сматривают страницы в интернете, проводят небольшие сделки, встречаются с клиентами и пользователями, рисуют красивые графики, попросту говоря, весело проводят время.

Затем первого июля происходит чудо. Анализ завершен.

А почему мы так считаем? Потому что уже первое июля. Если по графику этап анализа должен завершиться первого июля, значит, что первого июля этот этап завершен. Мы ведь не опоздали? Поэтому устроим небольшую вечеринку с воздушными шарами и пламенными речами, отпразднуем наш переход от этапа анализа к этапу проектирования.

Этап проектирования

А что теперь делать? Конечно же, будем проектировать. Но что представляет собой *проектирование*?

Об этапе проектирования программного обеспечения нам известно чуть больше. На этом этапе мы разбиваем проект на отдельные модули и проектируем интерфейсы между этими модулями. На этом этапе мы также предполагаем, сколько команд нам понадобится и как эти команды будут связаны между собой. В общем, нужно уточнить график работ, чтобы составить правдоподобный осуществимый план по реализации.

Безусловно, на этом этапе что-то неожиданно меняется. Добавляются новые функции. Старые функции исчезают или корректируются. И было бы неплохо оглянуться назад и провести анализ изменений заново, но время — деньги. Поэтому мы всеми возможными способами стараемся внести изменения в проектирование.

И тогда случается новое чудо. Первого сентября мы внезапно завершаем проектирование. А почему так? Да потому что. Первое сентября. По графику работ мы должны были уже закончить. Нечем медлить.

Итак, еще одна вечеринка. Воздушные шары и речи. И мы прорываемся к следующему этапу — реализации.

Было бы замечательно провернуть такую схему еще разок. Эх, если бы точно так же можно было бы завершить этап реализации! Но так уже не выйдет. Потому что по завершении реализации требуется завершить и весь проект. Анализ и проектирование не приносят плодов в *двоичном виде*. У них нет однозначных критериев завершенности.

Нет объективного способа узнать, проведены ли они в реальности. Поэтому и получилось завершить эти этапы вовремя.

Этап реализации

А вот у реализации как раз есть отчетливые критерии завершенности. Тут уже не получится аккуратно схалтурить, выдав мнимый результат за действительный¹.

¹ Однако разработчики сайта healthcare.gov определенно пытались.

На этапе реализации полностью отсутствует двусмысленность задач. Мы просто пишем код. И нам приходится писать код второпях, высунув язык, потому что четыре месяца просто выкинули на ветер.

Между тем требования к проекту продолжают меняться. Добавляются новые функции. Старые функции исчезают или корректируются. Нам бы вернуться назад, провести новый анализ инести изменения в проектирование, но... осталось лишь две недели. И ударными темпами мы вбиваем все эти изменения в код.

По мере того как мы смотрим на код и сравниваем его с результатом проектирования, мы осознаём, что, должно быть, были не в себе на этапе проектирования, потому что сам код имеет мало общего с тем, что было изначально изображено на замечательных графиках. Но времени на раздумья нет, потому что часики тикают, а сверхурочной работы становится все больше.

Примерно 15 октября кто-то говорит: «Эй, а какое сегодня число? Когда сдавать?» И тут мы понимаем, что осталось всего две недели и к первому ноября мы ни за что не закончим. И вдруг впервые наши заказчики узнают, что с проектом возникают какие-то нувязочки.

Представьте их негодование. «А на этапе анализа нельзя было об этом сказать? Разве не тогда вы должны были оценить размер проекта и внимательно рассчитать график работ? А на этапе проектирования почему не сказали? Разве не тогда нужно было разбить проект на модули, распределить работу по всей команде и рассчитать человеческий ресурс? Почему мы узнаем обо всем за две недели до дедлайна?»

И ведь они правы, разве нет?

Марафон на выживание

И начинается марафон на выживание. Клиенты злятся. Заинтересованные стороны дошли до белого каления. Давление нарастает. Работаем сверхурочно. Кто-то уходит с проекта. Просто ад!

И уже где-то в марте мы с горем пополам выдаем результат, который лишь наполовину удовлетворяет требованиям клиентов. Все расстроены. У всех опускаются руки. И мы клянемся самим себе, что в следующий раз такого не произойдет. В следующий раз мы все сделаем по уму. В следующий раз анализ и проектирование будут выполнены на совесть.

Я называю это *раздуванием вышедшего из-под контроля процесса*. Мы собираемся в следующий раз еще лучше работать по методу, который не работает.

Преувеличение?

Очевидно, что история утрирована. В ней собрано воедино все отрицательное, что вообще может быть во время работы над проектом по разработке программ. Большинство проектов, где применялась каскадная модель, не терпели такого краха. Действительно, по счастливой случайности некоторые проекты удавалось завершить даже относительно успешно. С другой стороны, на подобной встрече я бывал не раз, мне доводилось работать не над одним таким проектом, и такое случалось не только со мной. История гиперболизированная, но такое все равно бывает.

Если меня спросить, сколько проектов, разработанных по каскадной модели, провалились с таким же треском, как в описанной выше истории, я отвечу, что сравнительно мало. С другой стороны, это больше, чем ничего, что тоже плохо. Кроме того, большая часть

таких проектов испытывала подобные трудности в большей или меньшей степени.

Каскадная модель не самая ужасная из того, что существует. Не все проекты, выполняемые по ней, разлетались в прах. Но она как была, так и остается плачевным способом ведения проекта.

Способ получше

Проблема в том, что каскадная модель кажется очень понятной. Сначала мы проводим анализ задачи, потом проектируем решение и уже потом осуществляем реализацию.

Просто. Доступно. Очевидно. И неправильно.

Подход, предлагаемый Agile, в корне отличается от того, что было написано выше, но при этом так же понятен. По мере прочтения, полагаю, вы увидите, что в нем гораздо больше смысла, чем в трех словах, описывающих каскадную модель.

Проект по Agile начинается с анализа, однако анализ сопровождает весь цикл разработки. На рис. 1.4 изображена схема, объясняющая принцип ведения проекта в целом. Справа — дата сдачи проекта, 1 ноября. Помните, первое, что надо знать, — это срок сдачи. Срок требуется поделить на закономерные интервалы, называемые итерациями, или спринтами¹.

Длительность одной итерации, как правило, составляет одну или две недели. Я предпочитаю недельный интервал, потому что за две недели можно натворить слишком много. Другие предпочитают

¹ Понятие «спрингт» используется в Scrum. Мне оно не нравится, потому что намекает на то, что нужно нестись изо всех сил. Проект по разработке — это марафон. А кому нужен спрингт во время марафона?

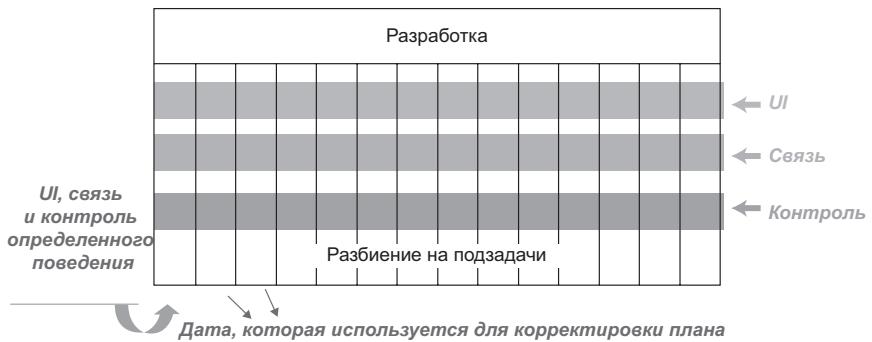


Рис. 1.4. Схема проекта

интервал в две недели, так как боятся не успеть выполнить задание за неделю.

Нулевая итерация

Во время самой первой итерации, которую иногда называют нулевой, создается краткий список функций – историй. Об этом будет рассказано подробнее далее. А сейчас давайте их рассмотрим как функции, которые нужно разрабатывать.

В процессе нулевой итерации также происходит развертывание среды разработки, оценка историй и построение первоначального плана. Такой план – это просто предварительное распределение историй по нескольким первым итерациям. Наконец, во время нулевой итерации разработчики, в том числе и разработчики архитектуры, творят чудо – создают первоначальный проект на основе предварительного списка историй.

Процесс написания историй, их оценки, планирования и проектирования никогда не прекращается. Поэтому через всю схему ведения проекта проходит горизонтальная линия, обозначенная

словом «исследование». В каждой итерации проекта, от его начала до конца, происходит анализ, проектирование и реализация. Согласно методологии Agile, постоянно нужно что-то анализировать и проектировать.

Некоторые думают, что Agile — это просто каскадная модель в миниатюре, которая повторяется многократно.

Это не так. Итерации не разделяются на три отрезка. В начале итерации выполняется не только сплошной анализ, а в конце — не только одна реализация. Скорее, анализ требований, архитектуры, проектирования и реализации непрерывно сопровождает всю итерацию.

Если вас это смущает, не переживайте. Об этом еще многое предстоит узнать в дальнейших главах. Просто имейте в виду, что итерации — не самая малая составляющая проекта при использовании Agile. Существует и много других уровней. Анализ, проектирование и реализация происходят на каждом из этих уровней. И все это не прерывается до самого конца.

Данные благодаря Agile

В начале первой итерации происходит оценка количества историй, которые нужно выполнить. Команда на протяжении всей итерации работает над выполнением собственно этих историй. О том, что происходит во время этой итерации, будет рассказано позже. Теперь скажите, что лишнего в работе команды, когда она пытается выполнить все истории, запланированные ранее?

Почти ничего. Так происходит из-за того, что разработка программного обеспечения плохо поддается точной оценке. Мы, программисты, просто не знаем, что сколько времени займет. Так происходит не потому, что мы тормозим или ленивы, а потому, что просто-на-

просто невозможно узнать, насколько сложно будет выполнить задание, до тех пор пока мы не принялись за него и не завершили. Но, как мы видим, не все так плохо.

В конце этой итерации будут выполнены некоторые фрагменты ранее запланированных задач. За счет этого мы можем предварительно измерить количество работы, выполняемой за одну итерацию. А это уже данные о том, как работа продвигается на самом деле. Если допустить, что все итерации будут схожи между собой, можно применить полученные данные для корректировки первоначального плана и рассчитать дату окончания проекта (рис. 1.5).

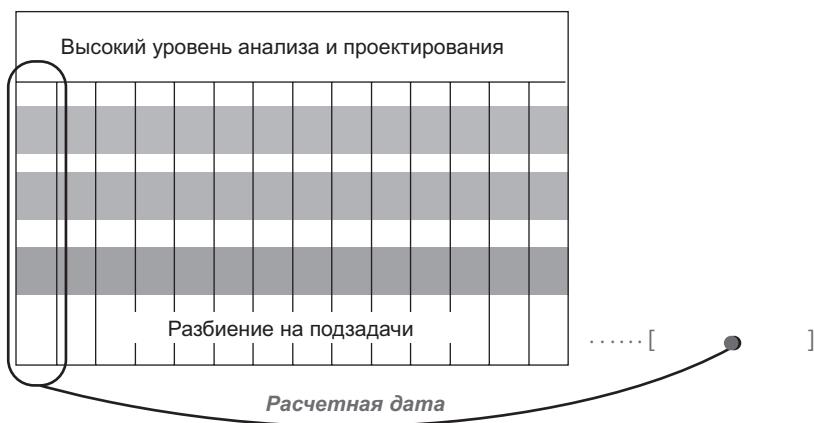


Рис. 1.5. Расчет новой даты завершения проекта

Этот расчет, вероятно, может огорчить. Почти всегда сроки будут в значительной мере выходить за рамки тех, которые были намечены изначально. С другой стороны, новые сроки основаны на *действительных данных*, поэтому не получится оставить их без внимания. Получившиеся сроки также не стоит принимать слишком близко к сердцу, так как они основаны лишь на одном замере. Погрешности при предварительном расчете данных довольно велики.

Чтобы уменьшить такие погрешности, должно пройти две, три и более итераций. По мере выполнения работ мы получаем больше данных о том, сколько историй выполняется за одну итерацию. Мы обнаружим, что их количество различается от итерации к итерации, но в среднем получается довольно стабильный расчет скорости продвижения. После четырех или пяти итераций у нас будет более ясное представление о сроках завершения проекта (рис. 1.6).

По мере прохождения итераций погрешности сводятся на нет, до тех пор пока не станет ясно, что первоначальный срок выполнения проекта в корне неверен.

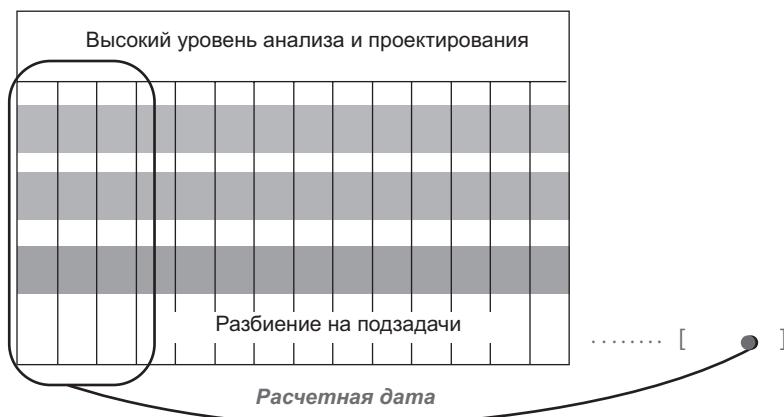


Рис. 1.6. Чем больше итераций, тем проще рассчитать сроки сдачи проекта

Надежда против управления

Зашита от самообмана — главная цель Agile. Мы применяем Agile, чтобы избавиться от ложных надежд, которые в итоге приведут проект к краху.

Надежды убивают проект. Надежды не позволяют команде сообщать менеджерам адекватные сведения о продвижении проекта. Когда менеджер спрашивает, как приводятся дела, именно надежда толкает программистов на ответ «все хорошо!». Надежда — никудышный способ управления проектом по разработке программного обеспечения. А Agile — это ушат с холодной водой, который непрерывно и своевременно возвращает к действительности.

Некоторые думают, что Agile способствует скорости выполнения проекта. Это не так. Agile никогда не ставил своей целью выполнить и сдать проект поскорее. Agile помогает вовремя понять то, где и насколько мы облажались. Это нужно для того, чтобы успешно справиться с поставленными задачами. Теперь посмотрим, в чем заключается задача руководителя. Для ведения проекта руководители собирают данные и потом уже на их основе принимают наилучшие решения. Благодаря Agile можно получить необходимые данные. Много данных.

Руководители используют эти данные для того, чтобы привести проект к наилучшему исходу. Наилучший исход из возможных — не всегда то же самое, что и желаемый. Наиболее благополучный исход может очень разочаровать, особенно заинтересованные стороны, изначально вложившиеся в проект. Но наилучший исход из возможных по определению является лучшим, что можно получить от проекта.

Как справиться с правилом креста?

Теперь вернемся к правилу креста в управлении проектами: хорошо, быстро, дешево, готово. Учитывая данные, полученные при выполнении проекта, руководство команды программистов может определить, насколько хорошо, быстро, дешево и когда будет готов проект.

Руководители, отталкиваясь от таких сведений, могут вносить изменения в объем и график работ, коллектив и задавать планку для качества результата.

Изменения графика

Начнем с графика работ. Можно задать вопрос: а что, если проект будет завершен не первого ноября, а первого марта? Обычно такие разговоры напрягают. Помните, что сроки устанавливают по объективным деловым причинам. Причины, конечно же, остались теми же. Перенос сроков зачастую означает, что компания потерпит какие-то убытки.

С другой стороны, менеджеры временами устанавливают сроки произвольно, исходя из удобства. Например, в ноябре будет проходить выставка, и компания просто хочет показать себя и представить свой проект. Вероятно, в марте будет проходить настолько же подходящая для него выставка. Помните, что все равно еще рано говорить о сроках окончания. Прошло только несколько итераций проекта. Лучше сказать заинтересованным сторонам, что проект будет готов в марте, чем дождаться, когда они оплатят стенд на выставке, проходящей в ноябре.

Много лет назад я вел группу разработчиков, которые работали над проектом для телефонной компании. В разгар проекта стало ясно, что сдачу проекта придется отложить на полгода. Мы сообщили об этом руководству компании как можно раньше, насколько это вообще было возможно.

Руководство компании впервые столкнулось с тем, что их предупредили о переносе сроков вовремя. Они просто зааплодировали нам стоя.

Невероятно. Но было именно так. Один раз.

Расширение команды

Как правило, никакие компании не хотят переноса сроков. Сроки установлены по объективным деловым причинам, и эти причины все еще имеют место. Можно увеличить количество сотрудников. На первый взгляд кажется, что если команду расширить вдвое, то и дело пойдет вдвое быстрее.

Но на самом деле прямой взаимосвязи нет. Закон Брукса¹ гласит: «Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше».

То, что происходит в реальности, можно увидеть на схеме, изображенной на рис. 1.7. Команда уже какое-то время работает над проектом с определенной отдачей. Приходят новички. Производительность проседает в течение нескольких недель, потому что новичкам необходимо учиться, и они донимают расспросами тех, кто уже работает давно. Хочется верить, что потом новички достаточно осваиваются и вносят свой вклад в проект.

Руководители делают ставку на то, что площадь под этой кривой будет строго положительна. Конечно, понадобится достаточно времени и усилий, для того чтобы компенсировать первонаучальные потери.

Другой фактор: безусловно, расширение коллектива стоит денег. Зачастую это непозволительная роскошь для бюджета проекта. Итак, давайте предположим, что команду нельзя расширять. Тогда следует ожидать изменения качества.

¹ Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Питер, 2020. https://ru.wikipedia.org/wiki/Мифический_человеко-месяц.

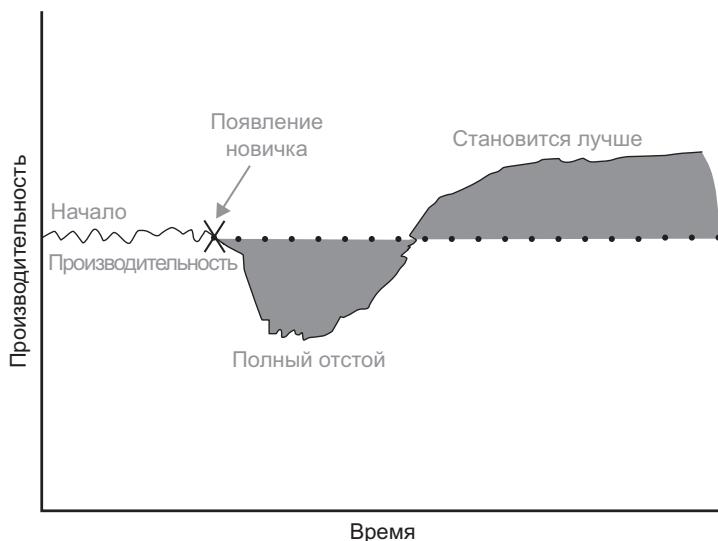


Рис. 1.7. Истинное следствие расширения команды

Снижение качества

Очевидно, что если делать фуфло, то и работа пойдет быстрее. Тогда зачем что-то тестировать, зачем пересматривать код, зачем нужен какой-то непонятный рефакторинг? Просто пиши код, и пошло все к чертовой матери! Если надо, пиши код хоть восемьдесят часов в неделю, главное — жги!

Думаю, вы понимаете, что я хочу сказать. Что это бесполезно. Клепая фуфло, вы не достигнете быстроты, вы будете тормозить. Поверьте моему опыту, это кристально ясно, когда программируешь уже двадцать или тридцать лет. Нельзя делать ерунду быстро. Ерунда — это всегда тормоз.

Есть только один способ быстро продвигаться вперед — нормально работать.

Поэтому планку качества нужно поднять до максимума и не снижать. Если нужно ускорить продвижение, тут без вариантов — извольте *повышать* качество.

Изменения объема работ

Это последнее из того, что можно поменять. Возможно, но это не точно, некоторые запланированные функции совсем не обязательно нужно предоставить именно первого ноября.

Расспросим всех заинтересованных лиц: «Господа, если вы хотите получить весь нужный вам функционал, он будет только в марте. Если вам нужно получить полностью весь функционал к ноябрю, некоторые функции придется исключить». На что нам наверняка ответят: «Нет, мы ничего исключать не будем! Нам нужно все! И нужно к первому ноября». Однако мы справедливо на то возразим: «Да, но вы не понимаете. Если вам нужно все, что вы хотите, придется подождать марта». Заинтересованные стороны, вероятнее всего, продолжат бодаться: «Нет, мы хотим получить все необходимое! И к первому ноября!»

Такой спор будет продолжаться какое-то время, потому что никто не хочет давать задний ход. У партнеров есть моральное право требовать в этом разговоре то, что им нужно, зато у программистов есть данные. И при грамотном раскладе побеждает тот, кто владеет данными.

Если проект организован правильно, то заказчики в конечном итоге задумчиво покивают головой, соглашаясь со сказанным, и начнут тщательный пересмотр плана. По очереди, методом исключения, они определят тот функционал, который им нужен к ноябрю, как собаке пятая нога. Неприятно, но что поделать, если мы хотим адекватно организовать работу? Итак, план под-

гоняют под действительность. Некоторые функции оставляют на потом.

Порядок реализации функционала

Непременно будет так, что заказчики прицепятся к какой-то функции, которую мы уже реализовали, и скажут нам: «Позорище! Зачем вы это сделали? Нам это не надо».

Мы не хотим снова это выслушивать! Так что теперь в начале каждой итерации придется спрашивать заинтересованные стороны, какие функции реализовать следующими. Да, разные функции зависят друг от друга, но мы же программисты и должны справляться с этим. Так или иначе, мы будем реализовывать функции в том порядке, в котором попросят заинтересованные стороны.

Завершение обзора

Вот мы и рассмотрели Agile. Но пока что только издалека. В обзоре опущены многие подробности, но в нем вся суть гибкой методологии. Agile – это процесс, в котором проект разделяют на отрезки, называемые итерациями. Объем работ, проделанный во время каждой итерации, измеряют и исходя из этого выверяют график. Функционал реализуется в том порядке, в котором это удобнее заинтересованным сторонам. Функции, необходимые в первую очередь, реализуют в первую очередь.

Планку качества нужно держать как можно выше. График в основном зависит от объема выполняемых работ.

Это и есть Agile.

ЖИЗНЕННЫЙ ЦИКЛ

Схема Рона Джейфриса, изображенная на рис. 1.8, объясняет принципы экстремального программирования. Эта схема также известна как «жизненный цикл».



Рис. 1.8. Жизненный цикл

Я посчитал, что для этой книги лучше всего подойдут методы экстремального программирования, потому что из всех методологий, составляющих Agile, экстремальное программирование является наиболее определенным, исчерпывающим и наименее замутненным.

Почти все прочие методологии, входящие в Agile, — это составляющие или разновидности экстремального программирования. Это не означает, что остальными методологиями, образующими Agile, нужно пренебрегать. Они могут быть очень ценны для различных проектов. Но чтобы понять, что такое Agile и с чем его едят, нет способа лучше, чем изучить экстремальное программирование.

Оно прообраз, лежащий в основе Agile, который одновременно является его лучшей составляющей.

Кент Бек — отец экстремального программирования, а Йорд Каннингем — его дедушка. Эти два человека, работая совместно в компании Tektronix в середине 1980-х, исследовали множество идей, которые в конечном итоге породили экстремальное программирование. Впоследствии Бек придал этим идеям точную форму. Таким образом, около 1996 года появилось экстремальное программирование. В 2000 году Кент Бек обнародовал свою окончательную работу: *Extreme Programming Explained: Embrace Change*¹.

Жизненный цикл подразделяется на три кольца. Внешнее кольцо отражает методы экстремального программирования при взаимодействии с клиентами. Это кольцо, по сути, эквивалентно тому, что предлагает методология Scrum². Эти методы обеспечивают структуру взаимодействия между командой разработчиков и клиентами, а также принципы, по которым заказчики и разработчики ведут управление проектом.

- Прием **«игра в планирование»** занимает центральное положение. Благодаря ему мы можем понять, как разбить проект по функциям, историям и задачам. Он содержит указания по оценке, постановке приоритетов, а также планированию соответствующих функций, историй и задач.

¹ Beck K. Extreme Programming Explained: Embrace Change. Boston, Massachusetts: Addison-Wesley, 2000. Também существует второе издание 2005 года, но мое любимое издание — первое, я считаю именно эту версию окончательной работой. Возможно, Кент не согласится со мной. (На русском языке: Бек К. Экстремальное программирование. — СПб.: Питер, 2002. — 224 с.: ил. — Примеч. ред.)

² Или, по крайней мере, так изначально задумывалось. Сейчас Scrum вобрал в себя множество методов экстремального программирования.

- **Небольшие и частые релизы** не позволяют команде «откусить» больше, чем возможно.
- **Приемочное тестирование** позволяет определять, какие функции реализованы, а также какие истории и задачи выполнены. Оно показывает команде, как определить однозначные показатели завершения.
- **«Одна команда»** позволяет понять, что в процессе разработки программного обеспечения принимают участие различные специалисты, в том числе программисты, тестировщики и руководители, а также то, что клиенты и сами принимают непосредственное участие, находясь на связи и будучи открытыми для вопросов. Все должны работать совместно для достижения общей цели.

Среднее кольцо жизненного цикла отражает методы экстремального программирования при взаимодействии внутри команды. Эти методы обеспечивают рамочную основу для взаимодействия между членами команды разработчиков, а также для самоуправления.

- **Постоянный темп** — это метод, который предохраняет команду разработчиков от перерасхода своих сил и банального выгорания до достижения финишной черты.
- **Коллективное владение** не позволяет членам команды тянуть одеяло на себя, благодаря чему каждый вносит посильный вклад в проект и несет ответственность за всю работу.
- **Непрерывная интеграция** позволяет команде сосредоточиться на частом слиянии рабочих копий в основную ветвь разработки и частом создании сборок, чтобы своевременно выявить ошибки и точнее отследить продвижение проекта.
- **Метафора** — это метод, который позволяет создавать и утверждать общую терминологию, благодаря которой команда

разработчиков и клиенты находят понимание при обсуждении вопросов, связанных с проектом.

Внутреннее кольцо жизненного цикла представляет собой технические методы, которые позволяют направлять и в чем-либо ограничивать программистов для достижения наиболее высокого уровня качества.

- **Парное программирование** — это метод, который помогает членам команды делиться сведениями и сотрудничать в парах, в том числе проводить совместный анализ, чем достигается высокий уровень обеспечения точности и внедрения новшеств.
- **Простота проектирования** — это метод, который позволяет команде не расходовать силы впустую.
- **Рефакторинг кода** способствует непрерывному совершенствованию и доработке всего, что получается в ходе работы.
- **Разработка через тестирование** — это страховочный канат, благодаря которому команда технических специалистов может быстро выполнять работу, не снижая планку качества.

Все методы очень тесно связаны с целями, которые провозглашает Манифест Agile, по меньшей мере в том, что перечислено ниже:

- **Люди и взаимодействие важнее процессов и инструментов.**
- «Одна команда», метафора, коллективное владение, парное программирование, 40-часовая рабочая неделя.
- **Работающий продукт важнее исчерпывающей документации.**
- Приемочное тестирование, разработка через тестирование, простота проектирования, рефакторинг кода, непрерывная интеграция.

- **Сотрудничество с заказчиком важнее согласования условий контракта.**
- Небольшие и частые релизы, игра в планирование, приемочное тестирование, метафора.
- **Готовность к изменениям важнее следования первоначальному плану.**
- Небольшие и частые релизы, игра в планирование, 40-часовая рабочая неделя, разработка через тестирование, рефакторинг кода, приемочное тестирование.

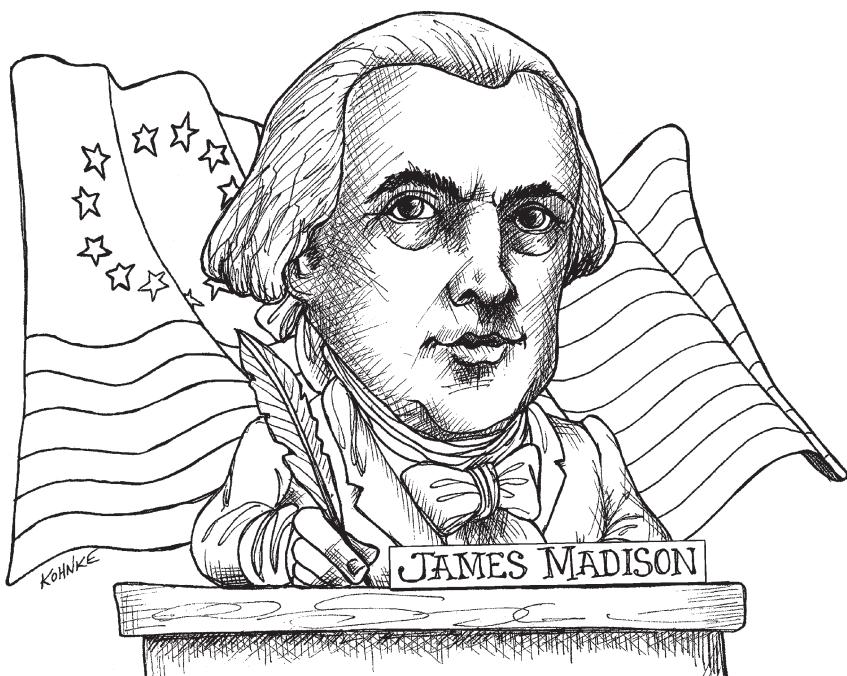
Однако по мере знакомства с этой книгой можно будет заметить, что связи между жизненным циклом и Манифестом Agile гораздо глубже и теснее, чем упрощенная модель, которая приведена выше.

ЗАКЛЮЧЕНИЕ

Таков Agile и история его появления. Agile – небольшая дисциплина, помогающая решению небольших задач, поставленных небольшими командами программистов для управления небольшими продуктами. Но несмотря на то что Agile невелик сам по себе, его значение и влияние достаточно велико, поскольку все большие проекты так или иначе состоят из множества маленьких.

С каждым днем программное обеспечение все больше переплетается с нашей повседневностью. Число людей, не мыслящих свою жизнь без него, постоянно растет. Не побоюсь сказать, что без программного обеспечения Земля перестанет вертеться. Если Земля остановится без программного обеспечения, то Agile – именно то, что позволяет наилучшим образом вести его разработку.

ПОЧЕМУ ЖЕ AGILE? 2



Прежде чем углубиться в тонкости разработки с помощью Agile, хотелось объяснить, что стоит на кону. Agile важен не только для отрасли разработки программного обеспечения, но и для промышленности, общества и, наконец, всей цивилизации.

Разработчики и руководители зачастую прибегают к Agile в силу каких-то временных обстоятельств. Они могут использовать его потому, что считают правильным подходом, или, возможно, просто потому, что купились на обещания уложиться в минимальные сроки при достижении самого высокого качества. Причины несознательны, неясны и могут противоречить друг другу. Многие отказались от Agile просто потому, что не смогли незамедлительно обрести ожидаемого, поскольку восприняли обещанное буквально.

Agile важен не по причинам, вызванным изменчивыми обстоятельствами. Agile важен в силу куда более глубоких философских и этических причин. Эти причины связаны с профессиональной деятельностью и обоснованными ожиданиями наших клиентов.

ПРОФЕССИОНАЛИЗМ

Agile привлекает меня тем, что ставит на первое место дисциплину, а не церемониальность. Чтобы правильно применять Agile, нужно работать в парах, в первую очередь писать тесты, проводить рефакторинг кода и соблюдать правила простоты проектирования. Придется работать короткими циклами, получая в результате каждого из них рабочий результат. А еще придется постоянно и непрерывно взаимодействовать с клиентами.

Взгляните на жизненный цикл и рассмотрите каждый из приведенных методов как обещание или обязательство — вам станет

понятно, откуда ноги растут. Для меня Agile — это очередной виток мастерства и приверженность идее продвигать профессиональный подход к работе по всей индустрии разработки ПО.

В этой отрасли крайне важно повышать професионализм. Мы слишком часто терпим крах. Слишком много занимаемся ерундой. Миримся с чрезмерным количеством ошибок. Заключаем ужасные сделки. Слишком часто мы ведем себя как сопляки, которым дали кредитку. Раньше все было проще, можно было позволить себе тормозить, так как на кону стояло меньше, чем сейчас. В 1970-х и 80-х, и даже в 1990-х цена ошибки в программе была не так высока. По крайней мере, убытки были ограничены и контролируемые.

Куда ни глянь, везде оно!

Времена меняются.

Прямо сейчас оглянитесь вокруг. Даже не надо вставать со своего места, просто оглянитесь на то, что в комнате вокруг вас. Сколько компьютеров в комнате?

Давайте теперь я. Сейчас я в своем летнем домике, который находится в лесу на севере штата Висконсин. Давайте посчитаем, сколько компьютеров и процессоров в них у меня здесь?

- **4.** Я набираю этот текст на ноутбуке Mac Book Pro с четырьмя ядрами. Производитель заявляет, что их восемь, но не стану считать виртуальные ядра. Также не стану брать в счет все малые вспомогательные процессоры, которые используются в MacBook.
- **+1.** Мыши Apple Magic Mouse 2. Уверен, что там больше одного процессора, но буду все считать за один.

- **+1.** Планшет iPad с монитором Duet в качестве дополнительного. Мне прекрасно известно, что в iPad много малых процессоров, но все равно посчитаю как один.
- **+1.** Ключ от машины (!).
- **+3.** Наушники Apple AirPods. По одному для каждого наушника и еще один в зарядном кейсе. Вероятно, процессоров больше, ну и ладно...
- **+1.** Мой iPhone. Да-да, на самом деле в iPhone, вероятно, около дюжины процессоров, но все равно, пускай будет один.
- **+1.** Взгляд упал на ультразвуковой датчик движения (в доме их куда больше, но я вижу и посчитаю один).
- **+1.** Термостат.
- **+1.** Панель управления системой безопасности.
- **+1.** Телевизор с плоским экраном.
- **+1.** DVD-проигрыватель.
- **+1.** Устройство для воспроизведения IP-телевидения Roku.
- **+1.** Apple AirPort Express.
- **+1.** Apple TV.
- **+5.** Пульты дистанционного управления.
- **+1.** Телефон (да, именно телефон).
- **+1.** Имитация камина (вы бы видели, какую красотищу он может выдавать!).
- **+2.** Старенький телескоп Meade LX 200 EMC с компьютером. Один процессор в приводе, а другой — в переносном блоке управления.
- **+1.** Флэшка у меня в кармане.
- **+1.** Стилус Apple pencil.

На свою душу я насчитал, по крайней мере, 30 компьютеров, и это только в этой комнате. Действительное количество можно увеличить примерно вдвое, поскольку в большинстве устройств по несколько процессоров. Но пока что давайте остановимся на тридцати.

А сколько насчитали вы? Уверен, что у большинства из вас получилось примерно столько же, сколько у меня. Действительно, бьюсь об заклад, что почти у каждого из 1,3 млрд человек, проживающих в западном мире, постоянно имеется рядом не один десяток компьютеров. Это что-то новое. В начале 1990-х это число в среднем было бы близко к нулю.

Что общего у всех компьютеров, которые мы видим рядом с собой? Их все надо программировать. Для них нужно программное обеспечение, которое как раз мы и пишем. И как вы думаете, каково качество этих программ?

Хорошо. Давайте рассмотрим вопрос с другого бока. Сколько раз на дню ваша бабушка пользуется программным обеспечением? У тех из вас, у кого она еще жива, дай бог ей здоровья, счет может идти на тысячи, потому что в современном обществе почти ничего нельзя сделать, не прикасаясь к программному обеспечению. У вас не получится:

- Говорить по телефону.
- Купить или продать что-либо.
- Пользоваться микроволновкой, холодильником или даже тостером.
- Постирать или высушить одежду.
- Помыть посуду.
- Слушать музыку.

- Водить машину.
- Подать страховую претензию.
- Регулировать температуру в помещении.
- Смотреть телевизор.

Но дела обстоят еще хуже. Сейчас в цивилизованном обществе буквально ничего значительного нельзя сделать без работы с программным обеспечением. Не получится рассмотреть, принять или привести в действие никакой закон. Правительство не сможет вынести на обсуждение ни один политический вопрос.

Самолеты не смогут летать. Машины не смогут ездить. Не получится запустить ракеты. Корабли не смогут ходить. На дороги станет невозможно нанести покрытие, не получится собрать урожай, остановится производство на сталелитейных заводах, автозаводы не смогут производить автомобили, кондитерские фабрики не произведут сладостей, прекратятся торги на биржах...

Без программного обеспечения наше общество сейчас как ноль без палочки. Каждое мгновение, когда мы не спим, мы сталкиваемся с программами. А многие даже во сне с ними сталкиваются — отслеживают фазы сна.

Куда без нас, программистов?

Наше общество сейчас целиком и полностью зависит от программного обеспечения. Оно стало играть роль крови, текущей в жилах нашего общества. Без него блага цивилизации, которыми мы сейчас наслаждаемся, были бы невозможны.

И кто пишет все программное обеспечение? Такие, как мы. Куда обществу без нас, программистов?

Другие думают, что их вклад в цивилизацию наиболее важен. Но они передают плоды своих трудов нам, а мы, в свою очередь, пишем алгоритмы, по которым работает различная техника, позволяющая отслеживать и задавать тон буквально всей деятельности в современном мире.

Получается, что без нас, программистов, никто не может и пальцем пошевелить.

Но в целом мы работаем плохо.

Как думаете, какая доля ПО, которое присутствует почти везде, протестирована должным образом? Сколько программистов могут сказать, что у них есть тестовый набор, который подтверждает с высокой долей вероятности, что программы, которые они написали, работают?

Работают ли сотни миллионов строк кода, из которого состоит ПО вашего автомобиля? Вы находили какие-нибудь ошибки в них? Я находил. А что скажете насчет кода, под управлением которого работают тормоза, газ и рулевой механизм? Там есть ошибки? Существует ли тестовый набор, который можно запустить в любое время и который подтвердит с высокой вероятностью, что когда ваша нога нажмет на педаль тормоза, машина действительно остановится?

Сколько людей погибло из-за того, что программное обеспечение в их автомобилях не смогло правильно отреагировать на давление ноги водителя на педаль тормоза? Точно сказать нельзя, но много. В 2013 году «Тойота» потерпела миллионы убытки, поскольку ПО содержало «возможное инвертирование разрядов, смертельные задачи, влекущие нарушение отказоустойчивости, повреждение содержимого оперативной памяти, одиночные неисправности элементов, влекущие за собой отказ всей системы, несовершенство

защиты от переполнения стека и буфера, одиночные неисправности отказоустойчивых систем и тысячи глобальных переменных», а сам код был запутан, как «спагетти»¹.

Программы, написанные нами, приводят к гибели людей. Что я, что другие наверняка становятся программистами не для того, чтобы кого-то убивать. Многие из нас постигли искусство программирования, потому что, еще будучи детьми, мы писали бесконечные циклы, которые выводили наши имена на экран, и нам это казалось невероятно крутым. Но сейчас от наших действий зависят жизни и судьбы. И с каждым днем появляется все больше кода, который ставит на кон жизни и судьбы все большего количества людей.

Катастрофа

Однажды наступит день, если еще не наступил, когда какой-нибудь несчастный программист по небрежности натворит глупостей, которые приведут к гибели десятка тысяч людей за раз. Задумайтесь об этом на минутку. Несложно представить себе несколько вероятных сценариев. И если так случится, то политики всего мира поднимутся в праведном гневе (как и полагается) и недвусмысленно укажут пальцем на нас.

Должно быть, вы подумаете, что пальцем покажут на наше начальство или руководство наших компаний, но мы отлично помним, что было, когда пальцем показали на исполнительного директора североамериканского подразделения «Фольксваген», и он предстал

¹ Safety Research & Strategies Inc., Toyota unintended acceleration and the big bowl of «spaghetti» code [blog post]. 7 ноября 2013 г. URL: <https://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code>.

перед Конгрессом. Политики поинтересовались, зачем «Фольксваген» устанавливал в свои автомобили программное обеспечение, которое целенаправленно обманывало оборудование для испытаний на количество и качество выбросов, которое применяется в Калифорнии. Он ответил: «Это не решение компании, насколько я знаю, менеджеры не имеют к этому отношения. Это сделала пара программистов, исходя из каких-то своих целей»¹.

Получается, в итоге крайними выставят нас. И это правильно. Потому что именно наши пальцы стучали по клавиатуре, набирая код, наша дисциплина хромала на обе ноги, а самой первопричиной послужила наша беспечность.

Именно это отдавалось эхом в моей голове, когда я возлагал большие надежды на Agile. Тогда, как и сейчас, я надеялся, что дисциплина, обретенная благодаря Agile, станет первым шагом навстречу тому, чтобы сделать профессию программиста по-настоящему почетной.

РАЗУМНЫЕ ОЖИДАНИЯ

Далее приведен вполне разумный список того, чего ожидают от нас руководство, пользователи и клиенты. Обратите внимание, что по мере прочтения списка, с одной стороны, вы согласны, что все эти ожидания вполне обоснованы. А с другой стороны, если в вас заговорит программист, будет страшновато. С точки зрения программиста сложно представить, как можно оправдать такие ожидания.

¹ O’Cane S. Volkswagen America’s CEO blames software engineers for emissions cheating scandal // The Verge. 8 октября 2015 г. URL: <https://www.theverge.com/2015/10/8/9481651/volkswagen-congressional-hearing-diesel-scandal-fault>.

Соответствие этим ожиданиям — это одна из основных целей Agile.

Принципы и методы Agile напрямую затрагивают большую часть ожиданий из этого списка. Подобное отношение к работе требует от своего коллектива любой хороший технический директор. Чтобы понять эту точку зрения, представьте, что я и есть ваш технический директор. И вот что я от вас ожидаю.

Фирма веников не вяжет!

Неприятно, что в нашей сфере вообще приходится упоминать о том, что код нужно писать качественно. А что поделать? Я уверен, дорогие читатели, что многим из вас довелось хотя бы раз не оправдать это ожидание. Мне доводилось.

Чтобы понять всю глубину проблемы, рассмотрим отключение сети управления движением воздушного транспорта над Лос-Анджелесом из-за сброса даты 32-битных часов. Или отключение всех двигателей на борту самолета Boeing 787 по той же причине. Или сотни жертв крушения Boeing737 Max из-за сбоя системы MCAS.

А как насчет моего собственного опыта с сайтом healthcare.gov на заре его существования? После первого входа, как и на многих современных сайтах, в целях защиты от взлома мне нужно было ответить на ряд вопросов. Одним из вопросов был «запоминающаяся дата». Я ввел: **21.07.73**. Сайт мне выдал, что я ввел неправильное значение.

Я программист. И знаю образ мышления программистов. В итоге я попробовал ввести дату в разных форматах: **21/07/73, 21-07-1973, 21 July 1973, 21071973** и тому подобное. Результат был одинаков.

Неправильное значение. Я расстроился. Что за приколы? Какой формат вам еще нужен?

Потом меня осенило. Программист, написавший код для сайта, не знал, какие вопросы будут задаваться. Он просто выдергивал вопросы из базы данных и сохранял ответы. Этот программист, вероятно, не предусмотрел поддержку нестандартных символов и чисел для ответов. Так что я набрал «годовщина свадьбы». На конец, получилось.

Думаю, что вполне справедливо сказать, что любая система, которая требует от пользователя мышления программиста, чтобы ввести какие-либо данные, — дрянь.

В этом разделе я мог бы рассказать массу историй об отстойном программном обеспечении. Но другие уже сделали это намного лучше. Если вам хочется ближе ознакомиться с масштабами этой проблемы, советую почитать книгу Гойко Аджича *Humans vs Computers*¹ и книгу Мэтта Паркера *Humble Pi*².

Вполне разумно, что боссы, клиенты и пользователи ждут от нас программ высокого качества с наименьшими недочетами. Никому не нужна ерунда, особенно если заплачены приличные деньги.

Обратите внимание, что упор Agile на тестирование, рефакторинг, простоту проектирования и обратную связь с заказчиком — это очевидное лекарство от низкого качества кода.

¹ Adzic G. Humans vs Computers. London: Neuri Consulting LLP, 2017. URL: <http://humansvsc.computers.com>.

² Parker M. Humble Pi: A Comedy of Maths Errors. London: Penguin Random House UK, 2019. URL: <https://mathsgear.co.uk/products/humble-pi-a-comedy-of-maths-errors>.

Всегда готовы

Самое последнее, чего от нас ожидают клиенты и руководство — что мы, программисты, будем как ненормальные переносить сроки поставки программного обеспечения. Но такие опоздания в мире разработки происходят сплошь и рядом. Причина таких задержек часто в том, что разработчики пытаются оснастить программу сразу всем функционалом, а не делать в первую очередь наиболее важные функции. Покуда есть функции, которые реализованы, протестированы или документированы лишь наполовину, программой пользоваться нельзя.

Другая причина переноса сроков — это доводка программ до стабильного состояния. Разработчики зачастую не принимают в расчет непрерывное тестирование, когда они наблюдают за тем, чтобы не было сбоев. Если в течение какого-то времени не обнаружено никаких сбоев, разработчики могут смело рекомендовать программное обеспечение к развертыванию.

Agile помогает решить эти проблемы с помощью простого правила, которое гласит, что программа должна быть технически готова к развертыванию в конце каждой итерации. «Технически готова» означает, что с точки зрения разработчиков продукт достаточно стабилен в техническом плане для развертывания. Код написан чисто, а тесты успешно пройдены.

Это значит, что работа, проделанная в течение итерации, включает в себя написание всего кода, прохождение всех тестов, написание всей документации и доводку до стабильного состояния всех историй, реализованных во время этой итерации.

Если в конце каждой итерации программа технически готова к развертыванию, то само развертывание производится по решению

клиента, а не из технических соображений. Клиент может посчитать, что для развертывания не хватает функционала, или может отложить развертывание по причинам, связанным с условиями рынка или обучением пользователей. В любом случае ПО соответствует необходимому качеству и технически готово к развертыванию.

Возможно ли такое, что программа технически готова к развертыванию в конце каждой недели или двух?

Конечно, да! Просто команде не нужно браться за непосильное количество историй, чтобы успеть выполнить все задачи и обеспечить развертываемость программы до окончания итерации. Также разработчикам стоит автоматизировать большую часть работ по тестированию.

Как и с точки зрения бизнеса, так и с точки зрения клиентов ожидание постоянной технической готовности продукта вполне естественно. Когда клиент видит, что функция работает, он думает, что работа над ней завершена. Он никак не ожидает того, что нужно подождать еще пару месяцев, чтобы обеспечить гарантию качества и стабильности. Ему не понять того, что функция заработала лишь потому, что программисты для демонстрации возможностей пошли в обход нерабочего кода.

Стабильная производительность

Вы, наверное, заметили, что зачастую команды программистов продвигаются очень быстро первые несколько месяцев, пока проект еще нов. Когда не существует основного кода, который замедляет работу, можно написать много рабочего кода за короткое время.

К сожалению, через некоторое время в коде появляется бардак. Если в коде не поддерживать порядок, то он будет тянуть команду

назад и замедлять ход проекта. Больше бардака — меньше скорость работы команды. Чем меньше скорость, тем сильнее поджимают сроки, а следовательно, больше спешки, приводящей к еще большему беспорядку. Принцип снежного кома приводит к тому, что команда практически впадает в ступор.

Руководители, озадаченные медленной работой, могут принять решение добавить специалистов в команду и увеличить производительность. Но, как мы уже знаем, расширение команды замедляет работу еще на несколько недель.

Есть надежда, что по прошествии нескольких недель новички выйдут на требуемый уровень производительности и проект пойдет быстрее. Только вот кому придется их обучать? Тем, кто и сотворил бардак. Новички, конечно, вберут от своих учителей не только самое лучшее.

Что еще хуже, они будут равняться на имеющийся код. Они посмотрят на уже написанный код и сделают вывод, как нужно работать в этой команде. А затем будут точно так же «бардакить». Именно поэтому производительность неуклонно падает, несмотря на подключение новых специалистов.

Менеджеры могут повторить эти действия еще несколько раз, ведь делать одно и то же и ждать иного результата — это показатель вменяемости руководства в некоторых организациях. В конечном итоге боссы узнают правду, но будет уже поздно. Уже ничто не сможет остановить неотвратимое погружение в бездну медлительности.

Отчаявшись, менеджеры спросят разработчиков, что можно сделать для повышения производительности. У разработчиков уже готов ответ. Уж они-то понимают, что делать. Они только и ждут того, чтобы их спросили.

И они отвечают: «Надо все переделывать».

Представьте, в какой ужас приходят менеджеры. Сколько денег и времени было вложено в проект! И что мы видим? Разработчики советуют выбросить все наработки и начать проект с чистого листа!

И как считаете, верит ли руководство разработчикам, когда те обещают, что «в этот раз будет все по-другому»? Конечно, нет! Они ж не дураки, чтобы поверить этому. Но что же им теперь делать? Производительность уже и так ниже плинтуса. Такими темпами работа не может продолжаться. Поэтому после многочисленных стенаний они соглашаются на рефакторинг.

У разработчиков засверкал в глазах огонь. «Слава богу! Наконец-то можно вернуться в те времена, когда трава зеленее и код чище», — скажут они. Но, конечно же, все не так просто, как кажется. Команда разбивается на два лагеря. Отбирается лучшая десятка — ударная группа, те самые орлы программирования, которые больше всех ответственны за беспорядочный код. Им выделяют отдельную комнату. Теперь они поведут всех остальных в мир рефакторинга. Остальные же их на дух не переносят, потому что из-за них приходится сопровождать тот отстой, который уже написан.

Откуда наша ударная группа берет требования? Есть ли на сегодняшний день какой-нибудь документ, где представлены требования? Да. Это код, который уже написан. Этот код — единственный документ, который точно дает понять, как должна выглядеть программа после рефакторинга.

Поэтому орлы вчитываются в написанный код, пытаясь сообразить, что с этим кодом не так и как нужно писать новый. Между тем

другие программисты вносят изменения в старый код, устраниют ошибки и добавляют новый функционал.

Гонка продолжается. Ударная группа пытается поймать движущуюся добычу. И как Зенон показал в своей апории об Ахиллесе и черепахе, поймать движущуюся добычу не так-то просто. Каждый раз, когда они доходят до этапа разработки, на котором находился написанный код, этот код уже изменился.

Чтобы доказать, что Ахиллес все равно догонит черепаху, нужно применить математический анализ. В разработке ПО это не всегда удается. Я работал в компании, где новое программное обеспечение не получалось развернуть в течение десяти лет. Клиентам обещали представить новый пакет программ еще восемь лет назад. Однако им постоянно не хватало функционала в новых программах, старые справлялись со своими задачами лучше. Поэтому клиенты не стали пользоваться новым пакетом программ.

Несколько лет спустя клиенты просто перестали обращать внимание на обещания о предоставлении новой системы.

С их точки зрения, подходящей системы никогда не было и не могло появиться.

В то же время компания, занимавшаяся написанием программ, оплачивала работу сразу двух команд разработчиков: тех самых ударных групп и сопровождающих. В конечном итоге руководство очень расстроилось и уведомило клиентов о том, что компания собирается развернуть новый пакет программ, несмотря ни на какие возражения. У клиентов началась истерика, но она не шла ни в какое сравнение с той, которую закатили разработчики, та самая ударная группа, а точнее сказать — то, что осталось от их команды. Всю команду, которая занималась разработкой с самого начала,

повысили до руководящих постов. Нынешние члены команды дружно встали и в один голос заявили: «Нельзя это поставлять, вы что? Это же фуфло. Надо все переделывать!»

Да-да, еще одна байка от Дяди Боба. Она основана на реальных событиях, но я чуть-чуть приврал для красного словца. И все же основной посыл вполне правдив. Масштабный рефакторинг — это чудовищно дорого и редко востребовано.

Клиенты и менеджеры не рассчитывают, что разработчики сбавят темпы. Скорее, они думают, что если реализация функции заняла одну или пару недель в начале проекта, то и через год подобная функция будет реализована в те же сроки. Они ожидают, что производительность будет стабильна на протяжении всего проекта.

Разработчики должны ожидать этого не меньше. Благодаря непрерывному поддержанию чистоты архитектуры, проекта и кода настолько, насколько это возможно, сохраняется высокая производительность, и команда не попадает в ловушку медлительности и перепроектирования.

Далее вы увидите, что такие методы Agile, как тестирование, парное программирование, рефакторинг и простота структуры проекта, помогают избежать попадания в эту ловушку. А игра в планирование — это противоядие от давления сроков, подталкивающего разработчиков в эту ловушку.

Недорогие изменения

Программное обеспечение — это словосочетание. Со словом «программное» все понятно. Рассмотрим слово «обеспечение». Оно означает, что пользователь обеспечен функционалом, необходимым

мым для успешного выполнения своих задач. Получается, что программное обеспечение — это программы, обладающие необходимым функционалом. А полнота функционала невозможна без изменяемости кода. Программное обеспечение изобрели потому, что возникла необходимость легко и быстро вносить изменения в функциональность техники. Если мы хотим, чтобы функционал изменялся только на аппаратном уровне, то пользуемся понятием «аппаратное обеспечение».

Разработчики беспрерывно сетуют на то, что требования к продукту изменяются. Мне часто доводилось слышать заявления вроде: «Это изменение полностью рушит архитектуру нашей программы». Солнышко, у меня для тебя плохие новости. Если изменение требований ломает архитектуру твоей программы, то твоей архитектурой можно только подтереться.

Мы, разработчики, должны радоваться изменениям, потому что ради них и работаем.

Изменение требований — основное правило игры. Эти изменения дают нам работу и кормят нас. Наша работа заключается в способности мириться с изменениями требований и находить решения, которые будут относительно недороги.

Эффективность программного обеспечения измеряется в том числе и тем, насколько безболезненно в него можно внести какие-либо изменения. Клиенты, пользователи и руководители ожидают, что программное обеспечение будет можно легко изменить и что стоимость таких изменений будет соизмеримой и как можно ниже.

Далее вы узнаете о том, как методы Agile, разработка через тестирование, рефакторинг и простота проектирования при одновремен-

ном применении и с наименьшими усилиями помогают вносить изменения в программы без ущерба для них.

Постоянное совершенствование

Со временем люди повышают уровень своего мастерства. Художники лучше пишут картины, певцы лучше исполняют песни, строители лучше возводят здания и так далее. То же должно быть справедливым и для отрасли разработки программного обеспечения. Чем дольше существует программа, тем лучше она должна становиться.

Структура и архитектура программного обеспечения со временем должны становиться только лучше.

Структура кода должна становиться лучше и, соответственно, должно произойти улучшение производительности программы. Разве это не очевидно? Разве это не то, что вы ожидаете от команды, работающей над чем-либо?

То, что мы с течением времени работаем все хуже — самое серьезное обвинение и наиболее очевидное свидетельство нашей некомпетентности.

Самое безответственное отношение, которое только может быть, пожалуй, проявляется в случае, когда разработчики настроены на то, что по мере продвижения код будет становиться беспорядочным, а программа — неработоспособной, нестабильной и уязвимой.

Постоянное и уверенное совершенствование — вот чего от нас ждут клиенты, пользователи и руководители. Они ожидают, что «детские болезни» программы со временем пройдут, что в дальнейшем

она будет становиться только лучше. Методы Agile, а именно разработка через тестирование, парное программирование, рефакторинг и простота структуры проекта, призваны укрепить здоровые ожидания.

Компетенция без страха и упрека

По какой причине большая часть программ со временем не становится лучше? Эта причина — страх. Если быть точнее, страх перед изменениями.

Представьте, что вы смотрите на экран, на котором находится написанный ранее код. Первая мысль, которая приходит в голову: «Господи, какой ужас! Нужно почистить его». Но следующая мысль будет примерно такой: «Нет, я в это не полезу!» А почему? Да потому что вы думаете, что если влезть в код, то он перестанет работать. А если он перестанет работать, виноваты будете вы. Так вы отказываетесь от того единственного, что может улучшить код, — от его очистки.

В вас говорит страх. Вы боитесь кода, и этот страх вынуждает вас вести себя некомпетентно. Вы чувствуете некомпетентность и боитесь проводить очистку кода, потому что исход непредсказуем. Вы позволили коду, который создали своими собственными руками, стать таким самостоятельным, что боитесь что-либо делать для его улучшения. Это в высшей мере безответственно.

Клиенты, пользователи и менеджеры ожидают, что вы профессио-нал без страха и упрека. Они ожидают, что если в коде будут ошибки или избыточность, вы это увидите, исправите и вычистите. Вы не допустите разрастания и усугубления несовершенств и сделаете все возможное для поддержания высокого качества и чистоты кода.

Так как же избавиться от этого страха? Представьте, что у вас есть кнопка, а возле нее две лампочки — красная и зеленая. Представьте, что когда вы нажимаете на кнопку, то загорается одна из них. Зеленая — если программа работает правильно, красная — если неправильно. Представьте, что нажатие на кнопку и получение результата занимает лишь считанные секунды. Насколько часто вы будете на нее нажимать? Вы будете нажимать на нее без остановки. Вы будете нажимать на нее постоянно. Каждый раз, когда вы внесете изменения в код, вы нажмете на кнопку, чтобы понять, все ли работает правильно.

Теперь представьте, что на экране вы видите уродливый код. Первая мысль, которая проносится у вас в голове: «Надо его почистить». И после этого вы просто берете и чистите его, нажимая кнопку после каждого внесенного изменения, чтобы удостовериться в том, что все работает.

Больше нет никакого страха. Вы спокойно чистите код. Чтобы устраниТЬ недостатки кода, можно использовать методы Agile: рефакторинг, простоту структуры проекта и парное программирование.

Где взять эту кнопку? Разработка через тестирование дает вам ту самую кнопку. Если вы применяете этот метод дисциплинированно и решительно, то достигнете компетенции без страха и упрека.

Контроль качества

Контроль качества должен показать, что неполадок нет. Продукт должен быть сделан так, чтобы после прохождения тестирования специалист сообщил, что все работает должным образом. Каждый

раз, когда при прохождении контроля качества обнаруживаются какие-то ошибки, разработчики должны выяснить, что пошло не так во время работы, и устранить недостатки, чтобы в следующий раз такого не было.

Специалисты по контролю качества (Quality Assurance) должны задаться вопросом, почему они застряли в конце системы проверки процесса, которая всегда работает. Так что пусть проверяют кого-нибудь другого, для этого есть более достойные кандидаты.

Методы Agile, а именно приемочное тестирование, разработка через тестирование, непрерывная интеграция, позволяют не бояться контроля качества.

Автоматизация тестирования

На рис. 2.1 можно увидеть руки. Это руки руководителя QA-отдела. В них он держит *оглавление плана для тестирования*, проводимого вручную. В нем содержится 80 тысяч тестов, которые должна проводить армия индусов раз в полгода. Чтобы провести это тестирование, нужно потратить миллион долларов.

QA-менеджер принес мне этот документ, когда вышел от своего начальника. Его начальник, в свою очередь, до этого был в кабинете финансового директора. Это был 2008 год. Начало мирового финансового кризиса. Финансовый директор каждые полгода сокращал этот миллион вдвое. Руководитель отдела контроля качества, протягивая мне план, попросил отобрать из списка тестов половину, которую можно не проводить.

Я объяснил ему, что неважно, какие тесты будут проводить, а какие нет. Важно то, что мы так не узнаем, работает ли половина программ.

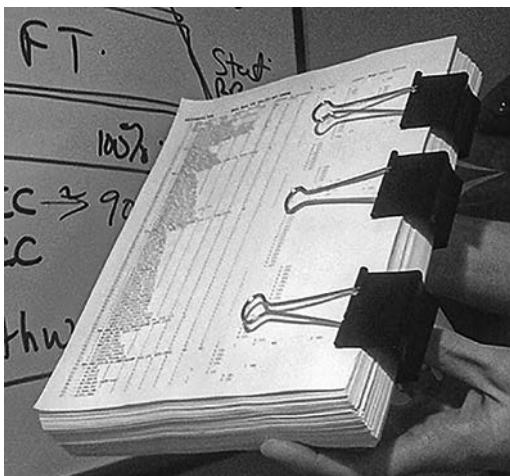


Рис. 2.1. Оглавление плана тестирования вручную

Тестирование вручную всегда в итоге провальное и неизбежно приводит к такому положению дел. Вы только что прочитали наиболее очевидное доказательство того, что тестирование вручную несостоит, поскольку оно дорого стоит и всегда попадает под сокращение финансирования.

К тому же существует еще один коварный нюанс в доказательство того, что тестирования вручную лучше избегать.

Разработчикам редко удается представить продукт на контроль качества вовремя. Получается, что у специалистов по качеству остается меньше времени на требуемые тесты, чем отводилось изначально. Поэтому им приходится на свое усмотрение выбирать, какие тесты нужны больше, чтобы закончить проверку в положенные сроки. Некоторые тесты остаются без внимания. Это провал.

Кроме того, люди не роботы. Требовать от них выполнять ту же работу, что и машины, — дорого, неэффективно и бесчеловечно. Для

специалистов, проводящих контроль качества, есть работа и занятия гораздо интереснее — там, где можно проявить воображение и творчество, присущие человеку. Но вернемся к нашим баранам.

Клиенты и пользователи ожидают, что каждый новый релиз пройдет тщательное тестирование. Никто даже не подумает, что разработчики решили что-либо не тестировать лишь по той причине, что не было времени или средств. Поэтому нужно автоматизировать все тесты, какие только возможно. Вручную стоит тестировать только то, что нельзя подвергнуть автоматической проверке, и тогда, когда требуется творческий подход, который применяется в исследовательском тестировании¹.

Методы Agile, а именно приемочное тестирование, разработка через тестирование, непрерывная интеграция, позволяют оптимизировать тестирование.

Друг за друга горой

Как технический директор я ожидаю от команды слаженной работы. Что значит «слаженная работа»? Представьте футбольную команду, гоняющую мяч по полю. Один из игроков спотыкается и падает. Что делают остальные игроки? Они закрывают образовавшуюся брешь, защищая проход к воротам, и стараются увести мяч как можно дальше.

На борту корабля у каждого своя работа. Незаменимых нет, каждый умеет выполнять не только свою, но и чужую работу. Потому что на корабле необходимо выполнение всех работ.

¹ Agile Alliance. Exploratory testing. Ссылка: <https://www.agilealliance.org/glossary/exploratory-testing>

Так же и в команде разработчиков. Если Боб захворал, его подменит Джек. Это значит, что Джеку нужно знать, чем занимается Боб, где Боб хранит все свои исходные файлы, скрипты и тому подобное.

Я ожидаю от каждого члена любой команды разработчиков готовности подменить товарища. В свою очередь, каждый член команды разработчиков может положиться на своих коллег, если вдруг что-то случится. Личное дело каждого сделать так, чтобы кто-то из команды мог вас подменить.

Если Боб спец по части баз данных и вдруг заболел, я не опасаюсь того, что работа над проектом из-за этого подвиснет. Кто-то, даже если он особо не «шарит» в базах данных, должен занять место Боба. Я ожидаю, что никто в команде не станет утаивать сведения, — сведениями нужно делиться. Если требуется перевести половину команды на другой проект, я уверен, что половина всех сведений никуда не исчезнет вместе с ушедшими, но останется в команде.

Методы Agile, а именно парное программирование, «одна команда» и коллективное владение, призваны оправдать ожидания технического директора.

Честность оценок

Я ожидаю, что будут какие-то оценки сроков и что они будут честными. Самая честная оценка — это «я не знаю». Тем не менее эта оценка не полноценна. Нельзя знать все. Вы хоть что-то, да знаете. Так что я ожидаю, что ваши оценочные суждения будут основаны на имеющихся знаниях.

Например, вы можете не знать, сколько времени займет та или иная задача, но можете сопоставить ее с другой и сделать вывод

на ее основе. Можно не знать, сколько времени уйдет на создание страницы входа в личный кабинет, но можно предугадать, что на страницу смены пароля уйдет примерно половина времени, потраченного на страницу входа. Относительные оценки, как та, что приведена выше, весьма ценные. Мы еще в этом убедимся на страницах этой книги.

Вместо относительной оценки можно высказать предположения с некоторым вероятностным разбросом. Например, вы можете предположить, что на страницу входа уйдет где-то от пяти до пятнадцати дней, а в среднем на это уходит двенадцать дней. Такие оценки сочетают в себе все имеющиеся знания, из которых можно вывести некоторую вероятность и передать сведения руководству.

Методы Agile, а именно «игра в планирование» и «одна команда», помогут провести оценку запланированных задач.

Умение говорить «нет»

Хотя и очень важно стремиться найти решение задачи, лучше сказать «нет», если такого решения не найдено. Просто осознайте, что вас взяли на работу даже не потому, что вы хорошо пишете код, а потому что вы можете сказать «нет», когда это нужно. Вы, программисты, — как раз те, кто знает, что возможно, а что нет. Как технический директор я ожидаю от вас сведений, дабы не упасть в пропасть. Я рассчитываю на это независимо от того, поджимают ли сроки, независимо от того, что желают услышать менеджеры. Просто ответьте «нет», когда это действительно нужно.

Метод Agile «одна команда» научит честно говорить «нет», если того требует положение дел.

Тяжело в учении, легко в бою

Как технический директор я ожидаю от вас стремления постоянно учиться. В нашей сфере все быстро меняется. И мы должны быть способны меняться вместе с ней. Так что век живи — век учись! Иногда компания сможет себе позволить отправить сотрудника на курсы или конференции. А иногда — закупить обучающую литературу и видеоуроки. Но даже если этого всего не будет, нужно искать возможность учиться самому, не дожидаясь помощи дяди из вашей компании.

Метод Agile «одна команда» поможет достичь новых вершин в обучении.

Мудрый коуч

Как технический директор я ожидаю от вас и ваших коллег взаимного обучения. Правда в том, что лучше всего мы учимся тогда, когда сами кого-то учим. Поэтому когда в команду приходят новички, не стесняйтесь их учить чему-то. Учитесь помогать друг другу. И снова на помощь приходит метод Agile «одна команда», который научит вас друг друга поддерживать.

БИЛЛЬ О ПРАВАХ

Во время встречи в Сноуберде Кент Бек заявил, что цель Agile — построить мост над пропастью, существующей между клиентами и разработчиками. Для этого Кент Бек, Уорд Каннингем, Рон Джейфрис и некоторые другие составили «Билль о правах».

Обратите внимание на то, что права клиентов и права разработчиков дополняют друг друга. Они подходят друг к другу, словно ше-

стерни одного механизма. Благодаря им существует некий баланс ожиданий между одними и другими.

Права клиента

У клиента есть следующие права:

- Ознакомиться с общим планом, знать, что и когда можно получить и за какие деньги.
- Получать наилучшую, насколько это возможно, отдачу от каждой итерации.
- Отслеживать ход работ, назначать необходимые тесты, получить рабочее и многократно протестированное программное обеспечение.
- Изменять решения, функциональность и приоритеты, не неся непомерных расходов.
- Получать сведения о графике работ, изменениях сроков, чтобы вовремя принять решение о том, как сократить объем работ и успеть к нужному числу. Отменить проект в любое время и получить полезный рабочий код, который оправдывает текущие вложения средств.

Права разработчика

У разработчика есть следующие права:

- Знать, что требуется от команды, а также иметь четкое представление о поставленных приоритетах.
- Выполнять работу качественно, несмотря ни на какие обстоятельства.

- Получать помощь от коллег, руководителей и самих клиентов.
- Проводить оценку задачи и уточнять ее в зависимости от обстоятельств.
- Брать на себя личную ответственность и не позволять возлагать на себя лишнее.

Все эти утверждения звучат весьма громко. Давайте рассмотрим их по очереди.

Клиенты

В данном контексте слово «клиент» относится к представителям бизнеса. Сюда относятся непосредственно клиенты, менеджеры, начальники, менеджеры проекта и им подобные, то есть все, кто способен нести ответственность за соблюдение сроков и расходование средств, либо те, кто платит деньги и получает выгоду от использования программы.

Клиенты могут потребовать предоставить им общий план на ознакомление, также имеют право знать, что и когда можно получить и за какие деньги.

Многие утверждают, что такое предварительное планирование не является частью методологии Agile. Самое первое из прав клиента уже опровергает это высказывание. Безусловно, в бизнесе требуется планирование. И, конечно, в план должны входить график работ и их стоимость. Разумеется, план должен быть как можно более точным и осуществимым.

Именно на последнем пункте мы часто не знаем, что делать, так как единственный способ построить точный и выполнимый план можно только в процессе выполнения проекта. Невозможно, почти

ничего не делая, создать нормальный план. Так что мы, разработчики, должны убедиться, что наши планы, оценки и графики работ правильно создают представление об уровне нашей неопределенности, и найти способы снизить этот уровень, чтобы гарантировать это право.

Проще говоря, нельзя соглашаться с невозможностью изменить объем и сроки работ. И объем работ, и сроки должны быть гибкими. Мы представляем такую гибкость с помощью кривой вероятностей. Например, согласно нашей оценке, вероятность того, что мы выполним первые десять историй к назначенному числу — 95 %. С вероятностью 50 % мы выполним вовремя следующие пять. И с вероятностью 5 % мы выполним еще пять историй в требуемый срок.

У клиентов есть право ознакомиться с этим планом, основанным на вероятности, потому что они не могут вести дела без плана.

Клиенты имеют право получать наилучшую, насколько это возможно, отдачу от каждой итерации.

Agile позволяет разбить объем работ на временные отрезки, называемые итерациями. Клиенты имеют право ожидать от разработчиков работы над наиболее важными задачами в любое время, а от каждой итерации — максимально возможную пользу для бизнеса. Приоритеты устанавливаются клиентами на этапе планирования в начале каждой итерации. Клиенты выбирают истории, которые приносят наибольшую отдачу от капиталовложений и которые команда разработчиков может, по ее оценкам, выполнить за итерацию.

Клиенты имеют право на отслеживание хода работ, назначение необходимых тестов, получение рабочего и многократно протестированного программного обеспечения.

Это право кажется очевидным с точки зрения клиента. Конечно, у них есть право отслеживать, как продвигается ход работ. Безусловно, у них есть право определять показатели того, что работа действительно продвигается. Несомненно, у них есть право неизменно и неоднократно проверять, действительно ли ход работ соответствует требуемым показателям.

Клиенты имеют право на изменения решений, функциональность и приоритеты, не неся при этом непомерных расходов.

В конце концов, мы говорим об отрасли разработки программного обеспечения. Цель программного обеспечения — легко изменять функциональность техники, которую мы используем. Первым делом ПО изобрели затем, чтобы функциональности придать гибкость. Поэтому, конечно же, у клиентов есть право на изменение своих требований.

Клиенты имеют право на получение сведений о графике работ, изменениях оценки сроков, чтобы вовремя принять решение о том, как сократить объем работ и успеть к нужному числу.

Клиент может отменить проект в любое время и получить полезный рабочий код, который оправдывает текущие вложения средств.

Обратите внимание, что у клиентов нет права требовать, чтобы их график соответствовал графику команды разработчиков. Их право на управление графиком ограничено правом на изменение объема работ. Самое главное, чем наделяет клиента это право — правом знать, что график работ находится под угрозой срыва, и иметь возможность своевременно привести его в соответствие с текущими условиями.

Разработчики

В данном контексте разработчиком признается каждый, кто принимает участие в написании кода. Это понятие включает в себя программистов, QA-специалистов и бизнес-аналитиков.

Разработчики имеют право на ознакомление с тем, что требуется от команды, а также на четкое представление о поставленных приоритетах.

Опять же сосредоточимся на *знании*. У разработчиков есть право на получение точных требований и сведений о том, какие из них важны и насколько. Конечно, требования ограничены возможностью их выполнимости примерно так же, как и сроки сдачи проекта. Как и с оценкой сроков, не всегда получается точно выразить требования. Не стоит забывать и о том, что у клиентов есть право изменять свои решения.

Это право применимо только в рамках одной итерации. За пределами итерации требования и приоритеты будут смещаться и изменяться. Но в пределах итерации у разработчиков есть право считать, что они незыблемы. Однако всегда нужно помнить, что разработчики могут отказаться от этого права, если считают изменения требований несущественными.

Разработчики имеют право на качественное выполнение работы, несмотря ни на какие обстоятельства.

Это должно быть самое главное право среди всех прочих. У разработчиков есть право делать свою работу хорошо. Клиенты не имеют права требовать от команды вести разработку спустя рукава. Или, другими словами, у клиентов нет права вынуждать разработчиков портить свою профессиональную репутацию или переступать через профессиональную этику.

Разработчики имеют право на помощь от коллег, руководителей и самих клиентов.

Такая помощь может выражаться по-разному. Программисты могут просить друг у друга помощи в решении задач, сверке результатов, в изучении фреймворка и во многом другом. Разработчики могут попросить клиентов лучше объяснить требования или уточнить приоритеты. По большей части, этот постулат дает разработчикам право на взаимодействие. И с правом просить о помощи приходит и ответственность — помогать, когда просят.

Разработчики имеют право на оценку задачи и ее уточнение в зависимости от обстоятельств.

Никто не сможет оценить задачу за вас. И если вы даете оценку задаче, у вас всегда есть возможность ее изменить по мере появления новых факторов. Ведь оценки основаны на предположениях. Конечно, эти догадки небезосновательны, но они и остаются догадками. Эти догадки становятся точнее со временем. Оценка не может быть обязательством.

Разработчики имеют право самостоятельного принятия на себя ответственности и недопущения возложения на себя лишнего.

Профессионалы принимают предложения по работе, а не назначение заданий. Разработчик имеет полное право говорить «нет», если его не устраивает какая-либо работа или задание. Может быть такое, что разработчик не уверен в своих способностях выполнить ту или иную задачу или, может быть, считает, что кто-то другой справится лучше, чем он. Или, допустим, разработчик отказывается из личных или моральных соображений¹.

¹ Вспомните разработчиков из «Фольксваген», которые «согласились» выполнить задание обойти испытания на количество выбросов вредных веществ в атмосферу. https://ru.wikipedia.org/wiki/Дело_Volkswagen.

В любом случае за право выбора приходится платить. Принятие предложения налагает ответственность. Согласившийся разработчик принимает на себя ответственность за качественное выполнение задания, за постоянное уточнение оценок, чтобы выверять график работ, и за взаимодействие со всей своей командой. Он не должен стесняться просить о помощи, когда она ему необходима.

Программирование в команде подразумевает тесное сотрудничество, где есть старшие и младшие программисты. У команды есть право согласованно решать, кто чем будет заниматься. Технический руководитель может попросить разработчика взяться за задание, но ни в коем случае никого не принуждать.

ЗАКЛЮЧЕНИЕ

Agile – это набор методов, помогающий разработчикам достигать и поддерживать высокий уровень профессионализма. Специалисты, которые придерживаются этих методов, принимают и отвечают разумным ожиданиям руководителей, заинтересованных сторон и клиентов. Agile также наделяет определенными правами как разработчиков, так и клиентов и накладывает на них некоторые обязательства. Взаимное соблюдение прав и принятие ожиданий, признание методов Agile – основа этической нормы для сферы разработки ПО.

Agile не процесс. Agile не модное увлечение. Agile не просто набор правил. Скорее, Agile – это набор прав, ожиданий и методов, который составляет основу профессиональной этики специалистов, связанных с разработкой программного обеспечения.

МЕТОДЫ ВЗАИМОДЕЙСТВИЯ С КЛИЕНТАМИ



Существует куча методов взаимодействия с клиентами, которые разработчику нужно соблюдать, чтобы преуспеть. Среди них планирование, небольшие и частые релизы, приемочное тестирование и «одна команда».

ПЛАНИРОВАНИЕ

Как провести оценку проекта? Самое простое: его нужно разбить на составные части, а затем провести их оценку. Подход-то хороший, но что делать, если эти части слишком велики, чтобы достоверно их оценить? Тогда нужно просто разбить эти части на меньшие по объему и оценивать уже их. Уверен, вы подумаете, что это попахивает рекурсией.

Как далеко можно зайти в такой разбивке? Проект можно делить вплоть до каждой строчки кода. Это как раз то, чем занимаются программисты. Программистом можно назвать того, кто постиг искусство разбиения задач на отдельные строки кода.

Если вы хотите провести наиболее точную и достоверную оценку проекта, разбейте его вплоть до отдельных строк кода. Да, это займет какое-то время, но зато вам станет доподлинно известно, сколько времени займет проект, ведь вы его уже выполнили.

Конечно, это уже никакая не оценка. Оценки построены на догадках: нам нужно знать, сколько времени займет выполнение проекта, не выполняя его непосредственно. Хочется, чтобы оценка проекта не стоила дорого. Поэтому оценка по определению не может быть точной. Неточность позволяет нам сократить сроки, необходимые для проведения оценки. Чем ниже степень точности, тем меньше времени понадобится на оценку.

Это не означает, что оценка должна быть неточной. Нужно давать как можно более точную оценку, но точную настолько, чтобы это не было непомерно дорого. Вот пример для лучшего понимания. По моим оценкам, я закончу свой жизненный путь в течение следующей тысячи лет. Это совершенно верно, но точность очень низка. Я буквально не потратил ни минуты на то, чтобы провести настолько достоверную оценку, потому что она совсем неточная. Такая хоть и достоверная, но неточная оценка обозначает временной отрезок, в течение которого оцениваемое событие произойдет почти наверняка.

Задача разработчиков — потратить как можно меньше времени, чтобы произвести достоверную оценку, и при этом в наибольшей степени сократить временной промежуток.

Трехвариантный анализ

Есть один метод, который неплохо подходит для больших задач. Это *трехвариантный анализ*. Такие оценки включают в себя три положения: *лучший случай, допустимый случай и худший случай*. С помощью такого анализа можно предречь исход с различной вероятностью. Худший случай — это когда мы уверены в сроках на 95 %. Допустимый — на 50 %, а лучший — на 5 %.

Например, я могу быть на 95 % уверен, что задание будет выполнено в течение трех недель. Только на 50 % я могу быть уверен, что оно будет выполнено за две недели. И 5 % вероятности, что получится уложиться в одну неделю.

Можно представить вероятность и иначе. Представим, что у нас 100 схожих заданий. Пять из них будут выполнены в течение недели, 50 — в течение двух недель, а 95 из них — в течение трех недель.

Существует целый математический метод, связанный с управлением трехвариантными оценками. Кому интересно, могу предложить изучить технику оценки и анализа программ и проектов (PERT)¹. Это мощный метод управления крупными проектами и портфелями проектов. Если вы отдельно не изучали эту технику, не стоит думать, что прочитав о ней, вы уже все знаете. PERT выходит далеко за рамки графиков, которые вы видели в Microsoft Project.

Несмотря на то что трехвариантный анализ хорошо подходит для долгосрочной оценки всего проекта, он слишком неточен для повседневного управления, которое необходимо в течение проекта. Для этого у нас есть другой подход — *единицы сложности* историй.

Истории и единицы сложности

Метод единиц сложности историй позволяет соблюсти достоверность и точность благодаря коротким циклам обратной связи, с помощью которых можно многократно выбирать оценку в сравнении с действительностью. Сначала точность невысока, но через несколько циклов она повышается до приемлемого уровня. Но прежде чем мы углубимся в метод, поговорим немного о самих историях.

Пользовательская история — это сокращенное описание функции программы с точки зрения пользователя. Например:

Когда я веду машину, то нажимаю сильнее на педаль газа, чтобы поехать быстрее.

Это один из наиболее распространенных видов пользовательских историй. Некоторым людям нравится так. Другие предпочитают

¹ <https://ru.wikipedia.org/wiki/PERT>.

ее в более коротком виде: «ускориться». Оба вида достаточно хороши. Обе истории представляют собой упрощенную версию того, что описывается большим количеством слов. Многое из того, что описано в истории, еще не произошло. Это произойдет ближе к моменту разработки функции программы. А вот само действие начинается тогда, когда историю уже пишут. В то время разработчики и заинтересованные стороны обсуждают некоторые возможные подробности истории и затем записывают все в упрощенном виде.

Еще рано точно говорить о подробностях, поэтому подробности опущены, а описание упрощено. Мы стараемся отложить уточнение подробностей как можно на более долгий срок, до тех пор пока не начнется разработка по этой истории. Поэтому оставим историю в сокращенном виде как предпосылку для дальнейших действий¹.

Как правило, мы записываем истории на каталожных карточках. Понимаю-понимаю. С чего бы мы использовали такую примитивную древность, когда в современном мире у нас есть компьютеры, планшеты и прочее, спросите вы? Но выходит, что возможность держать карточки у себя в руках, передавать их друг другу через стол, писать на них и делать с ними многое другое представляется чрезвычайно важной.

Иногда автоматизированные средства могут их заменить, я расскажу о них в другой главе. Но сейчас будем считать, что истории написаны на карточках.

Вспомните, что во Вторую мировую войну боевые действия планировались² с помощью карточек, поэтому я считаю этот метод проверенным.

¹ Это одно из определений истории, данное Роном Джейфрисом.

² Ну, в какой-то степени, все равно.

Истории для банкоматов

Давайте представим, что мы находимся на нулевой итерации и наша команда пишет истории для банкомата. Что представляют собой эти истории? Первые три довольно легко вычислить: выдача наличных, внесение наличных и перевод. Конечно, нужно научить банкомат идентифицировать пользователя. Можно назвать эту историю «вход». А раз есть вход, значит, вероятно, должен быть и выход.

Теперь представим, что у нас есть пять карточек. Их будет почти наверняка больше, как только мы начнем по-настоящему углубляться в работу банкомата. Мы можем представить, что есть такие задачи, как аудит, платеж по кредиту и много всего прочего. Но давайте пока что остановимся на пяти карточках.

Что у нас на них? То, о чем мы недавно упоминали — вход, выход, выдача наличных, внесение наличных и перевод. Конечно, это не все слова, которые мы перечислили во время нашего исследования. Во время встречи мы обговорили много подробностей. Мы обсуждали, как пользователь осуществляет вход посредством того, что вставляет карту в приемник и вводит пин-код. Мы обсуждали, что собой представляет внесение наличных, как их вставляют в купюropриемник и как он пересчитывает эти средства. Мы обсуждали, как выдаются наличные средства и что делать в случае, если купюры застряли или просто закончились. Мы прорабатывали многие из этих подробностей.

Пока что все эти подробности неточны, поэтому мы их и не записываем. Мы записываем только слова. Нет ничего плохого в том, что вы сделаете несколько пометок на карточке, если вам нужны напоминания по каким-то вопросам, но это необязательно. На карточки не накладывается формальных ограничений.

Отказ от подробностей возможен благодаря дисциплине. И это непросто. Каждый член команды посчитает нужным так или иначе обсудить все подробности, ничего не упустив. Сдерживайте эти порывы!

Как-то раз я работал с менеджером проекта, который настаивал на том, что нужно записывать на карточке подробности абсолютно каждой истории. Карточки с историями были разбиты на единицы сложности, а сами единицы сложности были написаны мельчайшим шрифтом. Их было невозможно прочесть, и они стали непригодными. На них было столько нюансов, что оценить задания просто не оставалось шансов. Составить график работ по ним не получалось. Пользы от них не было никакой.

Хуже того, в каждую карточку с историей была вложена уйма усилий, поэтому от них нельзя было просто избавиться.

Именно временная нехватка подробностей делает историю осуществимой, планируемой и оцениваемой. Истории должны создаваться с небольшими затратами, потому что многие из них придется изменить, разделить, объединить или даже забыть. Просто не забывайте о том, что они являются заменителями, а не требованиями.

Теперь у нас есть кипа карточек, созданных во время нулевой итерации. Остальные создадут позже, по мере возникновения новых идей и функций. На самом деле создание историй никогда не прекращается. Истории постоянно пишут, изменяют, отбрасывают и, что самое главное, развиваются в ходе проекта.

Оценка историй

Представьте, что эти карточки лежат на столе напротив вас, а вокруг стола сидят остальные разработчики, тестировщики и заинтересованные стороны. Все вы встретились для того, чтобы оценить

истории, помещенные на эти карточки. Еще состоится много встреч вроде этой. Они будут проводиться каждый раз, когда будут добавлены новые истории или уточнено что-то насчет старых. Стоит ожидать, что такие встречи будут непринужденными, но закономерными для каждой итерации.

Однако еще начало нулевой итерации, и это самая первая, оценочная встреча. До этого оценка историй не проводилась.

Итак, мы выбираем из кипы историю средней сложности. Допустим, это история «вход». Многие из нас присутствовали при написании этой истории, поэтому слышали разнообразные подробности, которые, как представляли себе заинтересованные стороны, должны быть приведены в ней. Мы, скорее всего, попросим партнеров рассмотреть эти подробности сейчас, чтобы убедиться в том, что понимаем сложившиеся обстоятельства одинаково.

Потом мы оценим реализацию истории в пунктах. История «вход» получит 3 единицы по сложности разработки соответствующей функции (рис. 3.1). Почему 3? А почему бы и нет? История «вход» средняя по сложности, поэтому она получает средний балл. Три – это средний балл при шестибалльной шкале оценки историй.



Рис. 3.1. История «вход» получает три единицы сложности

Теперь история «вход» становится эталоном. С ней будут сравнивать все истории при проведении оценки. Так, например, выход из системы гораздо проще, чем вход в нее. Поэтому история «выход» получает единицу. Выдача наличных, вероятно, в два раза сложнее, чем вход, поэтому эта история получает 6 единиц. Внесение наличных представляет собой примерно то же самое, что и выдача, однако эту историю реализовать чуть легче. Отдадим ей 5 единиц. И, наконец, реализация перевода не сложнее, чем реализация входа, поэтому она получает те же 3 единицы.

Мы записываем эти числа в одном из верхних углов карточки с историей, которую мы оценивали. Я еще расскажу подробнее о том, как даются оценки. А сейчас давайте считать, что у нас есть кипа карточек с историями, которые мы оцениваем в единицах от 1 до 6. Почему от 1 до 6? А почему бы нет? Существует много схем, по которым можно оценивать сложность. Обычно чем проще система оценок, тем лучше.

В этой связи, возможно, у вас появится вопрос: а что в действительности измеряют единицы сложности? Возможно, вы подумаете, что время — часы, дни, недели или что-то подобное. Но нет. Скорее это единица измерения затрачиваемых *усилий*, а не времени. Действительно, они позволяют измерить сложность выполнения истории.

Единицы сложности историй должны быть линейными. История на карточке, оцениваемая в 2 единицы, должна быть вдвое легче той, что оценена в 4. Впрочем, линейность не обязана блюстись в совершенстве. Помните, что это оценка, поэтому точность намеренно остается размытой. На историю, оцененную в 3 единицы, Джим может потратить два дня, если больше не появится ошибок, на которые приходится отвлекаться. А Пэт сможет выполнить ее за день, если работает из дома. Эти оценки слишком расплывчаты,

неотчетливы и неточны, поэтому их нельзя связывать с определенными промежутками времени.

Но в таких расплывчатых и неотчетливых оценках есть своя прелесть. Эта прелест — закон больших чисел¹. При многократном выполнении задания размытость сводится на нет! Это преимущество мы будем использовать позднее.

Планирование первой итерации

Между тем пришло время планировать первую итерацию. Встречу по ее планированию можно считать началом. Эта встреча по плану должна составлять одну двадцатую продолжительности всей итерации. То есть если итерация длится две недели, на нее требуется затратить полдня.

На встрече должна быть вся команда, в том числе заинтересованные стороны, программисты, тестировщики и менеджер проекта. Заинтересованные стороны должны заблаговременно прочитать оцениваемые истории и отсортировать их в порядке выполнения в зависимости от своих потребностей. Бывает, что команды оценивают приоритет историй в зависимости от требований клиента тем же способом, что и сложность выполнения задач. Есть команды, которые большее внимание уделяют приоритетам.

На встрече задача заинтересованных сторон заключается в том, чтобы выбрать истории, которые программисты и тестировщики будут выполнять во время итерации. Чтобы это сделать, им нужно знать, сколько единиц программисты, по их мнению, смогут выполнить. Количество историй за итерацию называется скоростью. Конечно, поскольку это только первая итерация, никто не может сказать, ка-

¹ https://ru.wikipedia.org/wiki/Закон_больших_чисел.

кова будет скорость хода работ. Поэтому приходится делать предположения. Предположим, что скорость будет 30 единиц за итерацию.

Важно осознавать, что скорость не является обязательством. Команда не дает обещание, что 30 единиц будут выполнены за итерацию. Она даже не обещает постараться выполнить эти 30 единиц. Эта скорость не более чем предположение о том, сколько единиц в лучшем случае будут выполнены к концу итерации. А такое предположение не может отличаться высокой достоверностью.

Прибыль от вложений

Теперь заинтересованные стороны нарисовали квадрат и поделили его на четыре участка (рис. 3.2).



Рис. 3.2. Квадрат с четырьмя участками

Приоритетные и несложные истории можно начинать выполнять хоть сейчас. Приоритетные, но более сложные можно немного отложить на потом. Неприоритетные и несложные можно сделать за день. А о тех, что не представляют особой ценности, да еще и сложны, можно забыть.

С помощью этого квадрата выполняется расчет окупаемости инвестиций (ROI). Здесь нет формальностей и не нужно никакой математики. Заинтересованные стороны просто смотрят на карточку и делают вывод, основываясь на оценках приоритета и сложности истории.

Например: «*Вход* довольно важен, но также и сложен. С ним подождем. *Выход* тоже важен, но будет попроще. О! Давайте его! *Выдача наличных...* сложно, очень сложно. Но нам в первую очередь важно показать как раз эту функцию. Поэтому делайте ее».

Мы только что увидели, как заинтересованные стороны расставляют приоритеты. Заинтересованные стороны рассматривают карточки с историями и ищут те, от которых можно получить наибольшую отдачу. Они останавливаются, когда выбранные истории в сумме дают 30 единиц сложности. Это план на итерацию.

Промежуточная проверка

А теперь за работу! Позже я расскажу в подробностях о том, как ведется разработка по историям. А сейчас просто представим, что есть какой-то способ превратить истории в работающий код. Представьте этот способ как перекладывание карточек с историями из кучи *запланированное* в кучу *сделанное*.

К середине итерации должно быть выполнено много историй. А сколько единиц должно оставаться? Правильно, столько же, сколько и сделано. В нашем случае это 15. Чтобы провести промежуточную проверку, нужно уметь делить на два.

Поэтому давайте созвонем всех на промежуточное совещание. Пускай это будет понедельник, первый день второй недели итерации. На встречу должны прийти члены команды и заинтересованные стороны — последние оценивают ход работ.

Ой-ей, да истории выполнены только на 10 единиц! Осталась всего неделя, вряд ли за нее выполнят больше двадцати! Поэтому заинтересованные стороны вычтут некоторое количество историй из плана, чтобы осталось только 10 единиц до конца итерации.

В пятницу после обеда подводится итог итерации с демонстрацией. Оказывается, выполнено лишь 18 единиц. Итерация провальна?

Нет! *Итерации не бывают провальными*. Целью итерации является сбор данных для менеджеров. Конечно, было бы здорово, если бы во время каждой итерации появлялся код, но даже когда этого не происходит, главное — собрать данные.

Вчерашняя погода

Теперь мы знаем, сколько единиц сложности историй мы можем выполнить за неделю — около 18. Как думаете, сколько единиц запланируют заинтересованные стороны в понедельник, в начале следующей итерации? Без сомнений, восемнадцать. Это называют вчерашней погодой. Вчерашняя погода лучше всего предскажет, какая погода будет сегодня. А вернее всего ход текущей итерации может предсказать то, что получено в ходе предыдущей.

Поэтому на встрече по планированию заинтересованные стороны добавляют столько историй, чтобы получилось 18 единиц. Но на этот раз на таком же промежуточном совещании происходит кое-что, что не укладывается в голове. Выполнено 12 единиц. Надо ли говорить об этом партнерам?

Нет, не стоит. Они сами все увидят. Поэтому заинтересованные стороны добавляют шесть дополнительных единиц, чтобы по плану вышло 24.

Конечно, выходит так, что команда успевает выполнить только 22. Поэтому на следующую итерацию планируется 22 единицы.

Окончание проекта

И на этой итерации происходит то же самое. По завершении итерации скорость, с которой выполняются истории, наносится на соответствующий график, чтобы все могли видеть, как быстро команда выполняет свою работу.

Предположим, что так продолжается итерация за итерацией, месяц за месяцем. Что происходит с кипой карточек с историями? Представьте, что кругооборот итераций — это насос, перекачивающий ROI из этой кипы. И представьте, что непрерывное появление требований в ходе проекта — это насос, перекачивающий окупаемость обратно в кипу. Пока входящие вложения превосходят их отток, проект будет продолжаться.

Однако может случиться и так, что количество новых функций для реализации постепенно сведется к нулю. Когда это произойдет, денежные средства, которые содержат карточки, будут исчерпаны через несколько итераций. Придет день, когда на встрече по планированию заинтересованные стороны посмотрят на карточки с историями и поймут, что работы больше не осталось. Проект завершен.

Проект заканчивается не тогда, когда выполнены все истории. Проект можно считать завершенным, когда больше не остается карточек с историями, которые стоит выполнять.

Иногда удивляет, какие карточки с историями остаются в кипе, когда проект заканчивается. Однажды я работал над проектом, который длился год. Самая первая история, написанная для проекта и давшая ему название, так и не была реализована. Эта исто-

рия была важна в свое время, но позже появились более срочные истории, за которые брались разработчики. И к тому времени, когда срочные истории были выполнены, оказалось, что важность первоначальной истории испарилась.

Истории

Пользовательские истории — это простые инструкции, которые напоминают нам о свойствах функций. Мы стараемся не записывать слишком много подробностей, истории мы пишем как раз потому, что знаем, что подробности наверняка не раз изменятся. Подробности нужно записывать позже, в качестве приемочных тестов.

Истории должны соответствовать следующим атрибутам качества (критерии INVEST).

- **Независимость (Independent).** Пользовательские истории независимы друг от друга. Это значит, что истории необязательно выполнять в каком-то определенном порядке. Например, *выход* не нужно выполнять после *входа*.

Это гибкое требование: бывает и так, что какие-то истории будут зависеть от других, которые нужно реализовать в первую очередь. К примеру, если мы реализуем *вход* без функции *восстановления забытого пароля*, то однозначно *восстановление пароля* в какой-то мере зависит от *входа*. И все же мы стараемся отделить истории так, чтобы они были зависимы друг от друга в наименьшей мере. Это позволяет выполнять истории в нужном клиенту порядке.

- **Обсуждаемость (Negotiable).** А это еще одна причина, по которой мы не записываем все подробности. Нужно, чтобы разработчики и клиенты могли обсуждать эти подробности.

Например, клиенты могут запросить классный интерфейс с возможностью перетаскивания в него чего-либо. Разработчики могут посоветовать интерфейс попроще, с флагками, объяснив, что разработка в таком случае будет проще и дешевле. Обсуждения важны, поскольку это один из немногих способов, с помощью которых клиенты получают представление о том, как управлять стоимостью разработки ПО.

- **Ценность (Valuable).** Клиенты хотят видеть, что у истории есть определенная измеримая ценность.

Рефакторинг нельзя считать историей. Архитектуру тоже нельзя. И чистка кода никакая не история. История — это всегда что-то, имеющее ценность для бизнеса. Не переживайте, нам доведется иметь дело с рефакторингом, архитектурой и чисткой кода, но не с самими историями.

Это значит, что история будет проходить через все уровни разработки программы. То есть она может частично затрагивать часть реализации графического интерфейса, промежуточного программного обеспечения, баз данных и так далее. Представьте, что история — это тонкий вертикальный срез, проходящий сквозь горизонтальные слои проекта.

Количественное измерение ценности можно проводить без формальностей. Некоторым командам может вполне подойти шкала приоритетов «высокий — средний — низкий», кому-то больше понравится десятибалльная шкала. Не имеет значения, какую шкалу использовать, главное — уметь чувствовать разницу между историями, ценность которых значительно отличается.

- **Поддаваемость оценке (Estimable).** Пользовательская история должна быть достаточно конкретной, чтобы разработчики могли сделать прогноз.

История «программа должна быть быстрой» не поддается оценке, потому что быстрота не имеет предела. Это сопутствующее требование, которое относится ко всем историям.

- **Компактность (Small).** Пользовательская история должна быть небольшой, чтобы один-два разработчика смогли с ней справиться за одну итерацию.

Не нужно, чтобы одна история огромным одеялом накрывала всю команду в течение целой итерации. Желательно, чтобы в итерации было примерно то же количество историй, что и разработчиков в команде. Если в команде 8 разработчиков, нужно подбирать от 6 до 12 историй для каждой итерации. Вы ведь не хотите завязнуть на одном этапе, верно? Это скорее совет, чем правило.

- **Тестируемость (Testable).** Клиенты должны четко назвать тесты, позволяющие подтвердить, что история выполнена.

Как правило, эти тесты пишут QA-специалисты. Тесты должны быть автоматизированы, с их помощью выполняется проверка историй на завершенность. Подробнее об этом вы узнаете позже. А пока что не забывайте, что каждая история должна быть достаточно конкретной, чтобы можно было подобрать необходимые тесты.

Но, может, это идет вразрез с принципом обсуждаемости? Нет, потому что когда мы пишем историю, то не обязаны знать, какие тесты нужны. Когда возникнет необходимость, тогда и будет написан тест. Хотя я еще не знаю всех нюансов истории *вход*, я уверен, что ее можно протестировать, потому что *вход* — это конкретное действие. А теперь посмотрим на историю *пригодный к эксплуатации*. Ее никак не протестируешь. И даже никак не оценишь. Действительно, прогнозируемость и тестируемость идут рука об руку.

Оценка историй

Существует множество способов оценки историй. Большинство из них — разновидности подхода Wideband Delphi¹.

Один из простейших способов — *оценка на пальцах*. Разработчики сидят вокруг стола, вчитываются в историю и, если необходимо, обсуждают ее с заинтересованной стороной. Потом разработчики прячут одну руку за спину так, чтобы никто не мог ее увидеть, скрывают пальцы так, чтобы по пальцам можно было понять, сколько единиц сложности требуется, по их мнению, затратить на выполнение истории. После этого кто-нибудь начинает счет, и на счет «три» все одновременно достают руки с зажатыми пальцами из-за спины.

Если все показали одинаковое число пальцев либо отклонение невелико и в среднем все показывают одно и то же, то число записывается на карточке с историей, а команда переходит к следующей истории. Но если между участниками встречи есть значительные разногласия, то разработчики обсуждают причину и повторяют действия до тех пор, пока согласие не будет достигнуто.

Можно оценивать истории по *размерам одежды*: большие, средние и маленькие. Если хочется задействовать все пять пальцев, продолжайте. С другой стороны, использование шкалы с большим количеством баллов почти всегда абсурдно. Помните, мы хотим дать достоверную оценку, а не наиболее точную.

*Покер планирования*² — похожий способ, но там нужны карты. Существует много популярных колод карт для такого покера.

¹ https://en.wikipedia.org/wiki/Wideband_Delphi.

² Grenning J. W. 2002. Planning Poker, or How to avoid analysis paralysis while release planning. URL: <https://wingman-sw.com/articles/planning-poker>.

В большинстве из них применяется что-то вроде рядов Фибоначчи. В одной распространенной колоде содержатся такие карты: ?, 0, $\frac{1}{2}$, 1, 2, 3, 5, 8, 13, 20, 40, 100 и ∞ . Если у вас такая колода, мой совет — уберите оттуда большую часть карт.

Преимущество рядов Фибоначчи в том, что с помощью них можно провести оценку больших историй. Например, вы можете взять 1, 2, 3, 5 и 8, благодаря чему получите восьмикратный размер.

Можно еще взять карты 0, ∞ и ?. В способе с пальцами можно вместо этих знаков показать палец вниз, палец вверх и открытую ладонь. Ноль означает «слишком незначительно, чтобы оценивать». С такими историями нужно быть осторожными! Возможно, стоит объединить несколько таких историй в одну побольше. Бесконечность (∞) означает, что история слишком большая, чтобы провести ее оценку, а это значит, что ее нужно разделить. И знак вопроса (?) означает неизвестность — нам не от чего отталкиваться.

Разбиение, слияние и костыли

В слиянии историй нет ничего сложного. Можно просто скрепить карточки вместе и рассматривать эти истории как одну. Просто посчитайте сумму единиц сложности. Если есть истории, оцененные в ноль единиц, хорошо подумайте, как оценить их при сложении. В этом случае пять нолей не могут дать ноль в сумме.

Разбивка историй будет сложнее: нужно сделать так, чтобы истории соответствовали правилам их написания. В качестве простого примера разбиения истории рассмотрим *вход*. Если мы захотим разбить эту историю на несколько историй поменьше, то можем создать на ее месте *вход без пароля*, *вход с одной попытки*, *вход с нескольких попыток* и *забыли пароль?*.

История, которую нельзя разбить на несколько, — большая редкость. Особенно это касается тех крупных историй, которые не можно, а нужно разбивать. Помните, что работа программиста заключается в том, чтобы разбивать истории вплоть до отдельных строк кода. Поэтому разбиение возможно почти всегда. Самое сложное — соблюдать правила написания историй.

Костыль — это метаистория, точнее история для оценки истории. Ее называют костылем, так как она напоминает длинный тонкий клин, который пронизывает всю разработку программы насквозь, подпирая другую историю.

Допустим, есть история, которую у вас не получается оценить. Пускай это будет *печатать PDF*. Почему вы не знаете, как ее оценить? Потому что до этого вы никогда не использовали библиотеку PDF и у вас нет точного представления, как она работает. Поэтому вы пишете новую историю, которую называете *оценка печати в PDF*. Теперь оцениваем уже ее, и ее-то оценить проще. Вы уже знаете, что нужно сделать, чтобы выяснить, как работает библиотека PDF. Обе истории отправляются в кипу с карточками.

На одной из последующих встреч по планированию заинтересованные стороны могут попробовать выбрать карточку *печатать PDF*. Но не тут-то было — костыль встанет им поперек горла. Поэтому придется выбрать карточку с костылем. Это позволит разработчикам выполнить работу, необходимую для оценки первоначальной истории, которую можно провести в одной из следующих итераций.

Управление итерацией

Цель каждой итерации — получение данных посредством выполнения историй. Команде скорее следует сосредоточиться на

историях, чем на задачах, которые под ними скрываются. Предпочтительнее выполнить 80 % историй, чем выполнить все истории на 80 %. Сосредоточьтесь на доведении историй до их выполнения.

Как только заканчивается совещание по планированию, программисты должны выбрать истории, за которые будут нести личную ответственность. В некоторых командах выбирают истории, находящиеся сверху, а остальные откладывают на потом, чтобы взяться за них по выполнении выбранных ранее. Как бы то ни было, программисты сами выбирают и выполняют истории.

Менеджеры и лидеры могут поддаться искушению назначать истории программистам. Но этого нужно избегать. Пускай программисты договорятся между собой.

Например:

Джерри (опытный спец):

— Если не возражаете, я возьму на себя вход и выход. Думаю, есть смысл выполнить их вместе.

Жасмин (опытный спец):

— Не вопрос, только почему бы вам вдвоем с Альбертом не взяться за базы данных? Он постоянно расспрашивает, как мы используем источники событий. Мне кажется, что вход даст ему некоторую ясность. Что скажешь, Альберт?

Альберт (стажер):

— О! Звучит круто! Я видел, как это делается. Я даже за выдачу наличных взяться могу!

Алексис (ведущий программист):

— А почему бы мне не взять выдачу наличных? Альберт! Давай ты поработаешь над ней вместе со мной. Потом возьмешь перевод.

Альберт:

— Угу, ладно. Наверное, так будет лучше. Кто сначала перекладывает камешки, потом передвигает горы, ведь так?

Жасмин:

— Ну конечно, Альберт! И остается внесение наличных. Я возьму это на себя. Алексис, мы с тобой в одной упряжке, нам нужно поработать над пользовательским интерфейсом, ведь они у нас, вероятно, мало отличаются. И мы должны иметь возможность делиться кодом.

В этом примере можно увидеть, как ведущий программист направляет новичка, амбициозного стажера, не давая ему взять на себя больше, чем он может, а еще то, как члены команды сообща выбирают себе истории в работу.

Контроль качества и приемочное тестирование

Если QA-специалисты еще не начали писать автоматизированные приемочные тесты, то нужно это делать как можно скорее, прямо после встречи по планированию. Тесты для историй, рассчитанные на быстрое выполнение, должны быть готовы как можно раньше. Не нужно ждать, пока приемочные тесты созреют для выполненных историй.

Приемочные тесты должны появляться быстро. Мы ожидаем, что их полностью напишут еще до середины итерации. Если к середине итерации готовы не все приемочные тесты, кто-то из разработчиков должен бросить работу над историей и помогать писать тесты.

Это означает, что за эту итерацию не удастся выполнить все запланированные истории, однако в любом случае без приемочного тестирования ни о каком выполнении не может идти речи. Только

убедитесь, что программисты, работающие над какой-либо историей, не пишут приемочные тесты для нее же. Если специалисты по качеству каждый раз не успевают написать тесты к середине итерации, это, вероятнее всего, означает, что соотношение разработчиков и QA-специалистов выбрано неверно.

Если по достижении середины итерации все приемочные тесты написаны, то QA-специалисты должны приняться за тесты к следующей итерации. Это рискованно, поскольку встреча по планированию еще не состоялась, однако заинтересованные стороны могут дать рекомендации насчет историй, которые они выберут с наибольшей вероятностью.

Разработчики и QA-инженеры должны обсудить такие тесты. Мы не хотим, чтобы разработчики молча брали то, что им бросают QA-специалисты. Необходимо обсуждать структуру тестов и писать их сообща, вплоть до парного программирования.

До середины итерации команде нужно постараться выполнить истории, чтобы успеть к промежуточному совещанию. А до конца итерации разработчикам нужно постараться успеть подвергнуть оставшиеся истории соответствующим приемочным тестам.

Если мы говорим, что история выполнена, то подразумеваем, что она прошла приемочное тестирование.

В последний день итерации разработчикам предстоит муки выбора: какие истории стоит завершить, а от каких придется отказаться? Мы предпочитаем перераспределять усилия. Так получается выполнить столько историй, сколько возможно.

Опять же ни к чему заканчивать итерацию с историями, выполненными лишь наполовину, ведь есть возможность пожертвовать какой-то историей, чтобы полностью выполнить другую.

Здесь не нужна спешка. Нужны конкретные результаты и измеримость хода работ. Нужны надежные данные. Историю можно считать выполненной, когда пройдено приемочное тестирование. Когда программист говорит, что выполнил историю на 90 %, в реальности непонятно, насколько она готова. Поэтому на графике скорости работ стоит отмечать лишь истории, прошедшие приемочное тестирование.

Демонстрация

Каждая итерация заканчивается короткой демонстрацией новых функций заинтересованным сторонам. Такая встреча должна длиться не больше часа или двух, в зависимости от размера итерации. В демо-версии должно быть видно, что истории, в том числе все предшествующие, полностью прошли приемочное и модульное тестирование. Следует также показать свежий функционал, добавленный к программе. Лучше всего, если заинтересованные стороны сами проверяют работоспособность программы, чтобы у разработчиков не было соблазна скрыть то, что не работает.

Скорость

Завершающий аккорд итерации — обновление графика скорости и диаграммы сгорания задач. На графике нужно отмечать единицы только тех историй, которые прошли соответствующие приемочные испытания. После нескольких итераций графики пойдут на спад. Спад диаграммы сгорания задач помогает выявить, какого числа будет достигнута следующая важная веха. График скорости же показывает нам, насколько хорошо организована работа команды.

Во время начальных итераций график скорости будет довольно неточным, так как команда еще толком не разобралась в основах

проекта. Но после первых нескольких итераций точность графика возрастет и достигнет уровня, позволяющего четче определить истинную скорость выполнения работ.

Мы ожидаем, что после нескольких первых итераций угол наклона приблизится к нулю, то есть график примет горизонтальный вид. Мы не ждем от команды ускорения или замедления в долгосрочной перспективе.

Возрастание скорости

Если мы видим, что график пополз вверх, это вряд ли означает, что команда действительно стала работать быстрее. Скорее всего, это означает, что менеджер проекта выжимает из разработчиков все соки, подгоняя их. Как только руководство прибегает к давлению, то команда бессознательно начинает мухлевать с оценками, чтобы создать впечатление, будто стала работать быстрее.

Это обычное надувательство. Единицы сложности можно представить как своеобразную валюту, которая обесценивается командой из-за напряжения, создаваемого извне. Вернемся к такой команде через год, и что мы увидим? Этих единиц за итерацию у них будет больше 9000! Мораль такова, что не нужно гнаться за скоростью — это всего лишь способ измерения. Зарубите себе на носу: нельзя подгонять то, что измеряешь.

Оценка итерации на встрече по планированию проводится лишь затем, чтобы дать знать заинтересованным сторонам, сколько историй возможно выполнить. Она помогает выбирать истории и планировать их выполнение. Однако такая оценка не является обязателькой — ни о какой провальной итерации не идет речи, если скорость вдруг оказалась ниже. Помните, что только та итерация провальная, из которой не удалось получить данных.

Снижение скорости

Если график скорости неуклонно ползет вниз, то, скорее всего, причина в качестве кода. Команда, вероятнее всего, проводит мало рефакторинга, и код начинает портиться. Одна из причин нехватки рефакторинга — команда пишет недостаточно модульных тестов и боится, что из-за рефакторинга перестанет работать то, что работало прежде. Борьба с этим страхом — важнейшая задача лидеров команды разработчиков, она полностью зависит от дисциплины при тестировании. Подробнее об этом дальше.

Когда скорость падает, усиливается давление на команду. Это ведет к обесцениванию единиц сложности. И за таким раздуванием единиц может скрываться падение скорости выполнения проекта.

Золотая история

Один из способов избежать обесценивания единиц сложности историй — постоянно сравнивать истории с первоначальным «золотым эталоном», историей, с которой сравнивают все остальные. Вспомните, нашей первоначальной «золотой» историей был *вход*, который мы оценивали в 3 единицы. Если новая история, например *исправить опечатку в меню*, оценена в 10 единиц, то очевидно, что оценка этой истории раздута.

НЕБОЛЬШИЕ ЧАСТЫЕ РЕЛИЗЫ

Согласно методу «небольшие и частые релизы», команде разработчиков нужно выпускать релизы своего программного обеспечения как можно чаще. В конце девяностых, когда Agile только появился, мы думали, что норма — это выпуск релиза раз в месяц-два. Но сейчас этот срок стал гораздо короче. Сокращать срок можно до

бесконечности. Увеличение частоты релизов происходит благодаря *непрерывной доставке* (continuous delivery). Этот метод заключается в том, что релиз происходит после каждого изменения кода.

Это толкование может ввести в заблуждение, потому что выражение «непрерывная доставка» создает впечатление, что мы хотим сделать короче только цикл доставки. Это не так, мы хотим сократить каждый цикл.

К сожалению, в силу исторических обстоятельств мы имеем препятствие в виде некой значительной инерции. Эта инерция –rudимент тех способов, к которым мы прибегали при работе с исходным кодом, когда трава была зеленее, а деревья выше.

Краткая история управления исходным кодом

История управления исходным кодом — это повесть о циклах и их размерах. Она берет начало в 1950–1960-х годах, когда исходный код хранили в отверстиях, пробитых на кусочках бумаги (рис. 3.3).

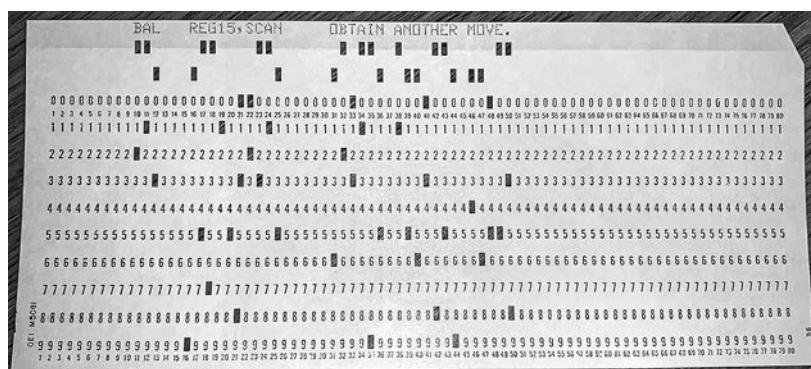


Рис. 3.3. Перфокарта

Многие из нас тогда пользовались перфокартами. Кarta вмещала на себе 80 символов и представляла собой одну строку программного кода. Сама программа занимала целую колоду таких карт. Обычно их перетягивали резинкой и хранили в коробке (рис. 3.4).

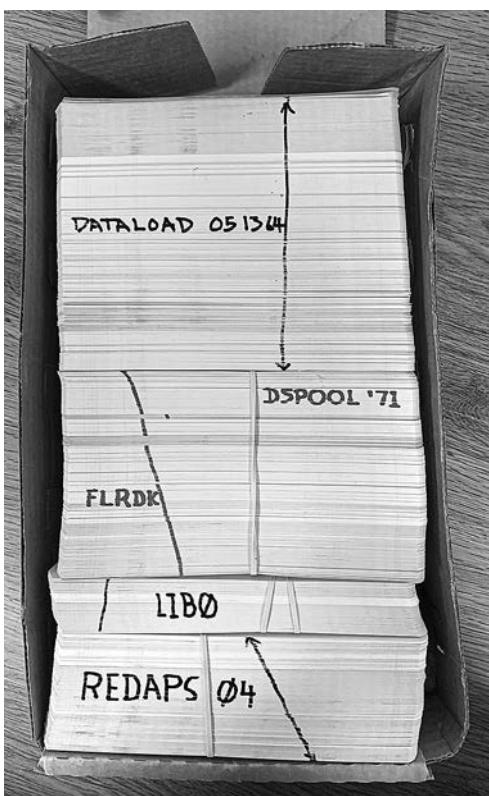


Рис. 3.4. Колоды перфокарт в коробке

Владелец программы хранил колоду перфокарт в выдвижном ящике или шкафчике. Если кому-то нужно было проработать исходный код, это приходилось делать прямо из ящика или шкафчика с позволения владельца.

Если у вас получилось проверить исходный код, то вы были единственным, кто мог внести в него изменения, поскольку имели возможность физически посмотреть перфокарты. Больше никто не мог их касаться. Когда дело было сделано, нужно было отдать колоду владельцу, который клал ее в ящик или шкафчик.

Цикл для этой программы составлял столько, сколько времени у программиста был к ней физический доступ. Счет мог идти на дни, недели, месяцы.

Ленты

В 1970-х мы плавно перешли к тому, что стали хранить образы перфокарт с исходным кодом на магнитной ленте. На магнитной ленте можно держать большое количество программных модулей, а еще ее было проще копировать. Редактирование модулей выглядело так:

1. Достать главную ленту с оригиналом из главной стойки.
2. Скопировать модули, которые нужно отредактировать, с главной ленты на ленту с рабочей копией.
3. Положить главную ленту на место, чтобы другие могли получить доступ к прочим модулям.
4. Воткнуть цветную кнопку на доску выдачи рядом с названием модулей, которые нужно отредактировать. (У меня была синяя, у начальника красная, у остальных программистов из моей команды были желтые. Да-да, в конце концов у нас закончились цвета.)
5. Редактировать, компилировать и тестировать на ленте с рабочей копией.

6. Снова достать главную ленту.
7. Скопировать измененные модули с ленты с рабочей копией на главную ленту.
8. Положить обновленную главную ленту в стойку.
9. Вынуть кнопку из доски.

И снова цикл составлял ровно столько, сколько кнопка находилась на доске выдачи. Это могло занимать часы, могло дни и даже месяцы. Покуда кнопка находилась на доске выдачи, никто больше не мог обращаться к модулям, которые вы закрепили за собой.

Разумеется, эти модули были и на главной ленте, и в крайнем случае кто-нибудь мог в обход правил отредактировать модули непосредственно на ней. Кнопки были условным соглашением, но никак не физической преградой.

Диски и системы управления исходным кодом

В 1980-х годах исходные коды переместились на диски. Поначалу еще использовалась доска выдачи и кнопки, но потом начали появляться настоящие средства управления исходным кодом. Первое из того, что я помню, — это Система управления исходным кодом (SCCS). Она работала по тому же принципу, что и доска выдачи. Происходила блокировка модуля на диске, из-за чего никто не мог получить доступ к его редактированию. Такое блокирование называется пессимистическим. И снова то же самое. Длительность цикла зависела от длительности блокирования. Блокировка могла длиться часами, днями, а то и месяцами.

На смену Системе управления исходным кодом пришла Система управления редакциями (RCS), которая, в свою очередь, уступила место Системе одновременных версий (CVS). Во всех так или иначе применялась пессимистическая блокировка. Поэтому цикл по-прежнему длился долго. Тем не менее хранение данных на диске было куда удобнее, чем на магнитной ленте. При копировании модулей с оригинала на ленту с рабочей копией очень соблазнительно было сделать модули крупными сами по себе.

С другой стороны, диски позволяли нам стремительно уменьшать размер модулей. Множество маленьких модулей просто-напросто не вело к таким издержкам, как несколько крупных. За счет уменьшения модулей продолжительность цикла стала короче, поскольку количество времени, затрачиваемое на проверку модуля, также относительно уменьшилось.

Проблема состояла в том, что изменения в программе влекли за собой изменения во многих модулях. В случаях, когда модули были тесно связаны, на их проверку все еще уходило много полезного времени. Некоторые из нас научились отделять связанные модули, чтобы уменьшить эти сроки. А кто-то так и не научился.

Subversion

Потом пришло время системы Subversion (SVN). В ней появилось оптимистическое блокирование. Оптимистическое блокирование по сути никакое и не блокирование. Разработчик мог проверять модуль одновременно с другим разработчиком. Система позволяла отслеживать действия разработчиков и автоматически объединять изменения в модулях. Если система обнаруживала конфликт, когда два разработчика работали над одними и теми же строками кода, то вынуждала программиста

сперва разрешить этот конфликт, прежде чем подтвердить принятые изменения.

Это значительно сократило продолжительность цикла до времени, требуемого на редактирование, компиляцию и тестирование последовательности небольших изменений. Связанность модулей еще представляла собой проблему. Тесно связанные модули замедляли цикл, потому что приходилось вносить изменения во много модулей одновременно. Однако программы, где модули не были так тесно связаны, проходили цикл гораздо быстрее. Сам срок проверки больше не служил ограничивающим фактором.

Git и тесты

В наши дни мы используем Git. Сроки проверки при использовании Git сошли на нет. Это понятие больше не существует. На-против, любое изменение в модуль можно внести когда угодно. Программисты разрешают конфликты между этими изменениями, как и когда этого пожелают. Маленькие несвязанные модули позволяют молниеносно и часто вносить изменения. Поэтому циклы составляют считанные минуты. Добавьте к этому возможность создавать всеобъемлющие и быстрые тестовые наборы, которыми можно протестировать практически все. Вот вам и все нужное для непрерывной доставки.

Историческая инерция

К сожалению, организации с трудом отказываются от унаследованных подходов. Циклы продолжительностью в дни, недели и месяцы глубоко укоренились в культуре многих команд, откуда затем перешли к QA-специалистам, менеджерам и даже заинтересованным сторонам. С колокольни такой «культуры»

мысль о непрерывной доставке может показаться бредом сумасшедшего.

Небольшие и частые релизы

Agile пытается порвать шаблоны исторической инерции, предлагая как можно сильнее сократить циклы релизов. Если у вас релиз проходит раз в полгода, постарайтесь выпускать раз в три месяца, раз в месяц, раз в неделю, в конце концов. Продолжайте уменьшать циклы релизов, асимптотически стремясь к нулю.

Чтобы достичь этого, организации требуется разорвать связь между релизом и развертыванием. Понятие «релиз» означает, что программа технически готова к развертыванию. Решение о развертывании полностью ложится на плечи клиентов.

Возможно, вы заметили, что теми же самыми понятиями мы описывали итерации. Итерации с технической точки зрения можно развертывать. Если продолжительность итерации составляет две недели, но мы хотим выпускать релизы чаще, придется итерации сократить.

Можно ли сокращать итерации асимптотически, стремясь к нулю? Да, можно. Но это уже тема отдельного разговора.

ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ

Приемочное тестирование — один из самых непонятных и запутанных методов Agile, который используется реже всего. Это любопытно, потому что основной посыл удивительно прост: требования указывает клиент.

Проблема, несомненно, в понимании слова «указывает». Многие клиенты считают, что могут, поводив руками в воздухе, расплыв-

что и туманно описать функционал, а потом произойдет чудо — разработчики сами догадаются обо всех мельчайших нюансах. А многие программисты хотят, чтобы клиенты *точно* объяснили, как должна работать программа, вплоть до координат и цвета каждого пикселя на экране.

Из этих двух крайностей нужно вывести золотую середину.

Что же такое спецификация? Спецификация по своей сути — это функция, которую можно протестировать. Например:

Когда пользователь вводит правильное имя учетной записи и пароль, а потом нажимает «войти», на экране появится страница приветствия.

Вот это и есть указание, то есть спецификация. Очевидно, что это можно протестировать.

Очевидно, что тест для этой функции можно автоматизировать. Нет причин на то, чтобы компьютер не мог удостовериться в выполнении указанного требования.

Вот так работает приемочное тестирование. Опыт подсказывает, что насколько это вообще осуществимо, требования к системе должны представлять собой автоматизированные тесты.

Погодите! А кто же пишет эти тесты? Первый абзац этого раздела отвечает на наш вопрос: требования указывает клиент. Тогда получается, что клиенты должны писать автоматизированные тесты, ведь так?

Погодите! Автоматизированные тесты должны быть написаны на каком-то формальном исполняемом языке. И это работа для программистов, не иначе! Получается, что программистам придется писать эти тесты?

Погодите! Если программисты будут писать тесты, то это будет не то, чего хочет клиент. Это будут технические тесты с множеством нюансов, понятных только программистам. Они не отражают ценность тестируемого элемента для клиента. Тогда автоматизированные тесты будет писать клиент? Ведь так?

Погодите! Если автоматизированные тесты будет писать клиент, то они не будут соответствовать технологии, которую мы используем. И программистам тогда придется их переписывать, не так ли?

Вот тут-то и становится понятно, почему этот метод вводит многих людей в заблуждение.

Инструменты и методологии

А еще хуже то, что наш метод погряз в инструментах и методологиях.

Пытаясь облегчить клиентам написание автоматизированных тестов, программисты написали целое изобилие инструментов, оказывающих медвежью услугу. Имеются в виду поделки вроде FitNesse, JBehave, SpecFlow и Cucumber. Каждый из этих инструментов предоставляет формы, призванные отделять техническую сторону автоматизированного теста от пользовательской. Гипотеза состоит в том, что клиент может написать пользовательскую часть автоматизированного теста, в то время как программисты могут написать код, связывающий эти тесты с тестируемой программой.

Звучит круто, и инструменты вроде как неплохо способствуют разделению труда. Тем не менее клиенты неохотно берутся за подобное. Представители компаний, ответственные за указание требований в спецификациях, побаиваются формальных языков. Они в своем большинстве предпочитают человеческие языки

вроде русского или английского для написания спецификаций с требованиями.

Увидев, что клиенты ни в какую не хотят писать приемочные тесты, программисты решили исправить положение, и в надежде на то, что клиенты хотя бы прочитают документы, написанные формальным языком, уже сами написали для них тесты. Но и это не сработало, ведь клиенты терпеть не могут формальные языки.

Они предпочтут посмотреть на работающую программу или в лучшем случае доверят тестирование QA-специалистам.

Разработка через поведение

С наступлением нового тысячелетия Дэн Норт начал работу по пересмотру разработки через тестирование. Получившуюся методологию он назвал разработкой через поведение. Он поставил себе цель избавиться от жаргона технарей в тестах, чтобы тесты больше напоминали спецификации с требованиями, которые так любят клиенты.

Сначала это было еще одной попыткой формализации языка тестирования, в этом случае применялось три ключевых слова: «дано», «когда» и «тогда». Было создано или модифицировано несколько инструментов для поддержки этого языка. В их числе JBehave, Cucumber и FitNesse. Но с течением времени упор стал делаться не на инструменты и тесты, а на требования и спецификации.

Сторонники разработки через поведение предполагают, что клиенты могут извлечь большую пользу, указывая требования к своим программам на формальном, основанном на сценариях языке вроде того, который использует ключевые слова «дано», «когда» и «тог-

да», независимо от того, автоматизированы ли в действительности такие требования в виде тестов.

Это избавляет клиентов от соответствия техническим требованиям, которые налагает написание действительно исполняемых тестов, в то же время позволяя быть тестам формальными и точными.

Как показывает практика...

Несмотря на всю противоречивость и запутанность, которые мы увидели выше, в приемочном тестировании нет ничего сложного. Клиенты пишут формальные тесты, которые содержат описание каждой пользовательской истории, а разработчики эти тесты автоматизируют.

Эти тесты пишутся бизнес-аналитиками и QA-специалистами до завершения первой половины итерации, когда должны разрабатываться истории, которые впоследствии будут проходить эти тесты. Разработчики объединяют эти тесты в непрерывную сборку. Такие тесты соответствуют критериям готовности для историй, разработанных во время итерации. Требования к истории не считаются указанными, пока для нее не написан приемочный тест. История не считается завершенной, пока не пройдено приемочное тестирование.

Бизнес-анализ и контроль качества

Приемочные тесты — результат совместной работы бизнес-аналитиков, QA-специалистов и разработчиков. Бизнес-аналитики указывают, что должно происходить в лучшем случае. Они являются связующим звеном между заинтересованными сторонами и программистами и выражают их желание получить хороший продукт.

Специалисты по контролю качества, напротив, обрисовывают наихудший исход. Сейчас способов получить такой исход намного больше, чем раньше. QA-специалисты зарабатывают на хлеб тем, что вычисляют уязвимости программы. Технари до мозга костей, они способны предвидеть все финты и фокусы, которые может выкинуть программа. Они также знают, как мыслят программисты и как определить халтуру.

И, конечно, не обойтись без самих разработчиков. Они, работая со специалистами по качеству и бизнес-аналитиками, обеспечивают гарантии того, что тесты будут иметь смысл с технической точки зрения.

Контроль качества

Здесь, конечно, QA-специалисты играют уже более важную роль. Если сначала они выполняют тыловую работу тестировщиков, то теперь выходят на первый план, находясь на передовой линии проекта. В начале работ они дают обратную связь после написания кода, указывая на ошибки и упущения, а потом занимаются предотвращением этих ошибок, заблаговременно предоставляя необходимые данные разработчикам.

Нагрузка на специалистов по качеству многократно возрастает. Контроль качества теперь должен проходить в начале каждой итерации, а не выявлять недостатки и несоответствие на финальной стадии. Однако в любом случае важность контроля качества не преуменьшается — именно инженеры по качеству определяют, можно ли развернуть систему.

Тесты в конце бесполезны

Проведение контроля качества и автоматизация тестирования решают еще одну важную задачу. Когда специалист по контролю

качества проводит тестирование в конце, да еще и вручную, он попадает в щекотливое положение.

Работа должны быть выполнена до развертывания программного обеспечения. Но менеджерам и заинтересованным сторонам просто не терпится скорее начать развертывание, и они наседают на QA-инженеров.

Когда контроль качества переносится на финал проекта, весь удар берут на себя специалисты по качеству. А если разработчики отдадут продукт на проверку с опозданием, будет тогда перенос сроков? Часто сроки определяются очень важными деловыми причинами, поэтому перенос будет дорого стоить или даже в корне расстроит планы клиента. В итоге козлом отпущения остаются QA-специалисты.

Как специалистам по качеству тестировать программу, если на это не остается времени в графике работ? Как им выполнить работу быстрее? Очень просто: пропустить некоторые тесты. Протестировать только то, что изменилось. Провести анализ воздействия на функции, которые появились недавно или претерпели изменения, а потом протестировать уязвимости. Не тратить время на тестирование того, что не изменилось.

Так пропускаются необходимые тесты. Под давлением сроков QA-специалисты пропускают все регрессионные тесты, надеясь, что проведут их в следующий раз. Но чаще всего «следующий раз» не наступает.

Болезнь контроля качества

Все перечисленное выше не самое худшее, что происходит с контролем качества на финише. Если контроль качества проводится

в конце, как клиенты узнают, хорошо ли выполняется работа? Естественно, по количеству выявленных недочетов. Если специалисты обнаруживают тонну ошибок, то они явно не просто так получают зарплату. QA-специалисты могут завышать количество таких ошибок, чтобы показать, как замечательно справляются со своими задачами.

И считается, что раз что-то найдено, значит, *все хорошо*.

Кому еще выгодны недочеты? Среди старых программистов есть такая присказка: «Уложусь-то я в любые сроки, а как оно будет работать – это уже другой разговор». Так кто еще выигрывает от найденных недочетов?

Разработчики, которым нужно уложиться в сроки.

И здесь не нужны слова. Не требуется никаких договоренностей. Обе стороны понимают, что только выиграют. Начинается черная торговля недочетами. Эта болезнь свойственна многим компаниям, она не то чтобы смертельна, но весьма изматывает.

Разработчики в роли тестировщиков

Эти проблемы лечатся методом приемочного тестирования. QA-специалисты пишут приемочные тесты для историй, выполненных за итерацию. Но само тестирование проводят не они. Не QA-специалисты должны проводить тестирование программы. Тогда кто? Программисты, конечно!

Это работа программистов. Разработчики ответственны за подтверждение того, что их код проходит все тесты. Поэтому проведение тестирования, безусловно, работа программистов. Проведение тестов – единственный способ проверить, выполнены ли истории.

Непрерывная сборка

А будет так, что программисты автоматизируют тестирование¹, установив сервер непрерывной сборки. Каждый раз, когда программист запускает проверку какого-нибудь модуля, сервер будет запускать необходимые программы, с помощью которых будут проводиться тесты, в том числе все модульные и приемочные. Подробнее о непрерывной интеграции далее в этой книге.

ОДНА КОМАНДА

В экстремальном программировании практика «одна команда» изначально называлась «заказчик всегда рядом» (On-Site Customer). Идея заключалась в том, что чем меньше дистанция между пользователями и программистами, тем лучше коммуникация, и разработка таким образом становится быстрее и точнее. Заказчик был метафорой кого-то или группы людей, которые понимали потребности пользователей и были рядом с командой разработчиков. В идеале клиент сидел в одной комнате с командой.

В Scrum заказчик называется «владелец продукта». Это человек или группа, которые выбирают истории, ставят приоритеты и дают своевременную обратную связь.

Метод переименовали в «одна команда», чтобы стало ясно, что команда разработчиков — это не дуэт «заказчик — программист». В команде разработчиков вклад вносят все, в том числе руководители, тестировщики, разработчики технической документации и т. д. Цель этого метода — в наибольшей степени улучшить кон-

¹ Потому что автоматизация — это работа программистов!

такт между всеми участниками. В идеале все участники должны сидеть в одной комнате.

Вряд ли вызывает сомнения, что нахождение всей команды в одном пространстве увеличивает ее эффективность. Команда может общаться быстро и с минимум формальностей. Ответ на вопрос можно получить за несколько секунд. Рядом всегда есть опытные товарищи, которые могут подсказать.

Более того, это повышает вероятность непреднамеренных интуитивных открытий. Представитель заказчика всегда может посмотреть в экран программиста или тестировщика и заметить, что происходит что-то не так. Тестировщик может случайно услышать, например, что программисты, работающие в паре, обсуждают требования, и понять, что они пришли к неверному выводу. Такой синергетический эффект нельзя недооценивать. Когда одна команда работает в одном пространстве, происходят чудеса.

Обратите внимание, что этот метод относится к методам взаимодействия с клиентом, а не к методам взаимодействия внутри команды. Основные выгоды от метода «одна команда» получает клиент.

Когда команда находится в одном пространстве, дело идет слаженнее.

В одном пространстве

В начале 2000-х я помогал некоторым организациям внедрить методы Agile. Во время предварительных визитов, до того как начинать активный коучинг, мы попросили наших клиентов подготовить пространство и расположить в нем всех членов команды. Заказчик неоднократно сообщал, что эффективность команд за-

метно возросла просто потому, что ее члены находились в одном пространстве.

Размещение другим способом

В 1990-х интернет открыл широкие возможности использования труда программистов в странах с очень низкой стоимостью рабочей силы. Искушение воспользоваться этой возможностью было бешеным. Бухгалтеры делали расчеты и с горящими глазами представляли, сколько средств можно было сэкономить.

Но мечта мечтой, а действительность опустила всех с небес на землю. Оказалось, что рассылать мегабиты исходного кода по всему миру не совсем то же самое, что расположить в одном пространстве команду из клиентов и программистов. Была огромная разница как в расстоянии, так и в часовых поясах, языке и культуре.

Несогласованность зашкаливала. Качество оставляло желать лучшего. Нужно было много переделывать¹. В последующие годы технологии в какой-то мере стали совершеннее. В наши дни скорость передачи данных позволяет регулярно связываться по видео и передавать изображение на экране. Два разработчика в разных концах света теперь могут работать в паре над тем же кодом почти так же, как если бы сидели за одним столом. Конечно, такой прогресс не решает проблему разных часовых поясов, культурных и языковых различий, но электронное написание кода лицом к лицу несомненно предпочтительнее, чем пересылка исходного кода туда-обратно по электронной почте.

¹ Это мои личные впечатления, основанные на разговорах с ребятами, которые напрямую сталкивались с подобными проблемами. Сейчас у меня нет актуальных данных. Действуйте на свой страх и риск.

Можно ли так работать по методам Agile? Я слышал, что можно. Но сам никогда не видел, чтобы это хорошо удавалось. Может быть, вы видели.

Удаленная работа из дома

Повышение пропускной способности интернет-соединения существенно облегчило людям возможность работы из дома. В этом случае различия в языке, культуре и часовом поясе не составляют существенной проблемы. Кроме того, не нужно передавать данные через океаны, нет сбоев. Совещания команды могут проходить почти так же, как если бы ее члены находились в одном помещении.

Не поймите меня неправильно. Когда члены команды работают из дома, есть значительная нехватка невербального общения. Разговоры, приводящие к случайным открытиям, происходят гораздо реже. Неважно, насколько хороша связь посредством электроники, члены команды все равно физически не в одном пространстве. Поэтому люди, работающие из дома, находятся в невыгодном положении. Они всегда пропускают какие-нибудь разговоры или импровизированные встречи. Несмотря на широкополосный доступ, они будто смотрят через глазок по сравнению с теми, кто находится рядом друг с другом.

Если в своем большинстве команда находится в одном пространстве, но один-два члена команды пару дней в неделю работают из дома, скорее всего, никаких трудностей не возникнет, особенно если используются средства связи с хорошей пропускной способностью. С другой стороны, если члены команды почти все работают из дома, такая команда никогда не сработается так же хорошо, как если бы была вместе.

Не поймите превратно. В начале 1990-х мы с моим партнером Джимом Ньюкирком благополучно управляли командой, все члены которой находились на удаленке. Почти все работали только из дома. Некоторые жили в других часовых поясах. Мы лично встречались от силы пару раз в год. С другой стороны, мы все говорили на одном языке, у нас был один менталитет, а разница во времени не превышала двух часов. У нас получалось работать. И мы работали. Весьма хорошо. Но мы бы работали лучше, если бы находились в одной комнате.

ЗАКЛЮЧЕНИЕ

На встрече в Сноуберде в 2000 году Кент Бек выразил мысль, что одна из наших задач — построить мост над пропастью, существующей между клиентами и разработчиками. Методы взаимодействия с клиентами играют важную роль при выполнении этой задачи. Если применять этот метод, то у разработчиков и клиентов будет простой и однозначный способ общения. Такое общение порождает доверие.

МЕТОДЫ ВЗАИМОДЕЙСТВИЯ ВНУТРИ КОМАНДЫ



Средняя полоса модели жизненного цикла Рона Джейффриса состоит из методов взаимодействия внутри команды. Эти методы регулируют отношения членов команды и их отношение к создаваемому продукту. Методы, которые мы обсудим, — это *метафора, 40-часовая рабочая неделя, коллективное владение и непрерывная интеграция*.

А потом кратко обсудим *стендап-митинги*.

Метафора

В годы накануне и после подписания Манифеста Agile метафора была методом расплывчатым, нам было стыдно, что не могли дать ей нормального описания. Мы знали, что это важно, поэтому могли привести удачные примеры. Но четко выразить то, что имели в виду, у нас не получалось. В некоторых наших беседах, на курсах и лекциях мы просто вскакивали и, выпучив глаза, восклицали: «Да вы сами все поймете, когда увидите!»

Для плодотворного общения команде требуется ясный и упорядоченный словарный запас из понятий и концепций. Кент Бек предложил понятие «метафора», так как это связывало его проекты с чем-то, о чем у всех команд было общее представление.

Первым примером Бека была метафора, которая использовалась в проекте расчета заработной платы «Крайслер»¹. Он сравнил оформление зарплатных чеков с конвейером. Чеки движутся от одной точки к другой, где к ним присоединяют разные «запчасти».

¹ https://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System.

Пустой чек перемещается на точку, где на нем ставят идентификационный номер сотрудника. Потом он попадает на точку, где начисляется зарплата до вычета налогов. Затем он добирается до точки, где вычитываются налоги, затем до точки, где вычитываются расходы на медицинское страхование, затем до точки, где идут отчисления в пенсионный фонд... Ну вы поняли.

Программисты и клиенты могут довольно просто применить такую метафору к подготовке зарплатного чека. Она дала им словарный запас, чтобы говорить о программах.

Но метафоры часто заводят не туда.

Например, в конце 1980-х я работал над проектом, в котором измерялось качество передачи данных в сетях Т1. Мы загрузили данные счетчиков ошибок с конечных точек каждой линии связи. Эти данные были объединены в слои по полчаса. Мы рассматривали эти слои как ломтики чего-то сырого, из чего можно приготовить еду. Когда мы нарезаем хлеб ломтями, где мы жарим ломтики? В тостере! И тут мы метафорически стали описывать хлеб.

У нас были ломтики, батоны, сухари и так далее.

У программистов такой словарный запас работал неплохо. Мы отлично понимали друг друга, когда разговаривали о сырых и приготовленных ломтиках, батонах и так далее. С другой стороны, менеджеры и клиенты, которые слышали наши разговоры, крутили пальцем у виска и выходили из комнаты. Им казалось, что мы несем чушь.

А есть пример еще хуже. В начале 1970-х я работал над системой разделения времени. Она перемещала приложения в память, которая была ограничена, и выгружала их из нее. За время, за которое

приложение находилось в памяти, она загружала в буфер текст и отправляла его на медленный телетайп. Когда буфер заполнялся, приложение уходило в спящий режим и выгружалось из памяти на диск, в то время как буфер постепенно очищался. Мы называли такие буфера мусоровозами, которые ездили между мусорными баками и свалкой.

Мы думали, что это гениально. Мы не могли сдержать смешков, когда разговаривали о метафорическом мусоре. По сути, мы говорили, что наши клиенты — торговцы мусором. Такая метафора была удобна для общения между нами, но она выражала неуважение к тем, кто нам платил. Они никогда не узнали об этой метафоре.

Эти примеры показывают как преимущества, так и недостатки метафоры. Метафора формирует словарь, который позволяет нам успешно общаться внутри команды. С другой стороны, некоторые метафоры глупо звучат и являются оскорбительными по отношению к клиенту.

Предметно-ориентированное проектирование

В своей прогрессивной книге Domain-Driven Design¹ Эрик Эванс решил нашу проблему с метафорами, и наконец мы избавились от чувства стыда. В этой книге он ввел понятие повсеместного языка — как раз так стоило назвать метод, который получил название «метафора». Команде была нужна именно модель предметной области, которую описывают теми словами, с которыми согласны все.

¹ Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston, Massachusetts: Addison-Wesley, 2003.

Под «всеми» я имею в виду всех: программистов, QA-специалистов, менеджеров, клиентов, пользователей и так далее.

В 1970-х Том Демарко назвал такие модели «словарями данных»¹. Они были простыми представлениями данных, которыми манипулирует приложение, и процессов, которые манипулировали этими данными. Эванс значительно развил этот простой замысел в дисциплину моделирования предметной области. И Демарко, и Эванс использовали эти модели как транспортные средства для общения с партнерами.

Как простой пример: я недавно написал видеоигру Space War. Элементы данных носили названия «корабль», «клингон», «ромуланин», «выстрел», «удар», «взрыв», «база», «транспорт» и прочее. Я внимательно относился к тому, чтобы изолировать эти понятия в их собственных модулях и использовать эти названия исключительно в приложении. Это был мой «повсеместный язык».

Такой язык используется во всех частях проекта. На нем говорят клиенты. Разработчики говорят на нем. И QA-специалисты. Специалисты по DevOps тоже на нем говорят. Даже клиенты берут на вооружение те части, которые будут им полезны. Повсеместный язык применяется в бизнес-моделях, требованиях, архитектуре и приемочном тестировании. Он прочной нитью последовательно объединяет составляющие проекта на каждом этапе его жизненного цикла².

¹ DeMarco T. Structured Analysis and System Specification. Upper Saddle River, New Jersey: Yourdon Press, 1979.

² «Это энергия, создаваемая всеми живыми существами. Она снаружи и внутри нас. Она связывает всю Галактику». Лукас Дж. Звездные войны. Эпизод IV: Новая надежда. 1979. Кинокомпания «Лукасфильм».

40-ЧАСОВАЯ РАБОЧАЯ НЕДЕЛЯ

Не быстрый побеждает в беге...

Екклесиаст 9: 11

Претерпевший же до конца спасется.

От Матфея 24: 13

На седьмой день Бог решил взять отдых. А позже Бог велел в этот день отдохнуть всем. Видимо, даже Богу нужен метод «40-часовая рабочая неделя», или равномерная работа.

В начале 1970-х, когда я был восемнадцатилетним, меня и моих школьных приятелей взяли работать джуниор-программистами для работы над *краине важным проектом*. Менеджеры установили сроки. Сроки были *жесткими*. Требовалось выкладываться *по полной*. Мы были *незаменимыми* винтиками в сложном механизме компании. *Мы были важны!*

Хорошо, когда тебе восемнадцать, не правда ли?

Мы, молодые и горячие, только окончившие школу, работали как волы. Мы работали долгими часами месяц за месяцем. В среднем мы работали по 60 часов в неделю. Были недели, когда мы работали даже по 80 часов. Десятки раз мы работали по ночам.

И мы гордились тем, что работали сверхурочно. Вот мы-то были настоящими программистами. Мы посвятили себя проекту. Нас ценили. Потому что мы были единственной силой, которая могла спасти такой важный проект. Мы. Были. Программистами.

А потом мы сгорели, причем жестко. Так жестко, что ушли всем скопом. Мы вылетели оттуда, оставив компании еле работающую

систему разделения времени, при этом в компании не было только толковых программистов, которые могли бы ее сопровождать. Вот так им!

Хорошо, когда тебе восемнадцать и ты в ярости, да?

Не беспокойтесь, компания выкарабкалась. Оказалось, что там все же были толковые программисты помимо нас. Ребята, которые спокойно себе работали 40 часов в неделю. Ребята, которых мы представляли безразличными к работе и ленивыми, над которыми мы во время сумасшедшихочных марафонов презрительно смеялись, пока они не видели. Эти ребята без лишней суэты взяли систему в свои руки, обеспечив вполне годное сопровождение. Не побоюсь сказать, они были рады избавиться от кучки шумных и надоедливых сопляков.

Работа сверхурочно

Как думаете, я усвоил урок из того, что вам только что рассказал? Нет, конечно. В течение последующих 20 лет я точно так же горел на работе ради своих работодателей. Я продолжал прельщаться байками о том, что проект чрезвычайно важен. Я, конечно, не сходил с ума, работая сутками, как в 18 лет. В среднем я работал уже где-то по 50 часов в неделю. Ночные посиделки стали происходить реже, но никуда не исчезли.

Когда я вырос, то понял, что самые худшие технические ошибки я совершил вочные часы, когда из меня ключом била маниакальная энергия. Я осознал, что эти ошибки были огромными помехами, которые мне постоянно приходилось исправлять в часы, когда я по-настоящему был работоспособен.

Затем случилось кое-что, что заставило меня задуматься. Я и мой будущий деловой партнер Джим Ньюкирк занимались ночным

бдением. Где-то около двух часов ночи мы пытались выяснить, как переместить единицу данных из низкоуровневой части нашей программы в другую часть, которая находилась намного выше в цепочке выполнения. Решение возвращать эти данные в стек не подходило.

Мы создали систему транспортировки внутри нашего продукта по типу почты. Мы применяли ее для пересылки информации между процессами. В наших жилах тек кофеин, мы были на пределе возможностей. Внезапно пришло осознание, что можно было сделать так, чтобы низкоуровневая часть процесса отсыпала единицу данных самой себе, где высокоуровневая часть могла бы ее считать.

Даже сейчас, спустя более трех десятилетий, каждый раз, когда мы с Джимом хотим описать чье-то неудачное решение, мы говорим: «Да уж. Они просто отослали это самим себе».

Я не буду грузить вас скучными подробностями, почему это решение было дурацким. Достаточно сказать, что на него ушло намного больше усилий, чем мы, какказалось, сберегли. И, конечно же, это решение привело к слишком глубоким и труднообратимым изменениям, поэтому мы потеряли много времени¹.

Марафон

И в то время я понял, что проект по разработке программного обеспечения — это марафон, а не спринт или серия спринтов. Чтобы победить, надо рассчитывать силы. Если вы высакиваете из стар-

¹ Это произошло за десятилетие до того, как я узнал о разработке через тестирование. Знай мы с Джимом этот метод в то время, мы могли бы легко откатить это изменение.

товых блоков и летите на полной скорости, у вас закончатся силы прежде, чем вы пересечете финишную черту.

Таким образом, вы должны работать в темпе, который можно поддерживать в течение длительного времени. Должна быть 40-часовая рабочая неделя. Если пытаться бежать в более быстром темпе, чем вы можете поддерживать, все равно придется замедляться и отдыхать до пересечения финишной черты. Средняя скорость будет ниже, чем при умеренном темпе. Когда финишная черта близко, еще остается немного сил на то, чтобы совершить рывок. Но нет необходимости делать его раньше времени.

Менеджеры могут просить вас поторопиться. Ни в коем случае не поддавайтесь. Ваша работа — грамотно распоряжаться своими ресурсами, чтобы выстоять до конца.

Самоотдача

Работа сверхурочно — плохой способ выразить самоотверженность перед работодателем. Это лишь показывает то, что вы плохо умеете планировать, что соглашаетесь со сроками, на которые не нужно соглашаться, что даете обещания, но не в состоянии их сдержать, что вы управляемый чернорабочий, а не профессионал.

Это не означает, что работать сверхурочно неправильно или что никогда не нужно этого делать. Есть смягчающие обстоятельства, при которых работа сверхурочно — единственный выход. Но это должно быть исключением. И вы должны прекрасно понимать, что работа сверхурочно может в итоге привести к большим временным затратам.

Та ночная посиделка с Джимом несколько десятилетий назад была не последней. Она была предпоследней. Последний раз я сидел всю ночь, когда меня вынудили обстоятельства.

Шел 1995 год. На следующий день была запланирована к выходу из печати моя первая книга, и мне кровь из носу нужно было представить корректуру. Было 6 утра, у меня уже все было на руках. Надо было просто залить файл на FTP-сервер издателя.

Но потом я волей случая наткнулся на способ вдвое увеличить разрешение сотен графиков в книге. Джим и Дженинфер помогали мне с подготовкой корректуры, и мы уже были готовы запустить FTP-клиент. И тут я показываю им пример графика с увеличенным разрешением.

Мы посмотрели друг на друга и тяжело вздохнули. Джим сказал: «Нужно все переделывать». Это был не вопрос. Это была констатация факта. Все трое посмотрели друг на друга, потом на часы. Потом снова друг на друга. А потом пошли горбатиться.

Но когда мы все сделали, посиделки закончились. Мы отправили книгу. И пошли спать.

Сон

В жизни программиста достаточный сон ценится на вес золота. Мне хватает семи часов. Могу выдержать день-два, если спать по шесть часов. Если сна немного меньше, то производительность резко падает. Определите, сколько часов вам нужно, чтобы физически выспаться, и выставьте этому времени наибольший приоритет. Поспать в эти часы оправданно более чем полностью. По своему опыту могу сказать, что один час недосыпа стоит мне двух часов рабочего времени днем. Если не поспал еще час, то смело вычеркивайте еще четыре часа полезной работы. И, разумеется, если недосып составит три часа, то можно забыть о какой-либо плодотворной работе вообще.

КОЛЛЕКТИВНОЕ ВЛАДЕНИЕ

В проекте, где применяется Agile, у кода нет владельца. Код — это коллективное владение команды как единого целого. Любой член команды в любое время имеет право проверить и усовершенствовать любой модуль проекта. Команда *коллективно* располагает всем кодом.

Я усвоил этот метод еще в начале своей карьеры, когда работал в Teradyne. Мы работали над большой системой, состоявшей из пятидесяти тысяч строк кода, разделенного на несколько сотен модулей. Но никто из команды не владел ни одним из этих модулей. Мы все стремились учиться и совершенствовать эти модули. Да, некоторые из нас лучше знали некоторые части кода, нежели остальные, но мы старались поделиться знаниями, а не копить их в себе.

Эта система стала одной из первейших распределенных сетей. Она состояла из центрального компьютера, который сообщался с несколькими десятками периферийных по всей стране. Компьютеры соединялись по модему с пропускной способностью более 300 бод. Программисты не делились на тех, кто работал над ПО для центрального, а кто — для периферийных компьютеров. Все работали над программным обеспечением для тех и других.

У центрального компьютера и периферийных были совершенно разные архитектуры. Один был такой же, как PDP-8, только его слово составляло 18 бит. У него было 256 килобайт оперативной памяти, а данные загружались с кассеты с магнитной лентой. Другой имел 8-битный микропроцессор 8085 с 32 килобайтами оперативной памяти и 32 килобайтами постоянной памяти.

Мы писали программы на ассемблере. Ассемблер для каждой из двух машин значительно отличался, среды разработки также были

очень разными. Мы все работали над обеими машинами в равных условиях.

Коллективное владение не означает того, что у вас не может быть специализации. Поскольку системы становятся все сложнее, от специализации никуда не уйти. Существуют системы, которые просто нельзя понять как полностью, так и в мелочах.

Однако даже если вы специализируетесь в чем-то, нужно уметь обобщать. Разделите работу между вашей специализацией и другими областями кода. Поддерживайте умение работать не только в рамках специализации.

Когда в команде применяется метод коллективного владения, знания распространяются среди всех участников команды. Все члены команды начинают лучше понимать границы между модулями и принципы работы системы в целом. Это существенно повышает способность команды взаимодействовать и принимать решения.

За свой достаточно долгий карьерный путь я видел, как некоторые компании делали наоборот. У каждого программиста были свои модули, и никто больше не имел права к ним прикасаться. Такие команды были чрезвычайно разложены, в них царило непонимание, и все сваливали вину друг на друга. Работа над модулем прекращалась, когда автор был не в офисе. Никто даже и не думал браться за чужой участок работы.

Секретные материалы

Компания X, производившая принтеры высокого класса, была одним из таких печальных случаев. В 1990-х компания переходила от преимущественно аппаратных решений к интеграции аппаратных

и программных средств. Им пришло осознание, что можно значительно сократить расходы на производство, если для управления оборудованием внутри компании применять программное обеспечение.

Однако привычка производить аппаратное обеспечение укоренилась слишком глубоко, поэтому разделение труда при разработке ПО осуществили с таким же подходом. При разработке аппаратного обеспечения каждая команда занималась своим устройством: податчик, принтер, стопоукладчик, сшиватель и так далее. ПО для устройств писали те же, кто занимался их производством. Одна команда писала программы для податчиков, другая — для сшивателя, и так же и для других устройств.

В компании X политический вес работника зависел от устройства, над которым он работал. Поскольку X была компанией, выпускающей принтеры, работа над ними была самой почетной. Чтобы работать над принтерами, инженерам приходилось продвигаться по службе. С теми, кто занимался сшивателями, никак не считались.

Эта система политических рангов передалась командам, писавшим программное обеспечение. Разработчики, писавшие код для стопоукладчиков, были политически бессильны, но когда на собрании начинал говорить разработчик, работавший над принтером, все внимательно его слушали. Из-за такого политического разделения никто не делился кодом. Ключом к политическому влиянию команды, занимавшейся принтерами, был код. Поэтому код, написанный для принтеров, находился за семью замками. Никто, кроме членов команды, не мог даже на него взглянуть.

Проблем от этого было море. Во взаимодействии возникают очевидные трудности, если невозможно проверить используемый

код. В таком случае неизбежно перекладывание ответственности и подставы.

Но хуже всего было несусветное и смехотворное дублирование. Оказывается, что ПО для податчика, принтера, стопоукладчика и сшивателя не особо-то и отличается. У всех этих устройств есть моторчики, реле, соленоиды и муфты, расположенные на внешних входах и внутренних датчиках. В основном внутреннее строение этих модулей было одинаковым. А из-за всей этой борьбы за политическое влияние каждой команде приходилось изобретать свое собственное колесо.

Что еще важнее, само намерение разделить команды разработчиков программного обеспечения по устройствам противоречило здравому смыслу. Нет смысла разрабатывать ПО для контроллера податчика независимо от контроллера принтера.

Напрасная траты трудовых ресурсов, не говоря уже о страхе, враждебности и конкуренции, вели к очень нездоровой атмосфере. Я считаю, что атмосфера была причиной, по крайней мере отчасти, последующего крушения этой компании.

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ

В первые дни существования Agile метод «непрерывная интеграция» означал, что разработчики отмечали изменения в своем исходном коде и объединяли их с основной веткой каждые «пару часов»¹. Все модульные и приемочные тесты проходили успешно. Не оставалось веток, которые бы не были объединены. Все не-

¹ Beck K. Extreme Programming Explained: Embrace Change. Boston, Massachusetts: Addison-Wesley, 2000. P. 97.

нужные при развертывании изменения отключались с помощью тумблера.

В 2000 году на одном из уроков на курсах XP Immersion один студент угодил в классическую ловушку. Эти уроки были очень насыщены. Мы сократили циклы так, что итерации составляли всего один день. Цикл непрерывной интеграции составлял лишь от четверти до получаса.

Один студент работал в команде из шести разработчиков, пятеро из которых отмечали изменения чаще, чем он. Непонятно почему, он был без пары. И вышло так, что он не выполнял слияние кода более часа.

Когда он наконец решил отметить изменения и выполнить слияние кода, он увидел, что накопилось столько изменений, внесенных другими студентами, что ему пришлось хорошенко по-возиться, чтобы объединить код. Пока он копался и выполнял слияние, остальные программисты продолжали отмечать изменения каждые 15 минут. И когда он наконец разобрался и попробовал отметить изменения в своем коде, то увидел, что снова попал впросак.

Он так расстроился, что встал посреди кабинета и громко произнес: «Ваше экстремальное программирование — полная ерунда!» После этого он вылетел из класса и направился в бар гостиницы.

И тут случилось невероятное. Программист, с которым он отказался работать в паре, пошел вслед, чтобы побеседовать с ним. Две пары других программистов перераспределили свою работу, завершили слияние и вернули проект в колею. Через полчаса студент притих, вернулся в кабинет и, извинившись, продолжил работу,

в том числе стал работать в паре. Впоследствии он стал ярым сторонником методологии гибкой разработки Agile.

Мораль: метод «непрерывная интеграция» работает только тогда, когда интеграция действительно непрерывна.

А вот и непрерывная сборка

В 2001-м Thought Works значительно изменили положение дел. Они создали CruiseControl¹, первое средство непрерывной сборки. Я помню, как в 2001-м на XP Immersion Майк Ту² читал ночную лекцию об этом. Не сохранилось записи этой лекции, но история была примерно таковой:

CruiseControl позволяет сократить время отметки изменений до нескольких минут. Даже самые мелкие изменения быстро вносятся в основную линию. CruiseControl отслеживает работу систем управления версиями и создает сборку каждый раз, когда отмечено какое-либо изменение. При создании сборки CruiseControl запускает большинство автоматизированных тестов, а затем отправляет электронные письма с итогами всем членам команды.

«Боб сломал сборку».

Мы ввели простое правило в отношении тех, кто ломал сборку. В тот несчастный день, когда вас угораздило сломать сборку, вы должны надеть футболку с надписью «Я сломал сборку». Которую никто никогда не стирал.

¹ <https://ru.wikipedia.org/wiki/CruiseControl>.

² <http://wiki.c2.com/?MikeTwo>.

С тех пор было создано много других средств непрерывной сборки. К ним относятся инструменты наподобие Jenkins (или Hudson?), Bamboo и TeamCity. С помощью этих инструментов можно в наибольшей степени сократить время между слияниями. «Пара часов», о которой изначально говорил Кент Бек, превратилась в «несколько минут». «Непрерывная интеграция» стала «непрерывной отметкой».

Дисциплина при непрерывной сборке

При непрерывной сборке ничего *не должно* ломаться. Потому что программист, который не хочет надевать грязную футболку, как в случае с Майком Ту, проводит приемочное и модульное тестирование до того, как отметить изменения в коде. Если сборка сломалась (как так можно?), значит, случилось что-то очень странное.

Майк Ту рассматривал в своей лекции и этот вопрос. Он рассказал о календаре, висевшем на видном месте на стене помещения, где работала команда. Календарь выбирали большой, чтобы каждый день в году располагался в своей ячейке.

Если сборка не удавалась хоть раз, то этот день отмечали красной точкой. Если сборка проходила успешно, этот день отмечали зеленой точкой. Такой простой визуализации было достаточно, чтобы в течение месяца или двух превратить календарь, состоящий в основном из красных точек, в календарь, состоящий в основном из зеленых.

Внимание!

Напомню: при непрерывной сборке ничего не должно ломаться. Сломанная сборка — это событие, означающее, что нужно максимальное внимание. Я хочу, чтобы заорали сирены. Я хочу видеть

мерцание красного прожектора в кабинете исполнительного директора. Сломанная сборка — это полный трэйндец. Я хочу, чтобы все программисты бросили свои дела и сплотились вокруг сборки, и та снова прошла успешно. Фраза «при сборке ничего не ломается» должна стоять на повторе в голове каждого члена команды.

Стоимость обмана

Бывали случаи, когда команда, игнорируя неисправности, под давлением дедлайна продолжала непрерывную сборку. Это попытка самоубийства. В результате все устают от шквала писем о нарушениях, приходящих от сервера непрерывной сборки. Поэтому разработчики просто убирают тесты, которые не получается пройти, обещая вернуться к проблемам позже и решить их.

Кто бы сомневался: теперь сервер непрерывной сборки снова сообщает, что сборка прошла успешно. Все расслабляются. Сборка проходит тесты. И все благополучно забывают о куче непройденных тестов, которые убрали в сторонку, пообещав решить проблемы «позже». В итоге происходит развертывание сломанной системы.

СТЕНДАП-МИТИНГ

На протяжении многих лет было много путаницы в понятиях *ежедневный скрам* и *стендап-митинг*. Позвольте мне разъяснить.

Вот вам правда о стендап-митингах:

- Такие встречи не обязательны. Многие команды прекрасно обходятся без них.
- Они могут проводиться реже, чем раз в день. Подберите график, который считаете подходящим.

- Они должны занимать примерно 10 минут даже у больших команд.
- Встреча проводится по простому сценарию.

Смысль этого мероприятия в том, что каждый член команды встает¹ в круг и отвечает на три вопроса:

1. Что я делал после прошлой встречи?
2. Что я буду делать до следующей встречи?
3. Что мне нужно сделать?

И все. Никаких обсуждений. Никакого позерства. Никаких пристранных объяснений. Никаких грустей и печалей. Никаких жалоб и обвинений кого угодно на свете.

У каждого есть полминуты на то, чтобы ответить на три вопроса. Потом встреча заканчивается, и все идут работать дальше. Все, аллес. Финита. Ферштейн?

Наверное, еще лучше стендап-митинги описаны на «вики» Уорда:
<http://wiki.c2.com/?StandUp Meeting>.

Курицы и свиньи?

Если готовить омлет с ветчиной, то степень участия в нем двух животных будет разной. Курица для омлета просто снесет яйцо, а вот свинье придется пожертвовать собой для ветчины полностью. Суть в том, что только разработчикам разрешается говорить на стендап-митинге. Руководство и иже с ними могут послушать, но никак не вмешиваться.

¹ Потому они и называются стендап-митинги, то есть «стоя».

Как по мне, без разницы, кто говорит, главное, чтобы встреча проводилась в формате трех вопросов, а собрание длилось около 10 минут.

Красавчик

Мне понравилось одно изменение — это добавить дополнительный четвертый вопрос.

- Кто у нас сегодня красавчик?

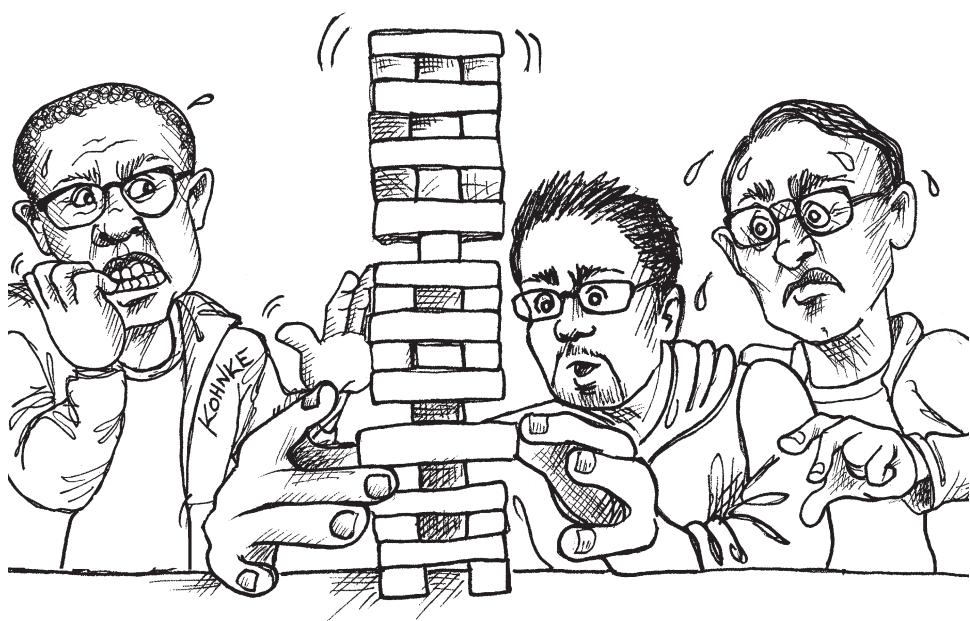
Это быстрое выражение признательности за помочь вам или за то, что этот человек, по вашему мнению, заслуживает похвалы за свой вклад.

ЗАКЛЮЧЕНИЕ

Agile — это набор принципов, методов и дисциплин, которые помогают небольшим командам разработчиков выполнять небольшие проекты. В этой главе приведены методы, которые помогают небольшим коллективам взаимодействовать как настоящие команды. Они помогают командам найти общий язык для взаимодействия, а также дают понимание того, чего ожидать членам команды от других в отношении друг друга и выполняемого проекта.

5

ТЕХНИЧЕСКИЕ МЕТОДЫ



Методы этой главы предлагают полный отход от тех, что господствовали среди большинства программистов на протяжении 1970-х годов. Раньше предлагался набор осмысленных ритуалов, проводимых периодично — ежеминутно и ежесекундно, которые большинство программистов изначально считают чушью. Поэтому многие программисты пробовали применять Agile, исключив эти методы. Однако у них ничего не получилось, потому что эти методы как раз лежат в основе Agile. Без разработки через тестирование, рефакторинга, простой структуры проекта и, да-да, даже парного программирования Agile становится бесполезным жалким подобием того, чем должен быть.

РАЗРАБОТКА ЧЕРЕЗ ТЕСТИРОВАНИЕ

Разработка через тестирование — это сложная и глубокая тема для обсуждения, и чтобы ее как следует раскрыть, понадобится целая книга. В этой главе приведен лишь общий обзор, который больше сосредоточен на обосновании и объяснении, чем на глубоких технических сторонах метода. В частности, в этой главе вы не найдете никакого кода.

Профессия программиста уникальна. Мы создаем огромные документы, написанные загадочными техническими символами. Каждый символ такого документа должен быть правильным, иначе может произойти что-то действительно страшное. Один неверный символ повлечет за собой потери больших средств или жизни. Разве есть другие такие профессии?

Бухгалтерский учет. Бухгалтеры также создают огромные документы, исписанные загадочными техническими символами. Каждый символ такого документа должен быть правильным, в противном случае это может стоить потери больших средств

или жизни. Как бухгалтеры проверяют, чтобы каждый символ был правильным?

Двойная запись

У бухгалтеров есть инструмент, который появился еще тысячу лет назад. Это двойная запись¹. Каждая операция, которую они проводят, заносится в книгу дважды: один раз как кредит в активном счете и еще раз, в дополнение, как дебет в пассивном счете. Затем эти счета сводят в единый документ, балансовый отчет, где из суммы финансовых активов вычитается сумма долговых обязательств и долей собственников. Бухгалтер должен выйти в ноль. Если в итоге получается не ноль, значит, где-то допущена ошибка².

Бухгалтеров учат вносить операции по одной за раз и каждый раз формировать баланс после проведения операции. Такой способ позволяет им быстро выявлять ошибки. Они не делают проводок между сверкой баланса, поскольку так будет тяжелее найти ошибки. Этот метод, который называется методом двойной записи, настолько важен для правильного учета денежных средств, что стал мировым стандартом.

Разработка через тестирование по сути то же самое, только для программистов. Каждое необходимое изменение проходит двойную проверку. Сначала под такое изменение пишут тест, затем

¹ https://ru.wikipedia.org/wiki/Двойная_запись.

² Если вы изучали бухгалтерское дело, то, скорее всего, кипите от возмущения. Да, это было грубым упрощением. С другой стороны, я описал разработку через тестирование упрощенно, одним абзацем. Программисты, наверное, тоже возмутились.

пишут готовый код, который позволяет пройти этот тест. Обе проверки дополняют друг друга, как активные и пассивные счета в бухгалтерском учете. После прохождения двойной проверки получается нулевой результат — ноль тестов провалено.

Программисты, изучающие разработку через тестирование, учатся вносить запись дважды — когда код не проходит специально написанный тест и еще раз, когда готовый код проходит тест. Такой способ позволяет им быстро выявлять ошибки. Их учат избегать написания большого количества готового кода и запуска партии тестов, поскольку так тяжело найти ошибки.

Эти две дисциплины, двойная запись в бухгалтерском учете и разработка через тестирование, схожи.

И то и другое служит одной цели — предотвращать ошибки в критически важных документах, где каждый символ должен быть правильным. Хотя программирование и стало неотъемлемой частью нашего общества, мы до сих пор не обязали проводить разработку через тестирование на законодательном уровне. Но учитывая то, скольких жизней и средств стоило плохо написанное ПО, может, такой закон не за горами?

Три правила разработки через тестирование

Разработку через тестирование можно описать тремя простыми правилами.

- Не пишите готовый код до того, как напишете тест, который не получится пройти из-за нехватки этого кода.

- Не пишите тестов больше, чем это необходимо для неудачи, — сбой при компиляции также считается неудачей.
- Не пишите готового кода больше, чем достаточно для прохождения теста, который был провален до этого.

Программист с опытом чуть больше нескольких месяцев наверняка посчитает эти правила странноватыми, если не сказать прямо, глупыми. Эти правила подразумевают, что цикл программирования длится пять секунд. Программист начинает с написания тестового кода для готового кода, который пока что не написал. Тест не удается скомпилировать почти сразу, потому что есть упоминание частей готового кода, которые еще не существуют. Программист должен перестать писать тест и начать писать готовый код. Но после нескольких нажатий на клавиши тест, который не получалось скомпилировать, теперь компилируется должным образом. Это заставляет программиста вернуться к тесту и дописывать к нему новый код.

Такое колебание между тестовым и готовым кодом составляет всего несколько секунд, и программист ограничен в действиях в рамках этого цикла. Программист никак не сможет написать целую функцию, или даже простое выражение с оператором `if`, или цикл с `while`, не прерывая себя написанием дополняющего тестового кода.

Большинство программистов изначально рассматривают такой подход как нарушение их мыслительного процесса. Это постоянное прерывание, вызванное нашими тремя правилами, не позволяет им должным образом думать во время написания кода. Им мешает ощущение, что три правила разработки через тестирование заставляют отвлекаться до невозможности часто.

Однако представьте группу программистов, которые соблюдают эти три правила. Посмотрите на любого из них в любой момент

времени. Все, над чем программист работал, запустилось или прошло соответствующие тесты меньше минуты назад. Неважно, на кого и когда вы посмотрите, у всех все работало меньше минуты назад.

Отладка

Что было бы, если бы минуту назад все *всегда* работало? Как долго пришлось бы выполнять отладку? Если минуту назад все работало, то почти каждый сбой, с которым вы столкнетесь, произошел не больше минуты назад. Отладка сбоя, который появился только что, зачастую пустяк. И в самом деле использовать отладчик для поиска проблемы, наверное, чересчур.

Вы ловко управляетесь с отладчиком? Помните его горячие клавиши? Ваша мышечная память позволяет непроизвольно нажимать эти клавиши так, чтобы вводить точки останова, входить в режимы: пошаговый, шаг с заходом в процедуры, шаг с обходом процедур? При отладке вы чувствуете себя в своей тарелке? *Это не тот на-вык, который желают получить.*

Единственный способ научиться хорошо пользоваться отладчиком — это проводить много времени за отладкой. А если за отладкой проводится много времени, значит, часто попадаются ошибки. Те, кто практикует разработку через тестирование, плохо умеют пользоваться отладчиками просто в силу их редкого использования. А если пришлось-таки воспользоваться отладчиком, это, как правило, не отнимает у них много времени.

Я хочу сказать, что не даю ложных надежд. Даже лучшие программисты, пользующиеся этим методом, натыкаются на ошибки. Это все-таки разработка, куда без них. Но частота и тяжесть таких

ошибок значительно снижается, если соблюдать три правила разработки через тестирование.

Документация

Вам хоть раз доводилось интегрировать сторонний пакет? Скорее всего, он пришел в zip-архиве, в котором были исходники, библиотеки, Java-архивы и тому подобное. Скорее всего, в архиве был и документ PDF, в котором содержались инструкции по интеграции. А в конце PDF, наверное, было уродливое приложение с примерами всего кода.

Что вы прочли первым делом, как открыли этот документ? Если вы программист, то промотали в самый низ и прочитали примеры кода, потому что в этом коде вся правда.

При соблюдении трех правил разработки через тестирование написанные вами тесты становятся примерами кода для всей программы. Если нужно узнать, как вызвать функцию API, есть тесты, которые вызывают эту функцию всевозможными способами, с учетом каждого возможного исключения. Если вы желаете узнать, как создать объект, есть тесты, которые создают нужный объект всеми способами, которыми он может быть создан.

Тесты — это один из видов документации, которая описывает тестируемую программу.

Такая документация написана на языке, который программисты отлично знают. Он совершенно недвусмыслен, настолько формален, что исполняем и не может не синхронизироваться с кодом приложения. Тесты являются отличной документацией для программистов, потому что представляют собой код.

Более того, тесты не образуют системы сами по себе. Они знать не знают друг о друге. Между ними нет никаких зависимостей. Каждый тест — это небольшой и независимый модуль кода, который описывает поведение только маленькой части всей системы.

Радость

Если вы хоть раз писали тесты после того, как работа выполнена, то согласитесь, что развлечение это так себе. Вообще не прикольно, потому что вы уже знаете, что код работает. Вы уже вручную его протестировали. Скорее всего, вы пишете эти тесты лишь потому, что вам так сказали. Чувствуете себя загруженным рутиной. Скучно.

Когда вы пишете тесты согласно трем правилам разработки через тестирование, это клево. Каждый новый тест — это вызов. Каждое успешное прохождение теста — это маленький праздник. Когда вы следите трем правилам, ваша работа представляется чередой маленьких вызовов и праздников. Нет чувства неблагодарной работы, вместо этого вы понимаете, что даете жизнь чему-то новому.

Полнота

Но вернемся к тестам, которые выполняют после того, как все уже готово. Кто-то зачем-то обязал вас написать эти тесты, причем вы уже протестировали все вручную и знаете, что все работает. Вы переходите от теста к тесту, не удивляясь тому, что программа спокойно проходит тесты.

Неизбежно вы дойдете до теста, который будет трудно написать. Его трудно написать, потому что во время написания кода вы забыли о тестируемости, а структура кода такова, что его просто так не протестируешь. Чтобы написать тест к этому коду, надо поменять

структурку кода. Придется разорвать некоторые связи, добавить несколько абстракций, а может, и перенаправить некоторые вызовы функций и аргументы. Кажется, что переделана гора работы, потому что вы и так знаете, что код работает.

У вас плотный график, и есть работа поважнее. Поэтому вы откладываете тест. Вы убеждаете себя, что в нем нет необходимости и написать его можно попозже. И вот теперь у вас пробел в тестовом наборе.

И поскольку вы оставляли пробелы в тестовом наборе, вы подозреваете, что все остальные тоже так делают. Когда вы запускаете тестовый набор и видите, что он пройден успешно, вы покрякиваете, ухмыляетесь или насмешливо отмахиваетесь, потому что знаете: прохождение тестов не означает, что программа работает.

Когда программа проходит такой набор тестов, нельзя принять решение. Все сведения, которые мы получаем от прохождения тестов,... — это то, что работает все, что тестировалось.

Неполнота тестов оставляет вас без вариантов. Но если вы будете следовать тем самым трем правилам, каждая строка кода будет написана так, что тест будет пройден. Таким образом, тестовый набор станет полным. Когда программа проходит набор тестов, можно принять решение. Решение о развертывании.

Вот это цель! Мы хотим создать набор автоматизированных тестов, который даст уверенность, что развертывание программного обеспечения безопасно.

И снова хочу сказать, я не пишу вам тут картину маслом. Соблюдение трех правил позволит создать полный тестовый набор, но и тут нельзя быть на 100 % уверенным. Есть ситуации, когда три правила

разработки через тестирование неуместны. Эти ситуации выходят за рамки этой книги, скажу лишь, что они ограничены, и есть решения, которые смягчают их. В результате даже самые прилежные приверженцы наших трех правил вряд ли смогут создать тестовый набор, который будет выполнен на 100 %.

Но в 100 %-ной полноте тестов нет необходимости для принятия решения о развертывании. Значения в 95 % вполне достаточно — и такая полнота тестов поистине достижима.

Я создавал настолько полные тестовые наборы, что они позволяли принять решение о развертывании. Я видел, как многие другие делают то же самое. В каждом из таких случаев полнота не достигала 100 %, но ее хватало, чтобы принять решение о развертывании.

ВНИМАНИЕ

Полнота тестов — это показатели для команды, а не для руководства. Менеджеры вряд ли знают, что на самом деле означают эти показатели. Менеджеры не должны ставить эти показатели своей целью. Команда должна применять их исключительно в целях сообщения сведений о стратегии тестирования.

ЕЩЕ РАЗ ВНИМАНИЕ

Не завалите сборку на основании недостаточного прохождения тестов. Если вы поступите так, программисты будут вынуждены избавиться от достаточного количества операторов подтверждения отсутствия ошибок, чтобы получить достаточно высокий процент выполнения. Полнота прохождения тестов — сложная тема, которую можно понять только в контексте глубокого знания работы кода и тестов. Не позволяйте делать ее метрикой для менеджеров.

Проектирование

Помните ту функцию, которую было трудно протестировать, а код уже до этого тестировали вручную? Может, ее трудно протестировать потому, что она связана с задачами, которые вы не хотите запускать в teste? Например, это может быть включение рентгеновского аппарата или удаление рядов из базы данных. Функцию тяжело протестировать, потому что она спроектирована не так, чтобы это можно было сделать легко. Вы сначала пишете код, а потом пишете тесты задним числом. Тестируемость конструкции была тем, о чем вы думали в последнюю очередь, когда писали код.

Теперь вы столкнулись с тем, что для прохождения теста нужно перепроектировать код. Вы смотрите на часы и понимаете, что тестирование длится слишком долго. Поскольку вы уже тестировали все вручную и знаете, что все работает, то просто уходите прочь, оставляя еще один пробел в тестовом наборе. А вот когда вы сначала пишете тест, то все происходит совершенно иначе. У вас просто не получится написать функцию, которую трудно протестировать. Поскольку вы сначала пишете тест, то естественным образом проектируете функцию так, чтобы ее легко было протестировать. Как писать функции так, чтобы они были легко тестируемы? Нужно убрать связи между ними. Тестируемость означает свободу от связей.

Когда вы сначала пишете тесты, то размыкаете связи внутри программы таким способом, о котором бы никогда даже не подумали. Программа будет тестируема целиком, то есть связи между функциями всей программы будут отсутствовать.

Именно по этой причине разработку через тестирование часто называют методом проектирования. Три правила в высшей мере способствуют уменьшению связанности.

Смелость

Пока мы видели, что если следовать трем правилам разработки через тестирование, то у нас появляется много преимуществ: меньше отладки, качественная низкоуровневая документация, радость и разделение связей. Но это лишь сопутствующие преимущества, ни одно из них не является главной причиной применения разработки через тестирование. Настоящая причина — это воспитание смелости.

Я уже рассказывал историю в самом начале книги, но стоит ее повторить.

Представьте, что вы смотрите на некий уже написанный код на экране. Там бардак. Первая мысль, которая приходит в голову: «Нужно почистить его». Но следующая мысль будет примерно такой: «Нет, я в это не полезу!» Вы думаете, что если влезть в код, то он перестанет работать. А если он перестанет работать, вина ваша. Поэтому вы отстраняетесь от кода подальше, оставляя его гнить и чахнуть.

В вас говорит страх. Вы боитесь кода. Боитесь что-либо делать с ним. Боитесь что-то сломать, потому что будут последствия. Так вы отказываетесь от того единственного, что может улучшить код, — от его чистки.

Если в команде каждый придерживается такого поведения, код будет портиться. Никто не возьмется его почистить. Никто не улучшит его. Каждая новая функция будет добавлена таким образом, чтобы свести на нет непосредственный риск для программистов. Будут добавлены связи и дубликаты, потому что они уменьшают непосредственный риск, даже если нарушают структуру и целостность кода.

В конце концов код становится чудовищно запутанным, как спагетти, его невозможно сопровождать, работа над таким кодом едва ли будет продвигаться. Сложность задач будет расти в геометрической прогрессии. Менеджеры в отчаянии. Они будут нанимать все больше программистов в надежде улучшить производительность, но улучшение не будет достигнуто.

Наконец, достигнув критической точки, руководство согласится на требование программистов переписать всю программу с самого начала. И начинается то же самое.

Представьте себе другой сценарий. Вернемся к монитору, на котором мы видим запутанный код. Первая мысль, которая вас посещает, — надо почистить код. Что, если бы у вас был настолько полный тестовый набор, что ему можно полностью доверять? А если этот тестовый набор работал бы быстро? Что бы вы подумали следующим делом? Наверное, что-то вроде этого:

Боже, думаю, просто надо поменять имя этой переменной. О, код все еще проходит тесты. Ладно, а теперь я разделяю большую функцию на две поменьше... Здорово, все еще удается пройти... Хорошо, теперь, думаю, можно перенести одну из этих новых функций в другой класс. Опа! Тест не пройден. Так, ну-ка, вернем все... А, я понял, надо было переместить и саму переменную. Да, тест снова пройден...

Когда у вас есть полный набор тестов, вы больше не боитесь вносить изменения в код. Вы больше не боитесь его чистить. Вы просто возьмете и почистите код. Код будет опрятным и чистым. Структура программы останется неизменной. Вы не будете плодить массу гниющего спагетти, которая вгонит команду в уныние, приводящее к низкой производительности и, в конце концов, к провалу.

Поэтому мы применяем разработку через тестирование. Мы применяем этот метод, потому что он вселяет в нас смелость поддерживать код в чистоте и порядке. Смелость вести себя профессионально.

Рефакторинг

Рефакторинг — еще одна тема, достойная целой книги. К счастью, ее уже написал Мартин Фаулер¹. В этой главе мы просто обсудим это тему, не углубляясь в отдельные методы. И как и прежде, в этой главе нет кода.

Рефакторинг — это метод улучшения структуры кода без изменения его поведения, определенного тестами. Другими словами, мы вносим изменения в имена, классы, функции и выражения, не проваливая никаких тестов. Мы улучшаем структуру программы без воздействия на ее выполнение.

Конечно же, эта дисциплина тесно связана с разработкой через тестирование. Чтобы без опасений перепроектировать код, нужен тестовый набор, который с высокой степенью вероятности укажет нам на то, что мы ничего не испортим.

Изменения, выполненные во время рефакторинга, разнятся от простых косметических до глубокой правки структуры. Такие изменения могут представлять собой просто изменения в названиях или сложную замену операторов switch на полиморфные отправки. Большие функции будут разбиты на те, что поменьше, с более удачными названиями. Списки аргументов будут изменены на

¹ Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston, Massachusetts: Addison-Wesley, 2019.

объекты. Классы с большим количеством методов будут разделены на множество мелких классов. Функции будут перемещены из одного класса в другой. Из классов будут выделены подклассы или внутренние классы. Зависимости будут инвертированы, а модули перемещены через границы архитектуры.

И пока все это происходит, наша программа непременно проходит тесты.

Красный/зеленый/рефакторинг

Ход рефакторинга естественным образом связан с тремя правилами разработки через тестирование приемом «красный/зеленый/рефакторинг» (рис. 5.1).



Рис. 5.1. Цикл «красный/зеленый/рефакторинг»

1. Сначала мы создаем тест, который не получается пройти.
2. Потом пишем код и проходим тест.
3. Затем подчищаем код.
4. Далее возвращаемся к шагу 1.

Написание рабочего кода и написание чистого кода — это две разные вещи. Делать одновременно то и другое необычайно сложно, так как это совершенно разная деятельность.

Довольно тяжело написать рабочий код, не говоря уже о соблюдении его чистоты. Поэтому мы сначала ориентируемся на написание рабочего кода, что бы там ни происходило в головах нашего сумрачного гения. Затем, когда все заработало, мы устранием беспорядок, который натворили.

Это дает понять, что рефакторинг кода — процесс непрерывный, и его не проводят по плану. Мы не плодим беспорядок несколько дней кряду, чтобы потом долго его подчищать. Мы лучше создадим легкий беспорядок и через минуту-две все исправим.

Слово «рефакторинг» никогда не должно появляться в графике работ. Это не тот род деятельности, который можно провести по плану. Мы не выделяем времени на рефакторинг кода. Рефакторинг — это часть нашей ежеминутной, ежечасной рутины при написании ПО.

Большой рефакторинг

Иногда требования меняются так, что вы осознаете: дизайн и архитектура программы не совсем подходят. Тогда вы решаете внести значительные изменения в структуру программы. Такие изменения вносят в цикле «красный/зеленый/рефакторинг». Мы не создаем программы специально для того, чтобы вносить изменения в структуру. Мы не выделяем времени в графике работ на такой глубокий рефакторинг кода. Маленькими порциями мы переносим код, продолжая добавлять новые функции за время обычного цикла Agile.

Такое изменение в структуру программы можно вносить несколько дней, недель или даже месяцев. Все это время программа проходит все необходимые тесты и готова к развертыванию, даже если изменение структуры не полностью завершено.

ПРОСТОТА ПРОЕКТИРОВАНИЯ

Метод «простота проектирования» — одна из целей рефакторинга. Простота проектирования — метод, предполагающий написание только необходимого кода, чтобы сохранять простоту структуры, его небольшой размер и наибольшую выразительность.

Правила простого проектирования Кента Бека.

1. Пройти все тесты.
2. Проявить намерение.
3. Удалить дубликаты.
4. Сократить количество элементов.

Номера пунктов означают порядок действий, в котором эти правила выполняются, и их приоритет.

Пункт 1 говорит сам за себя. Код должен пройти все тесты. Он должен работать.

В пункте 2 указано, что после того как код заработал, ему нужно придать выразительность. Он должен явно отражать намерения программиста. Код нужно писать так, чтобы он легко читался и содержал достаточно сведений. Как раз сейчас мы проводим косметический рефакторинг кода, в течение которого вносим много простых изменений. Нужно также разделить большие функции на мелкие, дав им более простые и понятные названия.

В пункте 3 говорится, что после того как код получился в высшей мере описательным и выразительным, мы старательно выискиваем и удаляем все дубликаты. Не нужно, чтобы в коде повторялось одно и то же. Во время такой деятельности проводить рефакторинг, как правило, сложнее. Иногда удалить дубликаты так же просто, как перенести дублирующийся код в функцию и вызвать его из разных мест. В других случаях требуются решения интереснее, например паттерны проектирования¹: *метод шаблонов, стратегия, декоратор или посетитель*.

В пункте 4 говорится о том, что как только мы удалили все дубликаты, нужно стремиться уменьшить количество структурных элементов, например классов, функций, переменных и так далее.

Цель метода «простота проектирования» — поддерживать наиболее возможную легковесность кода.

Легковесность

При проектировании программа может получиться как достаточно простой, так и необычайно сложной. Чем сложнее структура, тем больше умственная нагрузка на программиста. Эта умственная нагрузка — вес структуры программы. Чем больше вес программы, тем больше времени и усилий будет затрачено программистами на изучение и управление этой программой.

Таким же образом сложность требований также варьируется от небольших до огромных. Чем сложнее требования, тем больше времени и сил понадобится, чтобы изучить эту программу и управлять ею.

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — 448 с.: ил.

Однако обе причины не являются дополняющими. Сложные требования можно *упростить* за счет усложнения структуры программы. Часто предпочтительнее найти компромисс. Программу можно упростить, подобрав подходящую конструкцию под заданный набор функций.

Уравновешивание сложности конструкций и функций является целью «простоты проектирования». Применяя этот метод, программисты постоянно перепроектируют структуру кода, чтобы сохранялось равновесие между требованиями и достижением наиболее высокой производительности.

ПАРНОЕ ПРОГРАММИРОВАНИЕ

Метод парного программирования за годы своего существования оброс противоречиями и кривотолками. Многие отрицательно воспринимают мысль о том, что два (и больше) человека могут плодотворно работать над одной и той же задачей.

Во-первых, работа в паре не обязательна. Никого нельзя к ней принуждать. Во-вторых, работа в паре не обязательно постоянна. Существует много веских причин, почему иногда лучше писать код в одиночестве. Желательно, чтобы доля программистов, работающих в паре, в команде была 50 % или около того. Но это не так важно. Она может быть лишь 30 %, а может и все 80 %. Право выбора принадлежит членам команды.

Что такое парное программирование?

Парное программирование — это совместная работа двух программистов над одной задачей. Напарники могут работать на одной ма-

шине, с одним монитором, клавиатурой и мышью. Или они могут работать на двух машинах по сети, пока могут видеть один и тот же код и обращаться с ним. Последнее прекрасно можно осуществить с помощью систем доступа к рабочему столу. Такое ПО позволяет напарникам находиться далеко друг от друга, если у них хорошая пропускная способность и голосовая связь.

Программисты, работающие в паре, выполняют разную работу. Один будто водитель, а другой — штурман. У водителя в руках инструменты — клавиатура и мышь. Штурман же внимательно всматривается в монитор и дает советы. Другой вариант: один программист пишет тест, а второй добивается его прохождения, после чего пишет ответный тест первому программисту. Иногда этот метод называют «пинг-понг».

Но чаще всего никакого разделения нет. Программисты просто сидят за одной машиной и занимаются одним и тем же и по очереди используют клавиатуру и мышь.

Пары не назначаются. Они выбирают друг друга в зависимости от желания совместной работы над одной задачей. Менеджеры не должны вмешиваться со своими расписаниями или матрицами.

Чаще всего напарники быстро меняют партнера. Сессия парного программирования может длиться день, но чаще всего они делятся от силы час-два. Даже работа в паре в течение лишь четверти-половины часа может принести пользу.

Истории не распределяются по парам. За историю отвечают отдельные программисты, а не оба напарника. Продолжительность выполнения истории длится, как правило, гораздо дольше, чем работа с одним напарником.

В неделю каждый программист будет тратить около половины своего времени на выполнение своих собственных задач, привлекая к помощи некоторых других программистов. Оставшуюся половину времени, проводимого в паре, он потратит на помочь другим программистам с их заданиями.

Опытным программистам следует стараться как можно чаще работать в паре с младшими. Младшим программистам нужно просить помощи чаще у опытных, чем у других младших. Программисты со специализацией должны тратить значительное количество времени, работая в паре с программистами над задачами вне своей специализации. Цель состоит в том, чтобы распространять знания и обмениваться ими, а не накапливать их в одиночку.

Зачем работать в паре?

Работая в паре, мы укрепляем командный дух. Члены команды не изолируются друг от друга, а ежесекундно сотрудничают. Когда член команды не может работать, другие закрывают образовавшуюся брешь и двигаются к цели.

Работа в паре — однозначно лучший способ обмениваться знаниями в команде и избегать сокрытия информации отдельными членами. Это лучший способ организовать команду так, чтобы в ней не было незаменимых сотрудников.

Многие команды сообщали, что парное программирование сокращает количество ошибок и улучшает качество проектирования. Это верно в большинстве случаев. Как правило, во время работы над задачей лучше, если на нее смотрит не одна пара глаз. Действительно, во многих командах перешли от разбора кода к работе в парах.

Парное программирование как анализ кода

Парное программирование представляет собой вид анализа кода (Code Review), но имеет значительное преимущество. Работая в паре, программисты пишут код в соавторстве. Они видят уже написанный код и, как само собой разумеющееся, проводят его анализ с целью создания нового кода. Таким образом анализ кода — это не просто статическая проверка, которая проводится, чтобы убедиться в том, что код соответствует нормам, принятым в команде. Скорее это динамический обзор текущего состояния кода с прицелом на то, каким код должен быть в ближайшем будущем.

А каковы издержки?

Сложно измерить издержки, возникающие при парном программировании. Прямая издержка — это то, что над одной задачей работает два человека. Очевидно, что на решение задачи не затрачивается двойного усилия, но, вероятно, какие-то издержки все же есть. В разных исследованиях установлено, что издержки составляют примерно 15 %. Другими словами, потребуется 115 программистов, работающих в паре, чтобы выполнить работу 100 программистов, работающих индивидуально (без анализа кода).

Если считать упрощенно, получается, что в команде, где в паре работают половину от всего времени, потери в производительности составят менее 8 %. С другой стороны, работа в паре снимает необходимость анализа кода, и тогда нет никакой потери производительности.

Затем рассмотрим преимущества — обмен знаниями, взаимное обучение и глубокое взаимодействие. Эти преимущества невозможно просчитать, но они также весьма важны.

По моему опыту и опыту многих других, программирование в паре, если оно происходит непринужденно и по желанию самих программистов, приносит довольно много пользы всей команде.

Только два?

Слово «пара» подразумевает, что в сессии парного программирования работают только два программиста. Хотя чаще всего это так, это не строгое правило. Иногда для решения задачи может собраться группа из трех, четырех или большего количества программистов (опять же на усмотрение программистов). Это явление иногда называют «совместное программирование»¹.

Менеджеры

Программисты часто опасаются, что менеджеры не одобрят работу в парах или даже потребуют разойтись и не заниматься ерундой, чтобы не тратить драгоценное время. Я с таким не встречался. За все полвека, что я занимаюсь написанием кода, я никогда не видел, чтобы менеджеры вмешивались. В большинстве случаев, по моему опыту, они только рады видеть, что программисты сотрудничают, работая вместе. Это создает впечатление, что работа кипит.

Если же вы менеджер, который хочет разогнать программистов, работающих в паре, опасаясь, что такая работа неэффективна, то отбросьте свои опасения и дайте программистам возможность решить самим. В конце концов, они профессионалы. А если вы программист и ваш менеджер требует прекратить работу в паре,

¹ https://en.wikipedia.org/wiki/Mob_programming, <https://mobprogramming.org/mob-programming-basics/>.

напомните ему, что специалист здесь вы, поэтому только вы, а не менеджер, отвечаете за то, как будет вестись работа.

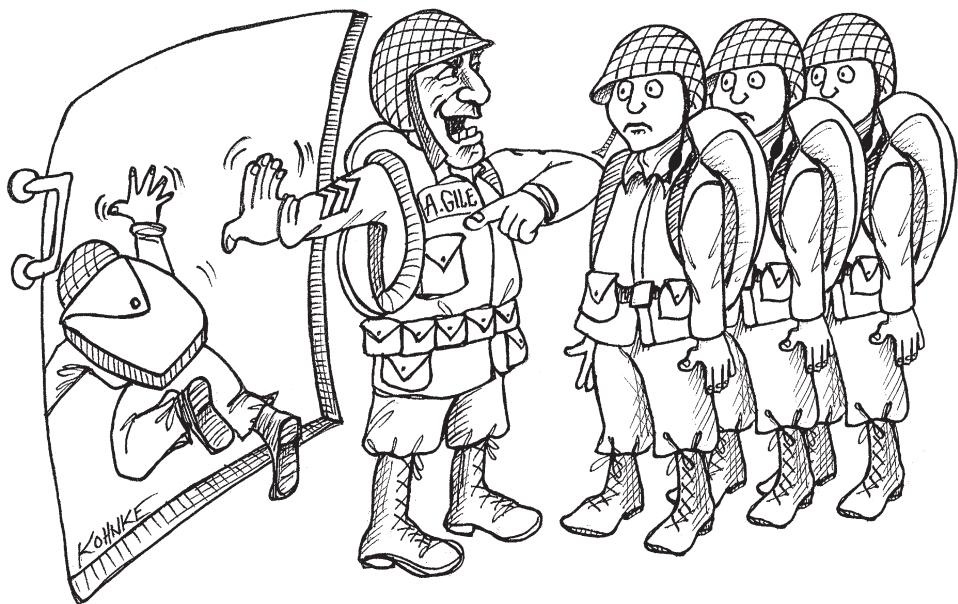
И наконец, никогда в жизни не просите разрешения на работу в паре. На проведение тестирования. На рефакторинг. И прочее.

Вы профессионал. Вам решать.

ЗАКЛЮЧЕНИЕ

Технические методы Agile — наиболее важная составляющая всей методологии. Любая попытка применить Agile без технических методов обречена на провал. Причина проста: Agile — действенный механизм при работе в большой спешке, образующей большой беспорядок. Без использования технических методов, позволяющих поддерживать высокий уровень качества кода, команда начнет стремительно и неумолимо утопать в бесконечной пучине низкой производительности.

6 ВНЕДРЕНИЕ AGILE



Когда я впервые услышал об экстремальном программировании, то подумал: «Что может быть проще? Просто соблюдать простые правила и применять методы. Вот и все».

Но судя по тому, сколько организаций безуспешно пытается внедрить Agile, реализация Agile сопровождается огромными сложностями. Возможно, причина всего этого в том, что во многих компаниях принимают за Agile что-то, что им не является.

ЦЕННОСТИ AGILE

Еще давно Кент Бек сформулировал четыре ценности Agile. Это смелость, взаимодействие, обратная связь и простота.

Смелость

Первая из ценностей — это смелость, или, по-другому, разумная степень принятия рисков. Члены команды, работающей по Agile, в первую очередь сосредоточены на качестве и возможностях, а не на каких-то политических мотивах. Они понимают, что лучший способ вести проект по разработке ПО в течение долгого срока — проявлять некоторую напористость.

Есть разница между смелостью и безрассудством. Чтобы развернуть наименее достаточный набор функций, нужна смелость. Смелость нужна и для того, чтобы поддерживать высокое качество кода и добросовестно применять методы. При этом безрассудно развертывать программу с неустойчивой структурой или такую, в качестве которой вы не до конца уверены. Безрассудно идти в ногу с графиком, принося в жертву качество.

Верить в то, что качество и дисциплина *повышают* скорость — смело, поскольку это убеждение будут постоянно оспаривать влиятельные, но наивные спешащие коллеги.

Взаимодействие

Мы ценим прямое и частое взаимодействие, которое воздвигает мосты между людьми. Члены Agile-команды хотят общения друг с другом. Программисты, клиенты, тестировщики и руководители не против находиться рядом друг с другом, часто общаться, и не только на встречах. Не только через электронную почту, сообщения и заметки. Они ценят личные непринужденные разговоры тет-а-тет.

Так команда становится сплоченной. Это происходит в быстром хаотичном потоке легкого и частого взаимодействия. Рождается огненная буря, несущая озарение, зажигающая лампочки в головах людей. Когда вся команда в сборе, а ее члены находятся рядом и постоянно общаются, то происходят чудеса.

Обратная связь

Методы Agile, которые мы изучили, все как один направлены на отдачу быстрой обратной связи ребятам, принимающим важные решения. *Игра в планирование, рефакторинг кода, разработка через тестирование, непрерывная интеграция, небольшие и частые релизы, коллективное владение, одна команда* и другие методы повышают частоту обратной связи и количество передаваемых сведений. Они позволяют нам своевременно понять, что что-то идет не так, чтобы внести исправления. Они дают важные уроки того, что все

решения, принятые ранее, влекут за собой последствия. Команды, применяющие Agile, успешны за счет обратной связи.

Именно обратная связь помогает команде работать столь плодотворно и приводит проект к благоприятному исходу.

Простота

Следующая ценность Agile — это простота, другими словами, *честность*. Часто говорят, что каждую проблему в разработке можно решить, добавив еще один слой неясности. Но такие ценности, как храбрость, взаимодействие и обратная связь, существуют для того, чтобы свести количество проблем к минимуму.

Таким образом, неясность можно свести к минимуму. Решения могут быть простыми.

Это касается ПО, но также относится и к команде. Пассивная агрессия — это неясность и уклончивость. Когда вы видите проблему, но молча перекладываете ее решение на кого-то другого, вы ведете себя нечестно. Когда вы соглашаетесь с требованиями менеджера или клиента, зная о губительности последствий, вы нечестны.

Простота — это честность. Честность при написании кода и честность во взаимоотношениях и поведении. При написании кода в некотором количестве неясности есть необходимость. Неясность — это механизм, посредством которого мы снижаем сложность взаимозависимости. При работе в команде в неясности куда меньше необходимости. Большую часть времени нужно быть честным, насколько это возможно.

Сохраняйте простоту кода. Не усложняйте отношения в команде.

МЕТОДОЛОГИЧЕСКИЙ БЕСТИАРИЙ

В огромном количестве существующих методик Agile легко запутаться. Я понимаю, что их тьма-тьмущая, но не обращайте на это внимания. В конце концов, независимо от того, на какие методики падет ваш выбор, вы будете подстраивать и отлаживать модель разработки под свои нужды. Таким образом, начни вы с экстремального программирования, Scrum или других 5328 методик, относящихся к Agile, вы придетете к одному и тому же.

Настоятельный совет, который я могу вам дать, — это полностью перенять жизненный цикл, главным образом, технические методы. Огромное количество команд переняли лишь внешнее кольцо, определяющее взаимоотношения с клиентами, и угодили в ловушку, которую Мартин Фаулер назвал «дряблый Scrum»¹. Признаки этой болезни: производительность медленно падает с высокого уровня в начале проекта до крайне низкого, когда проект подходит к концу. Причиной такой потери в производительности является искажение и ухудшение качества самого кода.

Оказывается, что методы взаимодействия с клиентами, предлагаемые Agile, — очень единственный способ создать огромный беспорядок. Кроме того, если вы не позаботитесь о чистоте структуры, которую выстраиваете, беспорядок будет замедлять ход работ.

Итак, выбирайте одну из методик или вообще не выбирайте. Убедитесь, что вы учитываете все дисциплины жизненного цикла. Согласуйте с командой. И вперед. Помните, что смелость, взаимодействие, обратная связь и простота позволяют с постоянством отлаживать дисциплины и методы. Не просите разрешений. Не

¹ <https://martinfowler.com/bliki/FlaccidScrum.html>.

боятесь сделать неправильно. Просто выполняйте свою работу наперекор возникающим проблемам и настойчиво ведите проект к лучшему.

ПРЕОБРАЗОВАНИЕ

Переход от других методологий и моделей к Agile ведет к изменению ценностей. Ценности при использовании Agile включают в себя принятие рисков, быструю обратную связь, а также глубокое и многостороннее взаимодействие между членами коллектива, которое стирает границы внутри команды, в том числе между начальниками и подчиненными. Они также нацелены на ясное и честное поведение, а не распределение ролей и поиски крайнего. Эти ценности прямо противоположны ценностям крупных организаций, которые вкладывают значительные средства в менеджеров среднего звена, ценности которых — безопасность, преемственность, административное управление и выполнение плана.

Разве возможно перевести такую организацию на Agile? Честно говоря, это не то, в чем я достигал больших успехов, у других я подобных успехов также не наблюдал. Я видел, как затрачивалось много усилий и средств, но чтобы организация действительно осуществила переход, я видел редко. Уклад ценностей слишком отличается от тех, которых придерживаются менеджеры среднего звена, чтобы их принять.

То, что я видел, — это переход команд и отдельных специалистов, потому что команды и одиночные программисты часто следуют ценностям, которыми руководствуется Agile.

Как ни странно, руководители тоже часто разделяют ценности, присущие Agile, например: принятие рисков, ясность и взаимо-

действие. В том числе по этой причине они пытаются осуществить переход в своих организациях.

Дело как раз в менеджерах среднего звена. Этих ребят взяли на работу не для того, чтобы принимать риски или вносить ясность, но для того, чтобы передавать ответственность при минимальном взаимодействии. Эта проблема остро стоит во многих компаниях. Руководство и работники в организациях разделяют мировоззрение, свойственное Agile, но среднее звено мыслит наоборот. Я ни разу не видел, чтобы менеджеры среднего звена стояли в основе изменений. И вправду, с чего бы? Сопротивление таким изменениям — это их работа.

Чтобы донести свою мысль, расскажу вам несколько историй.

Саботаж

Еще тогда, в 2000 году, я участвовал в переходе одной организации на Agile. Мы заручились поддержкой начальства и программистов. Переход сулил большие надежды. Возникли сложности с техническими руководителями и архитекторами. Эти ребята, неправильно оценив положение дел, подумали, что их значимостью стали пренебрегать.

Значимость архитекторов, технических руководителей проектов и многих других в команде, работающей по Agile, отличается, но никак не преуменьшена. К сожалению, ребята не понимали этого, возможно, и по нашей вине. Может, мы не донесли до них то, насколько они были значимы для команды, или они просто не хотели учиться новым навыкам, которые бы им пригодились.

Как бы то ни было, они тайком строили план саботажа, чтобы не допустить перехода на Agile. Не буду вдаваться в подробности

этого плана. Только скажу, что как-то раз их застукали за этим и немедленно выперли с работы.

Мне бы очень хотелось сказать вам о том, что после этого переход на Agile стал продвигаться семимильными шагами и закончился большим успехом. Но, увы, не могу.

Волчонок

У нас отлично получилось перевести на Agile одно из подразделений компании куда крупнее. Оно переняло опыт экстремального программирования и проделало за эти годы такую работу, что удостоилось статьи в журнале *Computerworld*. Собственно, за этот успех вице-президент по инженерно-техническому обеспечению, руководивший этим переходом, получил повышение.

Его заменил новый вице-президент. И, подобно возмужавшему волчонку, которому посчастливилось возглавить стаю, он занялся тем, что уничтожил все наследие своего предшественника. Это коснулось и Agile. Он полностью отказался от него и вернул команде старую модель разработки, которая отличалась не в лучшую сторону.

Это привело к тому, что многие члены команды стали искать новое место работы, чего, я полагаю, и добивался новый вице-президент.

Плакса

Последнюю историю мне рассказали. Я не присутствовал в самый важный момент. Мне рассказали мои сотрудники, работавшие в то время.

В 2003 году моя компания переводила на Agile одну известную брокерскую фирму. Все шло замечательно. Велась подготовка руководителей высшего и среднего звена, а также разработчиков. Они готовились вместе. Ничего не предвещало беды.

Потом пришла пора подвести итоги. Руководство и разработчики собрались в большой аудитории. Целью было оценить ход и успешность перехода на Agile. Начальство задало вопрос: «Как обстоят дела?»

С разных сторон послышались ответы: «Все отлично!»

Потом повисло гробовое молчание, которое резко прервалось всхлипами, доносящимися сзади. Кто-то заплакал. И тогда эмоциональный подъем обрушился, и аудитория погрузилась в уныние. Вдохновения как не бывало. «Это так сложно, — донеслось до собравшихся. — Мы это просто не тянем».

После этого начальство свернуло переход.

Мораль

Мораль всех этих историй: ожидайте любых странностей.

Притворяйтесь

Может ли команда, применяющая Agile, работать в организации, где есть сильное среднее звено, которое против этой методологии? Я видел, как время от времени это происходило. Некоторые команды разработчиков спокойно руководствуются Agile во время выполнения своей работы и в то же время выполняют строгие условия, навязанные менеджерами среднего звена. По-

куда менеджеры среднего звена довольны соблюдением правил и нормативов, они оставляют разработчиков в покое, не вмешиваясь в их работу.

Это как раз то, о чём говорили Буч и Парнас: «Притворяйтесь!»¹ Команда работает по Agile, не объявляя об этом, и в то же время делает все для того, чтобы менеджеры среднего звена оставались довольными. Вместо того чтобы бороться с ветряными мельницами, такие команды используют Agile на низком уровне, преподнося его так, что для менеджеров среднего звена он выглядит безопасным и совместимым с их ценностями.

Например, менеджеры хотят получить документ об анализе, проведенном на ранней стадии проекта. Команда, применяющая Agile, пишет большое количество первичного кода программы по всем канонам Agile, затем выпускает документ об анализе, запланировав череду историй по производству документации. Менеджеры получают необходимый им документ.

В этом есть смысл, поскольку первые несколько итераций написания кода в значительной мере ориентированы на анализ требований. То, что анализ выполнен благодаря непосредственно написанию самого кода, менеджерам среднего звена знать необязательно. Их не должно это волновать.

К сожалению, мне доводилось видеть компании, где такой дурдом, что если, не дай бог, среднее звено учитывает, что «что-то не так», оно прибегнет к различным уловкам, чтобы от Agile не осталось и следа. Это позор, потому что такие команды в действительности дают менеджерам все необходимое.

¹ Booch G. Object-Oriented Analysis and Design with Applications. 2nd ed. Reading, Massachusetts: Addison-Wesley, 1994. Pp. 233–234.

Успех в небольших организациях

Я видел, как некоторые средние по размеру компании переходили на Agile. В таких компаниях тонкая прослойка менеджеров среднего звена — это сотрудники, которые получили свои должности, поднимаясь с низов. У них сохранился образ мышления людей, готовых к прямому взаимодействию и принятию рисков.

То, что мелкие компании полностью переходят на Agile, отнюдь не редкость. Там нет менеджеров среднего звена, а ценности боссов и разработчиков совпадают.

Успешный переход отдельных специалистов

Наконец, в некоторых компаниях ценности Agile перенимают только отдельные сотрудники. Те, кто переходит на Agile индивидуально, чувствуют себя некомфортно в компании или команде, которые не собираются этого делать. Разница в ценностях обычно приводит к некоторому разделению. В лучшем случае люди, которые осуществляют переход, объединяются, чтобы сформировать новые гибкие команды, которым удастся скрыться от менеджеров среднего звена. Если это не получается, они, скорее всего, будут искать (и найдут ведь!) работу в другой компании, которая разделяет их ценности.

За последние двадцать лет мы видели, как в отрасли меняются ценности. Образуются новые компании, которые принимают ценности Agile, и программисты, которые хотят работать по методологии Agile, группируются в таких компаниях.

Создание Agile-организаций

Можно ли создать крупную организацию, где смогут успешно работать команды, использующие Agile? Несомненно! Только обратите внимание, я сказал «создать», а не «перевести».

Когда IBM решила выпустить персональный компьютер, руководство компании понимало, что ценности организации не позволяют быстро внедрить нововведения и принять необходимые риски. Поэтому она создала организацию с другой системой ценностей¹.

Было ли такое в мире разработки ПО? Было ли такое, что крупные организации создавали более мелкие, чтобы писать ПО с помощью Agile? На моей памяти были намеки на это, но не могу привести ни одного яркого примера.

Конечно, мы видели множество новых компаний, принимавших Agile. Также можно вспомнить много компаний, консультировавших по Agile более крупные компании, которые не применяли Agile в целом, но хотели выполнить отдельные проекты с большей скоростью и надежностью.

А вот мой прогноз. В конечном итоге мы увидим, как крупные компании будут создавать новые подразделения, которые будут заниматься написанием программ с применением Agile. Также консалтинговые компании, занимающиеся Agile, будут все чаще оказывать услуги крупным организациям, которым не удалось перевести на Agile своих разработчиков.

¹ The birth of the IBM PC. URL: https://www.ibm.com/ibm/history/exhibits/pc25_pc25_birth.html.

КОУЧИНГ

Нужен ли команде, работающей по Agile, коуч? Вообще — нет. Но если хорошо подумать, то иногда нужен.

Прежде всего, следует видеть грань между тренером и коучем. Agile-тренер обучает команду, как организовать себя, применяя Agile.

Часто они не работают в самой компании или работают в ней, но сами не являются членами команды. Их цель — прививать ценности Agile и обучать дисциплинам Agile. Их работа не должна длиться долго. Команде, состоящей из десятка разработчиков, понадобится одна-две недели тренингов.

Agile-тренер может говорить и делать что угодно, но всему остальному, чему нужно, разработчики научатся сами.

В начале перехода команды на Agile тренер может временно выступать в роли коуча, но продолжительность такого коучинга невелика. Это бремя должен на себя взять кто-то из членов команды, и чем скорее, тем лучше.

Как правило, коучи и тренеры — разные люди. Коучи — это члены команды, чья задача — обеспечивать соблюдение методологии внутри команды. Когда разработка кипит, у программистов может возникнуть соблазн сойти с колеи. Они ненароком могут перестать работать в паре, выполнять рефакторинг кода или обращать внимание на сбои, возникающие при непрерывной сборке. Коуч наблюдает за тем, чтобы такого не было, и при случае указывает команде на их ограхи. Коуч выступает в качестве совести команды, постоянно ей напоминая о данных себе обещаниях и о ценностях, которые те согласились соблюдать.

Эта роль по мере необходимости обычно переходит от одного члена команды к другому согласно неофициальному графику. Команда, уже стабильно сработавшаяся, не нуждается в коуче. С другой стороны, команда в напряженной обстановке, будь то из-за графика работ, проблем во взаимодействии с клиентами или внутри команды, может принять решение назначить кого-либо коучем на время.

Коуч не менеджер. Он не отвечает за распределение средств или график работ. Коуч не руководит командой и не представляет интересы команды перед менеджерами. Коуч также не является посредником между клиентами и разработчиками. Коуч выполняет свои функции исключительно внутри команды. Никто из менеджеров или клиентов не знает, кто является коучем и есть ли он вообще.

Скрам-мастер

В Scrum такой коуч называется мастером. Изобретение этого понятия и череда событий, последовавших за ним, были одновременно и лучшим, и худшим, что когда-либо случалось в сообществе Agile. Программы сертификации привлекли большое количество менеджеров проекта. Такой приток способствовал распространению Agile в самом начале, но в итоге он привел к тому, что функции коуча приравняли к тому, чем занимается менеджер проекта.

В наше время слишком часто происходит так, что скрам-мастера никакие не коучи, а обычные менеджеры проекта, которые выполняют соответствующую работу. К несчастью, название должности и наличие сертификата способствуют их непомерному влиянию на команду, применяющую Agile.

СЕРТИФИКАЦИЯ

Вся существующая сертификация по Agile смехотворна и нелепа. Нельзя относиться к такой «сертификации» всерьез. Обучение во время программ сертификации часто полезно, однако оно не должно сводиться к определенной роли и должно быть рассчитано на всех членов команды.

Например, от «сертификата» скрам-мастера никакого толка нет. Сертификаты значат не больше, чем то, что кому-то некуда было девать деньги и он в лучшем случае прошел двухдневные курсы. Лицо, выдающее сертификат, не гарантирует, что новоиспеченный мастер будет хорош в роли коуча. Бессмыслица наличие такого сертификата в том, что он наделяет «сертифицированного скрам-мастера» чем-то особенным, а это не имеет ничего общего с коучингом в команде. Чтобы стать коучем в команде, не нужно никаких обучений.

Нет ничего плохого в прохождении самого обучения, необходимого для сертификации. Просто обучать только одного человека отдельной роли в команде глупо. Каждый член команды, применяющей Agile, должен понимать ценности и методы Agile. И если один член команды подготовлен, значит, нужно подготовить и всех.

Настоящая сертификация

А как на самом должна выглядеть программа сертификации по Agile? Это должен быть семестровый курс, который сочетал бы обучение работе по Agile и небольшой учебный проект по гибкой методологии разработки. На этих курсах будет система оценок и высокие планки прохождения заданий. Тот, кто выдает сертификаты, должен гарантировать, что ученики усвоили ценности Agile

и показали высокий уровень мастерства в применении методов Agile.

AGILE В КРУПНЫХ МАСШТАБАХ

Движение Agile появилось в конце 1980-х. Его быстро признали способом организовать небольшую команду, размером от 4 до 12 разработчиков. Эти числа были нестрогими и редко озвучивались, однако все понимали, что Agile (или как мы его там называли до 2001-го) не подходит для гигантских команд из тысяч разработчиков. Это не та задача, над решением которой мы бились. Тем не менее вопрос подняли почти сразу. А что насчет больших команд? Что, если применить Agile в крупном масштабе?

Долгие годы люди искали ответ на этот вопрос. В самом начале авторы Scrum предложили метод Scrum-of-Scrums. Позже мы стали наблюдать появление некоторых фирменных подходов вроде SAFe¹ и LeSS². На эту тему написано несколько книг.

Я уверен, что в этих подходах нет ничего плохого. Я уверен, что эти книги замечательны. Но я не пробовал этих методов и не читал книг. Вы можете подумать, что я какой-то пустомеля, что высказываюсь на тему, которую не изучил. Может, вы и правы. Однако у меня есть своя точка зрения.

Agile создан для малых и средних команд. Точка. Он хорошо работает для таких команд. Agile никогда не предназначался для больших команд.

¹ https://ru.wikipedia.org/wiki/Scaled_Agile_Framework.

² <https://less.works/ru/>.

Почему мы не пробовали решить проблему больших команд? Да потому что проблема больших команд решается огромным количеством специалистов вот уже больше пяти тысяч лет. Эта проблема больших команд — проблема культур и цивилизаций. И если в какой-то мере судить о нашей нынешней цивилизации, эту проблему решили достаточно неплохо.

Как построили пирамиды в Египте? Надо было решить проблему больших команд. Как получилось победить во Второй мировой войне? Надо было решить проблему больших команд. Как удалось отправить человека в космос и благополучно вернуть его на Землю? Надо было решить проблему больших команд.

Но такие большие проекты — не единственные достижения больших команд, не правда ли? Как получилось развернуть телефонную сеть, построить автомагистраль, создать интернет, произвести мобильные телефоны или автомобили? Это все сотворили большие команды.

Инфраструктура и средства обороны нашей обширной, охватывающей весь земной шар цивилизации — прямое свидетельство того, что мы уже решили проблему организации больших команд.

Большие команды — проблема уже решенная.

Та проблема, которую все еще не решили тогда, в конце 1980-х, когда зарождалось движение Agile — это проблема организации работы малых команд разработчиков. Мы не знали, как эффективно организовать относительно малую группу программистов так, чтобы была максимальная отдача. И эту проблему решил Agile.

Важно понимать, что Agile создали для решения проблемы организации небольшой команды разработчиков, а не просто небольшой

команды. Проблему небольших команд решили еще в древние времена военные и производственные организации по всему миру. Римляне бы не покорили Европу, если бы не смогли решить проблему организации небольших отрядов.

Agile – это набор дисциплин, с помощью которых мы организуем небольшие команды разработчиков ПО. Зачем нам нужен отдельный способ для организации разработчиков? Потому что программное обеспечение особенно.

На него похожи только несколько областей знаний. Соотношения «вложение/выгода» и «риск/вознаграждение» в разработке ПО отличаются от тех, что имеются в других видах деятельности. Разработка похожа на строительство, за исключением того что не строится ничего осозаемого. Разработка похожа на математику, за исключением того что ничего нельзя доказать. Разработка похожа на естествознание своей эмпиричностью, но при этом не открывается никаких законов природы. Разработка похожа на бухгалтерское дело, за исключением того что она описывает поведение, упорядоченное по времени, а не факты о числах.

Разработка ПО действительно не похожа ни на что другое. Поэтому для того, чтобы организовать небольшую команду разработчиков, нужен набор особых дисциплин, которые подстроены под уникальность разработки.

Посмотрите на дисциплины и методы, о которых мы говорили на страницах этой книги. Обратите внимание, что они все до единого, почти без исключения, подстроены и отлажены под уникальные стороны разработки. Присмотритесь к методам, начиная от очевидных вроде разработки через тестирование и рефакторинга до более неоднозначных вроде игры в планирование.

Суть в том, что Agile создан для сферы разработки ПО. В частности, речь идет о небольших командах программистов. Мне неприятно, когда меня спрашивают, как внедрить Agile в сферу производства аппаратного обеспечения, строительства или в другой процесс. Я всегда отвечаю, что не знаю, потому что Agile существует для сферы разработки ПО.

А что, если масштабировать Agile? Думаю, ничего не выйдет. Организовать большие команды можно, разбив их на несколько мелких. Agile решает проблему небольших команд разработчиков. Проблема организации небольших команд в большие уже решена. Поэтому мой ответ на вопрос о применении Agile в крупном масштабе таков: просто распределите ваших разработчиков по небольшим командам, которые будут работать по Agile, а потом применяйте обычные способы управления и научно-исследовательские методы, чтобы руководить этими командами. Не нужно никаких особых правил.

Теперь мне могут задать еще один вопрос. Если разработка ПО в небольших командах настолько уникальна, что пришлось изобрести Agile, почему такая уникальность не относится к организации маленьких команд разработчиков в большие? Разве не существует чего-то уникального в области разработки ПО, что выходит за пределы организации небольших команд разработчиков и влияет на организацию больших?

Сомневаюсь, потому что проблема больших команд, которую мы решили более пяти тысяч лет назад, — это вопрос слаженного сотрудничества самых разных команд. Команды, работающие по Agile, — лишь один вид несметного числа команд, которые нужно скоординировать для создания чего-то большего. Координация команд различных назначений — уже решенная проблема. Я не вижу

никаких признаков того, что уникальность команд разработчиков плохо влияет на их включение в более крупные объединения.

Так что, опять-таки с моей точки зрения, не существует никакого Agile для применения в крупном масштабе. Agile – необходимое нововведение для организации небольших команд разработчиков ПО. Но будучи уже организованными, такие команды можно встроить в структуры, которые в крупных организациях применяются уже тысячелетиями.

Это не та тема, которую я усердно исследовал. Все, что вы только что прочитали, лишь мое мнение, я могу оказаться неправ. Возможно, я просто старый ворчун, который посыпает всех желающих применить Agile в крупных масштабах идти развлекаться в свой двор. Время лучший судья. Но теперь вы знаете, в чем я точно уверен.

ИНСТРУМЕНТЫ AGILE

*Авторы Тим Оттингер и Джессиф Лангр,
16 апреля 2019 года¹*

Мастера осваивают свои инструменты. Столяры овладевают молотком, метром, пилой, долотом, рубанком и уровнем. Все эти инструменты недороги и отлично подходят мастеру в начале его трудового пути. По мере роста своих потребностей столяр учится пользоваться инструментами посерьезнее (которые, как правило, и дороже): дрелью, гвоздезабивным пистолетом, токарным и фрезерным станком, САПР, ЧПУ и многое еще.

¹ Приводится с разрешения.

Однако мастера столярного дела не расстаются с ручным инструментом, который отлично подходит для работы. Используя только ручной инструмент, умелый мастер может выполнить работу качественнее и иногда даже быстрее, чем приводным инструментом. Как следствие, толковый столяр осваивает ручной инструмент, прежде чем перейти к более совершенным. Столяры изучают предел возможностей ручного инструмента, и поэтому у них есть понимание, когда нужно прибегнуть к приводному.

Вне зависимости от того, какой инструмент используется, ручной или приводной, столяр всегда стремится овладеть каждым инструментом из своего арсенала. Такое мастерство позволяет ему сосредоточиться непосредственно на ремесле, например на изготовлении изящной мебели высокого качества, а не на инструменте. Без должного овладения инструмент — плохой помощник, а при неумелом применении может даже нанести вред как изделию, так и незадачливому работнику.

Средства разработки

Разработчикам ПО в начале работы требуется освоить целый ряд инструментов:

- Хотя бы один язык программирования, а чаще больше.
- Интегрированную среду разработки или текстовый редактор, подходящий программисту (vim, Emacs и т. д.).
- Различные форматы данных (JSON, XML, YAML и т. д.) и языки разметки (в том числе HTML).
- Командную строку и скрипты для взаимодействия с операционной системой.
- Системы управления версиями (Git. Тут без вариантов).

- Средства для непрерывной интеграции и сборки (Jenkins, TeamCity, GoCD и т. д.).
- Средства развертывания и управления сервером (Docker, Kubernetes, Ansible, Chef, Puppet и т. д.).
- Средства коммуникации (электронная почта, Slack, английский язык).
- Инструменты тестирования (фреймворки для модульного тестирования, Cucumber, Selenium и т. д.).

Все эти инструменты необходимы для создания ПО. Без них на сегодняшний день невозможно ничего сделать. В некотором смысле, это «набор ручных инструментов» разработчика.

Чтобы освоить многие из этих инструментов и использовать их с отдачей, придется попотеть. Тем временем положение дел постоянно меняется, поэтому мастерски овладеть тем или иным инструментом становится все большим испытанием. Грамотный разработчик ищет пути наименьшего сопротивления и наибольшую пользу от применяемых инструментов, выбирая те, которые при затраченных усилиях дают большую отдачу.

Что делает инструмент эффективным?

Набор инструментов стремительно меняется, потому что мы постоянно узнаем более действенные способы достигать своих целей. За последние несколько десятков лет мы видели широкое разнообразие систем управления версиями: PVCS, Clear Case, Microsoft Visual Source Safe, Star Team, Perforce, CVS, Subversion, Mercurial и прочие. Все они страдали от каких-то недостатков: слишком нестабильные, слишком проприetaryные или закрытые, слишком медленные, слишком въедливые, слишком жуткие или сложные.

В итоге победил тот, который преодолел большинство ограничений, — Git.

Одна из сильных сторон Git в том, что он дает уверенность, что исходный код будет в сохранности. Если вы уже давно работаете, то наверняка использовали какие-то из перечисленных систем и, вероятно, время от времени нервничали. Требуется соединение с сервером в реальном времени, иначе ваша работа под угрозой. Репозиторий CVS время от времени повреждал файлы, после чего приходилось устраивать пляски с бубном в надежде восстановить данные. Сервер репозитория иногда падал, даже при наличии резервной копии, и можно было прождать пол рабочего дня. Некоторые проприетарные системы также страдали повреждением данных в репозиториях. Вы висите на телефоне часами, разговаривая с поддержкой, при этом отстегивая деньги на сторону за возможность привести их в порядок. При использовании Subversion вы опасались вести много веток, потому что чем больше файлов находилось в репозитории, тем дольше переключались ветки (иногда это занимало несколько минут).

Хороший инструмент должен быть удобен в использовании, а не заставлять вас содрогаться при одном лишь его виде. Git быстрый, он дает возможность вносить изменения в код локально, а не только на сервере, позволяет работать из локального репозитория без соединения по сети; он отлично поддерживает работу в нескольких репозиториях и нескольких ветках, а еще искусно выполняет слияние версий.

У Git достаточно ясный и понятный интерфейс. Получается, что научившись работать в Git однажды, вам не придется особо думать о самом инструменте. Вас будут волновать куда более насущные вопросы: безопасность хранения данных и управление версиями исходного кода. Инструмент стал прозрачен.

Git — это функциональный и сложный инструмент. И что значит изучить его хорошо? К счастью, работает принцип 80/20. Достаточно малая часть возможностей Git, скажем, процентов 20, поможет вам справиться с более чем 80 % повседневных задач, которые будут встречаться во время управления исходным кодом. Большую часть всего необходимого можно освоить за минуты. Сведения по всему остальному можно найти в Сети.

Простота и эффективность использования Git привели к совершенно непредвиденному новому подходу, как создавать программное обеспечение. Линус Торвальдс подумал бы, что использовать Git как инструмент для быстрого избавления от маленьких кусочков кода — сумасшествие, но это именно то, что продвигают сторонники метода Микадо¹ и TCR (Test&&Commit || Revert)². И даже, хотя ключевой стороной Git является его способность очень эффективно управлять ветками, бесчисленные команды почти без исключения ведут trunk-based разработку с помощью Git. Инструмент претерпел экзаптацию³, то есть эффективно используется способами, которые не предполагали авторы.

Хорошие инструменты выполняют следующие задачи:

- помогают людям достигать своих целей;
- позволяют их достаточно быстро освоить;
- стремятся быть прозрачными для пользователей;

¹ Ellnestam O., Brolund D. The Mikado Method. Shelter Island, New York: Manning Publications, 2014.

² Beck K. test && commit || revert. 2018. URL: https://medium.com/@kentbeck_7670/test-commitrevert-870bbd756864.

³ Классический пример экзаптации: оперение птиц первоначально использовалось для регуляции температуры, в дальнейшем оно было адаптировано для полета. — Примеч. ред.

- способны адаптироваться и экзаптироваться;
- доступны по стоимости.

Мы приводим Git в качестве примера хорошего инструмента... на 2019 год. Возможно, вы читаете это уже в будущем, и на дворе другой год. Времена меняются, меняются и инструменты.

Физические инструменты Agile

Пользователи Agile известны тем, что используют маркерные доски, клейкую ленту и наклейки разных размеров (маленькие и размером с флипчарт), чтобы работа была наглядной. Эти простые «ручные орудия» обладают всеми качествами хорошего инструмента:

- Помогают сделать ход работы наглядным и управляемым.
- Интуитивно понятны и не требуют особой подготовки.
- Не требуют значительной когнитивной нагрузки. Их можно легко использовать, сосредоточившись на других задачах.
- Легко экзаптируемы. Ни одно из этих средств не было создано именно для управления ходом разработки ПО.
- Легко адаптируемы под конкретные потребности. Можно использовать клейкую ленту или офисный пластилин, прикреплять картинки или значки, добавлять различные пометки, а еще по-своему использовать различные цвета и значки, чтобы не упустить ни одного нюанса.
- Все они недороги, и их легко приобрести.

Команды, располагающиеся в одном пространстве, могут легко управлять крупным и сложным проектом с помощью лишь этих простых и недорогих физических инструментов. Вы можете транслировать ключевую информацию с помощью листа из флипчар-

та, закрепив его на стену клейкой лентой. Такое представление обобщает важные тенденции и факты как для членов команды, так и для спонсоров. С помощью таких представлений можно изображать и представлять новые сведения прямо на лету. Гибкость почти не ограничена.

Но ограничения есть у каждого инструмента. Одно из основных ограничений физических инструментов в том, что они не очень действенны для распределенных команд. Только для сотрудников, находящихся в пределах видимости. Еще физические инструменты не сохраняют историю автоматически, поэтому есть только текущее состояние.

А может, автоматизируем?

Проект, в котором впервые применяли экстремальное программирование (Chrysler Comprehensive Compensation System), вели преимущественно с помощью физических инструментов. По мере распространения Agile рос интерес к автоматизированным программным средствам. На это есть вполне разумные основания:

- Программные средства хорошо позволяют осуществлять сбор данных в однородном виде.
- С помощью однородных данных можно легко составлять доклады, графики и схемы, выглядящие профессионально.
- Легко вести историю и хранить данные.
- Можно мгновенно делиться данными, вне зависимости от места нахождения адресата.
- Благодаря средствам вроде электронных таблиц, доступных по сети, можно работать в полностью распределенной команде в режиме реального времени.

Некоторым ребятам, больше привыкшим к вылизанным презентациям и программам, физические инструменты кажутся чем-то отсталым. И поскольку мы работаем в отрасли разработки ПО, для многих из нас автоматизация всего, чего только возможно, — естественное стремление.

Программные средства в студию!

Или... может, не надо? Давайте остановимся и хорошенько подумаем. В программных средствах может отсутствовать часть функций, которые нужны вашей команде. Если у вас есть инструмент, то путь наименьшего сопротивления — это исходить из возможностей инструмента, вне зависимости от того, отвечает ли он потребностям команды.

Команде сперва следует определиться, каким образом она собирается вести работы, затем уже подбирать средства именно под свои потребности.

Работники используют инструменты, а не инструменты — работников.

Никому не хочется зависеть от чужого мнения. Что бы вы ни делали, вам хочется разобраться в том, как нужно работать, прежде чем что-то автоматизировать. Но вопрос не в том, какие инструменты использовать: физические или программные. Вопрос должен стоять так: хорошие инструменты у нас или нет?

Системы управления жизненным циклом приложений

Вскоре после появления Agile были созданы многочисленные программы для управления проектами, которые ведутся с помощью

Agile. Существуют самые разные системы управления с жизненным циклом приложений (ALM) на базе Agile, как с открытым исходным кодом, так и в красивой блестящей обертке за приличные деньги. Они позволяют собирать данные, образующиеся в ходе работы, управлять длинными списками функций и еще не выполненных задач, создавать сложные графики, представлять сводки работы команд в совокупности, а еще выполнять некоторые операции с числами.

Автоматизированные системы, которые помогут справиться с по-добной работой, возможно, упорядочат эти процессы, для нашего же удобства. Помимо своих основных, системы ALM наделены другими полезными функциями. Большая часть ALM позволяет вести удаленное взаимодействие, отслеживать историю, выполнять некоторые рутинные бухгалтерские операции, а еще их можно гибко настроить под требования пользователя. Можно с помощью графопостроителя создавать профессиональные многоцветные графики на листах огромных форматов, которые можно развешивать как стенгазеты в пространстве, в котором находится команда.

И все же, несмотря на богатый функционал и коммерческий успех, ALM невозможно назвать хорошим инструментом. Эта неудача будет нам хорошим уроком.

- *Хороший инструмент можно изучить достаточно быстро.* ALM, как правило, громоздки и требуют специальной подготовки перед их использованием (давайте-ка вспомним, когда мы в последний раз были на обучении по индексным карточкам). Даже несмотря на обучение, команде приходится отвлекаться на поиск в интернете, чтобы выяснить, как выполнить ту или иную простую задачу. Многие нехотят принимают сложность этих систем, пытаясь вникнуть глубже и во всем разобраться, но в конце концов мирятся с тем, что работа идет медленно и неповоротливо.

- *Хорошие инструменты стремятся к прозрачности для пользователей.* Мы постоянно видим, как члены команды выискивают какую-то ведущую логику в попытке разобраться в программе. Во время работы с карточками с историями они ведут себя словно пьяницы, размахивающие кулаками. Они слоняются по веб-страницам, повсюду вставляя скопированный текст, пытаются связать истории между собой или с их родительскими эпиками. Они теряются в историях, задачах и заданиях в попытке выстроить их в слаженную систему. Это бардак. Эти средства требуют слишком много внимания.
- *Хорошие инструменты способны адаптироваться и экзаптироваться.* Хотите добавить поля в виртуальной карточке в ALM? Вам придется найти программиста экспертурного уровня, который посвятил (или даже пожертвовал) себя поддержке необходимой программы. Или все закончится тем, что вы отправите запрос на изменение поставщику. Дело пяти секунд при использовании простых физических инструментов обрачивается пятидневной, а может и пятинедельной, задержкой при использовании ALM. Эксперименты с быстрой обратной связью в ходе работ становятся невозможными. И, само собой, если вам вообще не нужны дополнительные поля, кто-то должен обратить изменения и перевыпустить программу в измененной конфигурации. Системы ALM плохо адаптируются.
- *Хорошие инструменты доступны по стоимости.* Лицензия на ALM, которая может стоить несколько тысяч долларов в год, — только начало. Установка и использование этих систем может потребовать значительных дополнительных расходов на подготовку, поддержку и, иногда, настройку под ваши потребности. Текущее обслуживание и администрирование выльются в дополнительные затраты к уже немалым имеющимся.

- *Хорошие инструменты помогают достигать людям своих целей.* ALM редко работает таким же образом, как ваша команда, а режим по умолчанию частенько не согласован с самими методами Agile. Например, многие системы ALM считают, что у каждого из членов команды есть свое рабочее задание, что делает невозможным их использование командами, члены которых работают совместно, перемежая задания между собой.

В некоторых системах ALM даже есть доски позора, которые показывают нагрузку, загруженность и объем выполненных работ (или нехватку их) для каждого участника команды. Вместо того чтобы мотивировать на ударную работу и поощрять коллективную ответственность, как это на самом деле принято в Agile, ALM становится средством пристыдить программиста и выжать из него все соки. Там, где команда раньше собиралась на утренний стендап-митинг (или ежедневный скрам), теперь она собирается, для того чтобы внести поправки в ALM. Система заменила личное взаимодействие автоматизированным отчетом о состоянии пользователя.

Хуже того, системы ALM часто не могут транслировать информацию, в отличие от физических средств. Вам нужно выполнить вход и копаться в данных, чтобы найти нужные сведения. Когда вы находитите нужные сведения, они часто идут вместе с кучей ненужных. Иногда два-три графика или изображения, которые вам нужны, могут находиться на разных страницах. Нет повода думать, что ALM никогда не станут хорошим инструментом. Но если вам нужна доска с карточками и нужно использовать ПО, я бы посоветовал какое-нибудь универсальное средство вроде Trello¹. Оно простое, быстрое, дешевое, расширяемое и неплохо выглядит.

Наши способы вести работы постоянно изменяются. Сначала была SCCS, потом RCS, потом CVS, потом Subversion и потом уже

¹ Опять же на 2019 год. Времена меняются.

Git. В течение многих лет мы видели море изменений в способах управления исходным кодом. Похожую эволюцию мы наблюдали на примере инструментов тестирования, средств развертывания и прочего (не будем перечислять). Вероятно, мы увидим и похожее развитие систем ALM.

Если смотреть на текущее состояние большинства систем ALM, то разумнее и безопаснее начать с простых физических инструментов. Возможно, позже вы задумаетесь о внедрении ALM. Удовстверьтесь, что систему легко изучить, она прозрачна для повседневного использования, легко адаптируется и в ваших возможностях ее приобрести и запустить. Самое главное, убедитесь, что с ее помощью удастся организовать работу так, как нужно вам, и что вложения будут не напрасны.

КОУЧИНГ — АЛЬТЕРНАТИВНЫЙ ВЗГЛЯД

Автор Дэймон Пул, 14 мая 2019 года¹

Дэймон Пул — это мой друг, который во многом со мной не соглашается. Коучинг в Agile как раз предмет наших разногласий. Вот я и рассудил, что его точка зрения также интересна и будет полезно ее вам поведать.

Дядя Боб

Множество путей к Agile

Можно прийти к Agile разными способами. И на самом деле многие из нас пошли по этому пути непреднамеренно. Кто-то может ут-

¹ Используется с разрешения.

верждать, что Манифест Agile появился из-за того, что его авторы заметили, что им было по пути, и они решили рассказать о нем, чтобы другие могли отправиться в этот путь вместе с ними.

Мой путь в Agile начался с того, что в 1977-м я посетил один магазин бытовой техники, в котором, как оказалось, продавали компьютеры TRS-80. Я был новичком и помогал одному опытному программисту проводить отладку игры Star Trek тем, что просто задавал ему вопросы. Сейчас это называется парным программированием. И, оказывается, задавать вопросы — важная часть коучинга.

С того времени примерно до 2001-го я, сам того не зная, работал по Agile. Я писал код только в небольших командах, где задачи тасовались между их членами. Клиентом в основном была фирма, в которой мы работали. Я уделял большое внимание тому, что сейчас называют карточками с историями, и мы выпускали продукт только небольшими и частыми релизами. Но потом, когда я уже работал в AccuRey, наши мажорные релизы стали выходить все реже, и в 2005-м разрыв дошел до полутора лет. Целых 4 года я непреднамеренно работал по каскадной модели. Это был ужас, а я даже не понимал почему. Более того, меня считали специалистом по каскадной модели. Если не углубляться в подробности, эта история знакома многим.

Путь к Agile

Мое знакомство с Agile было болезненным. Еще в 2005 году, до того как конференции Agile Alliance и им подобные стали сказочно популярными, были конференции, которые проводил журнал Software Development. Я выступал докладчиком на конференции Software Development East, и после моего выступления о методах

управления распределенными командами разработчиков, в котором не было ни слова об Agile, я вдруг обнаружил себя в окружении ведущих мыслителей отрасли, среди которых были Боб Мартин, Джошуа Кериевский, Майк Кон и Скотт Эмблер. Мне казалось, что все интересующие их темы сводились к карточкам, пользовательским историям, разработке через тестиирование и парному программированию. Я был в ужасе от того, чем были заняты мысли таких гигантов, их слова резали мне слух, словно бритвой.

Спустя несколько месяцев, во время изучения Agile с целью его разоблачить, меня будто ударило током. Как программиста и предпринимателя меня озарило, и я понял, что Agile — это алгоритм поиска наиболее ценных функций для рынка ПО и скорейшего превращения их в доход.

После такого воодушевления во мне развилась страсть советовать Agile всем. Я вел бесплатные вебинары, выкладывал посты в блог, выступал на конференциях, присоединился и участвовал во встрече Agile New England, проходившей в окрестностях Бостона. Делал все, чтобы распространить Agile повсюду. Когда люди делились, какие трудности у них возникали при внедрении Agile, я был полон решимости им помочь. Я перешел в режим решения проблем и объяснял, что нужно делать.

И я стал замечать, что мой подход часто вызывал возражения и все больше вопросов. Так было не только у меня. Я видел, как многие сторонники Agile на конференциях вступали в противоборство с теми, кто еще не осознал всю его прелесть. И до меня стало доходить, что людям, которые по-настоящему приняли Agile и с отдачей его применяют, нужен другой подход для передачи знаний об Agile и опыте его использования, который бы учитывал особенности и обстоятельства каждого обучаемого.

Зачем нужен коучинг в Agile?

Замысел Agile прост. Его описали всего 264 словами в Манифесте Agile. Но постичь Agile не так-то просто. Если бы все было так просто, каждый бы уже использовал Agile и не было бы потребности в специальных коучах. Людям вообще тяжело принимать какие-то изменения, не говоря уже о том, сколько всего нужно изменить, чтобы полностью принять Agile. Постижение Agile включает в том числе пересмотр застарелых убеждений, культуры, методов, мышления и подходов к работе. Задавить человека думать иначе и помочь ему понять, что же «будет от этого», довольно сложно. В масштабах целой команды сложности усугубляются, а когда обучение Agile происходит в среде, которая специально подготовлена под привычные способы работы, становится еще сложнее.

Для всех изменений справедливо, что люди делают то, что они хотят. Ключ к устойчивым изменениям — находить задачи или возможности, которые известны, в которые есть желание вкладывать средства, а потом помогать достигать людям своих целей. Общество требует и нуждается в компетентности специалистов. Все остальное ждет провала. Коучинг помогает людям находить пробелы и основополагающие убеждения, которые мешают им продвинуться вперед. Коучинг помогает преодолевать сложности и достигать своих целей, а не просто предписывает решение.

Становление коуча Agile

В 2008-м на сцену вышла Лисса Адкинс с совершенно другим подходом к коучингу в Agile. Она сделала упор на чистоту коучинга в Agile посредством внедрения навыков, полученных от профессиональных коучей, в мир Agile.

В то время, когда я узнавал больше о профессиональном коучинге и подходе Лиссы и стал применять это в своей работе, я пришел к тому, что можно получить потрясающую отдачу, будучи коучем самому себе. Полученная польза никак не связана с улучшением знания об Agile или набором опыта, которые коуч также может передать.

В 2010-м Лисса полностью описала свой подход к коучингу в книге *Coaching Agile Teams*¹. В то же время она начала предлагать курсы по коучингу. В 2011-м программы этих курсов легли в основу программы IC Agile's Certified Agile Coach (ICP-ACC), а затем консорциум *International Consortium for Agile* начал аккредитацию других тренеров, которые предлагают обучение по программе ICP-ACC. Курсы ICP-ACC в настоящее время представляют собой наиболее полный источник знаний для подготовки профессиональных коучей в области Agile.

За рамками ICP-ACC

Сертификация ICP-ACC включает в себя навыки, необходимые коучу: активное слушание, эмоциональный интеллект, подача себя, умение дать четкую и прямую обратную связь, задавать открытые и наводящие вопросы, а также держаться беспристрастно. Полный набор профессиональных качеств коуча еще шире. Например, Международная федерация коучинга (ICF), которая объединяет более 35 000 сертифицированных профессиональных коучей, различает 70 специализаций по 11 категориям. Чтобы стать сертифицированным профессиональным коучем, нужно пройти

¹ Adkins L. Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition. Boston, Massachusetts: Addison-Wesley, 2010.

сложную подготовку и строгую сертификацию, для которой требуется проявить навыки во всех 70 специализациях и документально подтвердить сотни часов оплаченного коучинга.

Инструменты коуча

Многие системы, практики, методы и техники, применяемые в сообществе Agile для обучения этой методологии и работе по ней, совместимы с целями профессионального коучинга. Существуют некие «инструменты коуча», чтобы помочь отдельным людям и группам открыть для себя, какие преграды встают у них на пути, и принять самостоятельное решение, как продвигаться вперед.

Коучинг дает еще ценный навык — многосторонний опрос, одно из назначений которого «задавать вопросы, которые приводят к открытиям, озарениям, заинтересованности или действиям». Взгляд в прошлое, особенно способы вроде «команда с лучшими результатами во все времена» или «шесть шляп», помогает команде самостоятельно находить возможности для изменений и независимо решать, как эти возможности использовать. Открытое пространство (иными словами, не конференция) — это способ провести многосторонний опрос в большой группе, даже в целой организации.

Если вы проходили формальное обучение по Agile или его методам, то, вероятно, участвовали в каких-то играх, которые давали представление о понятиях, принятых в Agile. Это игра с монетами, симуляторы Scrum, пицца Kanban или постройка городка из кирпичиков лего.

Благодаря этим играм участники получают наглядное представление о силе самоорганизации, размере небольших партий, коман-

дах с тасовкой функций, разработке через тестирование, Scrum и Kanban. Когда игры проводят с намерением повысить уровень осведомленности участников, а затем позволяют им решить, что делать дальше, участники чувствуют дух профессионального коучинга.

Число таких игр неуклонно растет. Многие из них можно найти на tastycupcakes.org, retromat.org и liberatingstructures.com.

Профессиональных навыков коуча недостаточно

Если мы работаем с командой, которая ничего никогда не слышала о Kanban, но ей это может пойти на пользу, никакие многосторонние опросы или другие профессиональные техники коучинга не помогут вдруг нарисовать Kanban в голове участника обучения. В таком случае Agile-коуч переключается в режим, в котором предлагает поделиться потенциально полезным опытом. Если участники проявляют интерес, тогда коуч делится своими знаниями, стараясь вернуться в русло проводимого обучения поскорее, как только команда усвоит новые знания.

Существует шесть областей знаний, которые стараются дать Agile-коучи: набор методов Agile, переход на Agile, управление продуктом в Agile, технические методы Agile, ведение встреч и коучинг. У каждого коучи свой набор навыков. Большинство организаций начинают с поисков Agile-коучи, у которого есть опыт работы с методами Agile. По мере того как компании продвигаются в своих поисках, они приходят к тому, что ценные все области знаний.

Одной из областей компетенции, которую компании постоянно недооценивают, является необходимость для каждого, кто зани-

мается написанием кода и тестированием, научиться писать код и создавать тесты, подходящие для Agile-среды, как описано в этой книге. Это важно, чтобы сосредоточиться на добавлении новой функциональности с новыми тестами, а не на постоянном обновлении существующего кода и тестов и/или возрастании технического долга, что увеличивает скорость.

Коучинг в нескольких командах

Где-то в 2012 году, по мере того как все больше организаций успешно налаживало работу с командами, произошел огромный всплеск интереса в применении Agile в крупных масштабах. То есть переход организаций с традиционной основы на поддержку методологии Agile.

Сейчас большинство коучей по Agile проводят обучение в условиях нескольких команд, а иногда даже десятков и сотен. И часто все начинается с того, что работники изолированно распределены по трем или более не связанным между собой проектам.

Не все из этих «команд» работают вместе для достижения общей цели, но все они работают в традиционной среде, где мыслят категориями многолетнего финансирования, планирования портфелей и разработки проектов, вместо того чтобы ориентироваться на командный подход и выпуск качественного продукта.

Agile в крупных масштабах

Проблема Agile в крупных масштабах весьма похожа на проблему Agile на уровне команды. Извлечь пользу из Agile — значит найти и удалить за каких-то пару недель все препятствия, возникающие на пути у команды при согласовании общих усилий, для того чтобы

перейти от заказа до готового релиза. В этом и состоит трудность, но она преодолима. Еще труднее сделать так, чтобы команда выпускала релизы по требованию.

При попытках согласовать усилия нескольких команд для получения единого результата такие трудности преумножаются и увеличиваются. К несчастью, Agile в крупной организации обычно стараются внедрить традиционным способом организации проекта. То есть происходит командно-административное внедрение огромного количества изменений, выбранных для предварительного преобразования. И когда я говорю об огромном числе, я говорю буквально о тысячах изменений. Речь идет о тысячах, потому что когда вы просите сотни человек предпринимать десятки изменений в их повседневной работе, у них это может как получиться, так и нет, в зависимости от того, насколько сильно эти изменения бывают по каждому из них лично. Достаточно даже сказать, что план изучить тот крупный набор методов Agile выглядит примерно как «по плану нам нужно реализовать вот этот огромный ворох требований».

Из своего опыта работы со многими организациями, пытавшимися перейти на Agile (многие из них насчитывали сотни команд), и работы со многими опытными Agile-коучами я понял самое важное: проблема успешного перехода на Agile — это точно такая же проблема, что и создание программного обеспечения.

Лучше всего создавать ПО, полагаясь на частое взаимодействие с клиентом. Так и здесь: приживаются только те изменения, которые напрямую связаны с тем, что люди, находящиеся под их влиянием, хотят и понимают в зависимости от своих собственных обстоятельств. Другими словами, я считаю, что самая действенная стратегия перехода на Agile — это воспринимать внедрение Agile как дерзкое предприятие с применением профессионального коучинга.

Внедрение Agile с помощью Agile и коуча

Манифест Agile сам по себе замечательный шаблон для коучинга и согласования работы нескольких команд: «Дайте им среду и поддержку, в которой они нуждаются, и доверьте им выполнение работы». В подтверждение вышесказанного у сообщества Agile есть несметное число паттернов для масштабирования, которые совместимы с ценностями и принципами Манифеста Agile. Говоря это, я имею в виду не наборы методов, а отдельные методы, из которых эти наборы состоят.

Все эти наборы суть готовые рецепты, состоящие из отдельных методов Agile. Вместо того чтобы действовать по одному из этих рецептов, можно подготовить свой собственный рецепт с Agile и коучем, который безупречно подходит именно вам. Если по вашему рецепту получается SAFe, Nexus, LeSS или Scrum@Scale, то замечательно!

Мы приводим краткий обзор того, как самые успешные Agile-коучи, работающие с крупными предприятиями, сочетают свое ремесло и Agile, и это лучшим образом оказывается на организации. На уровне отдельных людей смысл коучинга в том, чтобы помочь им решать проблемы самостоятельно. А коучинг на уровне команд и организаций помогает самостоятельно достигать своих целей целым командам.

Прежде всего, коуч рассматривает всех, кого затронет переход на Agile, в качестве клиентов. Затем, проводя ретроспективы, мероприятия в опенспейсах и прочее, он выясняет, что клиенты считают вызовами и возможностями. Становится понятно, сколько работы нужно провести, чтобы внедрить Agile. Затем с помощью групповых средств принятия решений, например точечного голосования, коуч определяет, с чего нужно начинать в первую

очередь. Потом он помогает организации провести несколько наиболее важных изменений. Затем проводит ретроспективу и повторяет действия.

Конечно, для многих участников такое внедрение Agile будет происходить впервые. Одного коучинга недостаточно, важно также проводить обучение и тренинги, чтобы сотрудники могли принимать решения, будучи достаточно осведомленными.

Наращивание внедрения Agile

Ниже приведен список отдельных методов для внедрения Agile. Этот список был изначально создан и периодически обновлялся посредством трех главных ступеней в Agile-коучинге — устранения дублей, сбора идей на стикерах и точечного голосования при участии группы из примерно десятка корпоративных коучей. Для справки здесь приведено обобщенное описание этих методов. Существует гораздо больше методов Agile, которые здесь не перечислены. Рассмотрим для начала этот список. Например, вместо того чтобы внедрять Scrum, Kanban, экстремальное программирование или один из наборов методов для масштабирования Agile, подумайте, какой метод из списка ниже наиболее уместен для текущих потребностей той или иной группы или команды, и внедрите его. Попробуйте его применять некоторое время, потом повторите действия.

- **Практики Kanban:** методы Kanban основаны на наглядности хода работ (с помощью карточек на стене), ограничении количества выполняемых работ и прохождении работы через разные стадии.
- **Scrum и экстремальное программирование (XP):** эти две методологии часто увязывают вместе, потому что они очень похожи,

за исключением технических методов XP. В SAFe, например, их упоминают совместно как ScrumXP. Обе методологии включают в себя большое разнообразие методов, например короткие ежедневные собрания команды, «владелец продукта», «facilitator процесса» (он же «скрам-мастер»), ретроспективы, кросс-функциональность команд, пользовательские истории, небольшие релизы, рефакторинг, заблаговременное написание тестов и парное программирование.

- **Распределение командных событий;** когда командные события, такие как стендап-митинги и ретроспективы, распределены по времени между несколькими командами, тогда возможно поднимать ежедневные и системные препятствия по дереву эскалации. Такое распределение по времени задает время начала и конца итераций, а также их продолжительность. Команды, не работающие по итерациям, которые могут выпускать релизы по требованию, могут соотносить свой рабочий график с любой другой каденцией.
- **Деревья эскалации:** если есть смысл всегда работать над чем-либо, что приносит наибольшую пользу, тогда есть смысл безотлагательно поднимать препятствия по строго намеченному пути эскалации. Они применимы к широко используемому методу Scrum of Scrums и не такому известному retrospective of retrospectives. Одним из паттернов для этого является фрактальный паттерн масштабирования для Scrum@Scale посредством Scrum, Scrum of Scrums и даже Executive Action Team.
- **Регулярное межкомандное взаимодействие:** этот метод предполагает регулярное взаимодействие между мастерами Scrum, владельцами продукта и членами команды, которые работают сообща на результат. Один из способов это обеспечить — проводить регулярные события на открытом пространстве.

- **Kanban портфеля:** традиционные способы управления портфелями способствуют распределению работников по нескольким командам, что ведет к неконтролируемой многозадачности. Многозадачность создает трение, увеличивает сложность и снижает производительность. Kanban портфеля накладывает ограничения на количество выполняемых работ на уровне инициативы и обеспечивает постоянное внимание на работе, приносящей наибольшую пользу. Одновременное управление меньшим количеством проектов также значительно упрощает (или даже решает) проблему согласования нескольких команд. Kanban портфеля лучше всего работает в паре с наименее возможным приростом (Minimum Viable Increment).
- **Наименее возможный прирост:** существует много вариантов развития этой идеи, но все они сводятся к обдумыванию того, какой путь короче и позволит скорее получить наибольшую пользу. Растущее число организаций принимают другую крайность — внедряют непрерывную доставку, регулярно и часто выпускают небольшие обновления, иногда по несколько раз на день.

Добиваться большого, сосредоточившись на меньшем

Большинство случаев внедрения Agile в нескольких командах одновременно сталкивается с проблемой того, что их члены больше думают о преодолении сложности, а не о решении задач для достижения простоты. По своему опыту могу сказать, что один из краеугольных камней применения Agile в крупных масштабах — это высокий уровень применения Agile на уровне команд и очень низкий уровень сложности во всех структурах. Когда у вас целая флотилия быстроходных лодок, то практически незачем связы-

вать их вместе. Ниже приведены некоторые методы, как правило, ассоциирующиеся с Agile на уровне команды, которые выполняют функцию двойного назначения в качестве инструмента согласования работы нескольких команд.

- **Принципы SOLID:** хотя эти принципы важны на любом уровне организации, они особенно полезны для упрощения согласования работы нескольких команд посредством значительного сокращения зависимостей.
- **Небольшие и ценные пользовательские истории:** небольшие, самостоятельно выпускаемые истории ограничивают количество зависимостей, а это упрощает согласование нескольких команд.
- **Небольшие и частые релизы:** независимо от того, будут ли эти релизы предоставлены клиенту, практика наличия продукта, готового к релизу, у всех команд помогает выявлять проблемы координации и архитектуры. Появляется возможность отыскать и устранить корень проблемы. Некоторые команды, работающие по Scrum, забывают об этом, но в самом Scrum говорится: «На каждом этапе продукт должен быть в рабочем состоянии, независимо от того, решит ли владелец продукта выпустить релиз». Это означает, что нужно согласовать работу команды с работой других команд, от которых зависит состояние продукта.
- **Непрерывная интеграция:** в экстремальном программировании делается еще больший упор на согласованность, этот метод призывает проводить слияние всего продукта после каждой отметки об изменении.
- **Простота проектирования:** этот метод также известен как «независимость проектирования», считается одним из труд-

нейших для изучения методов, поскольку он один из самых нелогичных. Команды тяжело справляются с этим методом, даже когда им не нужно согласовывать свои действия с другими командами. При согласовании работы нескольких команд монолитные, централизованные, заранее спланированные архитектуры создают огромные зависимости между командами, которые, как правило, заставляют их работать в тесной связи между собой, что нарушает большую часть обещаний Agile. Простота проектирования, особенно в сочетании с такими методами, как микросервисная архитектура, позволяет применять Agile в крупных масштабах.

Будущее Agile-коучинга

В последние несколько лет профессиональный коучинг и фасilitation все прочнее занимают свое место среди дисциплин Agile. На курсах скрам-мастера с расширенной сертификацией (ACSM), проводимых Scrum Alliance, есть несколько учебных задач, связанных с коучингом и фасилитацией, а программы «сертифицированный командный коуч» (CTC) и «сертифицированный корпоративный коуч» (SEC) требуют усвоения еще большего количества навыков фасилитации и коучинга. Руководство по Scrum дает определение скрам-мастера как того, кто проводит коучинг.

Поскольку все больше людей проходят курсы профессионального коучинга и встречают профессиональных коучей, которые работают в сообществе Agile, коучинг в Agile привлекает к себе все больше внимания.

Кажется, в последние пару месяцев наблюдается рост интереса к профессиональному коучингу. Люди все чаще пропускают обучение по программе ICP-ACC и сразу идут на обучение по ICF.

Появилась первая коучинговая школа Agile, аккредитованная ICF, и скоро появится, по крайней мере, еще одна. Agile-коучинг ждет большое будущее!

ЗАКЛЮЧЕНИЕ (СНОВА БОБ)

Во многих отношениях эта глава была больше о том, чего не нужно делать, чем о том, что стоит. Возможно, потому что я видел много примеров того, как не нужно переходить на Agile. В конце концов, я сейчас думаю так же, как и 20 лет назад: «Что может быть проще? Всего лишь соблюдать простые правила и применять методы. Вот и все».

МАСТЕРСТВО ВЫСШЕГО УРОВНЯ

Автор Сандро Манкузо, 27 апреля 2019 года



Волнение. Его чувствуют многие разработчики, когда впервые слышат об Agile. Для большинства из нас, разработчиков, которые приходят с фабрик программного обеспечения с менталитетом каскадной модели, Agile оставался надеждой на избавление от гнета. Надеждой на то, что мы будем работать сообща, к нашему мнению будут прислушиваться и уважать его. У нас будут модели и методы лучше, чем были. Мы будем работать малыми итерациями и безотлагательно давать обратную связь. Мы будем регулярно выпускать релизы нашего приложения. Мы будем поддерживать общение и обратную связь с пользователями. Мы будем постоянно анализировать и приспосабливать наши способы работы. Мы будем вовлечены с самого начала разработки. Мы будем ежедневно на связи с клиентами. Мы на самом деле будем одной командой. Мы будем регулярно обсуждать деловые и технические вопросы и договариваться о том, как двигаться вперед, к нам будут относиться как к профессионалам. Бизнес и технологии будут работать сообща на производство замечательных программных продуктов, приносящих пользу нашим компаниям и клиентам. В самом начале нам казалось, что Agile слишком хорош, чтобы быть правдой. Мы думали, что наши компании никогда не примут мировоззрение Agile, не говоря уже о самих методах.

Но большинство из них смогли это сделать и были приятно удивлены. Вдруг все изменилось. У нас появились списки невыполненных задач и пользовательские истории вместо документов с требованиями. У нас появились настоящие доски и диаграммы сгорания задач вместо диаграмм Ганта. У нас появились стикеры, которые мы передвигали каждое утро в соответствии с ходом работ. В стикерах чувствовалась какая-то мощная энергетика — что-то, что вызывало сильную психологическую зависимость. Они как будто представляли нашу приверженность Agile. Чем больше таких заметок было на стене, тем более вовлечеными в Agile мы себя

ощущали. Мы стали командой, работающей по Scrum, а не бригадой строителей. У нас больше не было менеджеров проекта. Нам сказали, что нам не нужны менеджеры, теперь наши менеджеры будут «владельцами продукта», а у нас будет самоорганизация. Нам сказали, что владельцы продукта и разработчики будут тесно сотрудничать, словно одна команда. И отныне, поскольку наша команда работает по Scrum, мы наделены правом принятия решений. Не только технических, но и тех, что связаны с самим проектом. Или мы так думали.

Agile бурей захватил отрасль программного обеспечения. Но, как и в игре в испорченный телефон, изначальный посыл Agile искали и упростили, преподнося его компаниям как методологию, которая *ускорит выпуск программного обеспечения*. Для компаний и руководителей, применяющих каскадную модель или RUP, это звучало райской песнью.

Менеджеры и заинтересованные лица пришли в восторг. В конце концов, кто бы не хотел испробовать Agile? Кто бы не хотел быстрее выпускать программное обеспечение? Даже среди скептиков Agile вызывал интерес. Если ваши конкуренты хващаются тем, что используют Agile, а вы нет, что вам мешает? Что о вас подумаю ваши потенциальные клиенты? Компании не могут позволить себе отказаться от Agile. За годы, последовавшие за саммитом Agile, компании по всему миру встали на путь перехода к Agile. Началась эра внедрения Agile.

ПОХМЕЛЬЕ ОТ AGILE

Переход из одной культуры в другую был непрост. Компаниям требовалась помочь извне, чтобы осуществить его в своих организациях. Появился новый вид специалистов — Agile-коучи.

Было создано много различных программ сертификации. Некоторые сертификаты можно получить, просто пройдя двухдневные курсы.

Продать методы Agile менеджерам среднего звена было легко — всем им хотелось, чтобы ПО выпускалось быстрее. «Инжениринг — это несложно. Если наладить процесс разработки, с ним тоже будет все в порядке, — говорили менеджерам. — Дело всегда в людях». И они покупали. Руководители работают с людьми и покуда занимают свою должность, они счастливы, когда их подчиненные работают быстрее.

Множество компаний на самом деле получили пользу от перехода на Agile, и сегодня их положение дел гораздо лучше, чем до этого.

Многие из этих компаний могут развертывать ПО несколько раз в день, бизнес и технологии у них работают действительно как одна команда. Но так, конечно, далеко не у всех. Менеджеры в попытке ускорить разработчиков используют полную прозрачность процесса для контроля каждого шага. Agile-коучи, у которых нет опыта ведения бизнеса и опыта технических работ, обучают *менеджеров* и говорят разработчикам, что им делать. Дорожные карты и вехи определяются менеджерами и навязываются командам разработчиков — разработчики могут оценивать работу, но их заставляют вписывать свои оценки в навязанные вехи. Довольно часто можно встретить проекты, в которых используются соответствующие итерации и пользовательские истории, уже распределенные руководством на следующие полгода-год. Если разработчик не в состоянии уgnаться за всеми единицами сложности историй за спринт, ему придется работать в следующем спринте больше, чтобы наверстать упущенное. Ежедневные стен-дап-митинги становятся встречами, где разработчики должны делать доклад о ходе работ владельцам продукта и Agile-коучам,

подробно рассказывая о том, над чем они работают и когда эти работы закончат. Если владелец продукта думает, что разработчики тратят слишком много времени на автоматизированные тесты, рефакторинг, парное программирование или что-то подобное, он просто команде запрещает это делать.

В их модели Agile нет никакой стратегическо-технической работы. В ней нет требований к архитектуре или проектированию. Порядок таков, что нужно просто сосредоточиться на какой-либо невыполненной работе из списка, которую нужно выполнить немедленно и которой присвоили наивысший приоритет. И так одно задание с наивысшим приоритетом следует за другим. Такой подход приводит к длинной последовательности итеративных тактических работ и накоплению технического долга. Хрупкое программное обеспечение, знаменитые монолиты (или распределенные монолиты, если говорить о командах, которые пробуют использовать микросервисную архитектуру) становятся в порядке вещей. Ошибки и неполадки оказываются излюбленной темой для обсуждения на ежедневном стендалап-митинге и ретроспективах. Релизы выходят не так часто, как ожидали клиенты. Тестирование вручную занимает целые дни, а то и недели. И надежда на то, что применение Agile убережет от всех напастей, бесследно уходит. Менеджеры обвиняют разработчиков, что те слишком медленно работают. Разработчики обвиняют менеджеров, что те не дают им проводить необходимые стратегические и технические работы. Владельцы продукта не считают себя частью команды, поэтому не берут на себя никакой ответственности за то, что дела пошли не так. Начинает преобладать порядок «свои против чужих».

Это то, что мы называем похмельем от Agile. После долгих лет вложения средств в переход на Agile компании понимали, что у них до сих пор много тех же проблем, которые были до него. И конечно, во всем виноват Agile.

ОЖИДАНИЕ И РЕАЛЬНОСТЬ

Переход на Agile, который полностью сосредоточен на процессе, нельзя назвать полным переходом. В то время как коучи по Agile направляют менеджеров и команды поставщиков в работе по Agile, никто не помогает разработчикам изучать технические методы Agile и инжиниринг. Предположение о том, что налаживание сотрудничества между людьми улучшит показатели по инжинирингу, в высшей мере ошибочно.

Слаженное сотрудничество убирает некоторые барьеры, которые мешают работать, но не обязательно добавляет мастерства работникам.

С внедрением Agile появляются и большие надежды: команды разработчиков должны выпускать ПО, готовое к релизу в производство, сразу как реализована какая-либо функция или, по крайней мере, в конце каждой итерации. Для большинства команд разработчиков это изменение значительно. Нет для них иного пути перехода на Agile, кроме изменения подхода к работе, а это означает, что нужно изучать и совершенствовать новые методы. Но встает несколько вопросов. Как правило, во время перехода на Agile на повышение квалификации разработчиков не выделяется бюджет. Клиенты не учитывают снижения темпов разработчиков при переходе на Agile. Большинство даже не знает, что разработчикам нужно изучить новые методы. Им сказали, что если они будут лучше сотрудничать, то разработчики будут работать быстрее.

Выпуск ПО в продакшен каждые две недели требует большой дисциплины и развитых технических навыков, тех навыков, которых обычно не отыщешь в командах, выпускающих ПО несколько раз в год. Все становится намного хуже, когда предполагается, что несколько команд с внушительным числом разработчиков в каждой

и работающих над одними и теми же системами, будут выпускать ПО в продакшен сразу, как только реализуют какие-либо функции.

Уровень мастерства команд в технических практиках и инжиниринге должен быть высоким, чтобы развертывать ПО в продакшен несколько раз в день, при этом не подрывая стабильность всей системы. Разработчики не могут просто выбрать что-то из списка невыполненных задач, начать писать код и думать, что все будет хорошо, когда релиз пойдет в производство. Им нужно стратегическое мышление. Им нужно модульное проектирование с возможностью параллельной работы.

Им нужно постоянно принимать изменения и при этом обеспечивать постоянную возможность развернуть систему. Для этого им постоянно нужно создавать ПО — и гибкое, и надежное. Но сохранять равновесие между гибкостью, надежностью и необходимостью непрерывно развертывать ПО в продакшен в высшей мере тяжело, и такого равновесия нельзя достичь без необходимых навыков инжиниринга.

Нельзя думать, что команды смогут развить эти навыки просто благодаря комфортной и сплоченной обстановке. В обретении этих технических навыков командам нужна поддержка. Эту поддержку можно оказать сочетанием коучинга, тренингов, экспериментирования и поощрения самообразования. Agile в бизнесе прямо связан с тем, как быстро компании могут выпускать ПО, а это означает эволюцию их навыков инжиниринга и технических методов.

ВСЕ ДАЛЬШЕ ДРУГ ОТ ДРУГА

Конечно, не каждый случай перехода на Agile сопровождается всеми проблемами, описанными выше, или, по крайней мере, не в та-

кой степени. С деловой точки зрения, честно сказать, большинство компаний, которые перешли на Agile хотя бы частично, сегодня находятся в лучшем положении. Они теперь работают короткими итерациями. Бизнес и технологии работают более слаженно, чем раньше. Проблемы и риски обнаруживаются преждевременно.

Предприятия чаще реагируют на новую информацию по мере ее поступления, по-настоящему выигрывая от итеративного подхода в разработке. Однако несмотря на то что компании действительно работают лучше, чем раньше, раскол между методами Agile и навыками инжиниринга до сих пор наносит им вред. У большинства современных коучей недостаточно (если вообще есть) технических навыков для обучения разработчиков техническим методам, и они редко говорят об инжиниринге. На протяжении многих лет разработчики стали считать Agile-коучей дополнительной прослойкой менеджмента: коучи говорят им, что делать, вместо того чтобы помочь развивать навыки.

Разработчики отдаляются от Agile или Agile отдаляется от разработчиков?

Вероятно, ответ на этот вопрос будет таким: и разработчики, и Agile. Кажется, что Agile и разработчики отдаляются друг от друга. Во многих организациях Agile и Scrum стали принимать за одно и то же. Экстремальное программирование, если оно вообще присутствовало, было представлено лишь несколькими техническими методами вроде разработки через тестирование и непрерывной интеграции. От коучей ждут обучения разработчиков некоторым методам экстремального программирования, но они на самом деле никак не помогают и не вовлекаются в то, чем занимаются разработчики. Многие владельцы продукта (или менеджеры проекта) до сих пор не ощущают себя частью команды и не чувствуют ответственности, когда что-то идет не по плану. Разработчикам до сих

пор приходится жестко настаивать в переговорах с представителями бизнеса на проведении необходимых технических улучшений для продолжения разработки и сопровождения системы.

Компании до сих пор недостаточно созрели для понимания того, что технические проблемы — на самом деле проблемы бизнеса.

Может ли Agile при снижении внимания на технических навыках значительно продвинуть выполнение проектов и обеспечить лучший результат, чем раньше? По-прежнему ли Agile сосредоточен на поиске эффективных способов разработки тем, ищет их и помогает в этом другим, как это написано в Манифесте Agile? Не могу уверенно сказать этого.

ВЫСШЕЕ МАСТЕРСТВО РАЗРАБОТКИ

Чтобы повысить планку профессиональных навыков разработки и восстановить некоторые изначальные цели Agile, группа разработчиков собралась на встрече в Чикаго в ноябре 2008 года, чтобы создать новое движение — мастеров разработки ПО (Software Craftsmanship). Эта встреча напоминала саммит Agile, который прошел в 2001 году, на ней разработчики утвердили основной набор ценностей и создали новый манифест¹ на основе Манифеста Agile:

Являясь устремленными к совершенству мастерами разработки ПО, мы повышаем уровень профессиональной разработки ПО, делая это сами и помогая другим осваивать наше ремесло. Занимаясь этой деятельностью, мы прежде всего научились ценить:

¹ <https://manifesto.softwarecraftsmanship.org/#/ru-ru>.

- Не только работающий продукт, но также и искусно разработанный продукт.
- Не только готовность к изменениям, но также и постоянное увеличение ценности.
- Не только людей и взаимодействие, но также и содружество профессионалов.
- Не только сотрудничество с заказчиком, но также и плодотворное партнерство.

Таким образом, в стремлении к тому, что слева, мы также считаем непременным следовать и тому, что справа.

Манифест мастеров разработки ПО содержит идеологию, менталитет. Он способствует профессионализму с разных точек зрения.

Искусно разработанный продукт означает хорошо спроектированный и протестированный код. Это код, в который мы не боимся вносить изменения, и код, который позволяет бизнесу быстрее реагировать на изменения. Это код и гибкий, и надежный.

Постоянное увеличение ценности означает, что независимо от того, чем мы занимаемся, мы всегда должны стараться непрерывно повышать ценность для наших клиентов и работодателей.

Содружество профессионалов означает, что мы должны обмениваться знаниями и учиться друг у друга, повышая тем самым уровень всей отрасли. Мы ответственны за подготовку следующего поколения разработчиков.

Плодотворное партнерство означает следование профессиональному в отношениях с клиентами и партнерами. Мы всегда будем, насколько возможно, проявлять этичность и уважительность в кон-

сультировании и работе с нашими клиентами и работодателями. Мы будем ожидать взаимного уважения и профессионализма, даже если нам придется взять на себя инициативу и показать пример.

Мы будем подходить к делу не как к чему-то, что просто нужно сделать по работе, а как к предоставлению профессиональных услуг. Мы возьмем карьеру в свои руки, будем вкладывать свое время и деньги, чтобы совершенствоваться. Эти ценности не только профессиональные, но и личные.

Мастера стремятся выполнять свою работу как можно лучше не потому, что кто-то за это платит, а потому что они сами хотят хорошо работать.

Тысячи разработчиков по всему миру немедленно подписались под принципами и ценностями, которые признаются мастерами разработки ПО. Изначальное волнение, которое разработчики почувствовали в начале появления Agile, не только вернулось, но и усилилось. Мастера, как они стали себя называть, решили больше не позволять спекулировать на их движении. Это движение разработчиков.

Движение, которое вдохновляет разработчиков работать как можно лучше. Движение, которое вдохновляет разработчиков становиться и чувствовать себя профессионалами высокого уровня.

ИДЕОЛОГИЯ ПРОТИВ МЕТОДОЛОГИИ

Идеология — это система идей и идеалов. А методология — система методов и практик. Идеология определяет идеалы, на которые нужно держать курс. Можно использовать одну или несколько методологий для достижения этих идеалов — они являются

средством достижения цели. Если посмотреть на Манифест Agile и его 12 принципов¹, мы можем явно разглядеть идеологию в этих строках. Главная цель Agile – это обеспечить гибкость бизнеса и удовлетворить запросы клиентов, а достичь этого можно через тесное сотрудничество, итеративный процесс разработки, быструю обратную связь и техническое совершенство. Методологии вроде Scrum, экстремального программирования (XP), метода разработки динамических систем (DSDM), адаптивной разработки ПО (ASD), методов Crystal, разработки, управляемой функциональностью (FDD), а также другие методологии Agile служат одной и той же цели.

Методологии и методы – как дополнительное колесико в детском велосипеде: они хороши поначалу. Как и в случае с ребенком, который учится кататься на велосипеде, такие колесики помогут научиться безопасно и легко кататься. Когда ребенок становится увереннее, мы поднимаем колесики, чтобы он мог потренироваться держать равновесие. Потом убираем одно из дополнительных колесиков. А за ним и другое.

Теперь ребенок может кататься без посторонней помощи. Но если мы слишком сильно сосредоточимся на важности тренировочных колес и не будем их долго убирать, ребенок привыкнет к ним и не захочет без них кататься. Чрезмерное внимание к методологии или набору методов отвлекает команды и организации от их действительных целей. Цель – научить ребенка ездить на велосипеде, а не внедрить тренировочные колеса.

Джим Хайсмит в своей книге *Agile Project Management: Creating Innovative Products* пишет: «Принципы без методов – ноль без

¹ <https://agilemanifesto.org/iso/ru/principles.html>.

палочки, в то время как методы без принципов, как правило, заучивают механически, без лишних раздумий. Принципы направляют методы. Методы воплощают принципы. Они идут рука об руку»¹. Хотя методологии и методы являются средством для достижения цели, мы не должны преуменьшать их важность. Профессионалов определяют по тому, как они работают. Мы не можем заявлять, что у нас есть какие-то принципы и ценности, если наши методы работы не согласуются с ними.

Хорошие специалисты могут точно сказать, как будут вести работу при тех или иных обстоятельствах. Они владеют широким набором методов и могут их применять в зависимости от потребностей.

ЕСТЬ ЛИ В МАСТЕРСТВЕ РАЗРАБОТКИ МЕТОДЫ?

В мастерстве разработки нет методов. Скорее оно способствует вечному поиску лучших методов и способов работы. Хорошие методы хороши до тех пор, пока мы не обнаружим новые, которые придут им на замену. Закрепление определенных методов за мастерством разработки ПО ослабило бы это мастерство с появлением новых методов. Но это не значит, что международное сообщество мастеров разработки не советует применять никакие методы. Наоборот, со времени создания в 2008 году и по сей день сообщество признает экстремальное программирование лучшим набором методов Agile для разработки ПО.

Разработка через тестирование, рефакторинг, простота проектирования, непрерывная интеграция и парное программирование высоко ценятся в сообществе мастеров разработки ПО, но это

¹ Highsmith J. Agile Project Management: Creating Innovative Products, 2nd ed. Boston, Massachusetts: Addison-Wesley, 2009. P. 85.

методы экстремального программирования, а не мастеров. И есть не только эти методы. Сообщество мастеров также одобряет принципы чистого кода и SOLID. Оно одобряет небольшие внесения изменений, небольшие частые релизы и непрерывную доставку. Оно одобряет модульность в проектировании ПО и автоматизацию, которая избавит от ручной и однообразной работы. И одобряет любые методы, которые повышают производительность, снижают риски и помогают произвести полезное, надежное и гибкое программное обеспечение.

Мастерство разработки ПО — это не только технические методы, инжиниринг и самосовершенствование. Это также профессионализм и помочь клиентам в достижении их деловых целей. И это как раз та область, где Agile, бережливое производство (Lean) и мастерство разработки прекрасно сочетаются. У всех трех концепций схожие цели, но они рассматривают проблему с разных, но одинаково важных точек зрения, которые друг друга дополняют.

СОСРЕДОТОЧЬТЕСЬ НА ЦЕННОСТЯХ, А НЕ НА МЕТОДЕ

Распространенная ошибка в сообществах Agile и мастеров разработки ПО в том, что они больше внимания уделяют методам, а не ценностям, которые лежат в основе этих методов. Возьмем, к примеру, разработку через тестирование. Один из наиболее частых вопросов, которые задаются в сообществах мастеров разработки ПО: «Как убедить моего руководителя/коллегу/команду применять разработку через тестирование?» Так вопрос ставить нельзя. Проблема здесь в том, что мы скорее предлагаем решение, чем принимаем проблему. Никто не станет работать по-другому, если не показать им ценности.

Вместо того чтобы силком тянуть к разработке через тестирование, можно найти согласие в том, что полезно будет сократить общее время тестирования программы. Сколько времени сегодня занимает тестирование? Два часа? Два дня? Две недели? Сколько людей этим занимается?

А что, если сократить время до 20 минут? Двух минут? Или даже 2 секунд? А что, если бы могли проводить его в любое время нажатием на кнопку? Даст ли это нам хорошую отдачу от вложений? Станет ли наша жизнь от этого легче? Сможем ли мы выпускать надежное ПО быстрее?

Если мы соглашаемся в том, что ответ будет «да», то можно уже говорить о методах, которые помогут нам достичь наших целей. Разработка через тестирование станет в этом случае естественным выбором. Тех, кому не нравится разработка через тестирование, мы спросим, какой метод они предпочитают. Какой метод, который позволит достичь поставленных целей с тем же или лучшим успехом, они смогут предложить?

При обсуждении методов необходимо в первую очередь договориться о целях. Единственное, чего не нужно допускать, — отказ от метода без предложения лучшей альтернативы.

ОБСУЖДЕНИЕ МЕТОДОВ

Обсуждение методов должно проходить на нужном уровне с компетентными людьми. Если мы хотим применять методы, которые улучшат сотрудничество в деловой и технологической областях, нам нужно вовлечь в обсуждение представителей этих областей. Если разработчики обсуждают методы, которые им понадобятся для улучшения процесса создания ПО, тогда не следует пригла-

шать к обсуждению посторонних. Представителей бизнеса стоит привлекать, только если изменения значительно коснутся стоимости или длительности проекта.

Существует разница между сменой монолитной архитектуры на микросервисную и разработкой через тестирование. Первое весьма значительно повлияет на стоимость и длительность проекта, последнее же не повлияет, пока разработчики не столкнутся с проблемами в области технического мастерства. Бизнесу не должно быть важно, автоматизируют ли разработчики свои тесты или нет. Это должно быть даже менее важно, чем то, написаны автоматизированные тесты до или после написания готового кода. Бизнесу лучше позаботиться о том, чтобы время от возникновения деловых идей до выхода программного обеспечения в производство постепенно уменьшалось. Сокращение количества средств и времени, потраченного на доработку (ошибки, ручная работа, такая как тестирование, развертывание и мониторинг производства), также является задачей бизнеса, в решении которой должны помочь команды разработчиков. Снижение затрат на эксперименты — тоже задача бизнеса. Эксперименты стоят очень дорого, когда ПО не является модульным и легко тестируемым. Представители бизнеса и разработчики могут вести разговоры о деловых ценностях, но не о технических методах.

Разработчики не должны спрашивать разрешения на написание тестов. У них не должно быть отдельных заданий для проведения модульных тестов и рефакторинга. У них не должно быть отдельных заданий для реализации какого-либо функционала. Такая техническая работа должна быть учтена при разработке каждой функции. Она обязательна. Руководство и разработчики должны обсуждать только то, что нужно будет выпустить и когда, а не способ выполнения. Каждый раз, когда разработчики добровольно

открывают секреты своей кухни, они дают повод руководителям перегибать палку в управлении.

Выходит, мы говорим о том, что разработчики должны скрывать ход своей работы? Вовсе нет.

Разработчики должны уметь доступно объяснить способы своей работы и преимущества этих способов всем, кому это может быть интересно. Что разработчики не должны позволять другим — так это решать за них, как им нужно работать. Разработчики и бизнес должны обсуждать «что», «зачем» и «когда», но вовсе не «как».

ВЛИЯНИЕ МАСТЕРСТВА НА ЛИЧНОСТЬ РАЗРАБОТЧИКА

Мастерство разработки ПО оказывает глубокое влияние на личность. Мы часто видим, как люди проводят разделение между личной жизнью и профессиональной деятельностью. Фразы вроде «я не хочу говорить о работе после того, как выйду из офиса» или «в жизни у меня другие интересы» произносят так, будто работа — это что-то плохое и скверное, или то, чем вы вынуждены заниматься без всякого желания.

Когда мы делим свою жизнь на несколько, эти жизни находятся в постоянном столкновении, что уже приносит проблемы. Ради одной жизни приходится жертвовать другой, независимо от той, что мы выберем.

Философия мастерства в том, что разработка — это профессия. Есть разница между наличием работы и профессии. Работа — это то, чем мы занимаемся, но это не часть нашей личности. Профессия же, с другой стороны, — часть нашего «я». Когда спрашивают про

род занятий, человек, который ходит на работу, обычно ответит что-то вроде «я работаю в компании такой-то» или «я работаю разработчиком программного обеспечения». Но человек, у которого есть профессия, ответит: «Я разработчик программного обеспечения». Профессия — это что-то, во что мы вкладываем душу. Это что-то, в чем мы хотим добиться большего. Мы хотим получить больше навыков, чтобы у нас был долгий и успешный трудовой путь.

Это не означает, что мы не будем проводить время с семьей, или что в жизни у нас больше нет других интересов. Скорее это означает, что мы будем искать равновесие между нашими обязательствами и интересами так, чтобы могли жить одной, гармоничной и счастливой жизнью. Однажды мы решим больше времени уделять семье, профессии или каким-то увлечениям. И это вполне нормально. В разное время у нас разные потребности. Но походы на работу не должны быть кошмаром, когда у нас есть профессия. Это просто еще кое-что, что доставляет нам удовольствие и разывает как личность. Профессия наполняет нашу жизнь смыслом.

ВЛИЯНИЕ МАСТЕРСТВА НА ОТРАСЛЬ РАЗРАБОТКИ

С 2008 года по всему миру растет число сообществ мастеров разработки ПО, организуются конференции, которые привлекают десятки тысяч разработчиков. В то время как в сообществах Agile больше внимания уделяется взаимодействию между людьми и процессу создания программного обеспечения, в сообществах мастеров больше внимания уделяется технической стороне вопроса. Они всегда были главными сторонниками экстремального программирования и многих других технических методов, распространяя их

среди большого числа разработчиков и компаний во всем мире. Именно благодаря сообществам мастеров разработки ПО многие разработчики стали изучать разработку через тестирование, непрерывную интеграцию, парное программирование, простоту проектирования, рефакторинг, принципы SOLID и чистого кода. Они также учатся создавать программы на основе микросервисной архитектуры, автоматизировать конвейеры развертывания и переносить программы в облако.

Они изучают новые языки и парадигмы программирования. Они изучают новые технологии и новые способы тестирования и сопровождения своих продуктов. Разработчики в сообществе мастеров создают безопасные и дружелюбные пространства, где могут встретиться с единомышленниками и поговорить о профессии.

В сообществах мастеров разработки ПО найдется место каждому. С самого начала одной из главных целей мастеров разработки было собрать самых разных разработчиков, для того чтобы они могли учиться друг у друга и повышать профессиональную планку.

В сообществах мастеров признаются участники любого уровня технического развития, на встречах приветствуется любой разработчик, независимо от его опыта. Сообщество преданно относится к подготовке нового поколения профессионалов, организует различные мероприятия, где люди, которые присоединяются к отрасли разработки, могут изучить основные методы искусственного создания ПО.

ВЛИЯНИЕ МАСТЕРСТВА НА КОМПАНИИ

Мастерство разработки ПО получает все большее признание. Многие компании, которые перешли на Agile, теперь смотрят в сторону

сообщества мастеров разработки, чтобы улучшить свои технические возможности. Однако мастерство разработки ПО не так привлекательно для бизнеса, как Agile. Экстремальное программирование для многих менеджеров — это часто то, что они не понимают или что вызывает у них тревогу. Руководство понимает Scrum, итерации, демонстрации, ретроспективы, сотрудничество и быструю обратную связь. Но им не особо интересны техники, связанные с программированием. Для большинства из них экстремальное программирование относится к написанию кода, а не к Agile.

В отличие от Agile-коучей начала 2000-х, у которых была хорошая техническая подготовка, большинство современных коучей не могут научить методам экстремального программирования или разговаривать с представителями бизнеса об инженерной стороне вопроса. Они не могут сесть рядом с разработчиком и программировать с ним в паре. Они не могут ничего рассказать о простоте проектирования или помочь с настройкой конвейера непрерывной интеграции. Они ничем не помогут разработчикам в рефакторинге уже написанного кода. Они не могут обсуждать стратегии тестирования и сопровождение нескольких сервисов в продакшене. Они не могут объяснить представителям бизнеса преимущества определенных технических методов, не говоря уже о каких-то технических стратегиях.

Но компаниям нужны надежные системы — системы, которые позволяют им реагировать быстрее, исходя из своих деловых потребностей. Компаниям также нужны целеустремленные и способные технические команды, которые могут хорошо создавать и сопровождать ПО. И это как раз те области, в которых мастера разработки выгодно отличаются.

Мышление мастера разработки ПО — источник вдохновения для многих разработчиков. Оно дает им чувство цели, гордости и про-

буждает врожденное стремление к совершенству. Большинство разработчиков, как и вообще люди, любят учиться и выполнять работу на совесть, просто им нужна поддержка и благополучное окружение. Компании, которые приняли идеи мастерства разработки, видят, как процветают сообщества мастеров внутри них. Разработчики организуют свои сессии, на которых вместе пишут код, практикуют разработку через тестирование и совершенствуют навыки проектирования. Им становится интересно изучать новые технологии и модернизировать системы, над которыми они работают. Они обсуждают, как лучше усовершенствовать базу кода и сократить технический долг. Мастерство разработки ПО поощряет культуру обучения, что делает компании прогрессивными и чуткими к изменениям.

ВЫСШЕЕ МАСТЕРСТВО И AGILE

Некоторые из факторов, побуждающих создание движения мастеров разработки ПО, были связаны с разочарованием многих разработчиков от пути развития Agile. Из-за этого некоторые считали, что движение мастеров и Agile противоречили друг другу. Участники движения мастеров разработки ПО, которые также участвовали в движении Agile, критиковали Agile за слишком большое внимание к процессу разработки и нехватку внимания к инжинирингу. Участники движения Agile критиковали движение мастеров за слишком узкий подход и пренебрежение реальными деловыми и человеческими проблемами.

Хотя обе стороны выказывали некоторое беспокойство, большая часть разногласий была связана с племенными инстинктами и, собственно, с принципиальным расхождением мнений. В сущности, стремления обоих движений очень схожи. Оба движения стремятся к удовлетворенности клиентов, ценят тесное сотрудничество

и быструю обратную связь. Они стремятся к выпуску продукта высокого качества, полезности выполняемой работы и професионализму. Чтобы достичь большой гибкости в бизнесе, компаниям требуются не только сотрудничество и итеративный процесс, но и хорошие навыки инжиниринга. Сочетание Agile и философии мастеров разработки ПО — прекрасный способ ее достичь.

ЗАКЛЮЧЕНИЕ

На встрече в Сноуберде в 2001 году Кент Бек выразил мысль, что одна из задач Agile — построить мост над пропастью, разделяющей бизнес и разработчиков. К сожалению, когда менеджеры проекта наводнили сообщество Agile, разработчики, которые в первую очередь создали сообщество, почувствовали себя обездоленными и недооцененными. Таким образом, они ушли, чтобы основать движение мастеров разработки ПО. Выходит, что старые терки никуда не делись.

И как бы то ни было, цели обоих сообществ — Agile и мастеров — почти одни и те же. Эти два движения не должны идти раздельно. Можно только надеяться, что однажды они снова воссоединятся.

ЗАКЛЮЧЕНИЕ



Вот такие дела. В этой книге — мои воспоминания, мнение, всякие разглагольствования и бредни об Agile. Надеюсь, вам было интересно, и вы почерпнули даже что-то полезное для себя.

Возможно, Agile — наиболее значительная и устойчивая революция в методах разработки ПО на нашей памяти. Такая значимость и устойчивость — свидетельство того, что те 17 ребят в феврале 2001 года в Сноуберде, что в Юте, спустили снежный ком вниз с очень высокого холма. Нестись на этом коме, наблюдая за тем, как он растет и набирает скорость, как сносит валуны и деревья, мне было очень весело.

Я написал эту книгу, потому что подумал, что наступило время, когда кто-то должен встать и сказать о том, каким Agile был и каким он должен быть и сейчас. Я посчитал, что настало время вернуться к основам.

Эти основы были, есть и будут теми самыми методами из круга жизненного цикла Рона Джейфриса. Эти основы представляют собой ценности, принципы и методы из книги Кента Бека *Extreme*

*Programming Explained*¹. Эти основы — также те самые устремления, техники и методы из книги Мартина Фаулера *Refactoring*². Эти основы были изложены Бучем, Демарко, Йорданом, Константином, Пейдж-Джонсом и Листером.

О них кричали Дейкстра, Даль и Хоар. Вы слышали о них от Кнута, Мейера, Якобсена и Рамбо. Им вторили Коплин, Гамма, Хелм, Влиссидес и Джонсон. Если вы внимательно слушали, то услышали, как о них шептали Томпсон, Ричи, Керниган и Плаугер. И где-то улыбались Черч, фон Нейман и Тьюринг, когда эти эхо и шепот касались их ушей.

Эти основы старые, проверенные и правильные. Неважно, сколько пуха накинули по краям, эти основы никуда не исчезли, они до сих пор актуальны и до сих пор составляют ядро гибкой методологии разработки Agile.

¹ Beck K. Extreme Programming Explained: Embrace Change. Boston, Massachusetts: Addison-Wesley, 2000.

² Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. Boston, Massachusetts: Addison-Wesley, 2019.

ПОСЛЕСЛОВИЕ

Автор Эрик Кричлоу, 5 апреля 2019 года



Я могу легко вспомнить свою первую работу, где решили перейти на Agile. Был 2008 год. Нашу компанию приобрела крупная организация. Происходили значительные изменения в политике, делопроизводстве и штате сотрудников. Еще я могу вспомнить парочку других работ, где акцент ставился на методы Agile. Ритуалы соблюдались неукоснительно: планирование спринта, демонстрация, обзор спринта... В одной из этих компаний всех штатных разработчиков направили на двухдневные тренинги по Agile, где они получили сертификаты скрам-мастеров. Я был разработчиком мобильных приложений, и меня попросили написать мобильное приложение для игры в Agile-покер.

Но за 11 лет, с тех пор как я впервые познакомился с Agile, я работал в нескольких компаниях, где уже точно не помню, использовался Agile или нет. Возможно, потому что Agile стал настолько вездесущим, что его легко принять как данность и даже не думать о нем. Или, может быть, потому что до сих пор значительное количество организаций не перешло на него.

Когда я узнал об Agile, то не испытал особого восторга. У каскадной модели, возможно, есть свои проблемы, но в моей компании не тратили много времени на написание проектной документации. У меня, разработчика, в основном был такой порядок: мне в устной форме передавали, какой функционал нужен к следующему релизу, назначали дату выпуска релиза и отпускали на все четыре стороны колдовать. Это, конечно, могло приводить к лютому марафону на выживание, но зато я мог свободно выстраивать свои действия так, как хочу. Мне не нужно было часто давать отчеты и проводить анализ на ежедневных стендах-митингах, где пришлось бы объяснять, над чем я работал вчера и что я буду делать сегодня. Если я решал потратить неделю на изобретение колеса, мне в этом никто не мешал, никто не осуждал мой выбор,

потому что все находились в блаженном неведении относительно того, чем я занимался.

Бывший директор по разработке, который шефствовал над нами тогда, называл нас «писаками». Нам просто нравилось лупить по клавиатуре на Диком Западе разработки ПО. Он был прав. И в какой-то мере методы Agile, будучи чем-то новым, царили у нас в голове, склоняя нас к инакомыслию.

Agile пришлось потрудиться, чтобы завоевать мое доверие.

Было бы самонадеянно полагать, что Agile — это стандарт де-факто в отрасли разработки ПО или что все разработчики принимают его с распластанными объятиями. С другой стороны, было бы неуважением отрицать значимость Agile в мире разработки. Но что это вообще значит? В чем, собственно, его значимость?

Спросите разработчиков в какой-нибудь компании, работающей по Agile, что такое этот самый Agile. И ответ, скорее всего, будет совсем другим, чем ответ любого из тех, кто находится в должности выше менеджера. Возможно, именно здесь эта книга наиболее поучительна.

Разработчики понимают Agile как методологию для оптимизации процесса разработки и для того, чтобы сделать разработку более предсказуемой, практичной и управляемой. Вполне логично, что мы смотрим на него с этой точки зрения, потому что это та точка зрения, которая самым непосредственным образом влияет на нас.

По своему опыту могу сказать, что многие разработчики понятия не имеют о том, что менеджеры используют метрики и данные, полученные при работе по методам Agile. В некоторых компаниях команда разработчиков принимает участие во встречах, когда эти

метрики обсуждаются. Однако во многих других компаниях разработчики остаются в неведении, что такие обсуждения вообще есть. Более того, возможно, в каких-то компаниях таких обсуждений не проводят в принципе.

Хотя я уже давно знал об этой особенности Agile, я все еще видел пользу в том, чтобы понять изначальный замысел и ход мышления основателей методологии, о которой рассказывается в этой книге. Было бы здорово и увидеть основателей Agile просто как людей. Они были не какими-то суперархитекторами разработки, не были рукоположены магистром Ордена инженерной мысли или избраны широкими народными массами разработчиков, чтобы передать каноны. Это были разработчики, наделенные опытом, обладавшие идеями, как облегчить себе жизнь и работу и избежать стрессов. Им надоело работать в командах, чья работа обречена на провал, поэтому им хотелось создать условия, способствующие благополучию.

Так можно сказать про большинство разработчиков, которых я встречал в каждой компании, где работал.

Если бы встреча в Сноуберде прошла на 15 лет позже, могло быть и так, что я бы сам организовал эту встречу и изложил бы те самые идеи, и на встрече было бы много разработчиков, с которыми я работал лично. Но будучи лишь еще одной группой опытных разработчиков, они были склонны к полетам фантазии, которые не всегда приживались в корпоративной реальности. Может быть, все это работает, как задумано, в мире высококлассных консультантов, наделенных властью выставлять требования и подчинять организации и руководство своим убеждениям, но большинство из нас — пехота, винтики в механизме фабрик по созданию программ. Нас можно заменить, у нас мало рычагов влияния. Поэтому когда дело доходит до таких штуковин, как «Билль о правах», мы по-

нимаем, что это идеал, но не то, что большинство из нас встречает в действительности.

Сегодня из сообществ в соцсетях я с радостью узнаю, что многие новые разработчики выходят за привычные рамки бакалавриата computer science и графика работы с девяти до пяти, что они сотрудничают с другими разработчиками по всему миру, учатся, применяют свои знания и опыт, чтобы обучать и вдохновлять новоиспеченных программистов.

Я весь в ожидании того, что следующая волна массовых изменений в методологии возникнет благодаря молодым звездам, которые смогут собираться вместе с помощью цифровых технологий.

Так что, пока мы ожидаем следующего большого события, которое принесет нам новое поколение, давайте уедим минутку и пересмотрим то, где сейчас находимся и с чем приходится иметь дело.

Теперь, когда вы прочитали эту книгу, у вас есть пища для размышлений. Рассмотрим Agile с тех сторон, о которых вы, возможно, знали, но о которых не особо задумывались. Подумайте о нем с точки зрения бизнес-аналитика, менеджера проекта или любого другого менеджера, непосредственно не связанного с разработкой, ответственного за планирование релизов или создание дорожных карт развития продукта. Подумайте над тем, какую пользу им приносит вклад разработчиков в методы Agile. Поймите, каким образом ваш вклад в работу влияет не только на вашу рабочую нагрузку в течение следующих двух недель. Затем вернитесь и снова просмотрите книгу. Если вы подойдете к ней с более широкой перспективы, думаю, вы по крупицам соберете еще больше полезных идей.

Как только вы это сделаете, попросите другого разработчика из компании прочитать эту книгу и провести такой же анализ. Може-

те дать почитать эту книгу даже кому-то... кто и вовсе не разработчик. Дайте ее кому-нибудь из тех, кто представляет вашу компанию на уровне бизнеса. Я почти гарантирую, что «Билль о правах» — это что-то из области того, о чем они никогда и не думали. Жить станет намного приятнее, если вы сможете до них донести, что эти права так же неотъемлемы для Agile, как и метрики, которые они получают при его использовании.

Можно сказать, что Agile стал чем-то вроде религии в области разработки. Многие из нас считают его лучшей практикой. Почему? Для многих — потому что так сказали. Это стало традицией: так надо. В понимании нового поколения корпоративных разработчиков просто так заведено. Они, и даже многие «старички», вообще не знают суть Agile — какие у него изначальные цели, задачи и методы. Можно что угодно говорить о религии, но истинные приверженцы — это те, кто старается понять то, во что они верят, помимо веры в то, о чем им говорят. Как и в случае с религией, нет единой общепринятой версии, которая подходит каждому.

Представьте, насколько большое значение имеет интерес к истокам своей религии, понимание событий и идей, которые образовали то, что впоследствии признали каноном. Когда речь заходит о профессиональной жизни, получается в точности то же самое. Делайте так же везде, где это того стоит: проповедуйте такой подход там, где имеете вес, восстановите первоначальную цель, цель, о которой вы и практически все, с кем вы когда-либо работали, мечтали, говорили и, вероятно, от которой в итоге отказались. Сделать достижимым успех в разработке программного обеспечения. Сделать достижимыми цели организации. Сделать процесс создания продукта лучше.

Роберт Мартин

Чистый Agile. Основы гибкости

Перевел с английского *И. Сигайлюк*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>Е. Тихонова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>М. Молчанова, Г. Шкатова</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 11.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 20.10.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



Р. Мартин

ЧИСТЫЙ КОД: СОЗДАНИЕ, АНАЛИЗ И РЕФАКТОРИНГ

КУПИТЬ

Плохой код может работать, но он будет мешать развитию проекта и компании-разработчика, требуя дополнительные ресурсы на поддержку и «укрощение». Каким же должен быть код? Эта книга полна реальных примеров, позволяющих взглянуть на код с различных направлений: сверху вниз, снизу вверх и даже изнутри. Вы узнаете много нового о коде. Более того, научитесь отличать хороший код от плохого, узнаете, как писать хороший код и как преобразовать плохой код в хороший.



Р. Мартин

ИДЕАЛЬНЫЙ ПРОГРАММИСТ. КАК СТАТЬ ПРОФЕССИОНАЛОМ РАЗРАБОТКИ ПО

КУПИТЬ

Всех программистов, которые добиваются успеха в мире разработки ПО, отличает один общий признак: они больше всего заботятся о качестве создаваемого программного обеспечения. Это — основа для них. Потому что они являются профессионалами своего дела. Книга насыщена практическими советами в отношении всех аспектов программирования: от оценки проекта и написания кода до рефакторинга и тестирования. Эта книга — больше, чем описание методов, она о профессиональном подходе к процессу разработки.



Р. Мартин

ЧИСТАЯ АРХИТЕКТУРА. ИСКУССТВО РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

КУПИТЬ

«Идеальный программист» и «Чистый код» — легендарные бестселлеры Роберта Мартина — рассказывают, как достичь высот профессионализма. «Чистая архитектура» продолжает эту тему, но не предлагает несколько вариантов в стиле «решай сам», а объясняет, что именно следует делать, по какой причине и почему именно такое решение станет принципиально важным для вашего успеха. Роберт Мартин дает прямые и лаконичные ответы на ключевые вопросы архитектуры и дизайна. «Чистую архитектуру» обязаны прочитать разработчики всех уровней, системные аналитики, архитекторы и каждый программист, который желает подняться по карьерной лестнице или хотя бы повлиять на людей, которые занимаются данной работой. Все архитектуры подчиняются одним и тем же правилам!