По книге: Вольфсон Борис. Гибкие методологии разработки. Версия 1.2

2.Agile-методологии

Семейство гибких методологий буквально ворвалось на софтверную сцену и перевернуло всё с ног наголову:

- Мы стали сосредотачиваться **на людях и улучшении коммуникаций между ними**, вместо выстраивания сверхжестких процессов;
- Мы стали концентрироваться **на продукте**, вместо того чтобы писать изощренную проектную документацию, которую никто не читает;
- Мы больше не заставляем расписываться заказчика кровью, ограничивая его жесткими и неудобными условиями договоров, мы строим действительно **партнерские отношения** и выясняем, что хочет заказчик и что ему нужно;
- Мы всегда **готовы к изменениям**, потому что понимаем, что мир вокруг нас меняется и то, что месяц назад казалось абсолютно необходимым в нашем проекте, сейчас уже не нужно вообще.

В более строгом варианте эти тезисы были сформулированы отцами-основателями гибких методологий в документе, который получил название Agile Manifesto:

- **Люди и их взаимодействие** важнее процессов и инструментов;
- **Готовый продукт** важнее документации по нему;
- **Сотрудничество с заказчиком** важнее жестких контрактных ограничений;
- **Реакция на изменения** важнее следования плану.

Рисунок 1. Визуализация ценностей манифеста гибкой разработки

Полный текст манифеста (и его переводы) доступны на сайте http://agilemanifesto.org. Каждый, кто хочет работать по гибкой методологии должен ориентироваться на эти четыре «взвешивания»: как только начинает перевешивать не та «чаша весов», надо задуматься: «А на верном ли я пути?». Таким образом, манифест станет вашим компасом, по которому можно определять направление движения.

Принципы Agile

1. Наш высший приоритет – это удовлетворение заказчика с помощью частых и непрерывных поставок продукта, ценного для него.
2. Мы принимаем изменения в требования, даже на поздних этапах реализации проекта.
3. Гибкие процессы приветствуют изменения, что является конкурентным преимуществом для заказчика.
4. Поставлять полностью рабочее программное обеспечение каждые несколько недель, в крайнем случае, каждые несколько месяцев. Чем чаще, тем лучше.
5. Представители бизнеса и команда разработки должны работать вместе над проектом.
6. Успешные проекты строятся мотивированными людьми. Дайте им подходящую окружающую среду, снабдите всем необходимым и доверьте сделать свою работу.
7. Самый эффективный метод взаимодействия и обмена информацией – это личная беседа.
8. Рабочее программное обеспечение – главная мера прогресса проекта
9. Гибкие процессы способствуют непрерывному развитию. Все участники проекта должны уметь выдерживать такой постоянный темп.
10. Постоянное внимание к техническому совершенству и качественной архитектуре способствуют гибкости.
11. Простота необходима, как искусство максимизации работы, которую не следует делать.
12. Лучшая архитектура, требования, дизайн создается в самоорганизующихся командах.
13. Команда постоянно ищет способы стать более эффективной, путем настройки и адаптации своих процессов.

Авторы манифеста

В феврале 2001 года 17 специалистов (консультантов и практиков) в местечке под названием Snowbird в штате Юта собрались, чтобы обсудить легковесные методики разработки. В результате родился документ «Манифест гибких методологий разработки» (Agile Manifesto). Позволю себе привести список авторов манифеста и краткую информацию о них.

Кент Бек – создатель разработки через тестирование и экстремального программирования. Автор нескольких книг на эти темы и соавтор JUnit.

Кен Швабер – президент компании Advanced Development Methods (ADM), которая занимается улучшением методов разработки ПО. Он является опытным разработчиком, менеджером продуктов и консультантом. В начале 90-х годов он работал с Джеффом Сазерлендом над первыми версиями Scrum. Он также является соавтором книги «Scrum, Agile Software Development».

Джефф Сазерленд – технический директор (CTO) компании PatientKeeper, которая занимается создание ПО для медицинских учреждений. За свою карьеру был техническим директором (или вице-президентом по технологиям) в девяти компаниях. Свою известность приобрел, как изобретатель методологии Скрам.

Дейв Томас верит в то, что сердцем проекта по разработке является не методология, а люди. По этой причине он является соавтором книги «The Pragmatic Programmer».

Scrum в двух словах

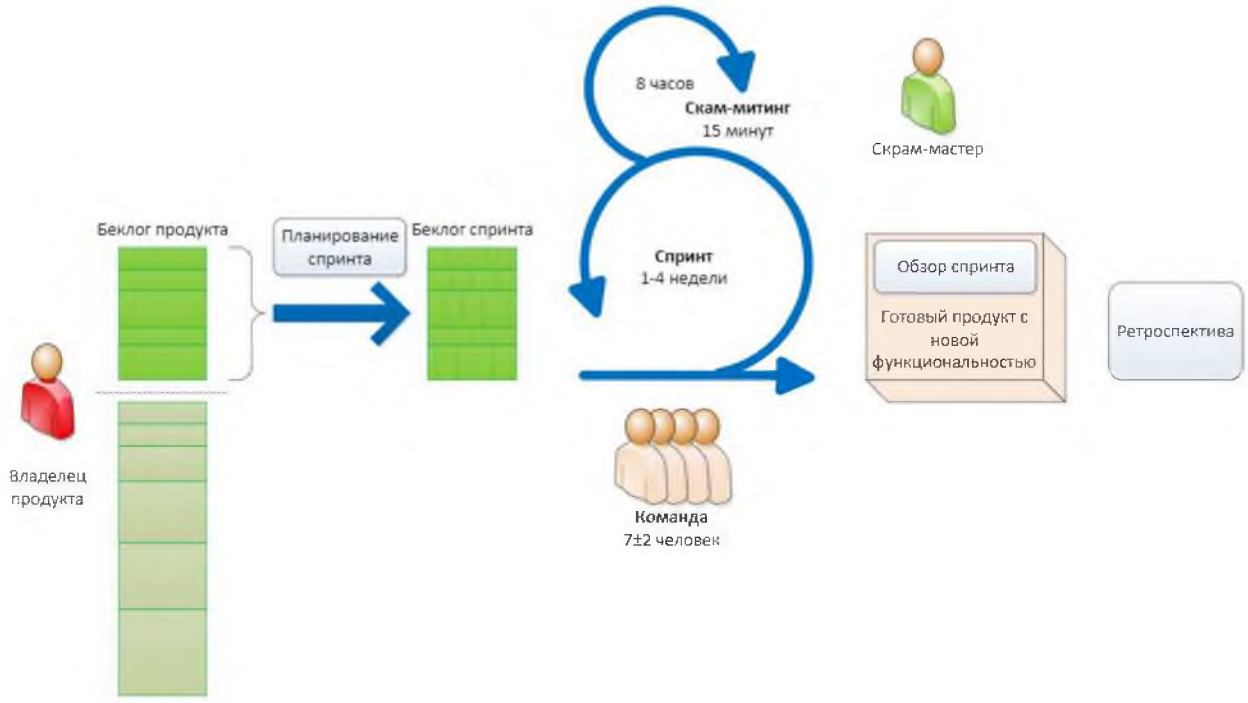

Рисунок 2. Общая схема Scrum

Организуйте работу в вашей организации в небольших кроссфункциональных командах, которые содержат всех необходимых специалистов. Выделите человека - скрам-мастера, который будет отвечать за соблюдение процессов в команде и конструктивную атмосферу.

Требования разбейте на небольшие, ориентированные на пользователей, функциональные части, которые максимально независимы друг от друга, в результате чего получите беклог продукта. Затем упорядочите элементы беклога по их важности и произведите относительную оценку объемов каждой истории. Выделите отдельного человека – владельца продукта, который будет отвечать за требования и их приоритеты, замыкая на себя всех заинтересованных лиц.

Всю работу ведите короткими (от 1 до 4 недель) фиксированными итерациями – спринтами, поставляя в конце каждого из них законченный функционал, который можно при необходимости вывести на рынок – инкремент продукта. Команда согласно своей скорости набирает задачи на неизменяемую по времени итерацию – спринт. Каждый день проводится скрам-митинг, на котором команда синхронизирует свою работу и обсуждает проблемы. В процессе работы члены команды берут в работу элементы беклога согласно приоритету.

В конце каждого спринта проводите обзор спринта, чтобы получить обратную связь от владельца продукта, и ретроспективу спринта, чтобы оптимизировать ваши процессы. После этого владелец продукта может изменить требования и их приоритеты и запустить новый спринт.

Не Scrum'ом единым

Scrum не является единственной гибкой методологией, хотя на данный момент он самый популярный среди собратьев. Но знание и понимание других методологий важно, чтобы понимать их преимущества и недостатки. Кроме того, на поздних этапах адаптации Scrum можно позаимствовать различные практики из этих методологий.

Также хотелось бы поделиться результатами исследования Agile Survey о популярности гибких методологий:

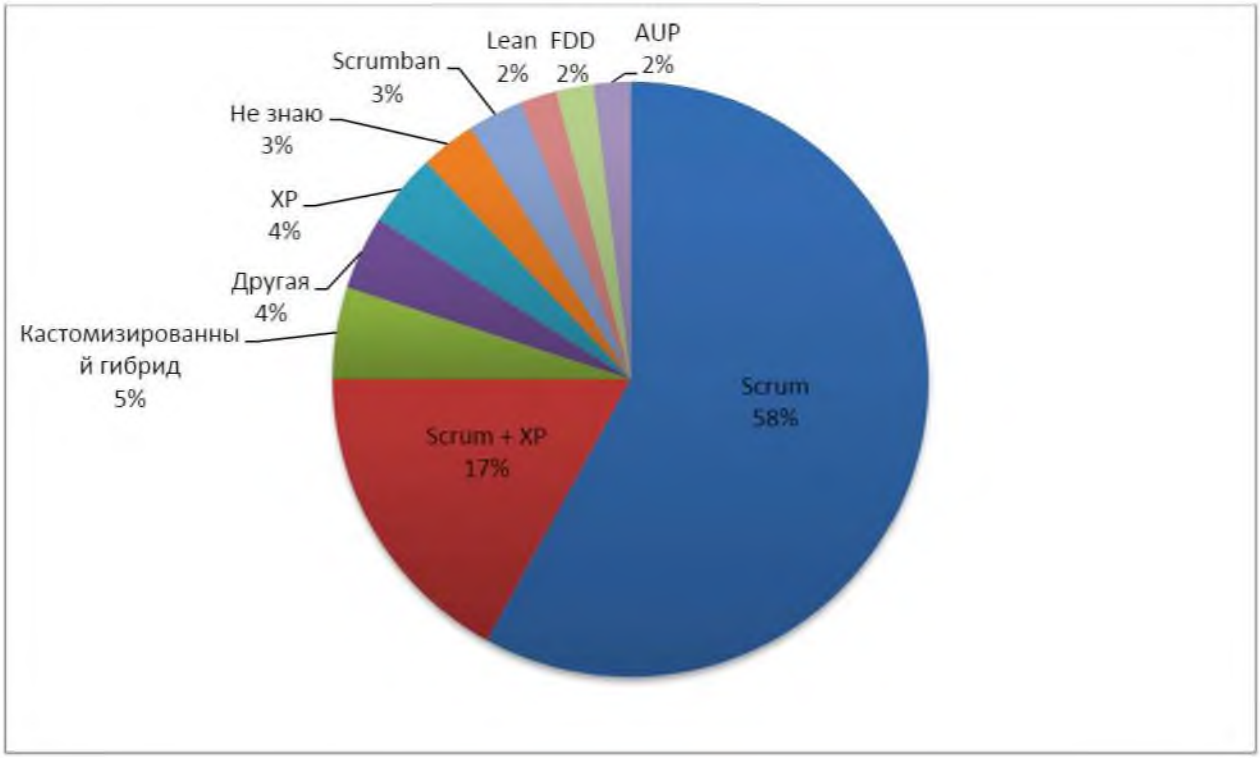

Рисунок 3. Популярность гибких методологий

Экстремальное программирование

Практики экстремального программирования можно разделить условно на управленческие и инженерные:

Рисунок 4. Практики экстремального программирования

Разделение весьма условное, но оно показывает, что экстремальное программирование имеет много общего со Scrum, но имеются и определенные отличия. Инженерные практики будут рассмотрены подробнее в соответствующем разделе, а управленческие практики применяются обычно из Scrum.

Crystal Clear

Crystal Clear – это легковесная гибкая методология, созданная Алистером Коуберном (Cockburn, 2004). Она предназначена для небольших команд в 6-8 человек для разработки некритичных бизнес-приложений. Как и все гибкие методологии Crystal Clear больше опирается на людей, чем на процессы и артефакты.

Crystal Clear использует семь методов/практик, три из которых являются обязательными:

1. **Частая поставка продукта**
2. **Улучшения через рефлексию**
3. **Личные коммуникации**
4. Чувство безопасности
5. Фокусировка
6. Простой доступ к экспертам
7. Качественное техническое окружение

Как вы видите, все практики характерны для семейства Agile-методологий. В графическом виде практики Crystal Clear можно изобразить таким образом:

3. Scrum – гибкий управленческий фреймворк

Сегодня самой популярной гибкой методологией разработки ПО является Scrum. Если вы спросите любого практика Agile, он обязательно подтвердит это, хотя каждое слово в предыдущей фразе неправда.

Начнем с конца – Scrum используют не только для разработки ПО, он отлично подходит для многих процессов по созданию продукта: от венчурных до маркетинговых продуктов. И соответственно подбираемся ко второму пункту – Scrum вовсе не методология, это гибкий управленческий фреймворк. Откуда следует и третий пункт – Scrum обычно дополняют инженерными практиками из других гибких методологий (например, практики разработки из экстремального программирования, или практики анализа и сбора требований из ICONIX). Так что в дальнейшем, если не оговорено иное, под Agile будем подразумевать семейство гибких методологий, а Scrum будем рассматривать в качестве управленческого фреймворка, дополненного практиками из других гибких методологий. Но довольно буквоедствовать, давайте начнем знакомиться со Scrum.

> Это был первый случай в моей жизни, когда я увидел, как методология работает "прямо из коробки". Просто подключи и работай. И при этом все счастливы: и разработчики, и тестеры, и менеджеры. Вопреки всем передрягам на рынке и сокращению штата сотрудников, Scrum помог нам выбраться из сложнейшей ситуации, позволил сконцентрироваться на наших целях и не потерять свой темп.
>
> Хенрик Книберг,
> Scrum и XP: Заметки с передовой

Классический Scrum состоит из следующих элементов:

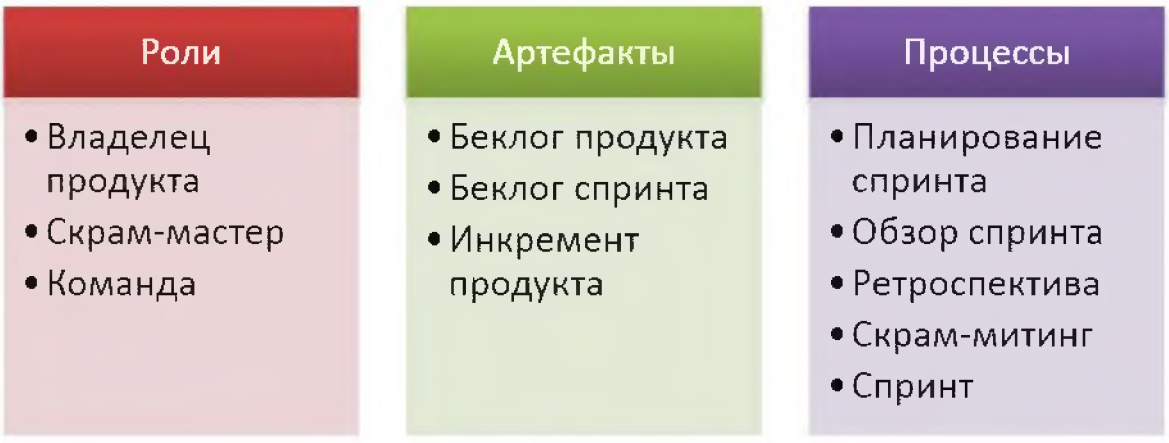

Рисунок 8. Элементы Scrum

Роли

В Scrum принято выделять три основных роли: владелец продукта, скрам-мастер и команда.

- **Владелец продукта** (Продукт оунер, Product owner, Менеджер продукта) – это человек, ответственный приоритезацию требований и часто за их создание.
- **Скрам-мастер** – член команды, который дополнительно отвечает за процессы, координацию работы команды и поддержание социальной атмосферы в команде.
- **Команда** – 7±2 человек, которые реализуют требования владельца продукта.

Обязанности скрам-мастера

Скрам-мастер должен ежедневно следить за тем, чтобы скрам-митинг начинался и заканчивался вовремя. Рекомендуется выделять определенное время каждому участнику, чтобы общая протяженность скрама-митинга не превышала заранее оговоренного времени (например, 15 минут).

Скрам-мастер в начале спринта помогает команде проводить планирование спринта и запуск спринта.

В конце спринта скрам-мастер организует демонстрацию результатов спринта при участии всех заинтересованных лиц и проводит ретроспективу при участии всех членов проектной команды.

Также в обязанности скрам-мастера входит мониторинг социальных аспектов команды и поддержание командного духа.

Артефакты

- **Беклог продукта (Product Backlog)** – приоритезированный список требований с оценкой трудозатрат. Обычно он состоит из бизнес-требований, которые приносят конкретную бизнес-ценность и называются элементы беклога.
- **Беклог спринта (Sprint Backlog)** – часть беклога продукта, с самой высокой важностью и суммарной оценкой, не превышающей скорость команды, отобранная для спринта.
- **Инкремент продукта** – новая функциональность продукта, созданная во время спринта спринта.

Процессы

Большинство процессов Scrum носят характер встреч, так как данная методология основана на качественных коммуникациях.

Скрам-митинг

Скрам-митинг (Scrum meeting, скрам, ежедневный скрам, планерка) – собрание членов команды (с возможностью приглашения владельца продукта) для синхронизации деятельности команды и обозначения проблем. Каждый член команды отвечает на три вопроса:

1. Что было сделано с предыдущего скрам-митинга?

2. Какие есть проблемы?
3. Что будет сделано к следующему скрам митингу?

Если первый и третий пункт служат для синхронизации деятельности команд, то второй пункт очень важен для выработки решений проблем: если проблема действительно небольшая, ее можно решить или выработать решение прямо на скрам-митинге, если серьезная и требует обсуждения, ее решить после скрам-митинга.

Планирование спринта

Для планирования спринта необходимо иметь качественный беклог, что означает следующее:

- все элементы бэклога должны иметь уникальную числовую важность;
- самые важные элементы бэклога должны быть уточнены и понятны всей команде и владельцу продукта;
- владелец продукта должен четко представлять, что будет реализовано в рамках каждого элемента бэклога.

Основным результатом планирования спринта является беклог спринта – список задач, которые команда планирует реализовать в рамках спринта. Поскольку длина спринта в Scrum жестко фиксирована, то команда определяет количество элементов беклога (объем работ), которые она может реализовать. Можно данную ситуацию отобразить на классическом «треугольнике управления проектами»:

Обзор спринта

Рисунок 9.Проектный треугольник в Scrum

Обзор спринта (также часто используется термин «демонстрация» или сокращенно «демо») – показ владельцу продукта (и заинтересованным лицам) работающего функционала продукта, сделанного за спринт. Основная задача проведения обзора спринта заключается в получении обратной связи, а общий цикл ее получения выглядит следующим образом:

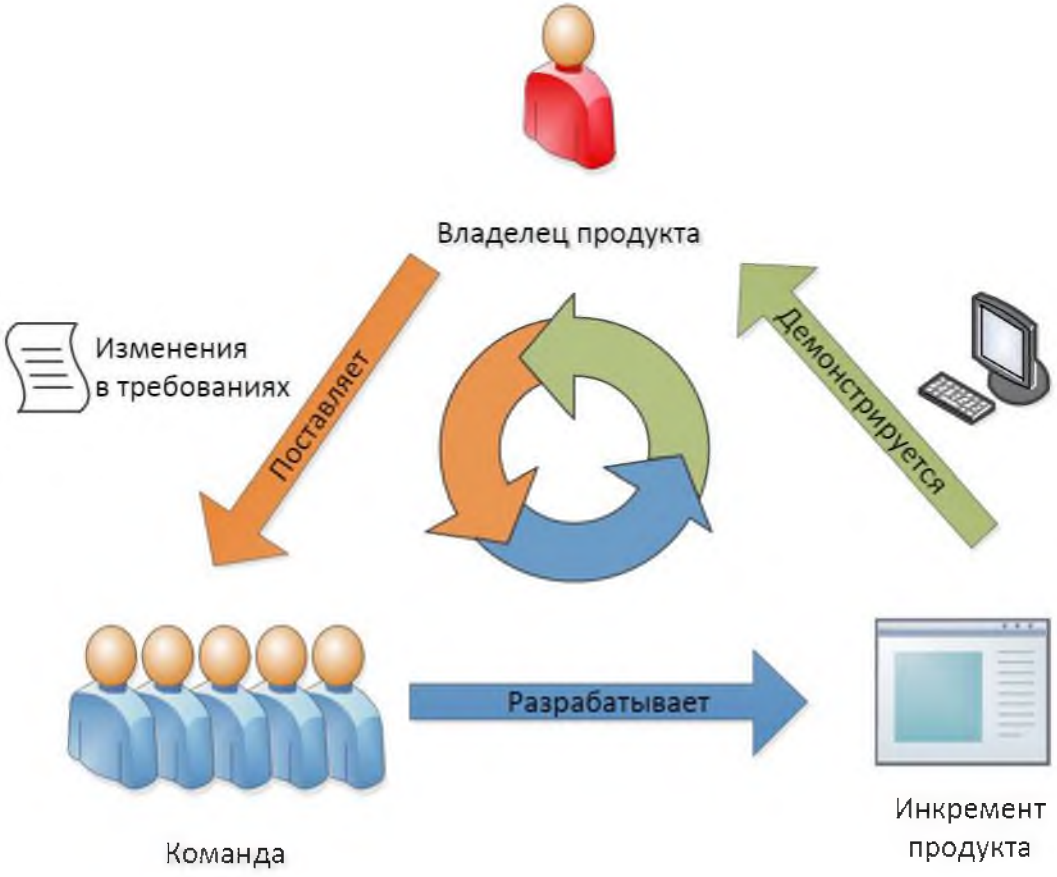

Рисунок 10. Получение обратной связи в рамках Scrum

Демонстрация результатов работы не только мотивирует команду, но и подталкивает реализовывать задачи полностью.

Если взглянуть еще раз на манифест Agile, то там есть пункт, который непосредственно касается демонстраций: «Готовый продукт важнее документации по нему». И действительно основной мерой прогресса является функционал нашего продукта, поэтому показывать на демонстрации надо именно программу. Если ваш заказчик находится не в одном помещении с вами, используйте специальные средства демонстрации. Гораздо хуже будет отправить заказчику презентацию или отчет по сделанному функционалу, ведь мы хотим получить качественную обратную связь.

В обзоре спринта обязательно должна принимать участие вся команда, при этом возможны разные стратегии показа. Антипаттерном можно назвать демонстрацию функционала одним человеком, например, скрам-мастером.

К паттернам можно отнести демонстрацию «чужих» реализованных элементов беклога и привлечение к демонстрации аналитиков, тестировщиков, верстальщиков, UI-специалистов и так далее. Такой подход позволяет выработать командную ответственность за результат.

Ретроспектива

В долгосрочном плане ретроспективы (или сокращенно «ретро») являются самой важной практикой Scrum: ведь именно они позволяют адаптировать и кастомизировать Scrum, делая из него по-настоящему гибкий фреймворк для управления проектами.

Ретроспективу традиционно проводят после обзора спринта спустя небольшое количество времени, чтобы оперативно получить фидбек. Скрам-мастер собирают всю команду для обсуждения результатов спринта. Рекомендуется на ретроспективу приглашать владельца продукта для получения дополнительной обратной связи.

Структура ретроспективы

Обычно ретроспектива занимает от 30 минут до 4 часов и ее продолжительность зависит от следующих факторов:

- Длина спринта: чем длиннее спринт, тем больше команда успевает сделать и тем больше материала для обсуждения;
- Размер команды: чем команда больше, тем больше надо времени, чтобы у каждого ее члена была возможность высказаться и тем больше функционала команда успевает сделать;
- Наличие проблем: со временем команда решает проблемы и ретроспективы сокращаются по времени.

В процентном соотношении принято выделять такую структуру:

Рисунок 11. Структура ретроспективы

Также традиционным является формат по сбору данных, который заключается в ответах каждого участника на три вопроса:

1. Что было сделано хорошо?
2. Что можно улучшить?

3. Какие улучшения будем делать?

Количество улучшений, которые команда берет в реализацию, не должно превышать 2-3, чтобы не снизить скорость реализации бизнес-функционала и не потерять фокус. Команда должна обязательно в том или ином виде составить план улучшений для контроля их исполнения.

Для максимальной открытости и прозрачности обсуждения необходимо использовать основное правило ретроспективы, которое можно озвучивать в начале:

«В независимости от того, что удастся выяснить в результате ретроспективы, каждый член команды сделал всё, чтобы добиться успеха»

Если у команды отсутствуют яркие проблемы, то желательно следующие темы обсудить на ретроспективе:

- скорость команды и ее изменение по сравнению с предыдущими спринтами;
- нереализованные истории пользователей и причины опоздания;
- дефекты и их причины;
- качество процессов (нарушения, отступления).

К паттернам можно отнести анализ сделанных улучшений за несколько прошлых спринтов. Такая «ретроспектива ретроспектив» может проводить раз в 4 спринта и позволяет контролировать уровень сделанных улучшений.

Экстремальное программирование

По книге

Кент Бек. Экстремальное программирование. 2002, СПб., "Питер" ISBN: 5-94723-032-1 224 стр.

Риск: основная проблема

Существующие дисциплины разработки программного обеспечения не срабатывают и не дают желаемого экономического эффекта. Эта проблема обладает огромным экономическим и гуманитарным значением. Мы нуждаемся в новом способе разработки программного обеспечения.

Основная проблема разработки программного обеспечения – это риск.

Вот несколько примеров риска.

• Смещение графиков– наступает день сдачи работы, и вы вынуждены сообщить заказчику, что разрабатываемая система не будет готова еще в течение шести месяцев.

• Закрытие проекта– после нескольких смещений графика и переносов даты сдачи проект закрывается, даже не будучи доведен до стадии опробования в рабочих условиях.

• Система теряет полезность – разработанное программное обеспечение успешно устанавливается в реальной производственной рабочей среде, однако после пары лет использования стоимость внесения в нее изменений и/или количество дефектов увеличиваются настолько, что становится дешевле заменить систему новой разработкой.

• Количество дефектов и недочетов – программная система устанавливается в реальной производственной рабочей среде, однако количество дефектов и недочетов столь велико, что система не используется.

• Несоответствие решаемой проблеме – программная система устанавливается в реальной производственной рабочей среде, однако выясняется, что на самом деле она не решает проблему бизнеса, для решения которой она изначально предназначалась.

• Изменение характера бизнеса – программная система устанавливается в реальной производственной рабочей среде, однако в течение шести последних месяцев проблема, для решения которой предназначалась эта система, потеряла актуальность, а вместо нее бизнес столкнулся с новой, еще более серьезной проблемой.

• Недостаток возможностей – программная система обладает множеством потенциально интересных возможностей, каждую из которых было очень приятно программировать, однако выясняется, что ни одна из этих возможностей не приносит заказчику достаточно много пользы.

• Текучка кадров – в течение двух лет работы все хорошие программисты, работавшие над проектом, один за другим возненавидели разрабатываемую программную систему и ушли на другую работу.

На страницах данной книги вы прочитаете об экстремальном программировании (eXtreme Programming, XP) – дисциплине разработки программного обеспечения, которая ориентирована на снижение степени риска на всех уровнях процесса разработки. XP способствует существенному

увеличению производительности и улучшению качества разрабатываемых программ, кроме того, это весьма занятная практика, доставляющая всем ее участникам массу удовольствия.

Каким образом XP снижает перечисленные ранее риски?

• Смещение графика – XP предлагает использовать очень короткие сроки выпуска каждой очередной версии. Предполагается, что каждая очередная готовая к использованию версия системы разрабатывается в течение максимум нескольких месяцев. Таким образом, объем работ в рамках каждой версии ограничен, а значит, если и происходит смещение, оно менее значительное. В рамках каждой версии предусматривается выпуск нескольких итераций запрашиваемых заказчиком возможностей, на разработку каждой из этих итераций уходит от одной до четырех недель. Так обеспечивается гибкая и чуткая обратная связь с заказчиком, благодаря чему он получает представление о текущем ходе работ. В рамках каждой итерации планирование в соответствии с XP осуществляется в терминах нескольких задач, которые необходимо решить для получения очередной итерации. На решение каждой из задач отводится от одного до трех дней. В результате команда может обнаруживать и устранять проблемы даже в процессе итерации. Наконец, XP подразумевает, что возможности с наивысшим приоритетом будут реализованы в первую очередь. Таким образом, любые возможности, которые не удалось реализовать в рамках данной очередной версии программного продукта, обладают меньшим приоритетом.

• Закрытие проекта – в рамках XP заказчик должен определить наименьший допустимый набор возможностей, которыми должна обладать минимальная работоспособная версия программы, имеющая смысл с точки зрения решения бизнес задач. Таким образом, программистам потребуется приложить минимальное количество усилий для того, чтобы заказчик понял, нужен ли ему этот проект или нет.

• Система теряет полезность – в рамках XP создается и поддерживается огромное количество тестов, которые запускаются и перезапускаются после внесения в систему любого изменения (несколько раз на дню), благодаря этому удается тщательно следить за качеством разрабатываемой программы. XP постоянно поддерживает систему в превосходном состоянии. Дефектам просто не дают накапливаться.

• Количество дефектов и недочетов – в рамках XP разрабатываемая система тестируется как программистами, создающими тесты для каждой отдельной разрабатываемой функции, так и заказчиками, которые создают тесты для каждой отдельной реализованной возможности системы.

• Несоответствие решаемой проблеме – в рамках XP заказчик является составной частью команды, которая работает над проектом. Спецификация проекта постоянно перерабатывается в течение всего времени работы над проектом, благодаря этому любые уточнения и открытия, о которых заказчик сообщает команде разработчиков, немедленно находят свое отражение в разрабатываемой программе.

• Изменение характера бизнеса – в рамках XP цикл работы над очередной версией программы существенно укорачивается. Таким образом, ко времени выхода очередной работоспособной версии программного продукта бизнес не успевает претерпеть существенных изменений. В процессе работы над очередной версией заказчик может попросить отказаться от разработки некоторой функциональности и вместо нее добавить в программный продукт другие, совершенно новые возможности. Команда разработчиков даже не обратит внимания на то, работает ли она над реализацией новых программных механизмов, или осуществляется разработка возможностей, определенных еще несколько лет назад.

• Недостаток возможностей – в рамках XP осуществляется реализация только наиболее высокоприоритетных задач.

• Текучка кадров – XP предлагает программистам брать на себя ответственность самостоятельно определять объем работы и время, необходимое для выполнения этой работы. Они получают возможность сравнить свои предварительные оценки с тем, что получилось на самом деле. В рамках XP содержатся правила, определяющие, кто именно имеет право делать предварительные оценки и изменять их.

За счет этого существенно снижается вероятность того, что программист окажется в растерянности перед возложенной на него задачей, которую заведомо невозможно решить. XP стимулирует интенсивное общение между членами команды разработчиков, снижая ощущение одиночества, которое может возникнуть в случае, если программист не доволен работой, которую он делает. Наконец, в рамках XP явно определяется модель смены кадров. Новые члены команды постепенно берут на себя все большую и большую ответственность.

В процессе этого они пользуются поддержкой друг друга, а также программистов, которые входят в состав команды уже давно.

Четыре ценности

Четыре ценности для XP – это:

• коммуникация (communication);

• простота (simplicity);

• обратная связь (feedback);

• храбрость (courage).

Первая ценность XP – это коммуникация. Проблемы, которые возникают в процессе работы над проектом, почти всегда связаны с тем, что кто то не сказал кому то о чем то важном. Иногда программист не сообщает кому то о важном изменении в дизайне. Иногда программист не задает заказчику важного вопроса, и в результате важное принятое им решение оказывается неправильным. Иногда менеджер не задает программисту важного вопроса, в результате у него складывается неполное, а иногда и неправильное представление о состоянии проекта.

Плохая коммуникация – это не случайность. Существует огромное количество предпосылок, которые ведут к нарушению коммуникации. Например, программист сообщает менеджеру плохие новости и менеджер наказывает программиста. Заказчик сообщает программисту нечто важное, а программист делает вид, что не понял или просто проигнорирует эту информацию.

Дисциплина XP нацелена на обеспечение непрерывной, постоянно осуществляемой коммуникации между участниками проекта. В рамках XP используются многие методики, которые невозможно реализовать без коммуникации. Эти методики направлены на достижение краткосрочных целей, например, тестирование модулей, программирование парами, а также оценка сложности задачи. В процессе тестирования, программирования парами и формирования

предварительных оценок программисты, заказчики и менеджеры вынуждены тесно взаимодействовать.

Это не означает, что излишние разговоры мешают работе. Люди часто попадают в ситуацию, когда они сомневаются, делают ошибки, отвлекаются. В рамках XP существует специальное ответственное лицо – инструктор (coach), в чьи обязанности входит следить за тем, чтобы люди общались тогда, когда это надо. Если инструктор замечает, что люди перестают общаться, он стимулирует коммуникацию.

Простота

Вторая ценность XP – это простота. Инструктор XP спрашивает команду: Какова самая простая вещь, которая скорее всего сработает? (эта фраза является часто употребляемым выражением, в определенном смысле девизом, так что в мире XP даже существует специальная аббревиатура – Do The Simplest Thing That Could Possibly Work[4], DTSTTCPW).

Простота – это далеко не так просто. Напротив, это очень даже сложно – не обращать внимания на вещи, которые вы намерены реализовать завтра, на следующей неделе, в следующем месяце. Вы должны понимать, что если вы намеренно пытаетесь предугадать, как в будущем будет развиваться проект, это значит, что вы боитесь экспоненциального роста стоимости изменений. Программист боится, что завтра ему придется тратить массу усилий для исправления ошибок, которые он сделает сегодня, в результате он зачастую делает код более сложным. Время от времени инструктор должен мягко напоминать программистам, работающим над проектом, что вместо того, чтобы заниматься решением текущих задач, они пытаются прислушаться к собственным внутренним страхам. Если ты пытаешься заставить работать это динамически сбалансированное бинарное дерево, значит, ты умнее, чем я. У меня складывается впечатление, что в данном случае можно обойтись обычным линейным поиском.

Обратная связь

Третья ценность в XP – это обратная связь. Инструктор XP часто произносит: Не спрашивай у меня, спроси у системы и Ты еще не написал для этого тестовый случай? Обратная связь, обеспечивающая точные и конкретные данные о текущем состоянии системы, – это воистину бесценная вещь. Оптимизм – это профессиональная болезнь всего программирования. Обратная связь – это лекарство от этой болезни.

Обратная связь работает в разных временных масштабах. Во первых, обратная связь работает в масштабе минут и дней. Программисты пишут тесты для всей логики в системе. Любой из этих тестов может не сработать. Так программист получает обратную связь, которая ежеминутно обеспечивает его сведениями о состоянии системы. Когда заказчик пишет новые истории (описания возможностей системы), программисты немедленно оценивают их, благодаря чему заказчик получает обратную связь, которая обеспечивает его сведениями о качестве его историй. Человек, который следит за своевременным решением задач в рамках проекта, обеспечивает всех членов команды сведениями о том, какова вероятность того, что запланированный объем работ будет реализован в установленные сроки.

Обратная связь также работает в масштабе недель и месяцев. Заказчики и те, кто обеспечивает функциональное тестирование системы, разрабатывают функциональные тесты для всех историй (говоря иначе – упрощенных тестовых случаев), реализованных в рамках системы. Таким образом они обладают обратной связью, которая обеспечивает их информацией о текущем состоянии используемой ими системы. Заказчики пересматривают график работ каждые две или три недели для того, чтобы убедиться, что скорость выполнения работ соответствует плану. В случае необходимости план пересматривается. Эксплуатация системы в реальных производственных условиях начинается, как только система становится способной решать хотя бы небольшую часть возлагаемых на нее обязанностей. В результате бизнес получает возможность на практике оценить, как именно выглядит система и в каком направлении ее лучше всего развивать в дальнейшем.

О том, что к эксплуатации системы следует приступать как можно раньше, следует рассказать подробнее. Одной из стратегий в процессе планирования является правило, в соответствии с которым наиболее полезные для заказчика истории реализуются программистами и начинают эксплуатироваться как можно раньше. Благодаря этому программисты получают обратную связь, которая обеспечивает их сведениями о качестве принятых ими решений. Процесс разработки начинает напоминать управление автомобилем – программисты прилагают усилия для того, чтобы проект развивался в направлении, удобном для заказчика, при этом колеса должны всегда оставаться на асфальте. Некоторые программисты занимаются разработкой системы длительное время до того, как она начнет эксплуатироваться в реальных рабочих условиях. Откуда же они могут узнать о том, что выполняемая ими работа действительно качественна и необходима?

В большинстве проектов используется прямо противоположная стратегия. Многие рассуждают так: Как только система начинает использоваться на реальном производстве, в нее нельзя будет внести «интересных» изменений, поэтому система должна находится в стадии разработки настолько долго, насколько это возможно.

В XP делается прямо противоположное. В разработке – это временное состояние, в котором система находится в течение очень небольшого времени своей жизни. Будет значительно лучше, если система будет жить самостоятельной жизнью независимо от разработки. Необходимо позволить ей дышать и действовать самостоятельно. Необходимо поддерживать функционирование системы в условиях реального производства и одновременно с этим, разрабатывать новую функциональность. Необходимо обеспечить параллельную разработку и эксплуатацию, и чем раньше вы этого добьетесь, тем лучше.

Обратная связь работает совместно с коммуникацией и простотой. Чем более исчерпывающей является обратная связь, тем легче осуществлять коммуникацию. Когда кто то недоволен написанным вами кодом и передает вам тестовый случай, который указывает на ошибку, это заменяет вам тысячу часов пространных дискуссий на тему эстетики программного дизайна. Если вы хорошо осуществляете коммуникацию, вы лучше знаете, что следует тестировать в системе. Простые системы тестировать проще. Разработка теста концентрирует ваше внимание на том, насколько простой может быть система; до тех пор, пока тест не сработает, вы не можете считать работу завершенной, а когда срабатывают все тесты, можно считать, что вы решили поставленную перед вами задачу.

Храбрость

В контексте первых трех рассмотренных ранее ценностей – коммуникации, простоты и обратной связи – необходимо действовать с максимально возможной скоростью. Если вы работаете со

скоростью, которая не является абсолютно максимальной, это будет делать вместо вас кто то другой, в результате этот кто то прибежит к финишу первым и съест ваш обед вместо вас.

Я расскажу вам о том, как храбрость срабатывает в реальной жизни. В середине восьмой итерации включающего в себя 10 итераций рабочего графика (25 из 30 недель) первой версии первого крупного XP проекта команда обнаружила фундаментальную ошибку в архитектуре системы. Поначалу функциональные тесты указывали на хорошее качество разрабатываемой системы, однако позже количество набранных нами очков резко снизилось. В результате исправления одного дефекта обнаруживался другой дефект. Количество дефектов увеличивалось. Проблема была в архитектурном изъяне.

Для любопытных скажу, что мы работали над системой начисления выплат. Для хранения долгов компании перед сотрудниками использовались поля данных с названием Entitlement (жалованье), а для хранения долгов сотрудников перед другими людьми использовались поля с названием Deduction (вычет). Для некоторых людей использовалось отрицательное жалованье, в то время как вместо этого надо было использовать положительный вычет.

Команда поступила так, как должна была поступить. Когда все поняли, что путь вперед закрыт, они исправили архитектурный изъян. При этом половина всех используемых в отношении системы тестов перестала срабатывать. Однако в течение нескольких дней напряженных усилий по исправлению ситуации тесты снова начали срабатывать и качество системы, оцениваемое при помощи функциональных тестов, повысилось. Однако для того, чтобы поступить описанным образом, потребовалась отвага.

Еще один смелый ход – отказ от ранее разработанного кода. Представьте, что в течение всего рабочего дня вы работаете над реализацией некоторой функциональности. Работа идет неплохо, но когда вы близки к ее завершению, компьютер зависает. На следующее утро вы приходите на работу и в течение получаса восстанавливаете то, над чем работали весь предыдущий день, однако на этот раз код получается более чистым и более простым.

Используйте это. Если приближается конец рабочего дня и код все еще не поддается контролю, выбросьте его. Может быть, следует сохранить тестовые случаи, если вам понравился разработанный вами интерфейс. Однако это не обязательно. Возможно, следующим утром будет легче начать с нуля.

Возможно, перед вами три варианта дизайна. Вы можете потратить по одному дню на реализацию каждой из альтернатив для того, чтобы почувствовать, как они будут вести себя на практике. Затем выбросьте код и начните с нуля развивать тот вариант дизайна, который показался вам наиболее многообещающим.

Стратегия проектирования в XP напоминает алгоритм взбирания на холм (hill climbing). Вы делаете простой дизайн, затем вы делаете его более сложным, далее вы его упрощаете, потом опять усложняете. Проблема подобных алгоритмов состоит в том, что вы ищете локальный оптимум, – при этом никакое незначительное изменение не может улучшить ситуацию, однако улучшения можно достичь, используя значительное изменение.

Достигнув локального оптимума вы, возможно, упускаете более эффективный вариант дизайна. Что поможет вам избежать этого? Как только у кого нибудь из вашей команды возникает сумасшедшая идея, в результате реализации которой сложность всей системы существенно уменьшится, он обязательно попробует реализовать свою идею, если конечно, у него хватит храбрости. Иногда это срабатывает. Если у вас хватит храбрости, вы приступите к эксплуатации новой версии системы в промышленных условиях. Считайте, что теперь вы забираетесь на совершенно новый холм.

Если у вас нет остальных трех ценностей, храбрость сама по себе является обычным взломом (в самом уничижительном смысле этого слова). Однако в сочетании с коммуникацией, простотой и надежной обратной связью храбрость становится чрезвычайно полезной.

Коммуникация идет на пользу храбрости, так как благодаря коммуникации вы обретаете возможность для осуществления более рискованных и более заманчивых экспериментов. Тебе это не нравится? Я просто ненавижу этот код! Давай вместе посмотрим, в какой мере мы сможем переделать его сегодня днем. Простота идет на пользу храбрости, так как, обладая более простой системой, вы сможете позволить себе более смелые действия в ее отношении. Маловероятно, что вы нарушите ее функционирование по неизвестным причинам. Храбрость способствует простоте, ведь как только вы видите способ упростить систему, вы немедленно пробуете его реализовать. Надежная обратная связь идет на пользу храбрости, так как в процессе серьезной модернизации кода вы чувствуете себя значительно увереннее, если вы можете щелкнуть на кнопке и увидеть, что в результате тестирования все тесты показывают зеленый цвет. Если зеленым цветом окрашиваются не все тесты, вы переделываете или просто выкидываете свой код.

Базовые принципы

Исходя из четырех ценностей мы сформулируем десяток (или около того) принципов, в соответствии с которыми будет формироваться наш стиль. В дальнейшем мы будем проверять рассматриваемые методики на соответствие этим принципам.

Мы должны извлечь из описанных ценностей конкретные принципы, которые мы сможем использовать, оценивая ту или иную методику.

Эти принципы помогут нам выбирать между несколькими альтернативами. Предпочтение будет отдаваться альтернативе, которая соответствует принципам в большей степени. Каждый из принципов является воплощением ценностей, о которых я рассказал в предыдущей главе. Ценность может быть туманной: например, то, что для одного человека является простым, для другого человека является сложным. Принцип – это нечто более конкретное. Либо вы используете быструю обратную связь, либо нет. Вот перечень фундаментальных принципов:

- быстрая обратная связь;
- приемлемая простота;
- постепенное изменение;
- приемлемое изменение;
- качественная работа.

Обратно к истокам

Мы хотим сделать все, что от нас зависит, для того чтобы получить стабильный, предсказуемый процесс разработки программного продукта. Однако у нас нет времени на что либо лишнее. Четыре основных рода деятельности, которые составляют собой процесс разработки, – это кодирование, тестирование, слушание и проектирование.

Кодирование

В конце дня у нас должна быть готовая программа. Таким образом, я прихожу к выводу, что кодирование – это как раз та самая деятельность, без которой нам не обойтись. Рисуете ли вы диаграммы, которые автоматически генерируют код, или вы набираете строки в текстовом редакторе, следует считать, что вы кодируете.

Что мы можем извлечь из кода? Наиболее важной вещью является обучение. Обучение происходит следующим образом: у меня появляется мысль, я тестирую ее, чтобы проверить, насколько она хороша. Код – это наиболее удобная вещь для того, чтобы реализовать этот метод на практике. Код не подвластен риторической силе и логике. На код нельзя воздействовать ученой степенью, общественным признанием и высоким окладом. Код просто сидит в вашем компьютере и делает то, что вы ему сказали делать. Если он делает не то, что вы ему сказали делать, – это ваша личная проблема.

Если вы что то закодировали, у вас появляется возможность понять, какая структура кода будет наилучшей. Внутри кода существуют некоторые признаки, которые сообщают вам о том, что вы пока еще не поняли, какой должна быть необходимая структура.

Код позволяет вам также обмениваться информацией четко, сжато и выразительно. Если у вас есть идея и вы пытаетесь объяснить мне ее, я вполне могу не понять вас. Однако если мы вместе реализуем вашу идею в коде, я могу увидеть в логике написанного вами кода точную формулировку вашей идеи. Опять же, я вижу формулировку идеи не так, как вы видите ее в своей голове, а так, как она выражается для всего остального мира.

Коммуникация подобного рода очень просто преобразуется в обучение. Я вижу вашу идею, и у меня появляется моя собственная, однако мне сложно объяснить ее на словах, поэтому я, так же как и вы, обращаюсь к кодированию. Так как наши идеи связаны между собой, мы используем связанный код. Вы видите мою идею, и у вас появляется еще одна.

Наконец, код является артефактом, без которого разработка программного продукта абсолютно невозможна. Я слышал истории о системах, в которых исходный код был утерян, и при этом они продолжали функционировать в производственных условиях. Подобные случаи чрезвычайно редки. Чтобы система продолжала жить, для нее должен существовать исходный код.

Раз мы обязаны обладать исходным кодом, значит, мы можем использовать его для максимально возможного количества задач, связанных с разработкой программ. Например, код можно использовать для общения. То есть при помощи кода вы можете объяснить свое тактическое намерение, описать алгоритм, указать на точки возможного будущего расширения или сокращения. Код можно использовать также для того, чтобы объяснить тесты. Тесты используются как для объективного тестирования некоторой операции системы, так и в качестве ценной спецификации системы на всех уровнях.

Тестирование

Английские философы позитивисты Лок (Locke), Беркли (Berkeley) и Хьюм (Hume) утверждают, что все, чего нельзя измерить, на самом деле не существует. Если речь заходит о коде, я с ними полностью согласен. Возможности программного продукта, которые нельзя продемонстрировать с использованием тестов, просто не существуют. Я запросто могу обмануть самого себя, убедив себя в том, что то, что я написал, есть то, что я имел в виду. Я также вполне могу обмануть себя в том, что то, что я имел в виду, является тем, что я должен был иметь в виду. Поэтому я не должен

верить ничему, что я написал до тех пор, пока я не напишу для этого тесты. Тесты позволяют мне думать о том, что я хочу, вне зависимости от того, как это реализовано. Если я что либо реализовал, тесты сообщают мне о моем представлении о том, что я реализовал.

Многие люди думают об автоматических тестах в контексте тестирования функциональности – то есть вычисления чисел. Чем более опытным я становлюсь в деле написания тестов, тем яснее для меня становится, что я могу разрабатывать тесты для тестирования нефункциональных требований, например для тестирования производительности, или соответствия некоторым стандартам кодирования.

Эрих Гамма (Erich Gamma) придумал термин инфицированный тестами (Test Infected) для описания людей, которые не приступают к кодированию до тех пор, пока у них не будет набор тестов для проверки разрабатываемого кода. Тесты сообщают вам о том, что ваша работа завершена, – когда все тесты сработали, считайте, что на данный момент кодирование успешно завершено. Когда вы больше не можете придумать ни одного теста, можете считать, что вы завершили работу.

Тесты – это ресурс и ответственность. Вы не можете написать всего один тест, добиться его работы и объявить, что на этом ваша работа закончена. Вы несете ответственность за разработку всех тестов, которые могут не сработать, которые вы только можете себе представить. Через некоторое время вы получите неплохое предощущение относительно тестов – если эти два теста сработали, значит, можно со всей уверенностью заключить, что этот третий тест также сработает, и его вовсе не обязательно писать. Конечно же, именно такие рассуждения ведут к появлению ошибок в программах, поэтому вы должны быть очень осторожными в этом отношении. Если в дальнейшем возникают проблемы, которые можно было бы обнаружить раньше, если бы только вы написали вовремя этот третий тест, вам необходимо должным образом воспринять этот горький опыт и в следующий раз заставить себя не отказываться от разработки подобного третьего теста.

Большая часть программного обеспечения разрабатывается без использования автоматического тестирования. Очевидно, что автоматические тесты – это необязательная составная часть разработки. Почему же я всетаки включил тестирование в список важнейших родов деятельности при разработке программного продукта? На этот вопрос я готов дать два ответа: один для краткосрочной перспективы, другой – для долгосрочной.

В долгосрочной перспективе тесты позволяют программе жить дольше (если конечно они работают и должным образом поддерживаются в рабочем состоянии). Если у вас есть тесты, вы можете вносить в программу более значительные изменения в течение более длительного времени. Если у вас нет тестов, вы теряете такую возможность, так как любое изменение перестает быть предсказуемым и может обернуться катастрофой. Если вы продолжаете писать тесты, со временем ваша уверенность в системе увеличивается.

Один из принципов предписывает, что необходимо работать совместно с природой человека, а не против нее. Если тестирование подкрепляется только лишь одним долгосрочным аргументом, об этом аргументе можно легко позабыть. В этом случае многие будут заниматься тестированием только потому, что они обязаны это делать, или потому, что кто то тщательно контролирует правильность их работы. Как только внимание надсмотрщика ослабевает, или в случае, если сроки сдачи работы стремительно приближаются, разработка новых тестов прекращается, уже имеющиеся тесты перестают запускаться, и в результате вся система разваливается на части. Таким образом, если мы хотим работать сообразно с человеческой природой и при этом мы хотим обеспечить тестирование, мы должны найти для тестирования краткосрочную эгоистическую причину.

К счастью, такая краткосрочная причина существует. Программирование в случае, если вы используете тесты, – это более приятный процесс, чем программирование без тестов. Вы кодируете со значительно большей уверенностью. У вас никогда не возникает страха наподобие: «В это место системы надо бы внести изменения, но вдруг я что нибудь сломаю?» Вы просто меняете код, щелкаете на кнопке, запускаются все тесты, если при этом на экране появляется зеленый цвет, вы можете продолжать работу с еще большей уверенностью.

Я помню, как я занимался этим на публичной программистской демонстрации. Каждый раз, когда я отворачивался от аудитории, чтобы продолжить программирование, я машинально щелкал на кнопке тестирования. Я не менял никакого кода. Абсолютно все в рабочей среде оставалось неизменным. Зачем же я раз за разом щелкал на тестирующей кнопке? Я всего лишь хотел получить заряд уверенности. Когда я в очередной раз видел, что все тесты по прежнему срабатывают и ничего в системе не нарушено, я получал такой заряд!

Совместное программирование и тестирование выполняется быстрее, чем просто программирование. Когда я только начинал использовать данную методику, я не ожидал такого эффекта, однако я со всей очевидностью заметил его. Мало того, помимо меня об этом сообщает множество других людей. Возможно, отказавшись от тестирования, вы сможете сэкономить полчаса, однако как только вы привыкнете к использованию тестов, вы быстро отметите разницу в производительности. Выигрыш в производительности получается за счет того, что уменьшается время, необходимое вам для отладки, – вместо того, чтобы заниматься поиском ошибки в течение часа, вы обнаруживаете ее в течение нескольких минут. Иногда вы никак не можете добиться, чтобы тест срабатывал. Это означает, что, скорее всего, вы столкнулись с существенно более крупной проблемой. В этом случае вы должны сделать шаг назад и убедиться в том, что все ваши тесты корректны. Вы также должны проверить весь дизайн системы – возможно, он требует серьезного пересмотра.

Однако существует опасность. Плохо организованное тестирование – это все равно, что розовые очки, сквозь которые вы смотрите на свою систему. Вы получаете ложную уверенность в том, что ваша система в порядке, – еще бы, ведь все тесты срабатывают. Вы продолжаете движение вперед, не подозревая, что оставляете позади себя ловушку. Как только вы в следующий раз пойдете этим же путем, ловушка может сработать.

Весь трюк, связанный с тестированием, заключается в нахождении приемлемого для вас уровня дефектов. Если вы в состоянии позволить себе одну жалобу со стороны заказчика в течение месяца, вкладывайте ресурсы в улучшение тестирования и улучшайте его до тех пор, пока не достигнете желаемого уровня. Затем, используя полученный стандарт тестирования, продолжайте движение вперед, так как состояние системы считается отличным в случае, если все тесты срабатывают.

Заглядывая вперед, отмечу, что в дальнейшем разговор пойдет о двух наборах тестов. Тесты модулей (unit tests) разрабатываются программистами для того, чтобы убедиться в корректной работе разрабатываемого ими кода. Функциональные тесты (functional tests) разрабатываются (или по крайней мере специфицируются) заказчиками для того, чтобы убедиться в том, что система как единое целое работает именно так, как она должна работать.

Таким образом, для тестов существуют две аудиенции. Программисты должны оформить свою уверенность в разрабатываемом коде в реальную форму для того, чтобы кто то другой также мог получить эту уверенность. Заказчики должны подготовить набор тестов для того, чтобы получить от системы подтверждение своей уверенности: Замечательно, я полагаю, что если вы сможете добиться срабатывания всех этих тестов, это значит, что система работает.

Слушание

Программисты ничего не знают. Говоря точнее, программисты не знают ничего такого, что является интересным для бизнесменов. Если бы бизнесмены могли обойтись без программистов, они бы вышвырнули нас вон в одну секунду.

К чему я веду? Если вы решили тестировать, вы должны получить откуда либо ожидаемые ответы. Так как вы (программист) ничего не знаете, вы должны спросить у кого то еще. Они сообщат вам, какие ответы являются ожидаемыми и какие случаи являются необычными с точки зрения бизнеса.

Если вы намерены задать вопрос, вы должны быть готовыми услышать ответ. Таким образом, слушание – это третий род деятельности в рамках разработки программного обеспечения.

Программисты должны слушать с большим вниманием. Они должны услышать от заказчика суть бизнес проблемы. Они должны помочь заказчику понять, что является простым, а что – сложным, это можно считать активной формой слушания. Обратная связь от программистов к заказчику помогает заказчику самому лучше понять суть стоящей перед ним проблемы.

Если вы просто скажете участникам проекта: Вы должны слушать друг друга, и вы должны слушать заказчика, то этим вы не добьетесь желаемого результата. Многие уже попробовали это и пришли к выводу, что подобные простые директивы не срабатывают. Мы должны найти способ структурировать коммуникацию так, чтобы в результате обсуждения речь шла именно о тех вещах, которые нуждаются в обсуждении, это должно происходить именно в то время, когда возникает надобность в подобном обсуждении, и именно в том объеме, в котором эти вещи должны обсуждаться. Кроме того, разрабатываемые нами правила должны избавлять команду от коммуникации, которая только мешает дальнейшему развитию проекта.

Проектирование

Почему нельзя просто слушать, затем писать тестовый случай, затем заставить его работать, затем опять слушать, опять писать тестовый случай и опять заставить его работать, и так далее? Потому что мы знаем, что это не сработает. Конечно, вы можете попробовать действовать именно так в течение некоторого времени. Вы можете даже действовать так в течение достаточно длительного времени. Однако в определенный момент вы не сможете продолжать работать над проектом. Вы попадете в ситуацию, когда для того, чтобы заставить работать некоторый тестовый случай, вам придется нарушить работу другого тестового случая. Или для того, чтобы заставить работать тестовый случай, вам придется затратить столько усилий, что это перестанет быть экономически выгодным. Энтропия проглатывает еще одну жертву.

Единственным способом избежать этого является проектирование. Проектирование – это создание структуры, которая организует логику в системе. Хороший дизайн организует логику так, что для внесения измене ний в одну часть системы вам не нужно обязательно вносить изменения в другую часть системы. Хороший дизайн предусматривает, что каждый логический раздел системы оформляется в виде самостоятельного независимого фрагмента программы. Хороший дизайн размещает логику рядом с данными, в отношении которых она действует. Хороший дизайн позволяет расширять систему, модифицируя только лишь одно ее место.

Плохой дизайн – это прямая противоположность. Если в рамках плохого дизайна вы пытаетесь внести в систему концептуальное изменение, вам приходится вносить изменения сразу в несколько разных мест системы. В рамках плохого дизайна одна и та же логика дублируется в нескольких местах. Со временем затраты, вызванные плохим дизайном, становятся чрезмерно большими. Вы просто забываете о том, в какие места системы необходимо внести все связанные между собой изменения. Вы не можете добавить в систему новую функцию, не нарушив при этом работу одной из уже существующих функций.

Сложность – это еще одна причина плохого дизайна. Если для того, чтобы выяснить, что же все таки происходит, дизайн предусматривает четыре уровня перенаправления и если эти уровни не обладают функциональным или смысловым предназначением, значит, дизайн плохой.

Таким образом, последним родом деятельности, который мы должны структурировать в рамках разрабатываемой нами новой дисциплины, – это проектирование, или, по другому, формирование дизайна. Мы должны сформировать контекст, в рамках которого создается только хороший дизайн, а плохой дизайн исправляется. Кроме того, в рамках этого контекста о текущем дизайне системы знает каждый, кому это необходимо.

Как будет показано в последующих главах, методы формирования дизайна в XP существенно отличаются от методов, традиционно используемых в рамках многих других дисциплин разработки программного обеспечения. В рамках XP проектирование является частью ежедневной работы каждого из программистов. Программисты XP занимаются проектированием прямо в процессе кодирования. Однако вне зависимости от стратегии, которая используется для получения хорошего дизайна, в процессе разработки программного продукта проектирование не выполняется по желанию. Это неотъемлемая часть любого программного проекта, и для того, чтобы сделать разработку программы эффективной, вы должны уделить проектированию очень серьезное внимание.

Решение

Целью данной книги является объяснение того, как работают входящие в XP методики, поэтому в данной главе я бегло перечислю основные группы используемых в рамках XP методик. В следующей главе я покажу, как подобные смехотворно простые решения могут дать столь значительный результат. Там, где некоторая методика слаба, сила остальных методик покрывает недостатки слабой. В последующих главах некоторые темы будут рассмотрены более детально.

Для начала перечислю все методики.

• Игра в планирование(planning game) – быстро определяет перечень задач (объем работ), которые необходимо реализовать в следующей версии продукта. Для этого рассматриваются бизнес приоритеты и технические оценки. Если со временем план перестает соответствовать действительности, происходит обновление плана.

• Небольшие версии(small releases) – самая первая упрощенная версия системы быстро вводится в эксплуатацию, после этого через относительно короткие промежутки времени происходит выпуск версии за версией.

• Метафора(metaphor) – эта простая общедоступная и общеизвестная история, которая коротко описывает, как работает вся система. Эта история управляет всем процессом разработки.

• Простой дизайн(simple design) – в каждый момент времени система должна быть спроектирована так просто, как это возможно. Чрезмерная сложность устраняется, как только ее обнаруживают.

• Тестирование(testing) – программисты постоянно пишут тесты для модулей. Для того чтобы разработка продолжалась, все тесты должны срабатывать. Заказчики пишут тесты, которые демонстрируют работоспособность и завершенность той или иной возможности системы.

• Переработка(refactoring) – программисты реструктурируют систему, не изменяя при этом ее поведения. При этом они устраняют дублирование кода, улучшают коммуникацию, упрощают код и повышают его гибкость.

• Программирование парами(pair programming) – весь разрабатываемый код пишется двумя программистами на одном компьютере.

• Коллективное владение(collective ownership) – в любой момент времени любой член команды может изменить любой код в любом месте системы.

• Непрерывная интеграция(continuous integration) – система интегрируется и собирается множество раз в день. Это происходит каждый раз, когда завершается решение очередной задачи.

• 40 часовая неделя(40 hour week) – программисты работают не более 40 часов в неделю. Это правило. Никогда нельзя работать сверхурочно две недели подряд.

• Заказчик на месте разработки(on site customer) – в состав команды входит реальный живой пользователь системы. Он доступен в течение всего рабочего дня и способен отвечать на вопросы о системе.

• Стандарты кодирования(coding standards) – программисты пишут весь код в соответствии с правилами, которые обеспечивают коммуникацию при помощи кода.

Игра в планирование

Ни соображения бизнеса, ни технические соотношения не должны превалировать. Разработка программного обеспечения – это всегда эволюционирующий диалог между желаемым и возможным. Природа этого диалога состоит в том, что в его процессе изменяется как то, что кажется возможным, так и то, что кажется желаемым.

Представители бизнеса должны принимать решения в следующих областях.

• Объем работ– какую часть проблемы достаточно решить для того, чтобы систему можно было эксплуатировать в реальных производственных условиях? Представитель бизнеса должен быть способен определить, какого объема будет недостаточно и какой объем будет чрезмерным.

• Приоритет – если с самого начала вы можете реализовать только возможности A или B, то какую из них следует реализовать в первую очередь? Ответ на этот вопрос должен быть определен в первую очередь представителем бизнеса, а не программистом.

• Композиция версий – как много или как мало должно быть сделано для того, чтобы бизнес лучше шел с программным обеспечением, чем без него? Программистская интуиция зачастую ошибается в отношении данного вопроса.

• Сроки выпуска версий – в какие важные даты очередные версии программного продукта должны появляться в производстве?

Бизнес не может принимать решения в вакууме. Разработчики должны сформировать набор технических решений, которые должны стать исходным материалом при формировании бизнес решений.

Разработчики должны принимать решения в следующих областях.

• Оценка – как много времени потребуется для того, чтобы реализовать ту или иную возможность?

• Последствия – существует набор бизнес решений, которые следует формировать, только ознакомившись с технологическими последствиями. Хорошим примером является выбор той или иной технологии управления базой данных. Бизнесу лучше иметь дело с крупной компанией, а не с новичками, однако и здесь существует набор факторов, которые необходимо тщательно изучить, так как, возможно, сотрудничество с компанией, появившейся на рынке недавно, оправданно по тем или иным причинам. Возможно, свежая, недавно разработанная система управления базой данных дает двукратный рост производительности. Возможно, разработка решения на основе этой базы данных обойдется бизнесу в два раза дешевле. Таким образом, риск, связанный с использованием новой технологии управления базой данных, вполне оправдан. А возможно, и нет. Разработчики должны объяснить последствия того или иного решения.

• Процесс – как будет организована команда разработчиков и как будет организована работа этой команды? Команда должна соответствовать культуре, в рамках которой будет осуществляться разработка, однако при этом не следует забывать, что ваша основная задача – получить качественный программный продукт, а не следить за чистотой культуры разработки.

• Подробный график работ– какие истории заказчика должны быть реализованы в первую очередь в рамках работы над очередной версией продукта? Программисты должны обладать свободой при решении вопроса о том, какие самые рискованные сегменты разработки должны быть реализованы в первую очередь. Благодаря этому снижается общий риск разработки. В рамках этого ограничения по прежнему следует в первую очередь работать над наиболее приоритетными для бизнеса задачами. Благодаря этому снижается вероятность того, что реализация чрезвычайно важных для бизнеса возможностей системы будет отложена до следующей версии.

Небольшие версии

Каждая версия должна быть настолько маленькой, насколько это возможно. В ней должны быть реализованы наиболее ценные для бизнеса требования. В целом версия должна быть логически завершенной и работоспособной, то есть вы не можете реализовать некоторую возможность только наполовину и внедрить ее в производство только для того, чтобы сократить время работы над версией.

Лучше планировать на месяц или два вперед, чем планировать на полгода или год вперед. Компания, которая за один раз передает в руки заказчика достаточно крупный программный продукт, не может выпускать очередные версии этого продукта достаточно часто. Эта компания должна сократить время работы над очередной версией настолько, насколько это возможно.

Метафора

Каждый программный проект XP направляется при помощи единой всеобъемлющей метафоры. Иногда эта метафора выглядит наивной, как, например, система управления контрактами, о которой рассказывается в терминах контрактов, заказчиков и индоссаментов. Иногда метафора требует дополнительных разъяснений, например, требуется отметить, что компьютер должен рассматриваться как рабочий стол или вычисление пенсии должно выглядеть как электронная таблица. Все это метафоры, так как на самом деле мы не говорим буквально, что система – это электронная таблица. Метафора просто помогает каждому участнику проекта понять базовые элементы программы и то, как они взаимосвязаны.

Слова, которые используются для идентификации технических элементов системы, должны постоянно заимствоваться из выбранной метафоры. По мере того как идет работа над проектом, метафора развивается и вся команда продолжает ее анализировать, получая при этом новые источники вдохновения.

В XP метафора во многом заменяет собой то, что другие люди называют термином архитектура. Проблема состоит в том, что архитектура – это, как правило, огромная по размерам схема системы, которая не дает представления о ее целостности. Архитектура – это большие прямоугольники и соединения между ними.

Конечно же, вы можете сказать: плохо сформированная архитектура – это плохо. Однако нам требуется подчеркнуть саму цель, для которой формируется архитектура, а это значит, что мы должны предоставить каждому участнику проекта связную историю о строении и функционировании системы. Эта история должна быть изложена на языке, понятном как технарям, так и бизнесменам. В рамках этой истории будет осуществляться работа над проектом. Спросив о метафоре, мы получаем в ответ сведения об архитектуре, причем эти сведения передаются нам в такой форме, которая удобна для общения и обдумывания.

Простой дизайн

В каждый момент времени правильным является дизайн системы, в рамках которого:

1. Выполняются все тесты.

2. Нет дублирующейся логики. (Опасайтесь скрытого дублирования, например, применения параллельных иерархий классов.)

3. Выражается каждая из идей, важных для программистов.

4. Существует наименьшее возможное количество классов и методов.

Каждый фрагмент дизайна системы должен доказать свое право на существование на основании всех перечисленных правил. Эдвард Туфт (Edward Tufte) придумал упражнение для разработчиков графов – нарисуйте граф так, как хотите, затем стирайте до тех пор, пока вы не удаляете из графа полезную информацию. Если вы не можете больше продолжать стирание, значит, перед вами наиболее удачный дизайн для графа. Простой дизайн программной системы можно сформировать точно таким же образом – уберите из системы все элементы, какие вы сможете убрать, не нарушив при этом правил 1 3.

Edward Tufte, The Visual Display of Quantitative Information (Визуальное отображение численной информации), Graphics Press, 1992.

Этот совет противоположен тому, что обычно приходится слышать программистам: Реализуйте для сегодняшнего дня, а проектируйте – для завтрашнего. Однако если вы уверены, что будущее – в неопределенности, если вы верите в то, что завтра вы можете сменить направление своих мыслей и не платить за это слишком дорого, значит, включение в дизайн функциональности только на основании абстрактных размышлений – это безумство. Добавляйте в дизайн то, что вам нужно, только тогда, когда это действительно вам нужно.

Тестирование

Любая возможность программы, для которой нет автоматических тестов, просто не существует. Программисты пишут тесты модулей, благодаря чему их уверенность в правильности функционирования программы становится частью самой программы. Заказчики пишут функциональные тесты, благодаря чему их уверенность в функционировании программы также становится частью программы. В результате всеобщая уверенность в работоспособности программы со временем все возрастает и возрастает. Эта уверенность выражена в наборе тестов, количество которых увеличивается и которые продолжают функционировать по мере продолжения работы над программой. Благодаря этому со временем программа становится не менее, а более приспособленной для внесения в нее изменений.

Нет необходимости писать тесты для каждого разрабатываемого вами метода, проверять надо только производственные методы, которые могут не сработать. Иногда вы тратите усилия только на то, чтобы понять, возможно ли в процессе функционирования кода возникновение той или иной ситуации. В течение получаса вы анализируете код. Да, это возможно. Теперь вы отбрасываете код и начинаете писать его заново – начиная с тестов.

Переработка

Когда программисты приступают к реализации некоторой возможности программы, они всегда задаются вопросом, существует ли способ изменения имеющейся программы для того, чтобы упростить добавление в нее требуемой новой возможности? После того как возможность добавлена, программисты спрашивают себя, можно ли теперь упростить программу и при этом обеспечить выполнение всех тестов? Это и называется переработкой кода (refactoring).

Обратите внимание, это означает, что иногда вы должны сделать больше работы, чем это на самом деле необходимо для того, чтобы добавить в программу новую возможность. Однако, работая в этом ключе, вы обеспечиваете снижение затрат, связанных с добавлением в программу последующих возможностей. Переработав код, вы упрощаете его дальнейшую модернизацию. Вы не выполняете переработку исходя из предварительных нечетких размышлений, напротив, вы перерабатываете код тогда, когда система нуждается в этом. Когда при добавлении новой возможности вы дублируете код, это означает, что система просит вас выполнить переработку.

Если программист видит некоторый черновой способ обеспечить выполнение теста, для реализации которого потребуется одна минута, и, помимо этого, он также видит другой, более элегантный способ обеспечить выполнение теста, предусматривающий также упрощение дизайна, но для реализации которого требуется десять минут, программист должен предпочесть

второй способ, то есть потратить больше времени, но в результате получить более простой и более элегантный дизайн. К счастью, в рамках XP вы получаете возможность вносить в дизайн системы любые, даже самые радикальные изменения, делая это небольшими, малорискованными шагами.

Программирование парами

Весь код системы вплоть до ее внедрения в производство пишется парами программистов, каждая из которых работает на одном компьютере, оснащенном одной клавиатурой и одной мышью.

В каждой паре существуют две роли. Один партнер, в руках которого находится клавиатура и мышь, думает о том, как прямо сейчас реализовать некоторый метод самым лучшим образом. Второй партнер думает более стратегически:

- Сработает ли используемый подход в целом?
- Какими могут быть другие, еще не рассмотренные тестовые случаи?
- Существуют ли какие либо способы упростить всю систему таким образом, что текущая проблема просто исчезнет?

Состав пар меняется динамически. Партнеры, которые утром входили в состав одной пары, днем могут стать членами совершенно других пар. Если вы отвечаете за решение задачи в области, которая является для вас малознакомой, вы можете попросить составить вам компанию кого либо, кто недавно работал в этой области. Скорее всего, вы побываете в паре с каждым из членов вашей команды.

Коллективное владение

Любой член команды, который видит возможность добавить что либо в любой раздел кода системы, может сделать это в любой подходящий для этого момент времени.

Сравните это с двумя другими моделями владения кода – полное отсутствие владения и индивидуальное владение. В давние времена различными кусками кода программы никто не владел. Если кто либо желал изменить какой либо код, он мог сделать это в соответствии со своими собственными пожеланиями. Результатом был хаос, в особенности если приходилось иметь дело с объектами, в которых взаимосвязь между строкой кода в одном месте и строкой кода в другом месте нельзя было в точности установить статически. Код разрастался очень быстро, и с такой же скоростью он стремительно терял стабильность.

Чтобы подвести ситуацию под контроль, программисты стали использовать индивидуальное владение кодом. Единственным человеком, кто обладал правом внесения в некоторый фрагмент кода изменений, являлся официальный владелец этого кода. Если кто либо, не являющийся владельцем, видел, что код необходимо изменить, он должен был обратиться с соответствующей просьбой к владельцу. В результате такой практики действительный код системы начинал расходиться с тем, каким его хотели бы видеть работающие в рамках проекта программисты. Изменение кода в рамках подобного подхода превращалось в своего рода бюрократическую процедуру – люди начинали избегать обращаться к владельцу кода для того, чтобы внести в код

желаемые изменения, вместо этого они предпочитали работать с тем, что есть. В конце концов, внести изменение требуется прямо сейчас, а не спустя некоторое время. Таким образом, код оставался относительно стабильным, однако он не эволюционировал с достаточно большой скоростью. А когда владелец кода находил другую работу и уходил из команды... возникали серьезные проблемы.

В рамках XP ответственность за весь код системы лежит на всех членах команды. Нельзя сказать, что каждый член команды хорошо знает каждую часть кода, однако можно сказать, что каждый член команды знает, по крайней мере, что то о каждой части. Если пара программистов работает над решением некоторой задачи и видит, что для упрощения работы требуется внести модификации в некоторую часть кода, тем самым улучшив этот код, изменения вносятся немедленно, благодаря чему решаемая этой парой задача упрощается.

Постоянно продолжающаяся интеграция

Код интегрируется и тестируется каждые несколько часов, минимум один раз в день. Для того чтобы обеспечить это, проще всего выделить для этой цели один специально предназначенный для этого компьютер. Когда этот компьютер освобождается, пара, у которой имеется код, подлежащий интеграции, садится за интеграционный компьютер, загружает текущую версию системы, добавляет в нее свои собственные изменения (проверяя и устраняя любые несоответствия и конфликты) и запускает тесты до тех пор, пока все они не сработают (все 100% тестов).

Интеграция одного набора изменений за один раз отлично срабатывает, так как становится очевидным, кто именно должен исправить тест, который не сработал, – мы должны, так как, должно быть, именно мы его сломали. Это связано с тем, что предыдущая пара, которая выполняла интеграцию, добилась срабатывания всех 100% тестов. И если мы не добьемся срабатывания всех 100% тестов, мы должны выкинуть из системы все, что мы написали, и начать решать задачу заново, так как очевидно, что в этом случае, приступая к решению, мы просто не знали всего того, что требуется для разработки требуемого кода (возможно, мы не знаем всего необходимого и сейчас).

40 часовая рабочая неделя

Я хочу быть свежим и энергичным каждое утро, равно как и уставшим и удовлетворенным каждый вечер. Каждую пятницу я хочу быть уставшим и удовлетворенным настолько, чтобы в течение последующих двух дней чувствовать себя в состоянии думать о чем либо, не связанном с работой. После этого, в понедельник, я хочу приходить на работу с горящими глазами и головой, наполненной идеями.

Конечно же, для этого необязательно, чтобы рабочих часов в неделе было бы ровно 40. Разные люди способны эффективно работать в течение различного времени. Один человек не может концентрировать свое внимание дольше, чем в течение 35 рабочих часов, другой способен успешно действовать в течение 45 часов в неделю. Но никто не способен работать по 60 часов в неделю на протяжении многих недель и при этом оставаться свежим, творческим, внимательным и уверенным в своих силах. Ни в коем случае не делайте этого!

Работа во внеурочное время – это признак серьезных проблем в проекте. В рамках XP действует очень простое правило – нельзя работать во внеурочное время две недели подряд. В течение одной недели можно поднапрячься и поработать несколько лишних часов. Но если в очередной понедельник вы приходите на работу и объявляете: Чтобы достичь поставленных целей, мы должны снова работать допоздна, это означает, что у вас возникли проблемы, которые вы не сможете решить простым увеличением рабочего времени.

Отпуск является близкой к этому темой для обсуждения. Европейцы часто отдыхают в течение двух, трех или четырех последовательных недель. Американцы редко когда берут для отдыха больше чем несколько дней подряд. Если бы это была моя компания, я настаивал бы на том, чтобы люди отдыхали в течение двух последовательных недель ежегодно и при этом у них в запасе было бы не менее одной или двух недель дополнительно для более коротких отпусков.

Заказчик на месте разработки

Для ответов на возникающие вопросы, решения споров и установки мелкомасштабных приоритетов рядом с командой разработчиков должен постоянно находиться реальный заказчик. Под термином реальный заказчик я подразумеваю человека, который действительно пользуется системой, когда эта система работает в производственных условиях. Например, если вы разрабатываете систему обслуживания клиентов, заказчиком будет служащий по работе с клиентами, пользующийся этой системой. Если вы разрабатываете систему обмена долговыми обязательствами, заказчиком будет биржевой брокер, работающий с этой системой.

Основным препятствием для воплощения этого правила является то обстоятельство, что реальные пользователи разрабатываемой системы обходятся для заказчика слишком дорого в смысле рабочего времени. Иногда менеджерам заказчика жалко жертвовать одним из реальных служащих компании только для того, чтобы он постоянно находился вместе с разработчиками и отвечал на их вопросы. Менеджеры должны решить, что является для них более важным – ускорение и повышение качества разработки или рабочее время одного или двух сотрудников. Если программная система не приносит бизнесу больше, чем приносит ему один из сотрудников предприятия, скорее всего, такая система не стоит того, чтобы ее разрабатывать и внедрять на производстве.

Кроме того, утверждение, что представитель заказчика, работающий вместе с командой, не может заниматься некоторыми своими повседневными делами, на самом деле не верно. Даже такие творческие люди, как программисты, не могут генерировать вопросы непрерывно в течение 40 часов каждую неделю. Конечно, сотрудник предприятия заказчика обладает тем недостатком, что он физически удален от тех людей, с которыми он должен взаимодействовать, однако при этом у него будет достаточно времени для выполнения некоторой своей обычной работы.

Недостатком такого подхода является то, что в случае, если проект в конце концов умирает, то сотни часов, которые сотрудник предприятия заказчика потратил на помощь и консультации разработчиков, оказываются временем, потраченным впустую. Получается, что заказчик потерял работу, которую его сотрудник делал вместе с разработчиками, а также он потерял работу, которую его сотрудник мог бы сделать в соответствии со своими прямыми обязанностями, если бы его рабочим временем не пожертвовали ради поддержки разработки проекта. XP делает все возможное для того, чтобы проект не умирал.

Однажды я работал над одним проектом, и фирма заказчик с большой неохотой выделила нам одного реального сотрудника, но только ненадолго. После того как мы завершили первый этап

разработки и успешно внедрили первую версию системы на производстве, когда стало очевидным, что система может продолжать эволюционирование, менеджеры заказчика выделили нам трех реальных сотрудников. Компания заказчик пришла к выводу, что сможет получить от системы больше прибыли, если пожертвует большим количеством своих сотрудников.

Стандарты кодирования

Если мы собираемся перебрасывать программистов с разработки одной части системы на разработку другой части системы, обеспечивать постоянную смену партнеров в парах по несколько раз на дню, обеспечивать постоянную переработку кода программистами, которые этот код не писали, мы просто не можем позволить себе использовать в рамках одного проекта применение нескольких разных стилей кодирования. Спустя некоторое время работы над системой уже нельзя будет сказать точно, какой именно член команды написал тот или иной код.

Стандарт должен требовать от разработчиков приложения минимально возможных усилий для реализации той или иной возможности. Он должен способствовать воплощению на практике правила трех О – Once and Only Once (запрет на дублирование кода). Стандарт должен способствовать коммуникации. Наконец, стандарт должен быть добровольно воспринят всей командой разработчиков.