1. Гибкая методология разработки программного обеспечения – 150 минут

Ключевые слова

Манифест гибкой методологии разработки программного обеспечения, гибкая методология разработки программного обеспечения, инкрементная модель разработки, итеративная модель разработки, жизненный цикл программного обеспечения, автоматизация тестирования, базис тестирования, разработка на основе тестов, тестовый предсказатель, пользовательская история

Цели обучения

1.1. Основы гибкой методологии разработки программного обеспечения

FA-1.1.1 (K1) Вспомнить основную концепцию гибкой методологии разработки программного обеспечения, основанной на манифесте гибкой разработки
FA-1.1.2 (K2) Понять преимущества командного подхода
FA-1.1.3 (K2) Понять преимущества ранней и частой обратной связи

1.2. Аспекты подходов гибких методологий

FA-1.2.1 (K1) Вспомнить подходы гибких методологий разработки программного обеспечения
FA-1.2.2 (K3) Написать тестируемые пользовательские истории в сотрудничестве с разработчиками и представителями бизнеса
FA-1.2.3 (K2) Понять, как ретроспективы могут быть использованы в качестве механизма улучшения процесса в проектах с гибкой методологией
FA-1.2.4 (K2) Понять применение и цель непрерывной интеграции
FA-1.2.5 (K1) Знать различия между планированием итерации и выпуска, и как тестировщик добавляет ценность для каждой из этих активностей.

1.1. Основы гибкой методологии разработки программного обеспечения

Тестировщик в проектах с гибкой методологией будет работать иначе, чем работающий в традиционных проектах. Тестировщик должен понимать ценности и принципы, лежащие в основе проектов с гибкой методологией, и как тестировщики являются неотъемлемой частью командного подхода вместе с разработчиками и представителями бизнеса. Участники проектов с гибкой методологией взаимодействуют друг с другом рано и часто, это помогает устранять дефекты как можно раньше и разрабатывать качественный продукт.

1.1.1. Гибкая методология разработки программного обеспечения и ее Манифест

В 2001 году группа людей, представляющих наиболее широко используемые облегченные методологии разработки программного обеспечения, договорились об общем наборе ценностей и принципов, которые стали известны как Манифест гибкой разработки программного обеспечения [Agilemanifesto]. Он включает заявления четырех ценностей:

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

Манифест гибкой разработки утверждает, что, хотя понятия справа имеют ценность, понятия слева имеют большую ценность.

Люди и взаимодействия
Гибкая разработка сильно ориентирована на людей. Команды создают программное обеспечение, и команды могут работать более эффективно благодаря непрерывному взаимодействию вне зависимости от используемых инструментов и процессов.

Работающее программное обеспечение
С точки зрения клиента, работающее программное обеспечение гораздо более полезно и ценно, чем слишком подробная документация, и это обеспечивает возможность дать группе разработчиков быструю обратную связь. Кроме того, поскольку работающее программное обеспечение, пусть даже с урезанной функциональностью, доступно гораздо раньше в жизненном цикле разработки, гибкая разработка может принести значительное преимущество от времени выхода на рынок. Поэтому гибкая разработка особенно полезна в быстро меняющихся бизнес-средах, где про-

блемы и/или решения не ясны или где бизнес хочет внедрять инновации в новых проблемных сферах деятельности.

Сотрудничество с заказчиком

Пользователи часто сталкиваются с большими трудностями при определении системы, которая им требуется. Непосредственное сотрудничество с пользователем повышает вероятность понимания того, что требуется клиенту. В то время, как наличие контрактов с клиентами может быть важным, работа в постоянном и тесном сотрудничестве с ними, вероятно, принесет больше успеха проекту.

Готовность к изменениям

Изменения в проектах программного обеспечения неизбежны. Окружение, в котором работает бизнес, законодательство, конкуренция, продвижение технологий и другие факторы могут иметь большое влияние на проект и его цели. Эти факторы должны учитываться в процессе разработки. Таким образом, наличие гибкости в методах принятия изменений важнее, чем просто строго придерживаться плана.

Принципы

Суть ценностей Манифеста гибкой разработки отражены в 12 принципах:

- Наивысшим приоритетом является удовлетворение потребностей заказчика, благодаря ранней и непрерывной поставке ценного программного обеспечения.
- Изменение требований приветствуется даже на поздних стадиях разработки. Гибкие процессы используют изменения для конкурентного преимущества заказчика.
- Поставляйте работоспособное программное обеспечение часто: с периодичностью от нескольких недель до нескольких месяцев, предпочитая более короткие интервалы.
- В течении всего проекта представители бизнеса и разработчики должны ежедневно работать вместе.
- Создавайте проекты в окружении мотивированных личностей. Предоставьте им окружение и поддержку, в которой они нуждаются, и доверьте им сделать работу.
- Наиболее эффективный и результативный способ передачи информации в команду и внутри нее - это непосредственное общение.
- Работающее программное обеспечение - основной показатель прогресса.
- Гибкие процессы способствуют устойчивому процессу разработки. Спонсоры, разработчики и пользователи должны быть в состоянии поддерживать постоянный темп бесконечно.
- Постоянное внимание к техническому совершенству и хорошему дизайну повышает гибкость.

- Простота - искусство максимизации объема невыполненной работы - необходима.
- Лучшая архитектура, требования и дизайн появляются в самоорганизующихся командах.
- На регулярной основе команда размышляет, как стать более эффективной, затем соответственно корректирует поведение.

Различные гибкие методологии предлагают предписывающие практики, чтобы привести эти ценности и принципы в действие.

1.1.2. Командный подход

Командный подход означает привлечение каждого, кто обладает знаниями и навыками, необходимыми для обеспечения успеха проекта. Команда включает представителей клиента и других заинтересованных лиц бизнеса, которые определяют функции продукта. Команда должна быть относительно небольшой, успешные команды наблюдались немногочисленные - от 3 до 9 человек. В идеале, вся команда разделяет одно и тоже рабочее пространство, поскольку совместное размещение сильно облегчает общение и взаимодействие. Командный подход поддерживается на ежедневных митингах (см. Раздел 2.2.1), в которых участвуют все члены команды, где сообщается о ходе работы и отмечаются любые препятствия на пути прогресса. Командный подход способствует более эффективной и результативной командной динамике.
Использование командного подхода в разработке продукта является одним из основных преимуществ гибкой разработки. Эти преимущества включают:

- Повышение коммуникации и сотрудничества внутри команды
- Предоставление различных наборов навыков в команде для использования в интересах проекта
- Общая ответственность за качество

Вся команда отвечает за качество в гибких проектах. Суть командного подхода заключается в том, что тестировщики, разработчики и представители бизнеса работают вместе не каждом этапе процесса разработки. Тестировщики будут тесно сотрудничать и с разработчиками, и с представителями бизнеса, чтобы обеспечить достижение желаемых уровней качества. Это включает поддержку и сотрудничество с представителями бизнеса, чтобы помочь им создать подходящие приемочные тесты, работу с разработчиками для согласования стратегии тестирования и принятии решения о подходах автоматизации тестирования. Таким образом, тестировщики могут передавать и распространять знания о тестировании другим членам команды и влиять на разработку продукта.
Вся команда участвует в любых консультациях или встречах, на которых функционал продукта представляется, анализируется или оценивается. Концепция привле-

чения тестировщиков, разработчиков и представителей бизнеса во всех обсуждениях функционала называется силой трех [Crispin08].

1.1.3. Ранняя и частая обратная связь

Гибкие проекты имеют короткие итерации, позволяющие проектной команде получать раннюю и постоянную обратную связь о качестве продукта на протяжении всего жизненного цикла разработки. одним из способов обеспечения быстрой обратной связи является непрерывная интеграция (см. раздел 1.2.4).
Когда используются подходы последовательной разработки, клиент часто не видит продукт до тех пор, пока проект не будет завершен. В этот момент для команды разработчиков зачастую слишком поздно эффективно решать любые проблемы, которые могут возникнуть у клиента. Получая часто обратную связь от клиента в процессе проекта, гибкие команды могу включить больше новых изменений в процессе разработки продукта. Ранняя и частая обратная связь помогает команде сосредоточится на функциях с наивысшим бизнес приоритетом или связанном с ними риском, и они предоставляются клиенту в первую очередь. Это также помогает лучше управлять командой, поскольку способность команды прозрачна для всех. Например, сколько работы мы можем сделать в спринте или итерации? Что может помочь нам двигаться быстрее? Что мешает нам сделать это?
Преимущества ранней и частой обратной связи включает:

- Избежание недоразумений в требованиях, которые возможно, не были обнаружены до тех пор в цикле разработки, пока они не стали более дорогостоящими для исправления.
- Уточнение требований пользователей к функционалу, делая его доступным для использования пользователем на ранней стадии. Таким образом, продукт лучше отражает то, что хочет клиент.
- Обнаружение (через непрерывную интеграцию), изоляция и решение проблем качества на ранней стадии.
- Предоставление информации гибкой команде относительно производительности и возможности поставки.
- Последовательная поддержка темпа проекта.

1.2. Аспекты подходов гибких методологий

Существует много подходов гибких методологий, используемых в организациях. Общепринятые практики большинства гибких методологий в организациях включают совместное создание пользовательских историй, ретроспективы, непрерывной интеграции и планирования каждой итерации, как и релиза. В этом подразделе описаны

некоторые подходы гибких методологий.

1.2.1. Подходы гибкой методологии разработки программного обеспечения

Существует несколько подходов гибкой методологии, каждый из которых по-разному реализует ценности и принципы Манифеста. В этой программе рассмотрены три представленных подхода гибких методологий: Экстремальное программирование (XP), Скрам, Канбан.

Экстремальное программирование

Экстремальное программирование (XP), первоначально представленное Кентом Бек [Beck04], является подходом гибкой методологии разработки программного обеспечения, характеризуемым определенными ценностями, принципами и практиками разработки.

XP включает в себя пять ценностей для руководства разработкой:

- коммуникация,
- простота,
- обратная связь,
- смелость
- уважение.

XP описывает набор принципов как дополнительные рекомендации:

- человечность,
- экономичность,
- взаимная выгода,
- сходство,
- улучшение,
- разнообразие,
- обдумывание,
- течение,
- возможность,
- избыточность,
- неудача,
- качество,
- маленькими шажками,
- принятая ответственность.

XP описывает тринадцать основных практик:

- сидеть вместе,
- команда как одно целое,
- информативное рабочее пространство,
- напряженная работа,

- парное программирование,
- истории,
- недельный цикл,
- ежеквартальный цикл,
- слабость,
- десятиминутная сборка,
- непрерывная интеграция,
- сначала тесты,
- инкрементное проектирование.

Многие из подходов гибких методологий разработки программного обеспечения, использующиеся сегодня, зависят от XP его ценностей и принципов. Например, команды в гибких методологиях, придерживающиеся Срам, часто используют методы XP.

Скрам

Скрам – это система управления гибкой методологией, которая содержит следующие составные инструменты и практики [Schwaber01]:

- Спринт: Скрам делит проект на итерации (называемые спринтами) фиксированной длины (обычно от двух до четырех недель)
- Приращение продукта: каждый спринт приводит к потенциально выпускаемому/поставляемому продукту (называемому приращением).
- Набор задач продукта: владелец продукта управляет приоритезированным списком запланированных функций продукта (называемым набором задач продукта)
- Набор задач спринта: в начале каждого спринта команда Скрам выбирает набор функций с наивысшим приоритетом (называемый набором задач спринта) из набора задач продукта. Поскольку команда Скрам, а не владелец продукта, выбирает функции, которые будут реализованы в рамках спринта, этот выбор считается принципом вытягивания, а не принципом проталкивания.
- Критерии готовности: чтобы убедиться, что в конце каждого спринта есть потенциальный релиз продукта, команда Скрам обсуждает и определяет критерии завершения спринта. Дискуссия расширяет понимание командой набор задач и требований продукта.
- Временная шкала: только те задачи, требования или функции, которые команда ожидает завершить за спринт, являются набором задач спринта. Если команда разработчиков не может завершить задачу за спринт, связанные функции продукта удаляются из спринта и задача возвращается в набор задач продукта. Временная шкала применяется не только к задачам, но и в других ситуациях (например, соблюдение времени начала и окончания встречи).
- Прозрачность: команда разработчиков ежедневно сообщает и обновляет статус спринта на встрече, называемой ежедневным скрам-митингом. Это делает содержание и прогресс текущего спринта, включая результаты тестирования, видимыми

для команды, руководства и всех заинтересованных сторон. Например, команда разработчиков может отобразить статус спринта на доске.

Скрам определяет три роли:

- Скрам мастер: обеспечивает выполнение и соблюдение практик и правил Скрам, а также устраняет любые нарушения, проблемы с ресурсами или другие препятствия, которые могут помешать команде следовать практикам и правилам. Этот человек не руководитель команды, а тренер.
- Владелец продукта: представляет клиента, а также создает, поддерживает и приоритезирует набор задач продукта. Он не руководитель команды.
- Команда разработки: разрабатывает и тестирует продукт. Команда самоорганизована: нет руководителя команды, поэтому команда сама принимает решения. Команда также является кросс-функциональной (см. Раздел 2.3.2 и Раздел 3.1.4).

Скрам (в отличие от XP) не предписывает конкретные методы разработки программного обеспечения (например, сначала тесты). Кроме того, Скрам не дает рекомендаций как должно проводиться тестирование в проекте Скрам.

Канбан

Канбан [Anderson13] это подход к управлению, который иногда используется в проектах с гибкой методологией. Основная цель - визуализация и оптимизация потока работ производственной цепи. Канбан использует три инструмента [Linz14]:

- Канбан доска: Цепочка создания ценности, которую нужно контролировать, Скрам визуализируется Канбан доской. Каждый столбец отображает состояние, которое представляет набор связанных действий, например, разработку или тестирование. Элементы или задачи, которые должны быть обработаны, символически изображаются карточками, перемещающимися слева направо по доске путем изменения состояний.
- Лимит незавершенных работ: Количество параллельных активных задач строго ограничено. Это контролируется максимальным количеством карточек, разрешенных для состояния и/или глобально для доски. Всякий раз, когда на состоянии есть свободное место, сотрудник перетягивает карточку от состояния-предшественника.
- Время выполнения: Канбан используется для оптимизации непрерывного потока задач для минимизации (среднего) времени выполнения задач.

Канбан имеет сходство со Скрамом. В обоих подходах визуализация активных задач (например, на общей доске) обеспечивает прозрачность содержания и прогресс задач. Задачи, которые еще не запланированы, ждут в наборе задач и переходят на Канбан доску, как только появится новое место (производственные мощности).
Итерации или спринты необязательны в Канбан. Процесс Канбан позволяет выпускать свои продукты функция за функцией, а не как часть релиза продукта. Таким об-

разом, временная шкала как механизм синхронизации необязателен, в отличии от Скрам, который синхронизирует все задачи в спринте.

1.2.2. Совместное создание пользовательской истории

Плохие спецификации часто являются основной причиной провала проекта. Проблемы со спецификацией могут возникнуть в результате отсутствия у пользователей понимания их истинных потребностей, отсутствия глобального видения системы, избыточного или противоречивого функционала и других недопониманий. В разработке по гибким методологиям пользовательские истории написаны для сбора требований с точки зрения разработчиков, тестировщиков и представителей бизнеса. При последовательной разработке это общее видение функции осуществляется посредством формальных рецензирований после написания требований; в разработке по гибким методологиям это общее видение достигается посредством частых неформальных рецензирований, пока пишутся требования.

Пользовательские истории должны учитывать, как функциональные, так и нефункциональные свойства. Каждая история содержит критерии приемки этих свойств. Эти критерии должны определяться в сотрудничестве с представителями бизнеса, разработчиками и тестировщиками. Они предоставляют разработчикам и тестировщикам расширенное видение функции, которую будут подтверждать представители бизнеса. Команда в гибкой методологии считает, что задача завершена, когда выполнен список критериев приемки.

Как правило, уникальное видение тестировщика улучшит пользовательскую историю, указав потерянные детали или нефункциональные требования. Тестировщик может внести свой вклад, обратившись к представителям бизнеса по открытым вопросам пользовательской истории, предложив способы проверки пользовательской истории и подтвердив критерии приемки.

При совместном написании пользовательской истории можно использовать такие методы, как «мозговой штурм» и «составление карт памяти». Тестировщик может использовать технику INVEST [INVEST]:

- Независимая (Independent)
- Обсуждаемая (Negotiable)
- Полезная (Valuable)
- Поддающаяся оценке (Estimable)
- Небольшая (Small)
- Тестируемая (Testable)

Согласно концепции «трех С» [Jeffries00], пользовательская история - это сочетание трех элементов:

- Карточка (Card): Карточка представляет собой физический носитель, описывающий пользовательскую историю. Она определяет требование, его критичность, ожидаемую продолжительность разработки и тестирования, а также критерии приемки этой истории. Описание должно быть точным, поскольку оно будет использоваться в наборе задач продукта.
- Диалог (Conversation): Посредством диалога разъясняется, как будет использоваться программное обеспечение. Диалог может быть задокументирован или оставаться устным. Тестировщики, имеющие другую точку зрения от разработчиков и представителей бизнеса [ISTQB_FL_SYL], привносят ценный вклад в обмен мыслями, мнениями и опытом. Диалог начинается во время этапа планирования релиза и продолжается даже после того, как история запланирована.
- Подтверждение (Confirmation): Критерии приемки, обсуждаемые в диалоге, используются для подтверждения того, что история выполнена. Эти критерии приемки могут охватывать несколько пользовательских историй. Для оценки критериев следует использовать как позитивные, так и негативные тесты. Во время подтверждения различные участники играют роль тестировщика. К ним могут относиться разработчики, а также специалисты по производительности, безопасности, совместимости и другим характеристикам качества. Чтобы подтвердить, что история выполнена, конкретные критерии приемки должны быть протестированы и должно быть продемонстрировано, что они достигнуты.

Команды в гибких методологиях различаются тем, как они документируют пользовательские истории. Независимо от подхода к документированию пользовательских историй, документация должна быть краткой, достаточной и необходимой.

1.2.3. Ретроспектива

В разработке по гибкой методологии ретроспектива - это встреча в конце каждой итерации для обсуждения того, что было успешным, что можно было бы улучшить, и как включить улучшения и сохранить успехи в будущих итерациях. Ретроспективы охватывают такие темы, как процесс, люди, организации, отношения и инструменты. Регулярно проводимые итоговые встречи, после которых следуют соответствующие действия, имеют решающее значение для самоорганизации и постоянного совершенствования разработки и тестирования.

Результатом ретроспективы могут стать решения по улучшению, связанные с тестированием, ориентированные на эффективность, производительность тестирования, на качество тестовых сценариев и удовлетворенность команды. Они также могут учитывать возможности тестирования приложений, пользовательских историй, функций или системных интерфейсов. Анализ первопричин дефектов может привести к совершенствованию тестирования и разработки. В целом команды должны реализовать только несколько улучшений за итерацию. Это позволяет непрерывно проводить улучшения с постоянным темпом.

Расписание и организация ретроспективы зависят от конкретного метода гибких методологий. Представители бизнеса и команда присутствуют на каждой ретроспективе в качестве участников, а координатор организует и проводит встречу. В некоторых случаях команды могут приглашать на встречу других участников.
Тестировщики должны играть важную роль в ретроспективе. Тестировщики являются частью команды и привносят свой уникальный взгляд [ISTQB_FL_SYL, раздел 1.5]. Тестирование есть в каждом спринте и жизненно важно для успеха. Все члены команды, тестировщики и не-тестировщики, могут вносить вклад как в тестирование, так и в активности, не связанные с тестированием.
Ретроспектива должна происходить в профессиональной среде, характеризующейся взаимным доверием. Атрибуты успешной ретроспективы такие же, как и для любого другого рецензирования, описанного в учебном плане Базового Уровня [ISTQB_FL_SYL, раздел 3.2].

1.2.4. Непрерывная интеграция

Осуществление приращения продукта требует надежного, рабочего, интегрированного программного обеспечения в конце каждого спринта. Непрерывная интеграция решает эту задачу, объединяя все изменения, внесенные в программное обеспечение, и регулярно интегрируя все измененные компоненты по крайней мере один раз в день. Управление конфигурацией, компиляция, сборка программного обеспечения, развертывание и тестирование - все в едином, автоматическом, повторяемом процессе. Поскольку разработчики постоянно интегрируют свою работу, постоянно собирают и постоянно тестируют, дефекты в коде обнаруживаются быстрее.

После кодирования, отладки и фиксации кода разработчиков в общем хранилище исходного кода процесс непрерывной интеграции состоит из следующих автоматизированных действий:

- Статический анализ кода: выполнение статического анализа кода и создание отчетов о результатах
- Компиляция: компиляция и линковка кода, генерация исполняемых файлов
- Модульное тестирование: выполнение модульных тестов, проверка покрытия кода и создание отчетов о результатах тестирования
- Развертывание: установка сборки в тестовом окружении
- Интеграционное тестирование: выполнение интеграционных тестов и создание отчетов о результатах
- Отчет (сводная таблица): публикация статуса всех этих активностей в общедоступном местоположении или отправка статуса команде по электронной почте

Автоматизированный процесс сборки и тестирования происходит ежедневно, рано и быстро обнаруживает интеграционные ошибки. Непрерывная интеграция позволяет тестировщикам в гибких методологиях регулярно запускать автоматические тесты, в

некоторых случаях, как часть процесса непрерывной интеграции, и предавать быструю обратную связь команде о качестве кода.

Эти результаты тестирования видны всем членам команды, особенно когда автоматизированные отчеты интегрируются в процесс. Автоматическое регрессионное тестирование может быть непрерывным на протяжении всей итерации. Хорошие автоматические регрессионные тесты охватывают как можно больше функциональности, включая пользовательские истории, реализованные в предыдущих итерациях. Хорошее покрытие автоматическими регрессионными тестами помогает поддерживать создание (и тестирование) больших интеграционных систем. Когда регрессионное тестирование автоматизировано, тестировщики в гибких методологиях освобождаются, чтобы сконцентрировать ручное тестирование на новых функциях, внесенных изменениях и подтверждающем тестировании исправленных дефектов.

Организации, использующие непрерывную интеграцию, в дополнение к автоматическим тестам обычно используют инструменты сборки для непрерывного контроля качества. В дополнение к запуску модульных и интеграционных тестов, такие инструменты могут запускать дополнительные статические и динамические тесты, измерять и профилировать производительность, извлекать и оформлять документацию на основании исходного кода и облегчать проведение процессов обеспечения качества. Это постоянное применение контроля качества нацелено на улучшение качества продукта, а также сокращение времени, затрачиваемого на его поставку, взамен традиционной практики применения контроля качества после завершения всей разработки.

Инструменты сборки могут быть связаны с инструментами автоматического развертывания, которые могут доставать соответствующую сборку с сервера непрерывной интеграции или с сервера сборки и развертывать его в одной или нескольких средах: разработки, тестирования, промежуточной или даже производственной среды. Это уменьшает ошибки и задержки, связанные с использованием специализированного персонала или программистов для установки релизов в этих средах.

Непрерывная интеграция может обеспечить следующие преимущества:

- Позволяет более раннее обнаружение и более простой анализ первопричины проблем интеграции и конфликтующих изменений
- Дает команде разработчиков регулярную информацию о работоспособности кода
- Сохраняет версию программного обеспечения, протестированного в течение дня разработанной версии
- Снижает риск регрессии, связанный с рефакторингом кода, благодаря быстрому повторному тестированию основного кода после каждого небольшого набора изменений

- Обеспечивает уверенность в том, что ежедневная работа разработки основывается на прочном фундаменте
- Создает прогресс видимого движения продукта к завершению, поощряя разработчиков и тестировщиков
- Устраняет риски расписания, связанные с интеграционным «большим взрывом»
- Обеспечивает постоянную доступность исполняемого программного обеспечения на протяжении всего спринта для тестирования, демонстрации или образовательных целей
- Уменьшает повторяющиеся действия ручного тестирования
- Обеспечивает быструю обратную связь по решениям, принятым для улучшения качества и тестов

Однако, непрерывная интеграция не лишена своих рисков и проблем:

- Необходимо внедрять и поддерживать инструменты непрерывной интеграции
- Процесс непрерывной интеграции должен быть определен и создан
- Автоматизация тестирования требует дополнительных ресурсов и может быть сложной для создания
- Полное тестовое покрытие имеет важное значение для достижения преимуществ автоматизированного тестирования
- Иногда команды излишни полагаются на модульные тесты и выполняют слишком мало системного и приемочного тестирования

Непрерывная интеграция требует использования инструментов, включая инструменты тестирования, инструменты автоматизации процесса сборки и инструменты контроля версий.

1.2.5. Планирование релиза и итерации

Как упоминалось в программе Базового Уровня [ISTQB_FL_SYL], планирование – это постоянный вид деятельности, и это также относится и к жизненным циклам гибких методологий. Для жизненных циклов гибких методологий выполняются два вида планирования – планирование релиза и планирование итерации.

Планирование релиза рассматривает перспективы релиза продукта, часто за несколько месяцев до начала проекта. Планирование релиза определяет и переопределяет набор задач продукта и может включать в себя детализацию больших пользовательских историй в коллекцию небольших историй. Планирование релиза обеспечивает основу для подхода к тестированию и планирования тестирования, охватывающего все итерации. Планы релиза – высокоуровневые.

При планировании релиза представители бизнеса совместно с командой устанавливают и расставляют приоритеты для пользовательских историй релиза. (см. Раздел 1.2.2). Основываясь на этих пользовательских историях, определяются риски проекта и качества и выполняется высокоуровневая оценка объема работ (см. Раздел

3.2).
Тестировщики участвуют в планировании релиза и особенно повышают ценность следующих активностей:

- Определение тестируемости пользовательских историй, включая критерии приемки
- Участие в анализе рисков проекта и качества
- Оценка объема работ по тестированию, связанных с пользовательскими историями
- Определение необходимых уровней тестирования
- Планирование тестирования релиза

После завершения планирования релиза начинается планирование первой итерации. Планирование итерации рассматривает одну итерацию до конца и связано с набором задач итерации.

При планировании итерации команда выбирает пользовательские истории из приоритезированного набора задач релиза, прорабатывает пользовательские истории, проводит для пользовательских историй анализ рисков и оценивает объем работы, необходимый для каждой пользовательской истории. Если пользовательская история слишком расплывчата и предпринятые попытки уточнить ее неудачны, команда может отказаться принять ее и использовать следующую пользовательскую историю, основываясь на приоритете. Представители бизнеса должны ответить на вопросы команды о каждой истории, чтобы команда могла понять, что они должны реализовать и как тестировать каждую историю.
Количество выбранных историй основано на установленной скорости команды и предполагаемом размере выбранных пользовательских историй. После того, как содержимое итерации согласовано, пользовательские истории разбиваются на задачи, которые будут выполняться соответствующими членами команды.
Тестировщики участвуют в планировании итерации и особенно повышают ценность следующих активностей:

- Участие в детальном анализе рисков пользовательских историй
- Определение тестируемости пользовательских историй
- Создание приемочных тестов для пользовательских историй
- Разбивка пользовательских историй на задачи (в частности, задачи по тестированию)
- Оценка объема работа по тестированию для всех задач тестирования
- Определение функциональных и нефункциональных аспектов тестируемой системы
- Поддержка и участие в автоматизации тестирования на нескольких уровнях тестирования

Планы релиза могут меняться в течении проекта, включая изменения пользовательских историй в наборе задач продукта. Эти изменения могут быть вызваны внутренними или внешними факторами. Внутренние факторы включают возможности поставки, быстроту поставки и технические проблемы. Внешние факторы включают

открытие новых рынков и возможностей, новых конкурентов или бизнес-угроз, которые могут изменить цели релиза и / или целевые даты. Кроме того, планы итераций могут меняться во время итерации. Например, конкретная пользовательская история, которая считалась относительно простой во время оценки, может оказаться более сложной, чем ожидалось.

Эти изменения могут быть сложными для тестировщиков. Тестировщики должны понимать общую картину релиза для планирования целей тестирования, и они должны иметь достаточный базис тестирования и тестовый предсказатель для каждой итерации для разработки целей тестирования, как описано в программе обучения Базового Уровня [ISTQB_FL_SYL, раздел 1.4]. Необходимая информация должна быть доступна тестировщику раньше, но все же изменение должно быть принято в соответствии с принципами гибких методологий. Эта дилемма требует тщательного принятия решений относительно стратегии тестирования и документации по тестированию. Подробнее о задачах тестирования в гибких методологиях см. [Black09, глава 12].

Планирование релиза и итераций должно касаться планирования тестирования, так же, как и планирования активностей в области разработки. К конкретным вопросам, связанным с тестированием, относятся:

- Объем тестирования, глубина тестирования областей из этого объема, цели тестирования и обоснования этих решений.
- Члены команд, которые будут проводить работы по тестированию.
- Необходимые тестовое окружение и тестовые данные, когда они нужны, и произойдут ли какие-либо изменения в тестовом окружении и / или в данных до или во время проекта
- Сроки, последовательность, зависимости и предварительные условия для функционального и нефункционального тестирования (например, как часто запускать регрессионные тесты, какие функции зависят от других функций или тестовых данных и т.д.), в том числе о том, как работы по тестированию связаны и зависят от работ по разработке.
- Риски проекта и качества, подлежащие рассмотрению (см. Раздел 3.2.1).

Дополнительно, большие команды оценивая объем работ должны включать учет необходимого времени и объема работ для завершение требуемого тестирования.

2. Ключевые принципы, методики и процессы в гибком тестировании – 105 минут

Ключевые слова
Верификационный тестовый сценарий, элемент конфигурации, управление конфигурацией

Цели обучения фундаментальным принципам, практикам и процессам гибкого тестирования

2.1. Различия процессов тестирования при традиционных и гибких подходах

FA-2.1.1 (K2) Описать различия тестовых активностей в проектах под управлением гибких методологий и в проектах не под их управлением
FA-2.1.2 (K2) Описать взаимосвязь активностей по разработке и тестированию в проектах под управлением гибких методологий
FA-2.1.3 (K2) Описать роль независимого тестирования в проектах под управлением гибких методологий

2.2. Статус тестирования в проектах под управлением гибких методологий

FA-2.2.1 (K2) Описать инструменты и техники, используемые обмена для информацией по статусу тестирования, включая прогресс и качество продукта
FA-2.2.2 (K2) Описать процесс развития тестов в течение множества итераций и объяснить почему в проектах под управлением гибких методологий для управления регрессионными рисками важна автоматизация тестирования

2.3. Роль и навыки тестировщика в команде, работающей по гибким методологиям

FA-2.3.1 (K1) Понять, какие навыки (в сфере общения с людьми, в профессиональной сфере и в тестировании) необходимы тестировщику в гибкой команде
FA-2.3.2 (K3) Понять роль тестировщика в гибкой команде

2.1. Различия процессов тестирования при традиционных и гибких подходах

Как описано в учебном плане базового уровня [ISTQB_FL_SYL] и в [Black09], тестовые активности связаны с активностями по разработке, и, как следствие, процесс тестирования строится по-разному в разных жизненных циклах. Тестировщики должны понимать различия между тестированием в традиционных моделях (например, в последовательных, таких как V-модель или итеративных, таких как RUP) и тестированием при гибком подходе, чтобы работать эффективно и продуктивно. Гибкие модели отличаются тем, как интегрируются процессы тестирования и разработки, тем как выстраивается работа над проектом, наименованиями, критериями входа и выхода, использующихся для различных уровней тестирования, использованием инструментов и тем, как можно эффективно использовать независимое тестирование.
Тестировщики должны помнить, что организации значительно различаются реализацией жизненных циклов ПО. Отклонение от идеалов гибких подходов (см. Раздел 1.1) может представлять собой разумную настройку и адаптацию разных практик. Способность адаптироваться к особенностям конкретного проекта, включая фактически применяемые методы разработки программного обеспечения, является ключевым фактором успеха для тестировщиков.

2.1.1. Активности тестирования и разработки

Одним из ключевых отличий между традиционными и гибкими подходами в разработке программного обеспечения является концепция сравнительно коротких итераций, каждая из которых приводит к созданию рабочего программного обеспечения, которое предоставляет функции, имеющие ценность для заинтересованных сторон бизнеса. В начале проекта осуществляется планирование выпуска релиза. За этим следует последовательность итераций. В начале каждой итерации осуществляется ее планирование. После определения перечня задач на итерацию осуществляется разработка выбранных пользовательский историй, интеграция их в общий продукт, а затем тестирование. Из-за высокой динамики протекания итерации процессы разработки, внедрения и тестирования происходят как параллельно, так и одновременно. Процесс тестирования происходит на протяжении всей итерации, а не в конце.
Тестировщики, разработчики и представители бизнеса - все они играют определенную роль в тестировании, как и в традиционных жизненных циклах. Разработчики выполняют модульные тесты, поскольку они разрабатывают функции из пользовательских историй. Тестировщики затем проверяют этот функционал. Представители бизнеса также проверяют истории во время реализации. Все заинтересованные стороны могут использовать как написанные тестовые сценарии, так и просто экспериментировать с использованием функционала, чтобы обеспечить быструю обратную связь с командой разработчиков.

В некоторых случаях могут возникать укрепляющие или стабилизирующие итерации для исправления давно найденных дефектов и других форм так называемого технического долга. Однако наилучшей практикой является то, что никакой функционал не считается выполненным до тех пор, пока он не будет интегрирован и системно протестирован [Goucher09]. Другая хорошая практика заключается в исправлении дефектов, оставшихся от предыдущей итерации, в начале следующей итерации, называемая «Сначала исправить ошибки». Иногда возникают жалобы, что эта практика приводит к ситуации, когда суммарная работа, которая должна быть выполнена в процессе итерации, неизвестна, и становится сложнее оценить, когда может быть реализован остальной функционал. В конце последовательности из нескольких итераций может проводиться ряд действий по подготовке релиза, однако в некоторых случаях релиз выпускается в конце каждой итерации.

Вследствие того, что тестирование на основе риска используется в качестве одной из стратегий тестирования, во время планирования релиза проводится анализ рисков высокого уровня, при этом тестировщики часто используют этот анализ. Однако конкретные риски качества, связанные с каждой итерацией, определяются и оцениваются при планировании итераций. Этот анализ рисков может влиять на последовательность разработки, а также на приоритет и глубину тестирования функционала. Это также влияет на оценку усилий, необходимых для тестирования каждой функции (см. Раздел 3.2).

В некоторых гибких методах (например, в экстремальной разработке) используется парная работа. Под ней подразумевается объединение тестировщиков, работающих вместе на проекте, в пары для проверки функционала. Парная работа может также объединять тестировщика с разработчиком для разработки и тестирования функционала. Парная работа может быть затруднена при распределенной работе тестовой группы, но существующие процессы и инструменты могут помочь обеспечить ее выполнение. Дополнительные сведения о распределенной работе см. в разделе [ISTQB_ALTM_SYL, раздел 2.8].

Тестировщики могут также выступать в качестве тренеров по тестированию и повышению качества в команде, делиться опытом тестирования и поддерживать работу по обеспечению качества. Это способствует осознанию коллективной ответственности за качество продукта.

Автоматизация тестирования на всех уровнях тестирования происходит во многих командах, работающих с применением гибких методологий, и это означает, что тестировщики тратят время на создание, выполнение, мониторинг и поддержку автоматизированных тестов и результатов. Из-за интенсивного использования автоматизации тестирования, большое количество ручного тестирования на проектах, как правило, осуществляется с использованием методов, основанных на опыте и основанных на найденных дефектах, таких как программные атаки, исследовательское тестирование и предположение об ошибках [ISTQB_ALTA_SYL, разделы 3.3 и 3.4] и [ISTQB_FL_SYL, раздел 4.5]).

В то время как разработчики будут сосредоточены на создании модульных тестов,

тестировщикам следует сосредоточиться на создании автоматизированных систем интеграции, системных и системных интеграционных тестов. Это приводит к тому, что команды, работающие по гибким методологиям, предпочитают тестировщиков с большим техническим опытом и опытом автоматизации тестирования.

Один из основных принципов гибких методологий заключается в том, что изменения могут возникать на протяжении всего проекта. Поэтому в проектах, выстроенных с применением таких методологий, работа с документацией проекта ведется в облегченном режиме. Изменения существующих функций влияют на тестирование, особенно на регрессионное тестирование. Использование автоматизированного тестирования является одним из способов управления трудозатрат по тестированию, связанных с изменением функционала. Однако важно, чтобы скорость изменения не превышала способность проектной команды справляться с рисками, связанными с этими изменениями.

2.1.2. Результаты проектной работы

Результаты работы проектной команды и тестировщиков в частности в гибких подходах обычно делят на три категории:

1. Бизнес-ориентированные результаты, описывающие предназначение функционала (например, спецификации требований) и способы его использования (например, пользовательская документация)
2. Результаты разработки, которые описывают, как система построена (например, диаграммы сущности объекта), как фактически реализована (например, код) или как устроены отдельные фрагменты кода (например, автоматические модульные тесты)
3. Результаты тестирования, которые описывают, как строился процесс тестирования системы (например, стратегии и планы тестирования), как фактически тестировалась система (например, ручные и автоматические тесты) и результаты тестирования.

В типичном проекте, выстроенном на основе гибких методологий, обычной практикой является избегание большого количества документации. Вместо этого основное внимание уделяется созданию рабочего программного обеспечения вместе с автоматическими тестами, которые демонстрируют соответствие требованиям. Это поощрение к сокращению документации относится только к документации, которая не представляет ценности для заказчика. В успешном проекте наблюдается баланс между повышением эффективности за счет сокращения документации и предоставления достаточной документации для поддержки бизнеса, тестирования, разработки и сопровождению. На этапе планирования релиза, проектная команда должна принять решение о необходимых результатах своей работы к завершению проекта и определить, какой уровень проектной документации необходим.

Типичные бизнес-ориентированные рабочие продукты на таких проектах включают пользовательские истории и критерии приемки при работе над проектом. Пользовательские истории - это гибкая форма спецификаций требований, целью которых является объяснение, как система должна вести себя в отношении единой, когерентной функции или функционала. Пользовательская история должна определять достаточно маленький функционал, чтобы его можно было реализовать за одну итера-

цию. Большие коллекции связанных функций или набор вспомогательных функций, составляющих один сложный функционал, принято называть «Бизнес-потребностями». Бизнес-потребности могут включать пользовательские истории из разных групп разработчиков. Например, одна пользовательская история может описывать, что требуется на уровне API (промежуточное ПО), в то время как другая история описывает, что необходимо на уровне интерфейса (приложение). Эти сочетания могут быть разработаны в рамках серии итераций. Каждая бизнес-потребность и ее пользовательские истории должны соответствовать критериям приемки.

Типичные продукты для разработчиков на проектах, построенных с применением гибких методологий, включают код. Эти же разработчики часто создают автоматизированные модульные тесты. Такие тесты могут быть созданы после разработки кода. Однако в некоторых случаях разработчики создают тесты постепенно, до того, как весь код будет написан, чтобы обеспечить проверку, как только часть кода будет написана, работает ли она так, как ожидалось.

Хотя этот подход и называется «сначала тестируем» или «разработка, управляемая тестированием», в действительности, тесты являются скорее формой исполняемых низкоуровневых проектных спецификаций, чем тестов [Beck02].

Типичные результаты работы тестировщика на проектах, построенных с применением гибких методологий, включают автоматические тесты, а также такие документы, как планы тестирования, каталоги рисков качества, ручные тесты, отчеты о дефектах и журналы результатов тестирования. Документы о тестировании должны быть написаны максимально простым языком и иметь максимально простую структуру, что также бывает характерно для традиционных подходов. Тестировщики также собирают метрики на основе результатов проведенной работы по тестированию и протоколов результатов тестов, где делается упор на упрощенный подход к этим действиям.

В некоторых реализациях гибких методологий, особенно в регламентированных, критически важных, распределенных или очень сложных проектах и продуктах, необходима дальнейшая формализация этих результатов. Например, некоторые команды преобразовывают пользовательские истории и критерии приемки в более формальные спецификации требований. Отчеты по вертикальной и горизонтальной возможности отслеживания могут быть подготовлены для анализа аудиторам, соответствия нормативам и других требований.

2.1.3. Уровни тестирования

Уровни тестирования - это активности тестирования, которые логически связаны друг с другом, зачастую по готовности или полноте проверяемого функционала.

В последовательных моделях жизненного цикла уровни тестирования часто определяются так, что критерии завершения одного уровня являются частью критериев начала для следующего уровня. В некоторых итеративных моделях это правило не применяется. Уровни тестирования перекрываются. Спецификация требований, проектная спецификация и действия по разработке могут перекрываться уровнями тестирования.

В некоторых жизненных циклах разработки с применением гибких методологий перекрытие происходит, потому что изменения в требованиях, дизайне и коде могут происходить в любой момент итерации. Хотя Скрам теоретически не позволяет изменять пользовательские истории после планирования итерации, на практике такие изменения иногда происходят. Во время итерации для любой пользовательской истории обычно выполняются последовательно следующие тестовые действия:

- Модульное тестирование, обычно выполняемое разработчиком
- Приемочное тестирование функционала, которое иногда разбивается на два вида деятельности:
  - Верификационное тестирование компонентов, которое часто автоматизировано, может выполняться разработчиками или тестировщиками и включает тестирование по критериям приемки пользовательской истории
  - Валидационное тестирование функционала, которое обычно является ручным и может выполняться силами разработчиков, тестировщиков и бизнес-партнеров, работающих совместно, чтобы определить, подходит ли эта функция для использования, улучшить наглядность достигнутого прогресса и получить реальную обратную связь от заказчика

Кроме того, часто наблюдается параллельный процесс регрессионного тестирования, происходящий на протяжении всей итерации. Он включает повторное выполнение автоматических модульных тестов и проверок функционала, реализованного в текущей итерации и в предыдущих итерациях, как правило, с помощью системы непрерывной интеграции.

В некоторых проектах может существовать уровень системного тестирования, который начинается, когда первая пользовательская история готова к такому тестированию. Это может включать выполнение функциональных тестов, а также нефункциональные тесты производительности, надежности, удобства использования и других соответствующих типов тестов.

Гибкие команды могут использовать различные формы приемочного тестирования (используя термин, описанный в учебном плане базового уровня [ISTQB_FL_SYL]). Внутренние альфа-тесты и внешние бета-тесты могут проводиться либо по завершении каждой итерации, либо после серии итераций. Приемочные испытания для пользователей, эксплуатационные приемочные испытания, нормативные приемочные испытания и контрактные приемочные испытания также могут проводиться либо по завершении каждой итерации, либо после серии итераций.

2.1.4. Управление тестированием и взаимодействием

Гибкие проекты часто связаны с использованием автоматизированных инструментов разработки, тестирования и управления разработкой программного обеспечения. Разработчики используют инструменты статического анализа, модульного тестирования и покрытия кода. Разработчики постоянно проверяют код и модульные тесты в системе управления конфигурацией, используя автоматизированные схемы сборки и тестирования. Эти интегрированные среды позволяют непрерывно добавлять новое

программное обеспечение в систему, при проверке которого повторяется и статический анализ, и модульные тесты.
Такие автоматизированные тесты могут также включать функциональные тесты на уровне интеграционного и системного тестирования. Эти функциональные автоматические тесты могут быть созданы с использованием функциональных испытательных заглушек, функциональных средств тестирования пользовательского интерфейса с открытым исходным кодом или коммерческих инструментов и могут быть интегрированы с автоматизированными сценариями, которые выполняются как часть непрерывной интеграции. В некоторых случаях из-за продолжительности функциональных тестов они отделяются от модульных и выполняются реже. Например, модульные тесты могут запускаться каждый раз, когда проверяется новое программное обеспечение, а более длинные функциональные тесты запускаются только каждые несколько дней.
Одна из целей автоматизированных тестов - подтвердить, что сборка работает и устанавливается. Если какой-либо автоматизированный тест завершается неуспешно, команда должна своевременно устранить обнаруженный дефект и сохранить изменения в коде. Для оперативного доступа к результатам тестирования требуется умение создавать отчеты о тестировании в режиме реального времени. Этот подход помогает сократить дорогостоящие и неэффективные циклы «разработка-установка-ошибка-исправление-переустановка», которые могут возникать во многих традиционных проектах, поскольку изменения, которые нарушают сборку или являются причиной сбоя программного обеспечения, обнаруживаются быстро.
Автоматизированные инструменты тестирования и сборки помогают управлять риском регрессии, связанным с частыми изменениями, которые часто возникают в проектах, построенных с применением гибких методологий. Однако чрезмерная зависимость от автоматизированного модульного тестирования для управления этими рисками может быть проблемой, поскольку модульное тестирование часто имеет ограниченную эффективность обнаружения дефектов [Jones11]. Также требуются автоматизированные тесты на уровне системного и интеграционного тестирования.

2.1.5. Варианты организации независимого тестирования

Как обсуждалось в программе базового уровня [ISTQB_FL_SYL], независимые тестировщики часто более эффективны при поиске дефектов. В некоторых гибких командах разработчики создают множество автоматизированных тестов. Один или несколько тестировщиков могут быть встроены в команду, выполняя множество задач тестирования. Однако, учитывая положение этих тестировщиков в команде, существует риск потери независимости и объективной оценки.
Другие гибкие команды сохраняют полностью независимые, отдельные тестовые группы и назначают тестировщиков по требованию в течение последних дней каждого спринта. Это может сохранить независимость, и такие тестировщики могут обеспечить объективную, независимую оценку программного обеспечения. Однако, временные рамки, непонимание новых функций продукта и проблемы взаимоотношений

с заинтересованными сторонами и разработчиками, зачастую, приводят к проблемам при таком подходе.

Третий вариант состоит в том, чтобы иметь независимую отдельную группу тестирования, в которой тестировщики назначаются в группы на долгосрочной основе в начале проекта, что позволяет им сохранять свою независимость, получая хорошее представление о продукте и крепкие отношениях с другими членами команды. Кроме того, независимая команда тестирования может иметь специализированных тестировщиков вне таких групп для работы в долгосрочных и / или итерационно-независимых мероприятиях, таких как разработка автоматизированных тестовых инструментов, проведение нефункционального тестирования, создание и поддержка тестовых сред и выполнения разно уровневого тестирования, которые не могут быть хорошо выполнены в рамках спринта (например, системное интеграционное тестирование).

2.2. Статус тестирования в проектах под управлением гибких методологий

Изменения в проектах под управлением гибких методологий происходят очень быстро. Это означает, что статус тестирования, ход тестирования и качество продукта должны постоянно контролироваться. Тестировщики должны искать способы доводить до команды информацию, позволяющую принимать правильные решения на каждой итерации. К тому же изменения в коде могут повлиять на функциональность из предыдущих итераций. Таким образом, ручные и автоматические тесты должны модифицироваться для предотвращения регрессионных рисков.

2.2.1. Обмен информацией по статусу тестирования, прогрессу и качеству продукта

Команды, работающие по гибким методологиям, должны выпускать работоспособное ПО в конце каждой итерации. Чтобы определить, когда у команды будет готово работоспособное ПО, необходимо отслеживать прогресс всех рабочих составляющих для каждой итерации и каждом релизе. Тестировщики в командах используют различные методы учета прогресса и статуса тестирования. Например, написание отчетов по результатам автоматического тестирования, представление последовательности тестовых задач и пользовательских историй на доске задач, изображение диаграммы «сгорания» задач и графиков, отражающих статус команды. Это может быть доступно другим членам команды с использованием базы знаний, электронных писем в стиле информационной доски, а также устно в рамках ежедневных митингов, команды могут использовать различные инструменты для автоматической генерации статусных отчетов, основывающихся на результатах и прогрессе тестирования. Эти ин-

струменты автоматически обновляют wiki доски, посылают письма, а также собирают метрики, которые могут быть использованы для улучшения процесса. Такой метод предоставления статуса тестирования экономит время тестировщиков, позволяя сосредоточиться на разработке и выполнении большего количества тестов.

Команды могут использовать диаграмму сгорания задач для отслеживания прогресса на протяжении всего релиза и в течении каждой итерации. Диаграмма сгорания задач показывает количество оставшихся работ, которые необходимо выполнить за время, выделенное на итерацию.

Для предоставления текущего детального статуса всей команды, включая статус тестирования, можно использовать доску задач. Пользовательские истории, задачи разработчиков, задачи тестировщиков, и другие задачи, созданные во время планирования итерации (см. раздел 1.2.5) отражаются на доске задач, обычно используя цветные карточки для определения типа задачи. Во время итерации ход разработки управляется движением этих задач по доске по колонкам «сделать», «в процессе», «проверить» и «сделано». Команды, работающие по гибким методологиям, могут использовать инструменты для автоматизации поддержки своих карточек с историями и досок задач.

Задачи тестирования на доске задач относятся к критериям приемки, определенным для пользовательских историй. Как только автоматические тесты, ручные тесты или исследовательское тестирование оканчиваются, они передвигаются в колонку «сделано» на доске. Вся команда регулярно следит за статусом работ на доске задач, обычно во время ежедневных митингов, что обеспечивает своевременное перемещение задач по доске. Если какие-либо задачи (включая тестовые) не двигаются или двигаются очень медленно, команда выявляет проблемы, которые могут блокировать выполнение этих задач.

Ежедневные встречи должны посещаться всей командой, включая тестировщиков. На этих встречах каждый сообщает статус своей работы. План отчета для каждого члена команды [согласно Agile Alliance Guide] выглядит так:

- Что было сделано с момента последней встречи
- Что планируется сделать до следующей встречи
- Какие есть проблемы

Так как любые проблемы, блокирующие работу, обсуждаются на ежедневных встречах, то вся команда о них знает и может помочь с их решением.

Для улучшения качества продукта многие команды проводят опросы удовлетворенности заказчиков, с целью получения их отзывов о продукте и выяснения соответ-

ствует ли продукт ожиданиям клиентов. Для работы над качеством продукта могут использоваться дополнительные показатели, аналогичные описанным в традиционных методологиях разработки. Например, соотношение количества прошедших / не прошедших тестов, количество обнаруженных дефектов, результаты приемочного и исследовательского тестирования, плотность дефектов, количество найденных и исправленных дефектов, покрытие требований тестами, покрытие рисков, покрытие кода, объем измененного кода.

Как и в любом жизненном цикле, собранные и предоставленные метрики должны быть актуальны и помогать в принятии решений. Метрики не должны использоваться для поощрения, наказания, или изолирования кого-либо из членов команды.

2.2.2. Управление регрессионными рисками, используя ручное и автоматическое тестирование

В проектах под управлением гибких методологий продукт увеличивается в размерах по окончанию каждой итерации. Таким образом, объем тестирования тоже увеличивается. Помимо тестирования изменений в коде, тестировщики должны проверить, не появилась ли необходимость регрессионного тестирования функциональности, разработанной и проверенной в предыдущих итерациях. Риск появления такого тестирования в гибкой разработке велик из-за экстенсивного изменения кода (добавленных, измененных или удаленных строк кода между двумя версиями). Поскольку повышенная ответственность за изменения является основным принципом гибких методологий, модификации могут затрагивать и завершенную ранее функциональность. Для поддержания скорости крайне важно, чтобы команды вкладывались в автоматизацию тестирования на всех уровнях тестирования как можно раньше. Также важно, чтобы все тестовые артефакты - автоматизированные тесты, ручные тестовые сценарии и др. - поддерживались в актуальном состоянии на каждой итерации. Рекомендуется, чтобы все тестовые артефакты содержались и поддерживались в специальном инструменте управления конфигурациями. Это необходимо для осуществления контроля версий, для обеспечения доступа к артефактам всеми членами команды, для управления изменениями, сохраняя историческую информацию о тестовых артефактах по мере изменения функциональности.

Поскольку выполнение всех тестов не всегда возможно, особенно в ограниченных по времени проектах, тестировщики должны выделять время в каждой итерации для оценки ручных и автоматических тестовых сценариев из предыдущего и текущего циклов. Это позволит выбрать тесты-кандидаты на регрессионное тестирование, а также те тесты, которые больше не нужны. Тесты, написанные на ранних итерациях для проверки конкретной функциональности, имеют меньшую ценность на поздних итерациях, поскольку функциональности изменены или написаны новые, которые влияют на поведение предшествующих.

Во время анализа тестов тестировщики должны оценивать их пригодность для авто-