Приоритетный национальный проект «ОБРАЗОВАНИЕ»

Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Государственное образовательное учреждение высшего профессионального образования
«Санкт-Петербургский государственный университет
информационных технологий, механики и оптики»

Д. Г. Шопырин

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Управление проектами разработки ПО. Дисциплина «Гибкие технологии разработки программного обеспечения»

Санкт-Петербург

1. Основные принципы гибких технологий разработки программного обеспечения

1.1. Технология программирования

Для того чтобы понять, что такое технология программирования вообще, и что такое гибкие технологии программирования в частности, необходимо ознакомится со спецификой проблем, которые возникают при разработке современного программного обеспечения. Грубо говоря, при разработке любого программного продукта необходимо полностью и в установленные сроки решить поставленную задачу, обеспечив при этом высокий уровень качества. Данные требования можно предъявить как к выполнению лабораторной работы по программирования, так и к разработке современного программного комплекса. Однако в последнем случае разработчик столкнется с решением задач, которые, на первый взгляд, не относятся к дисциплине программирования (в узком смысле). Такими задачами являются, например, определение и спецификация требований, разработка архитектуры системы, управление качеством продукта и так далее. Именно для решения подобных задач и необходима *технология программирования*.

При этом важно отметить, что технология программирования не является набором абстрактным умозаключений и формальных методик. Используемая технология программирования пронизывает все фазы жизненного цикла программного продукта. Любое действие, совершаемое в процессе разработки продукта, выполняется в соответствии с используемой технологией программирования.

Также важно отметить отличие между методологией и технологией программирования. *Методология программирования является формой предписаний и норм, в которых фиксируются содержание и последовательность определённых видов деятельности. Технология программирования является совокупностью приемов и способов выполнения определенных видов деятельности.* Исходя из вышеприведенных определений ясно, что технология программирования присутствует при разработке любого программного продукта, даже если это и не закреплено формальной методологией разработки.

Одним из ключевых понятий технологии разработки программного обеспечения, как и многих других областей деятельности, является понятие *проекта*. *Проект* есть уникальное временное предприятие, направленное на создание определенного, уникального продукта и услуги. Технология *управление проектом* есть совокупность знаний, навыков, инструментов и методов для планирования и реализации действий, направленных на достижение поставленной в рамках проекта цели.

Современные программные системы разрабатываются в исключительно сложной обстановке. На пути современного проекта по разработке программного обеспечения встают многочисленные организационные и технические препятствия. Можно выделить следующие

переменные, которые влияют на сложность проекта по разработке программного обеспечения:

- Наличие высококвалифицированных специалистов на рынке труда. Причем чем новее используемая технология, тем меньше доступно специалистов, ей владеющих.
- Стабильность используемой технологической платформы. Чем новее используемая платформа, тем меньше ее стабильность.
- Стабильность и функциональность используемых инструментов разработки. Чем новее и мощнее используемый инструмент разработки, тем меньше специалистов способны с ним эффективно работать, и тем менее стабильна его функциональность.
- Эффективность используемых методов разработки, включая методы моделирования, проектирования, тестирования и управления версиями.
- Доступность специалистов, обладающих экспертизой в прикладной области.
- Используемая методология и ее соответствие данному проекту.
- Ситуация на рынке и ее влияние на сроки проекта и планируемую функциональность продукта.
- Сроки и финансирование проекта.
- Множество других организационных и технических переменных.

Можно предположить, что сложность проекта является функцией от вышеприведенных переменных. Важно понимать, что любая из этих переменных может изменять свое значение в течение жизненного цикла проекта. Например, может существенно измениться ситуация на рынке, что может привести к сжатию сроков и увеличению объема запланированных работ.

Для управления проектами с высокой сложностью требуются развитые инструменты контроля и управления рисками. Однако в управлении проектами выделены следующие проблемы:

- Большинство процессов разработки неуправляемы. Исходные данные и желаемый результат многих процессов разработки неизвестны, или определены очень нечетко. Более того, процесс достижения желаемого результата не поддается формализации. Примерами неуправляемых процессов разработки являются разработка архитектуры и исчерпывающее тестирование продукта.
- Идентифицированные процессы разработки сопровождаются неизвестным количеством неидентифицированных процессов разработки. Например, в процессе моделирования и доказательства адекватности модели возникает множество

процессов разработки, которые с трудом поддаются идентификации и формализации.

- Требования к продукту часто меняются в течение жизненного цикла проекта, что требует сложной процедуры изменения и согласования требований.

Попытки предложить формальную, детализованную методологию разработки программного обеспечения оказываются безуспешны, потому что сам процесс разработки не поддается детализации и формализации. Слепое следование методологиям, предполагающим управляемость и предсказуемость процессов разработки приводит к непредсказуемым результатам проекта.

1.2. Традиционные методологии разработки программного обеспечения

Несмотря на то, что процесс разработки программного обеспечения является плохо определенным и динамичным процессом, известно несколько формальных и детализованных методологий разработки программного обеспечения. Подробное описание традиционных методологий разработки программного обеспечения выходит за рамки данного пособия. Однако мы кратко остановимся на двух наиболее распространенных традиционных методологиях разработки программного обеспечения: водопадной и спиральной.

Водопадная методология является одной из первых предложенных формальных методологий разработки программного обеспечения. Схематически водопадная модель может быть изображена следующим образом:

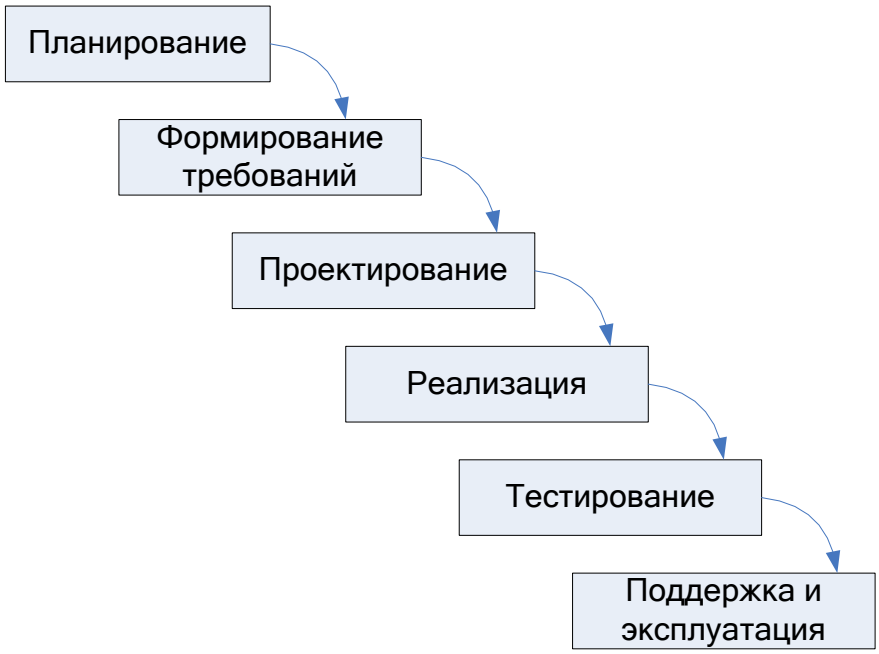

Рис. 1. Водопадная модель

Несмотря на то, что водопадная методология предполагает наличие неопределенных процессов разработки, ее линейная природа не определяет способа реакции на непредвиденные результаты любого из промежуточных процессов разработки. Более того, на каждом этапе проекта существует необходимость в некоторой степени предвидеть результаты последующих этапов. Например, на этапе планирования необходимо предвидеть результаты всех последующих этапов проекта, что часто оказывается практически невыполнимым из-за сложности и неопределенности процессов разработки программного обеспечения. Это является основными проблемами водопадной методологии.

Спиральная методология разработки программного обеспечения решает основную проблему водопадной методологии. Каждая фаза водопадного процесса разработки в спиральной методологии завершается

этапом прототипирования и управления рисками. Схематически спиральная модель может быть изображена следующим образом:

Рис. 2. Спиральная модель

Этап прототипирования после каждой фазы проекта позволяет определить, насколько текущее состояние проекта соответствует первоначальному плану. По итогам прототипирования выполняется либо переход к следующей фазе, либо возвращение на одну из предыдущих фаз. Однако фазы и последовательность фаз остаются линейными. Что по прежнему не вполне соответствует реальному положению вещей, когда процессы разработки программного обеспечения являются неопределенными и непредсказуемыми.

1.3. Гибкие технологии разработки программного обеспечения

В гибких технологиях разработки программного обеспечения используется принципиально другой подход к борьбе с возрастающей сложностью и непредсказуемостью проектов. Гибкие технологии предполагают максимальную гибкость всех процессов разработки при наличии необходимого уровня контроля над ходом проекта.

Как было показано выше, разработка программного обеспечения сталкивается с высокой сложностью и непредсказуемость процессов разработки ПО. Гибкие технологии разработки программного обеспечения минимизируют риски благодаря разделению процесса разработки на маленькие промежутки времени. Такой промежуток времени называется *итерацией* и обычно занимает от одной до четырех недель. Каждая итерация может рассматриваться как полноценный проект по разработке программного обеспечения. Так, итерация может включать в себя все основные процессы разработки, такие как планирование, анализ требований, проектирование, реализация, тестирование и документирование.

Обычно, результатом итерации не является продукт, готовый к выходу на рынок. Но целью каждой итерации является получение стабильной версии продукта. В конце каждой итерации происходит переоценка приоритетов проекта, что значительно сокращает риски.

Манифестом гибких технологий разработки программного обеспечения являются следующие основные принципы:

- Личности и взаимодействия важнее, чем процессы и инструменты.
- Работающее программное обеспечение важнее, чем всеобъемлющая документация.
- Сотрудничество с заказчиком важнее, чем переговоры по контракту.
- Адаптивность к изменениям важнее, чем следование плану.

Манифест гибких технологий основан на следующих основополагающих соображениях:

- Высший приоритет – это удовлетворение требований заказчика путем скорой и непрерывной поставки ценного и работоспособного программного обеспечения.
- Приветствуются изменяющиеся требования, даже на поздних фазах разработки. Гибкие технологии разработки программного обеспечения используют изменения для повышения конкурентоспособности продукта.
- Работоспособное программное обеспечение поставляется как можно чаще, периодами от пары недель до пары месяцев, с предпочтением к более коротким интервалам времени.

- Бизнесмены и разработчики на протяжении всего проекта ежедневно работают сообща.
- Проекты строятся вокруг мотивированных личностей. Этим личностям оказывается доверие и для них создаются все необходимые условия работы.
- Наиболее эффективным и действенным способом передачи информации (как внутри команды разработчиков, так и вовне) является разговор лицом к лицу.
- Основной мерой прогресса является работоспособное программное обеспечение.
- Гибкие технологии разработки программного обеспечения устанавливают удобный режим ведения разработки. Спонсоры, разработчики и пользователи должны быть способны постоянно придерживаться такого режима в течении всего проекта.
- Непрерывное внимание к техническому совершенству и хорошему дизайну повышает гибкость.
- Простота есть искусство максимизации количества невыполняемой работы.
- Лучшие архитектурные решения, наборы требований и дизайны создаются самоорганизующимися командами.
- Команда регулярно рассматривает и внедряет любые методы повышения собственной эффективности.

Гибкие технологии разработки программного обеспечения представлены разнообразными методологиями разработки, такими как *экстремальное программирование*, *Scrum, Feature Driven Development* и т.д. Однако все гибкие методологии имеют некоторые общие характеристики, такие как

- итеративная разработка;
- фокус на взаимодействии и коммуникации;
- полный или частичный отказ от создания дорогостоящих промежуточных артефактов проекта.

Применимость гибких технологий разработки может быть рассмотрена с нескольких точек зрения. С точки зрения создаваемого продукта, гибкие технологии особенно хорошо применимы в случае, когда требования к продукту часто и неожиданно меняются. С другой стороны, существует мнение, что гибкие технологии не применимы для разработки систем, к которым предъявляются высокие требования надежности и безопасности.

С точки зрения организации, на применимость гибких технологий влияют такие факторы, как культура организации, состав коллектива и сложившиеся коммуникации. Основными факторами, которые обеспечивают успех внедрения гибких технологий, являются:

- культура, сложившаяся в организации, должна способствовать переговорам и принятию компромиссных решений;
- персоналу должно оказываться доверие со стороны руководства;

- высококвалифицированный (но, возможно, менее многочисленный) персонал;
- технический персонал имеет право принимать решения, оказывающие влияние на ход проектов;
- условия труда должны способствовать интенсивному общению членов команды.

С точки зрения размера проекта, гибкие технологии хорошо применимы на проектах небольшого и среднего размера (до 40 человек). Использование гибких технологий для реализации крупных проектов ограничено тем фактором, что в случае большой команды достаточно сложно организовать коммуникацию «лицом к лицу».

Большинство гибких методологий разделяют общих набор практик разработки, таких как разработка через тестирование, игра в планирование и т.д. Каждая конкретная гибкая методология во многом опирается на разнообразные практики гибкой разработки, без эффективного выполнения которых невозможен успех проекта. В настоящем пособии будут подробно описаны все основные практики гибких технологий разработки программного обеспечения. Ниже будет произведен краткий обзор некоторых основополагающих практик.

Разработка через тестирование является краеугольным камнем гибких технологий программирования. Использование большинства других гибких практик разработки программного обеспечения (например, рефакторинга или непрерывной интеграции) без использования разработки через тестирование может оказаться неоправданным и даже опасным. Основные принципы разработки через тестирование:

- автоматические тесты пишутся для любой части реализации, которая гипотетически «может сломаться»;
- автоматические тесты пишутся непосредственно *перед* написанием соответствующего кода;
- уже существующий код никогда не меняется без написания соответствующих автоматических тестов;
- выполняется регулярный запуск всех автоматических тестов.

Стандарты кодирования позволяют внедрить такие практики, как парное программирование и коллективное владение кодом. Стандарты кодирования, используемые в гибких технологиях разработки программного обеспечения, удовлетворяют следующим основным требованиям:

- стандарт кодирования позволяет упрощать как написание, так и прочтение кода;
- стандарт кодирования облегчает коммуникацию между членами команды;
- стандарт кодирования относительно добровольно воспринимается всеми членами команды.

Парное программирование обеспечивает высокое качество всего кода, который создается в течение гибкого проекта. Во время парного

программирования два программиста работают за одним компьютером. Грубо говоря, один из программистов занят непосредственным написанием кода, в то время как другой мыслит стратегически, обдумывая такие вопросы, как работоспособность выбранного решение, нетривиальные варианты использования и т.д. Обязанности программистов внутри пары и сам состав пар непрерывно меняются в течение проекта. Основные достоинства парного программирования следующие:

- увеличивается дисциплина разработчиков.
- значительно увеличивается качество кода и степень его покрытия автоматическими тестами;
- информация о разрабатываемой системе свободно распространяется внутри команды в устной форме.

Игра в планирование позволяет быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как условия задачи становятся все более четкими. Основным артефактом игры в планирование является набор бумажных карточек, на которых записаны пожелания заказчика и приблизительный план работы по выпуску следующей версии. Основным плюсом игры в планирование является тот факт, что заказчик отвечает за принятие всех бизнес-решений, в то время как команда принимает все технические решения. При этом ответственность распределяется следующим образом.

Команда отвечает за:

- формирование оценки времени, необходимого для реализации каждого из пожеланий;
- оценку затрат, связанных с использованием той или иной технологии;
- распределение задач между членами команды;
- оценку рисков, связанных с реализацией пожеланий;
- определение порядка реализации пожеланий в рамках текущей итерации.

Заказчик отвечает за:

- определение объема работ (набора пожеланий на итерацию или релиз);
- определение даты завершения работы над релизом;
- назначение приоритетов для каждого из пожеланий.

Эти и другие практики гибких технологий разработки программного обеспечения формируют сложный и тесно взаимосвязанный фреймворк, позволяющий значительно минимизировать объем рисков, связанных с проектом по разработке программного обеспечения.

2. Разработка через тестирование

Тестирование – это процесс выполнения программного продукта с целью выявления дефектов. Невозможно полностью протестировать программу, поскольку число вариантов работы нетривиальной программы может быть бесконечно большим. Следовательно, тестирование не может доказать отсутствие ошибок в программном коде, оно может показать только наличие ошибок.

Тестирование – это очень важный и вместе с тем трудоемкий вид деятельности разработчиков программного обеспечения. Время, использованное на тестирование, требует значительных затрат, поэтому желательно начать тестирование как можно раньше, чтобы получить от этих затрат максимальную прибыль. Чем больше дефектов будет обнаружено при тестировании программы, тем выше выигрыш от вложений на тестирование.

Основным отличием процесса тестирования в гибких технологиях является то, что тестирование рассматривается как *разрушительный процесс*. Более того, в гибких технологиях применяется подход, известный как *разработка через тестирование* (*test-driven development*). Благодаря использованию этого подхода гарантируется тестирование всего кода разрабатываемой программной системы.

Разработка через тестирование осуществляется на всем протяжении процесса создания программного продукта. Оно является неотъемлемой частью процесса разработки и даже, более того, упрощает этот процесс:

- дизайн разрабатываемого программного продукта настолько прост, насколько это вообще возможно, поэтому разработка тестов – это не такая уж сложная процедура;
- программирование выполняется в паре, вместе с партнером, который постоянно инспектирует создаваемый код, оказывает помощь в составлении тестов и в любой момент может включиться в процесс разработки;
- постоянное растущее число корректно работающих тестов создает благотворную атмосферу для разработчиков;
- заказчик удовлетворен, когда он принимает в эксплуатацию версию, в которой выполняются все разработанные им тесты.

При выполнении этих условий разработчики и заказчики, скорее всего, с большим желанием будут тратить время на разработку тестов. Уверенность в том, что программный продукт работает корректно, будет подкрепляться набором тестов, которые продолжают функционировать по мере продолжения работы над проектом.

В гибких технологиях программирования в основном используется два вида тестирования:

- модельное тестирование (*unit testing*);
- функциональное тестирование (*functional testing*).

Кроме этих двух основных видов тестов могут использоваться и другие виды тестирования, использование которых может быть оправдано в определенных ситуациях. Остановимся подробно на каждом виде тестирования.

2.1. *Модульное тестирование*

Для успешного завершения проекта необходимо обеспечить автоматическое тестирование как отдельных модулей, так и всего программного продукта. Создание тестов для тестирования модулей (модульных тестов) должно предшествовать разработке программы. Эрих Гамма придумал термин «инфицированный тестами» (Test Infected) для людей, которые не приступают к кодированию до тех пор, пока у них не будет набора тестов для тестирования разработанного кода.

Модульное тестирование соответствует идеологии тестирования «белого ящика» (более правильным было бы название «стеклянного ящика»). Программист видит код, для которого пишет тест.

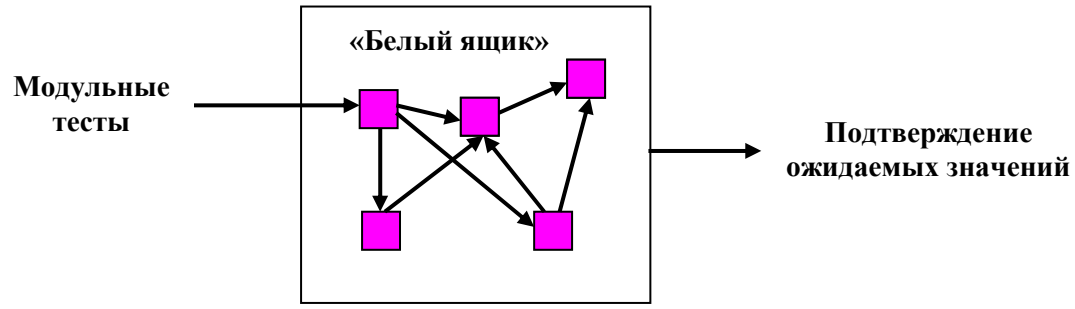

Рис. 3. Модульное тестирование

Целью модульного тестирования является проверка функционирования фрагментов программного кода (методов, классов и их комбинации, если известны связи между ними).

Модули (классы в объектно-ориентированной среде), к которым применяется модульное тестирование, являются блоками при построении программной системы, а не отдельными кирпичами, из которых строится дом. Хотя на качество дома не сильно повлияют несколько бракованных кирпичей, программный продукт может оказаться очень чувствительным к дефектам в отдельных блоках конструкции. Если дефектные модули будут встроены в программы, может понадобиться огромное количество времени на их нахождение и исправление. Поэтому модули должны быть абсолютно надежными, что и является целью модульного тестирования.

Модульное тестирование осуществляется программистами в процессе разработки программной системы. Все существующие модульные тесты должны выполняться корректно. Протестировать абсолютно все невозможно – для этого тесты должны быть столь же сложными и столь же защищенными от ошибок, как и сам код. Необходимо создавать тесты для всего, что может дать сбой.

Как правило, не могут содержать ошибки операторы присваивания, если присваивается константа или простая переменная (такие ошибки легко находятся при инспекции кода), но должны тестироваться операторы, присваивающие значения выражений. Не могут внести ошибку и простые

методы конструктора параметров, которые просто берут набор значений и присваивают эти значения переменным экземпляра, хотя более сложные конструкторы должны тестироваться. Как правило, написание теста должно предшествовать кодированию в следующих случаях:

- если интерфейс метода неясен;
- если интерфейс ясен, но при реализации могут возникнуть проблемы;
- если код должен работать в необычных условиях.

В случае обнаружения ошибки необходимо создать новый тест, чтобы предотвратить ее повторное появление. Процесс написание тестов по поводу всех обнаруженных ошибок называется регрессионным тестированием. По мере приобретения опыта, команда будет лучше чувствовать, что и когда действительно нужно тестировать. Всегда следует руководствоваться правилом, что лучше составить больше тестов, чем недостаточно протестировать программу.

Если предположить, что для разработки команда использует объектно-ориентированный язык, разработчики в первую очередь должны разработать все возможные тесты для каждого метода, а затем только закодировать эти методы. Реализация должна обеспечивать корректное выполнение всех тестов метода и не более того. Благодаря тому, что кодирование выполняется после того, как будут написаны все тесты, команда разработчиков получает:

- наиболее полный набор всевозможных тестов;
- наиболее простой код, который реализует заданную функциональность;
- четкое представление о том, какую функцию реализует код.

Вильям Уэйк (William Wake) [5] для описания порядка написания тестов и кодирования использует следующую *метафору светофора*.

Желтый свет – пишите тест и компилируйте. Скомпилировать его не удастся.

Красный свет – пишите код, необходимый для компиляции теста. Скомпилируйте тест, запустите его. Тестирование не пройдет успешно.

Зеленый свет – пишите код, который необходим для успешного прохождения теста. Запустите его. Он пройдет успешно.

Переход от состояния к состоянию, не соответствующий последовательности «желтый – красный – зеленый», является признаком проблемы. Рассмотрим разные варианты переходов.

От зеленого к зеленому

Только что написанный тест скомпилирован и работает. Это означает, что либо тест тестирует написанные ранее методы, либо тест фиктивный. Чтобы убедиться, что тест корректный, в него необходимо внести ошибку.

От зеленого к красному

Компиляция прошла успешно, но тест не сработал. Если такая ситуация возникает при работе над новым тестом для существующих

методов – это нормально. Если же это тест для новых методов, то допущена какая-то ошибка.

От желтого к желтому

В коде заглушки допущена синтаксическая ошибка, которую обнаруживает компилятор.

От желтого к зеленому

Тест для кода заглушки (пустой метод) сработал. Пустые методы, которые не реализуют никаких функций, не должны проходить тесты. Однако если в качестве языка программирования используется C++ или Java, то для методов, возвращающих значение, необходимо ввести оператор возврата, который должен будет в случае успешного прохождения теста возвратить 0 или *null*.

От красного к желтому

В методе, который только что добавлен, есть синтаксическая ошибка, которую обнаруживает компилятор.

От красного к красному

Добавленный метод не работает. Необходимо исправить ошибку и запустить тесты снова.

Уэйк утверждает, что на описанную процедуру должно уходить около 10 – 15 минут для каждого теста. Если разработчик не укладывается в это время, необходимо писать более короткие тесты.

Если выполнение полного набора тестов занимает слишком много времени, можно запустить только тесты, касающиеся задач, которые в данный момент разрабатываются (например, при программировании на Java это может быть пакет, над которым производится работа). Идеальным является подход, при котором тесты образуют иерархию. Это дает возможность легко запустить тесты для отдельного класса, группы классов пакета, всего программного продукта.

Модульные тесты необходимо помещать в библиотеку кодов вместе с кодом, который они тестирует. Тесты делают возможным коллективное создание исходных текстов и защищают их от неожиданных изменений. Тесты следует автоматизировать, чтобы их можно было выполнять регулярно. По мере продвижения работы над проектом, количество тестов постоянно увеличивается. Основываясь на результатах тестов, разработчики включают в очередную итерацию работу над ошибками.

Модульные тесты лежат в основе выполнения рефакторинга кода. При этом тестирование необходимо проводить как *перед факторингом*, так и *после рефакторинга*. В любой момент разработчик может модифицировать любой фрагмент кода, не опасаясь, что будет нарушена работоспособность кода, – тесты модулей немедленно сообщат ему, корректно ли были внесены изменения. Если при выполнении рефакторинга некоторого фрагмента кода обнаруживается, что существующие тесты не могут обеспечить необходимой уверенности (например, они тестируют не все методы класса или при их разработке учитывались не все комбинации параметров), прежде чем приступить к рефакторингу кода, необходимо выполнить *рефакторинг*

соответствующих тестов. Необходимо следить за чистотой тестов столь же ревностно, сколь и за чистотой кода. Чем лучше будут тесты, тем лучше будет код и тем проще будет выполняться его модернизация и добавление в него новых возможностей.

2.2. Функциональное тестирование

Функциональное тестирование соответствует идеологии тестирования «черного ящика», когда разработчик тестов ничего не знает о внутреннем устройстве программы.

Рис. 4. Функциональное тестирование

Функциональные тесты разрабатываются при участии заказчика для каждого из пожеланий. В идеале разработка функциональных тестов для некоторой итерации завершается еще до того, как разработчики завершат работу над этой итерацией. Как правило, заказчики не могут писать приемочные тесты самостоятельно. Они нуждаются в помощи разработчиков. Именно поэтому команда разработчиков должна включать в себя, по меньшей мере, одного разработчика, одной из основных обязанностей которого будет функциональное тестирование системы в тесном сотрудничестве с заказчиком.

Функциональные тесты могут быть описаны на обратной стороне карточки с пожеланиями заказчика или на отдельной карточке (рис. 5). Во втором случае пожелание заказчика и функциональный тест должны ссылаться друг на друга с помощью уникального номера пожелания заказчика. Цель этого теста – найти простой, конкретный пример, который отображает нужное поведение системы.

Функциональный тест состоит из трех частей:

1. *Установка теста* – описание минимальных действий, которые необходимо выполнить перед запуском приемочного теста.
2. Сам тест, который часто называют *операцией.*
3. *Подтверждени*е – результат выполнения теста (значение или состояние, которое достигается после корректного срабатывания теста).

Рассмотрим пример создания функционального теста для пожелания заказчика. Пусть, результатом корректного выполнения теста должен быть выведенный на экран список гостиниц со свободными одноместными номерами. Заказчик предлагает рассмотреть несколько вариантов:

- в базе данных гостиниц имеется всего одна гостиница, удовлетворяющая заданным условиям;
- в базе данных гостиниц имеется несколько гостиниц, удовлетворяющих заданным условиям;
- в базе данных гостиниц отсутствует гостиница, удовлетворяющая заданным условиям.

На рис. 5 приведено описание функционального теста для первого варианта.

№ 56
Установка: Занести в базу данных гостиницу, в которой есть свободный номер с *05.08.05* по *15.08.05*. Адрес гостиницы: *ул. Мира, д. 14*, стоимость номера 2500 руб.

Операция: Просмотреть базу данных гостиниц и выбрать гостиницу, в которых есть свободные номера на 10.08.05.

Подтверждение: На экран выводится адрес: *ул. Мира, д. 14*, стоимость номера 2500 руб.

Рис. 5. Пример приемочного теста

Функциональные тесты служат для интегрального тестирования функций системы. Они подтверждают, что разрабатываемый программный продукт делает то, что нужно заказчику. Эти тесты рассматривают систему как «черный ящик» и рассчитаны на получение вполне определенного результата работы программного продукта. При тестировании методом «черного ящика» каждый тест никак не взаимодействует с остальными тестами, поэтому сбой в одном из тестов не является причиной несрабатывания других тестов.

Функциональные тесты не обязательно должны срабатывать в любой момент времени на все 100%, поскольку невозможно получить синхронизацию функциональных тестов и кода программного продукта в такой степени, в какой синхронизированы с кодом модульные тесты. В то время как для тестирования модулей может быть только две оценки – 100% или 0%, работоспособность функциональных тестов, как правило, оценивается в некотором процентном отношении. Предполагается, что спустя некоторое время все функциональные тесты должны срабатывать на 100%. По мере приближения сроков выпуска очередной версии программного продукта, заказчик должен классифицировать не срабатывающие функциональные тесты по степени важности функций, которые они проверяют. Прогон и исправление ошибок для более важных функциональных тестов необходимо выполнять в первую очередь.

Функциональное тестирование системы также должно быть автоматизировано. С помощью автоматизации можно после каждой версии вернуться на предыдущий шаг. Это означает, что все функциональные тесты выполняются в каждой версии, даже те, которые ранее использовались в предыдущих версиях. Если при работе программного продукта возникает ошибка, в набор автоматизированных функциональных тестов добавляется

тест, с помощью которого была обнаружена ошибка, и эта ошибка никогда вновь не появится в выпускаемых версиях.

2.3. Другие виды тестов

Кроме модульных и функциональных тестов в разработке через тестирование могут использоваться:

Параллельные тесты

Параллельный тест (*parallel test*) предназначен для того, чтобы доказать, что новый программный продукт работает так же, как старый. На самом деле этот тест может только продемонстрировать, что новый программный продукт отличается от старого. При этом только заказчик может принять решение о том, насколько удовлетворительным для него является различие и можно ли он допустить программный продукт в эксплуатацию.

Стресс- тест

Стресс-тест (*stress test*). Этот вид тестов разрабатывается для имитации максимальной нагрузки на программный продукт. Стресс-тесты должны обязательно разрабатываться для тестирования сложных программных продуктов, для которых трудно делать предположения о производительности.

2.4. Тесты как одна из форм документации

Составление тестов – это неотделимая составная часть разработки кода, поэтому *тесты являются наилучшей формой документации* на уровне модулей (классов). Любые потери времени, связанные с написанием тестов, быстро окупаются благодаря существенному сокращению времени, которое приходится тратить на отладку и написание документации.

Тесты являются более подробной формой документации, чем документация, написанная на бумаге, вместе с тем они не содержат никаких лишних сведений. Они без лишних слов коротко и ясно объясняют, как работает тестируемый код.

Тесты являются более удобной и полезной формой документации, так как они дают представление о сценарии, в котором тестируемый код должен вести себя, так как это ожидается.

В отличие от других форм документации, модульные тесты всегда в точности соответствуют текущему состоянию кода при условии, что соблюдаются правила разработки и модульного тестирования.

Записанные и проверенные заказчиками и разработчиками технические требования и проектные документы, в отличие от тестов, не могут быть откомпилированы и запущены на выполнение, поэтому могут содержать неточности и ошибки.

Конечно, отсюда не следует, что документация при использовании гибких технологий разработки вообще не нужна. Письменное описание программного продукта может потребоваться для презентации, защиты проекта, получения лицензии и в других подобных ситуациях.

2.5. Сложности тестирования

В настоящее время программные продукты разрабатываются для самых разных проблемных областей и аппаратных платформ, поэтому разработка тестов может оказаться довольно сложным делом. Рассмотрим ряд примеров.

2.5.1. Тестирование пользовательского интерфейса

Создание тестов для тестирования пользовательского интерфейса (*User Interface – UI*), которые по эффективности и надежности могли бы сравниться с тестами для бизнес-логики, является серьезной проблемой. Как правило, эффект от применения таких тестов не окупает затраты на их разработку. Это объясняется тем, что пользователи могут задавать функциональность разрабатываемого программного продукта в форме графического пользовательского интерфейса. Сложность тестирования таких программ заключается в том, что в них тесно переплетены пользовательский интерфейс и модули, реализующие бизнес-логику.

Лучший способ тестирования пользовательского интерфейса – это отделить его от бизнес-логики и написать набор тестов для тестирования бизнес-логики. Чем больше кода будет отделено от пользовательского интерфейса, тем лучше он будет оттестирован. Действуя таким образом, можно оттестировать до 90% кода бизнес-логики.

Наиболее часто тестирование пользовательских интерфейсов сводится к тестированию *удобства и простоты использования* программного продукта. Критерии оценки удобства и простоты использования должны быть сформулированы заранее. Например, можно использовать следующие критерии.

- ***Доступность.*** Насколько легко пользователи могут входить в систему, ориентироваться и выходить из системы?
- ***Способность реагировать.*** Насколько быстро программный продукт позволяет пользователям достичь определенных целей?
- ***Эффективность.*** Сколько шагов необходимо сделать, чтобы получить выбранную функциональность?
- ***Ясность.*** Насколько часто пользователи пользуются документацией и вызывают помощь?

Необходимое количество пользователей, которых необходимо привлечь к тестированию, определяется статистически и зависит от предполагаемого числа пользователей продукта и желаемой вероятности правильного заключения.

2.5.2. Тестирование в ограниченном пространстве

Разработчики, создающие программное обеспечение для миниатюрных компьютерных устройств, например, сотовых телефонов или цифровых фотоаппаратов, работают в среде, далекой от условий применения программного обеспечения. Например, при разработке Java-приложений для обычных компьютеров можно использовать любые из стандартных классов и

возможностей Java, в то время, как небольшой размер устройства не позволит разместить все классы Java. В этом случае необходимо создать новую среду для тестирования, при этом дополнительные усилия, затраченные на разработку, модификацию или адаптацию библиотеки тестирования для миниатюрного компьютерного устройства, как правило, окупят себя.

3. Кодирование, рефакторинг и управление исходным кодом.

Разработка программного обеспечения – это не только технический, но и социально-культурный процесс. Большинство традиционных методик разработки программного обеспечения основано на инженерных подходах, которые подразумевают создание программ на основе фиксированных требований путем выполнения последовательности формальных шагов. Традиционные подходы не учитывают тот факт, что программы разрабатываются людьми, которые работают в условиях тесных контактов и культурных связей. В гибких же технологиях разработки программного обеспечения этот фактору уделяется большое внимание.

Основные методики, на которых базируется кодирование в гибких технологиях разработки программного обеспечения, связано с человеческим фактором:

- участие заказчика в разработке;
- применение стандартов программирования;
- составление тестов перед кодированием;
- парное программирование;
- частые интеграции кода;
- коллективное владение кодом;
- рефакторинг кода;
- 40-часовая рабочая неделя.

3.1. Применение стандартов программирования

Стандарты программирования – это важнейшая составляющая процесса разработки программного обеспечения. Гибкие технологии разработки программного обеспечения дают желаемый эффект, если они работают вместе. Невозможно внедрить парное программирование и коллективное владение кодом, если разработчики в рамках одного проекта не будут придерживаться одного стиля программирования. Стандарты программирования способствуют максимальной скорости работы команды и позволяют избавить команду от пустых споров о второстепенных вещах.

Требования к стандарту:

- стандарт должен облегчать прочтение кода системы;
- стандарт должен требовать от разработчиков приложения как можно меньших усилий для реализации любой возможности;
- стандарт должен способствовать воплощению на практике правила запрета на дублирование кода;
- стандарт должен содействовать взаимодействию;
- стандарт должен быть воспринят добровольно всей командой разработчиков.

Если в команде не используются единые стандарты программирования, при смене партнеров в парах возникают серьезные затруднения, труднее выполнять рефакторинг, в общем и целом, продвижение проекта вперед затрудняется. Необходимо добиться того, чтобы было невозможно понять, кто именно из членов команды написал тот или иной фрагмент кода.

Для того, чтобы вся команда могла работать унифицировано, как один человек, команда должна сформировать набор правил, а затем каждый человек должен неукоснительно следовать этим правилам. При этом если на предприятии работает несколько команд разработчиков, стандарт должен быть общим для предприятия. Если на предприятии отсутствует стандарт, не следует пытаться в самом начале работы определить абсолютно все правила кодирования. Это следует делать постепенно, по мере продвижения проекта.

Если недостаток стандартизации является причиной возникновения проблем у некоторых членов команды, необходимо организовать общее совещание для обсуждения этого вопроса. На этом совещании необходимо решить только самые необходимые задачи, не следует превращать это совещание в заседание комитета по формированию очередного стандарта ISO (International Standards Organization (ISO) – Международная организация по стандартизации).

Перечень правил, образующих стандарт, не должен быть исчерпывающим или слишком объемным. Самые общие рекомендации по стандартам кодирования могут быть следующими:

- члены команды должны соблюдать текущий стандарт кодирования, если он есть. Даже если текущий стандарт обладает некоторыми недостатками, лучше соблюдать его, чем вводить несколько *альтернативных* стандартов;
- стандарт кодирования не должен допускать разбрасывания числовых констант (*magic numbers*) по коду. Все числовые константы должны соответствующим образом декларироваться и снабжаться символьными именами;
- стандарт кодирования должен требовать комментирования всех *нетривиальных* мест в коде системы;
- стандарт кодирования не должен одобрять использование тавтологических комментариев в *тривиальных* местах кода системы;
- стандарт кодирования должен содержать точные соглашения по именованию переменных, классов, методов и форматировании кода, имена не должны быть аббревиатурами;
- стандарт кодирования должен отдавать предпочтения длинным идентификаторам, если аналогичные короткие идентификаторы могут быть непонятны или сложны для прочтения;
- стандарт кодирования должен возбранять написание *длинных* методов и функций.

Общий стиль программирования в командах, работающих в соответствии с гибкими технологиями разработки программного обеспечения, вырабатывается со временем. Как отмечает Кен Ауэр [2], «единственным действительно важным стандартом является общее соглашение о том, что разрабатываемый код должен соответствовать самым высоким стандартам».

3.2. Парное программирование

Парное, или совместное, программирование является, наверно, самой спорной практикой гибких технологий разработки программного обеспечения, поэтому остановимся на этом вопросе подробнее. Парное программирование – это процесс создания программного обеспечения двумя программистами, работающими одновременно за одним компьютером, оснащенным одной клавиатурой и одной мышью. На самом деле слово «программирование» в данном случае не очень удачно, так как программирование – это лишь один процесс из множества других, возникающих во время разработки программного обеспечения. Работая в паре, разработчики осуществляют анализ, дизайн, тестирование, кодирование, оптимизацию и рефакторинг кода. Когда используется термин «программирование в паре», то имеется в виду все перечисленные процессы, из которых лишь одно – программирование.

Необходимо также отметить, чем *не является* парное программирование:

- Парное программирование не предполагает, что один человек *работает*, а второй на это *смотрит*. В каждой паре существует две роли: первый партнер, который сидит за клавиатурой и непосредственно программирует, а второй – следит за направлением работы, помогает разобраться в сложных местах, предлагает альтернативное решение и исправляет ошибки. Таким образом, обе роли являются активными.
- Парное программирование также не является сеансом обучения. Конечно, иногда случается так, что пара программистов существенно отличается с точки зрения опыта или понимания предметной области. В этом случае первые сеансы программирования в такой паре будут похожи на уроки: менее опытный партнер задает много вопросов и разрабатывает относительно немного кода. Однако спустя относительно небольшое время уровень программистов в паре выравнивается настолько, что они начинают продуктивно работать на равных. В результате чего увеличивается производительность и возрастает удовлетворение, получаемое от сделанной работы.
- Парное программирование также не является средством от скуки. Два разработчика интенсивно работают над решением самых актуальных задач. При этом их совокупный опыт и знания гарантируют, что они выполняют свою работу максимально эффективным образом.

Состав пар может и должен меняться динамически. Партнеры, которые утром работали вместе, днем могут войти в состав других пар. При этом необходимо учитывать следующее правило: нежелательно менять партнера тогда, когда решение задачи, над которой работает пара, еще не завершено. Заметим, что для того, чтобы привыкнуть работать в паре, требуется

определенное время. Вначале партнеры чувствуют себя неловко. Со временем это чувство исчезает, и через некоторое время члены команды начинают эффективно работать в паре практически с любым возможным партнером.

В отношении парного программирования существуют прямо противоположные мнения: от резко отрицательных до восторженных. Противники парного программирования считают, что посадить двух программистов за один компьютер, значит поручить двум разработчикам выполнять работу одного из них. С точки зрения руководителя программист слишком ценный ресурс, чтобы тратить его понапрасну, удваивая количество людей. Многие программисты, особенно с большим опытом, отказываются работать в паре, мотивируя это тем, что они привыкли считать свою работу индивидуальным, а не коллективным трудом.

И в то же время довольно много известных и уважаемых программистов предпочитают парное программирование любому другому стилю работы. Что касается качества программы, то опыт показывает, что при парном программировании программный продукт имеет лучший дизайн и более простой код. Согласно опросам, даже программисты-новички, работающие в паре с опытными специалистами, вносят код гораздо меньше ошибок чем при одиночном программировании. Те же программисты, которые привыкли к парному стилю работы, утверждают, что так работать вдвое быстрее.

Доказательством эффективности парного программирования занимались многие исследователи. Наилучший обзор этих доказательств приводится Алистером Коуберном (Alistair Cockbern) в [7]. Рассмотрим более подробно, что дает парное программирование.

3.2.1. Советы по использованию парного программирования

Разные пары могут работать по-разному, отношения между напарниками со временем могут изменяться. Однако существуют некоторые тонкости использования парного программирования, которые необходимо знать.

Необходимость частых пауз

Парное программирование – это очень интенсивный процесс. Им невозможно заниматься без остановок в течение полного рабочего дня. Программирование в паре делает более эффективной работу над проектом, но оно требует от разработчиков большей отдачи сил. Программист, работающий в паре, устает быстрее, чем программист-одиночка. Для того чтобы поддерживать силы программистов, необходимо делать частые перерывы. Это может быть полномасштабный перерыв на обед, перерыв на чашку кофе или просто отдых в течение нескольких минут.

Эффект наблюдателя

Разработчик, который не работает за клавиатурой и мышью, как правило, находит быстрее нужные пункты меню, элементы интерфейса и элементы списка, чем разработчик, владеющий клавиатурой. Это может стать причиной напряжения и спровоцировать конфликт. Необходимо понимать, что это естественный процесс, на который обратили внимание многие программисты, работающие в паре. Если партнеры меняются ролями, эффект остается в силе: тот кто раньше находил быстрее нужные кнопки и пункты меню, получив в руки мышь и клавиатуру, как по волшебству, становится более медлительным и невнимательным. Если оба напарника будут знать об этом эффекте, они смогут избежать множества проблем.

Изучение инструментальных средств

Программисты, работающие в паре, должны обсуждать наиболее эффективные способы использование инструментальных средств. Современные инструментальные средства являются достаточно сложными программными продуктами, изучение всех встроенных в них возможностей может потребовать значительного времени. Пара сможет работать эффективнее, если партнеры будут делиться между собой опытом.

Формирование общего словаря

Программирование в паре основано на постоянном тесном общении. Если все члены команды пользуются единым словарем терминов и единым представлением о дизайне, общая скорость работы существенно повышается.

Общий для всей команды словарь терминов должен охватывать терминологию предметной области, для которой разрабатывается программный продукт, элементы и объекты разрабатываемой системы, имена паттернов из языка паттернов [6] и т. д.

Уточнение дизайна

Даже после того, как общие направления дизайна определены, в проекте остаются вопросы, которые должны быть решены во время работы над конкретной реализацией. Как правило, на ранних стадиях проекта попытки заполнить эти пробелы могут привести к длительным абстрактным дискуссиям. Если партнеры в течение часа не нашли приемлемого решения, это говорит о том, что они не понимают задачу достаточно хорошо. В этом случае рекомендуется выбрать самый простой вопрос и написать код, который на него отвечает. Такой подход либо поможет разработчикам найти правильное решение, либо продемонстрирует несостоятельность выбранного пути решения проблемы.

3.2.2. Преимущества парного программирования

Экономическая целесообразность

Ключевой вопрос, возникающий при обсуждении целесообразности перехода на парное программирование – это стоимость такого перехода. Скептики полагают, что переход на парное программирование влечет за

собой удвоение расходов на разработку программного обеспечения. Однако для правильной оценки затрат на разработку программного обеспечения необходимо учитывать затраты не только непосредственно на программирование, но и затраты, связанные с контролем качества и сопровождением программного продукта. Например, IBM утверждает, что они потратили около 250 миллионов долларов на устранение 30 000 дефектов, о которых заявили пользователи, т. е. по 8 000 долларов на каждую ошибку.

Как показывают эксперименты по анализу экономической целесообразности парного программирования [7], при программировании в паре после начального периода «притирки» партнеров, которая проходила во время работы над первой программой, затраты времени на разработку увеличиваются на 15% (рис. 6), при этом результирующий код содержит на 15% меньше дефектов (рис. 7).

Изначальное 15% увеличение стоимости разработки окупается за счет уменьшения количества ошибок. Предположим, что программа в 50 000 строк кода (50 000 LOC) разрабатывается программистом-одиночкой и программистами, работающими в паре. При средне статистической скорости разработки 50 LOC в час программист-одиночка напишет эту программу за 1000 часов, в то время как программисты, работающие в паре, затратят на эту программу на 15% времени больше, т. е. стоимость разработки вырастет на 150 часов. Согласно статистике, средний программист совершает 1 ошибку на 100 строк кода.

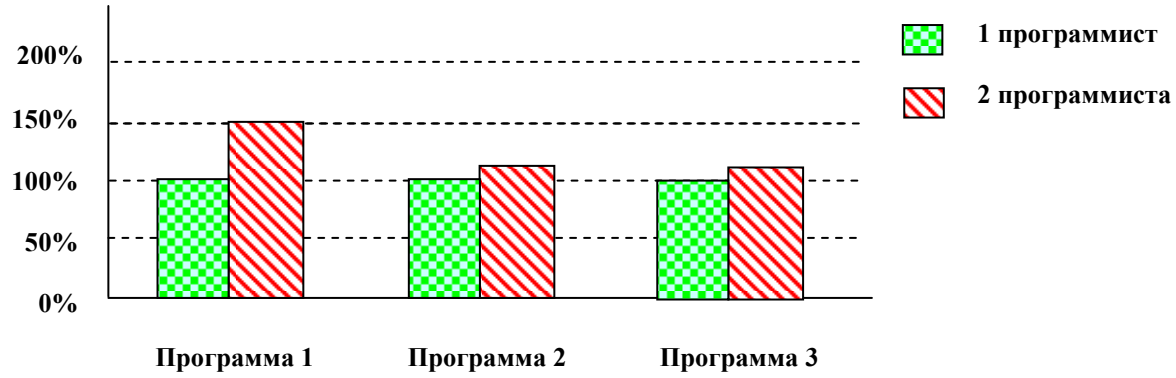

Рис. 6. Время, затраченное на программирование задач

Правильно поставленный процесс тестирования позволяет выявить около 70% этих ошибок. Следовательно, в первом случае в программе останется порядка 1500 ошибок, а в программе разработанной парой программистов, их будет на 15% меньше, т.е. 1275 ошибок.

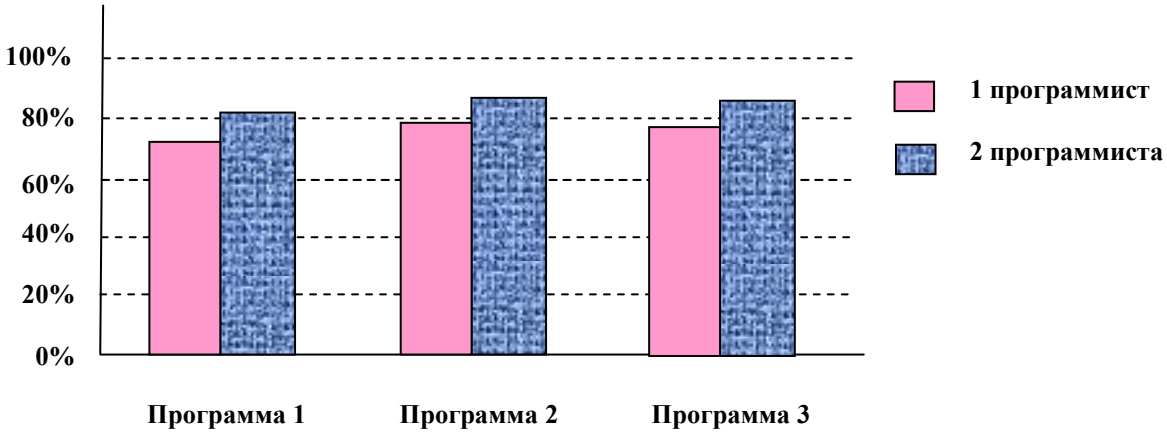

Рис. 7. Количество ошибок в программах

Обычно на тестирование и исправление компанией-разработчиком одной системной ошибки уходит от четырех до шестнадцати часов. Если взять среднее значение – 10 часов, то на исправление «лишних» 225 ошибок отделу тестирования или службе качества компании необходимо 2250 часов, а это в 15 раз больше, чем увеличение затрат, связанное с внедрением парного программирования. Если же по окончанию работ программа передается непосредственно заказчику, то парное программирование выигрывает еще больше. По данным статистики, после начала эксплуатации программы на исправление одной ошибки требуется от 33 до 88 часов. Если взять оптимистичный вариант – 40 часов на одну ошибку, то стоимость исправления дополнительных 225 ошибок обойдется компании-разработчику в 9 000 часов, что в 60 раз превышает затраты, требуемые для организации парного программирования.

Удовлетворение от работы

Опрос программистов-профессионалов, проведенный через Интернет, говорит о том, что 85% опрошенных предпочитают работать в паре. Одной из причин они называют возросшую уверенность в качестве своего кода, что подтверждается и статистическими данными.

Улучшение качества дизайна

Результаты исследований, приведенные в [7], демонстрируют, что работая в паре, программисты производят более качественный код и реализуют заданную функциональность при помощи меньшего количества строк, чем одиночный программист (рис. 8).

Ник Флор (Nick Flor) [7], исследовавший соотношение между вербальным и невербальным поведением программистов, с помощью видео и аудио аппаратуры фиксировал все виды обмена мнениями между двумя программистами, работающими в паре. Он сделал интересный вывод о распределении знаний у программистов, работавших в паре: «Распределенное знание - это один из разделов когнитологии, основное положение которого можно выразить словами: «Все, кому приходилось

изучать процесс осмысления, были поражены тем фактом, что «разум» очень редко работает в одиночку. Все данные, которые человек поднимает во время этого процесса, оказываются распределенными – по различным умам, людям, а также символическому и физическому окружению, в котором этот человек находится».

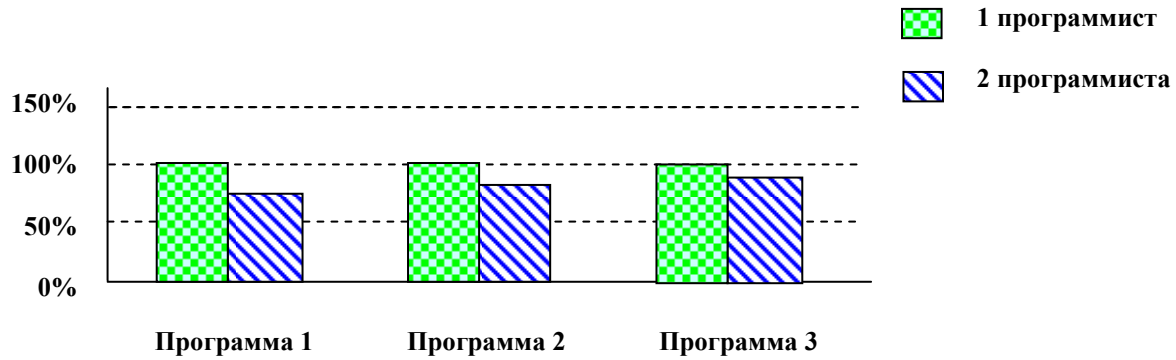

Рис. 8. Количество строк кода для одной и той же функциональности

Ник Флор также выделил три причины, по которым программисты, работающие в паре, обладает большим потенциалом. Во-первых, все действующие лица привносят в разработку свой уникальный личный опыт; во-вторых, каждый из них обладает своим подходом к информации, касающейся выполняемой задачи; в-третьих, все они находятся в разных отношениях к проблеме, поскольку выполняют разные функциональные роли. Вырабатывая единое общее решение, они перебирают гораздо большее количество альтернативных вариантов, чем мог бы в такой ситуации программист-одиночка. Именно это обстоятельство и ведет к снижению риска выбора плохого решения.

Повышение дисциплины

Замечено, что парное программирование значительно повышает дисциплину разработчиков. Особенностями психики и физиологии человека обусловлен тот факт, что для многих людей оказывается затруднительным постоянно поддерживать высокий уровень производительность труда. Поэтому многие разработчики достаточно часто отвлекаются на всякие мелкие и ненужные дела, иногда даже просто бездельничают на рабочем месте. Работая в паре, разработчики могут неявно оказывать положительное влияние друг на друга, что позволяет непрерывно поддерживать высокий уровень производительности труда.

Борьба с отвлекающими факторами

В современной корпоративной культуре существует достаточно много факторов, которые могут отвлекать от непосредственного решения задачи. Это телефонные звонки, постоянно возникающие мелкие организационные проблемы, общение с вспомогательным персоналом. С другой стороны,

замечено, что программирование является родом деятельности, которое для обеспечения высокой эффективности требует непрерывного погружения в задачу в течении, как минимум, нескольких часов. Второй программист, не занятый в данный момент непосредственно программированием, может легко разбираться с вышеперечисленными отвлекающими факторами не прерывая деятельности первого программиста.

Улучшение качества кода

Еще двадцать лет тому назад было установлено, что визуальный просмотр кода (*code review*) является самым эффективным, с точки зрения стоимости, методом исправления дефектов в программных продуктах. Однако, несмотря на это, большинство программистов не любят просмотр кода и не считают это занятие ни приятным, ни стоящим. В результате о просмотре кода, как правило, забывают, за исключением тех случаев, когда их выполнение требуется формально. В некоторых компаниях просмотр кода разработчиком, компетентным в данной узкой области, обязателен перед внесением *любого* изменения в код системы. Однако зачастую просмотр кода либо не осуществляется вовсе, либо осуществляется неподготовленными к этой задаче разработчиками.

Идея просмотра кода базируется на известном утверждении: чем раньше обнаружен дефект, тем проще и дешевле его исправить. Другим преимуществом постоянного просмотра кода является то, что с его помощью разработчики узнают новые способы и стили кодирования, более тщательно соблюдают стандарты кодирования.

Парное программирование гарантирует просмотр кода всех изменений, вносимых в систему. Это происходит *автоматически*, так как *второй* программист непрерывно наблюдает за тем, что и как программирует первый *программист*. Непрерывный просмотр кода при парном программировании создает уникальные условия для обучения, команда разработчиков учится общению и совместной работе.

Решение проблем

Работая в паре, программисты быстрее и качественнее решают проблемы. Программирование в паре является комбинацией мозгового штурма и взаимной поддержки (когда один партнер не в силах решить задачу, второй может оказаться более сообразительным в данной области). Благодаря этому факту у пары программистов больше шансов решить проблему, чем у одиночного программиста.

Коллективное владение кодом

Работая в разных парах, программисты вплотную знакомятся с большинством модулей системы, что автоматически обеспечивает коллективное владение кодом.

Обучение

Парное программирование практически является взаимообучением. Партнеры постоянно обмениваются знаниями: от навыков работы с инструментами разработки до изучения правил языка программирования, определенных способов проектирования и программирования, общего навыка построения дизайна системы. Также разработчики непрерывно передают друг другу знания о структуре и особенностях разрабатываемой ими системы.

Партнеры-программисты попеременно играют роли ученика и учителя. При этом они обмениваются даже навыками и привычками, которые затруднительно передать вербально, но можно *подсмотреть*.

Формирование команды и общение

Работая в паре, программисты постоянно общаются между собой. Благодаря смене партнеров в парах информационный поток усиливается, что в свою очередь усиливает эффективность всей команды. Вся документации системы передается в устной форме от разработчика к разработчику. Алистер Коуберн вообще ставит человеческий фактор в деле разработки программного обеспечения на первое место, утверждая, что это вовсе не второстепенный фактор, как многие считают.

Персонал и управление проектом

Руководство проектом только выигрывает от улучшения качества работы персонала и уменьшения рисков, которые с ним связаны. И компании, и команде разработчиков выгодна атмосфера постоянного обучения и обмена знаниями. Во время работы над проектом существенно возрастают профессиональные навыки разработчиков – как в области языков программирования, так и в области проектирования.

Снижается риск потери ключевых разработчиков, так как многие их коллеги хорошо знают каждую из частей системы.

3.3. Частые интеграции кода

При использовании гибких технологий разработки программного обеспечения интеграция нового кода в существующий код программного продукта выполняется несколько раз в день. При частой интеграции кода существенно снижается риск не успеть завершить работу в заданный срок. Если сначала полностью разработать каждую из составных частей продукта и только после этого попытаться интегрировать все части в единое целое, можно потратить дни и даже недели на исправление всех ошибок, которые обнаружились в процессе такой интеграции. Одна большая интеграция – это один большой риск, а несколько маленьких интеграций – это несколько маленьких рисков.

Интеграция выполняется редко, если разработчики исходят из следующего ошибочного положения: код, разработанный одним разработчиком, не может нарушить работоспособность кода, написанного другим разработчиком. Интегрируя код немедленно после его разработки, программист знает, что любые возникшие проблемы связаны с только что добавленным кодом, а значит, его проще локализовать и исправить. Постоянную интеграцию делают все члены команды, благодаря этому каждый из них получает доступ к самому последнему полностью работоспособному коду.

Основным элементом, делающим постоянную интеграцию возможной, является наличие «*интеграционного компьютера*». Это отдельный компьютер, на котором любая из пар программистов может выполнить всю работу, связанную с интеграцией разрабатываемого продукта. Интеграционный компьютер должен быть расположен так, чтобы разработчики, интегрирующие код, в любой момент могли обратиться к любому члену команды. Если в команде используется отдельный интеграционный компьютер, каждый член команды знает, какая пара разработчиков интегрирует в настоящий момент код. Интеграцию можно выполнять только тогда, когда интеграционный компьютер свободен, благодаря чему одновременно может интегрировать код только одна пара программистов.

Рекомендуется следующий порядок интеграции. Интеграция выполняется с *контрольной версией*, которая находится на сервере. Контрольная версия отвечает следующим требованиям:

- весь код обновлен;
- вся компиляция выполняется с нуля;
- система скомпонована;
- полный набор тестов выполняется без ошибок.

Перед началом работы разработчики должны переписать контрольную версию на интеграционный компьютер и удостовериться, что она работоспособна. Затем они копируют на интеграционный компьютер весь код, который был ими разработан или модифицирован. После этого они

запускают тесты. Если все тесты срабатывают, разработчики сообщают всем остальным членам команды, что интеграция прошла успешно. Если какие-либо тесты не срабатывают, разработчики пытаются исправить ошибку. Для этого они либо работают на интеграционном компьютере до тех пор, пока все тесты не сработают, либо восстанавливают изначальное состояние интеграционного рабочего пространства и возвращаются к своему компьютеру. При этом интеграционный компьютер освобождается, и другая пара может начать интеграцию кода.

Когда пара возвращается на свой компьютер, они переносят на него самый новый работающий код, включающий код других разработчиков, успешно выполнивших интеграцию.

3.4. Коллективное владение кодом

У разработчиков программного обеспечения постоянно возникает необходимость изменять код. Если этот код принадлежит разным людям, скорость разработки существенно снижается.

Коллективное владение кодом не является нормой в мире разработки программного обеспечения. Многие программисты ошибочно считают, что коллективное владение кодом – это то же самое, что полное отсутствие ответственности за разрабатываемый код. Некоторые из разработчиков опасаются потерять контроль над кодом, который они разработали.

При использовании гибких технологий разработки программного обеспечения правом и обязанностью изменения любой части кода с целью его улучшения обладает каждый из членов команды. При коллективном владении кодом каждый из разработчиков несет ответственность за корректное функционирование кода, который он разрабатывает или изменяет. Если код принадлежит всей команде, уход из команды одного из ее членов никак не повлияет на возможность дальнейшего совершенствования программного продукта.

Коллективное владение кодом возможно только, если все члены команды используют синхронизированный код. При любом изменении кода необходимо проверить, чтобы все модульные тесты, прошедшие раньше, срабатывали. Если при внесении изменений в код программист нарушает работу некоторых тестов, он *обязан восстановить работоспособность кода*.

Для того чтобы внедрение практики коллективного владения кодом прошло более успешно, необходимо, чтобы все разработчики отказались от идеи эксклюзивного владения кодом. Введение общего стандарта кодирования также значительно упрощает внедрения практики коллективного владения кода.

При переходе к коллективному владению кода могут помочь инструментальные средства, которые обеспечат разработчиков надежными средствами контроля версий и позволят легко исправить ошибки в случае их возникновения. Коллективное владение кодом лучше внедрять совместно с парным программированием и использованию общих стандартов кодирования.

3.5. Рефакторинг кода

Рефакторинг или переработка кода – это методика улучшения качества кода без изменения его функциональности и нарушения его работоспособности.

Если разрабатываемый программный продукт должен эксплуатироваться достаточно долго, его код будет неизбежно меняться. Традиционные методики разработки программных продуктов отрицательно относятся к частым изменениям программного кода. Считается, что если программный продукт близок к завершению или уже эксплуатируется, вносить в него изменения слишком рискованно. Поэтому разработчики боятся любых изменений кода.

Рефакторинг, напротив, формирует привычку у разработчиков постоянно менять код. Постоянное изменение кода – это реальность, с которой приходится мириться, поэтому одной из первоочередных задач является сокращение затрат на эти изменения. Получив код, обладающий нужной функциональностью, необходимо убрать из него весь «строительный мусор». Сопровождение чистого кода существенно облегчается. Рефакторинг не только обеспечивает чистоту кода, но и упрощает его модернизацию. Это происходит за счет того, что рефакторинг улучшает структуру программного обеспечения. Плохо структурированный программный код труден для понимания и модификации. Он, как правило, содержит дублирующий код. Удаление дублирующего кода напрямую связано с улучшением его структуры.

Код, в отношении которого выполнен рефакторинг, проще оптимизировать, так как, во-первых, упрощается поиск наиболее критичных участков кода, а во-вторых, упрощается внесение изменений, повышающих производительность кода.

Рефакторинг увеличивает скорость создания программ. На первый взгляд, это утверждение кажется спорным, поскольку для выполнения рефакторинга требуется дополнительное время. Однако это время расходуется только в процессе разработки кода и значительно меньше времени, которое потребуется в дальнейшем на модернизацию кода, если для этого кода рефакторинг не выполнялся. Итак, прилагая дополнительные усилия в начале, можно получить выигрыш в будущем. Но одним из основных принципов гибких технологий является «не делать сегодня того, что можно отложить на завтра». Чтобы разрешить это противоречие, необходимо принять во внимание, что в гибких технологиях разработки ПО не рекомендуется делать то, что *может оказаться* полезным завтра, а рефакторинг – это то, что *обязательно окажется* полезным завтра.

Еще одним преимуществом рефакторинга является то, что он помогает найти ошибки в программном обеспечении.

Рефакторинг рекомендуется выполнять:

- при добавлении новой функции;
- при исправлении ошибок;

- при обзорах кода.

Обзоры кода дают возможность большому числу людей изучить программный продукт и высказать по этому поводу полезные мысли, а рефакторинг позволяет изменить код, исходя из новых полученных знаний, т. е. возвратить знания в код.

Рефакторинг должен выполняться небольшими порциями. Работа по переработке кода должна начинаться с тестирования, которое является неотъемлемой частью рефакторинга. Рефакторинг может выполняться только в случае, если код работоспособен.

Лучше всего рефакторинг выполнять в двух случаях:

- перед тем, как реализовать некоторую дополнительную функциональность;
- после того, как дополнительная функциональность реализована.

Когда разработчик приступает к реализации новой функциональности, он изучает существующий код, пытаясь определить, можно ли каким-либо образом переделать его, чтобы упростить реализацию новой функции.

Когда разработчик завершает реализацию новой функции, он анализирует только что написанный им код, пытаясь определить, существует ли способ упростить его

Важно не откладывать рефакторинг. Как только разработчик обнаруживает необходимость выполнить рефакторинг, так сразу же он должен этим воспользоваться.

Каким образом выполняется рефакторинг? Для многих источников плохого кода существуют *шаблоны рефакторинга*, представляющие собой инструкции, каким образом следует поступать при возникновении источника плохого кода. Шаблоны объединяют в группы по категории производимых действий, например, шаблоны составления методов, шаблоны перемещения функций между объектами, шаблоны организации данных, шаблоны упрощения условных выражений, шаблоны упрощения вызовов методов.

Рассмотрим пример шаблона рефакторинга, для фрагмента кода, который выполняет присваивание параметру. Результатом рефакторинга является создание временной переменной.

Исключить присваивание параметрам(Remove Assignments to Parameters)

- Создать для параметра временную переменную.
- Заменить все обращения к параметру, осуществляемые после присваивания, временной переменной.
- Изменить присваивание так, чтобы оно производилось для временной переменной.
- Выполнить компиляцию и тестирование.

Если передача параметра осуществляется по ссылке, необходимо проверить, используется ли параметр в вызывающем методе снова. Проверить также, сколько в этом методе параметров, передаваемых по ссылке, которым присваивается значение и которые в дальнейшем используются. Попытаться сделать так, чтобы метод возвращал одно значение. Если необходимо возвращать несколько значений, попытаться преобразовать группу данных в объект или создать отдельные методы.

Рассмотрим еще один важный вопрос, когда нужно завершить рефакторинг. Некоторые сторонники рефакторинга считают, что рефакторинг никогда не должен завершаться. Это неправильно. Не следует переделывать код больше, чем это нужно заказчику. Нельзя в программный продукт добавить абсолютно всю возможную функциональность, точно так же нельзя выполнить весь возможный рефакторинг. Важно соблюдать два правила:

1. Нельзя добавлять в код фрагмент, если известно, как его можно улучшить.
2. Следует всегда удалять плохой код, если рядом с ним размещается код, реализующий новую функциональность.

И, наконец, когда не нужно делать переработку кода. Рефакторинг не нужен тогда, когда код не работает и его требуется переписать заново.

4. Проектирование и управление требованиями

Один из основных принципов гибких технологий разработки программного обеспечения – отказ от длительного проектирования перед началом работы и выполнение проектирования на протяжении всего выполнения проекта.

Методики, предписывающие выполнять все проектирование до начала работы над проектом, основаны на ошибочном представлении о том, что заранее можно определить все требования к программному продукту, разработать план и выполнить проектирование. В гибких технологиях разработки программного обеспечения предполагается, что в начале работы разработчики располагают только приблизительным планом реализации, который постоянно вместе с программным продуктом развивается и уточняется в процессе работы.

В гибких технологиях разработки ПО этап проектирования принято называть *дизайном*. Противники гибких технологий утверждают, что дизайн в гибких технологиях и проектирование в классических методиках разработки программного обеспечения – совершенно разные понятия. Это неправильное утверждение. Гибкие технологии предполагают непрерывное осуществление процесса проектирования, а не только в самом начале работы над проектом. В начале проекта выполняется лишь небольшая часть работы – *формирование общего представления*. Для этой цели в гибких технологиях используются *системные метафоры*, на основе которых формируется высокоуровневая схема проекта.

Эволюционный подход изначально лежал в основе проектирования программного обеспечения, но в условиях неорганизованности и неструктурированности процесса разработки от него отказались в пользу предварительного проектирования. При этом в качестве прототипов процессов проектирования в классических технологиях разработки программных продуктов принимались методы проектирования, хорошо зарекомендовавшие себя при проектировании сложных строительных или промышленных объектов. Такой подход оказывается совершенно неприемлемым при создании программных продуктов, требования к которым перед началом работы окончательно не сформированы.

Если в условиях неполной определенности (заказчик не до конца осознал предъявляемые требования или разработчики неправильно интерпретировали его пожелания) попытаться выполнить проектирование, а затем в соответствии с разработанным проектом реализовать программный продукт, то полученный результат, как правило, не сможет удовлетворить заказчика. Решение, позволяющее выйти из создавшейся ситуации, может варьироваться от нескончаемого ряда беспорядочных изменений в проекте (это возвращает разработчиков к первоначальному, беспорядочному способу проектирования) до полного перепроектирования на основе новых требований, при этом стоимость внесения изменений в проект экспоненциально растет (рис. 9).

Традиционная стратегия сокращения с течением времени затрат на разработку программного обеспечения заключается в том, чтобы снизить вероятность его перепроектирования и затраты, связанные с повторным перепроектированием. Гибкие технологии разработки ПО предлагают действовать с точностью до наоборот. При этом достигается положительный эффект за счет того, что риск – это точно такие же деньги, как и время. Можно получить определенный выигрыш, если сегодня включить в проект некоторую возможность, хотя сегодня она и не нужна. Однако при этом необходимо учитывать вероятность того, что эта возможность может вообще не понадобиться, и затраты на ее реализацию окажутся лишними.

Дизайн не обходится бесплатно, для более сложного дизайна требуются дополнительные расходы, связанные с сопровождением и развитием программного продукта. Более сложный дизайн требует также больших затрат на тестирование и обучение. С учетом этих факторов разница между сегодняшними и завтрашними инвестициями становится еще более ощутимой. Таким образом, идея отложить решение завтрашних проблем до завтра выглядит более оправданной.

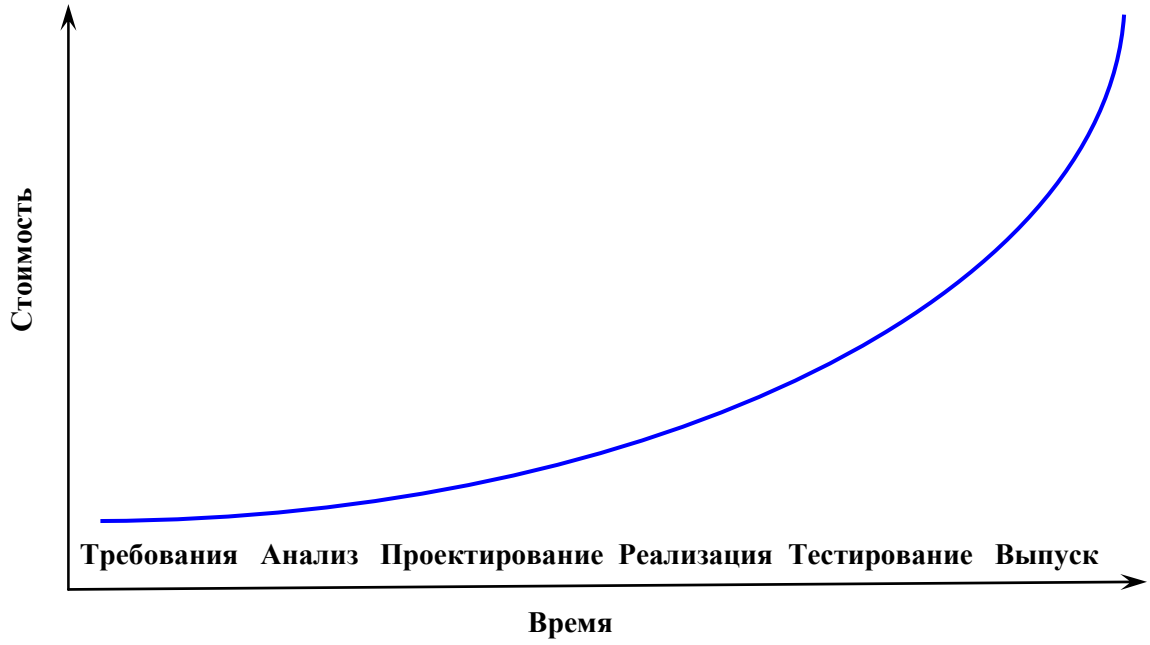

Рис. 9. Классическая кривая стоимости изменений

Таким образом, если стоимость сегодняшнего решения высока, вероятность того, что оно окажется правильным, низка, вероятность того, что завтра будет найден лучший способ решения проблемы, высока, а стоимость внесения изменений в дизайн завтра низка, то можно сделать простой вывод: если сегодня можно обойтись без решения, значит, мы ни в коем случае не нужно принимать это решение сегодня.

Девиз гибких технологий: «Количество сложностей ровно на один день и не более того»

Используя принципы гибких технологий, кривую стоимости изменений можно сгладить, сделав стоимость изменений на любой стадии проекта приблизительно одинаковой (рис. 10).

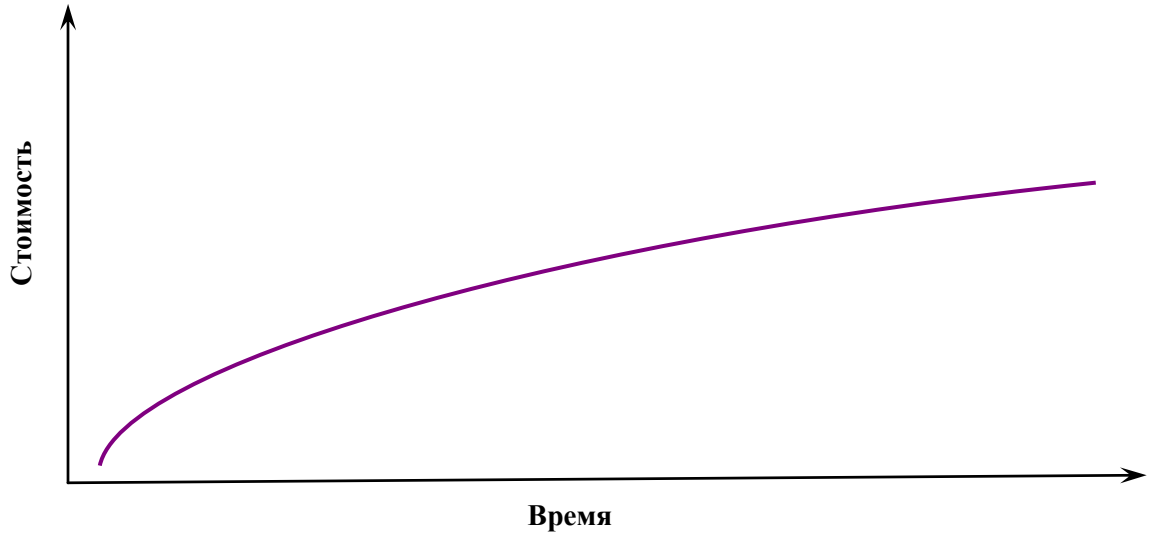

Рис. 10. Кривая стоимости изменений в гибких технологиях

Процесс *гибкой* разработки программного обеспечения (рис. 11) состоит из большого количества очень коротких циклов (чтобы полностью сосредоточиться на циклах разработки, на рис. 11 не отображен этап планирования). Конечный результат этапа планирования – список задач, подлежащих реализации на следующей итерации. Разработчик получает задачу и выбирает себе напарника. Затем они берут соответствующий фрагмент разрабатываемого кода, выполняют рефакторинг, необходимый для упрощения написанного кода, составляют тесты, а только затем создают сам код, который должен пройти тесты.

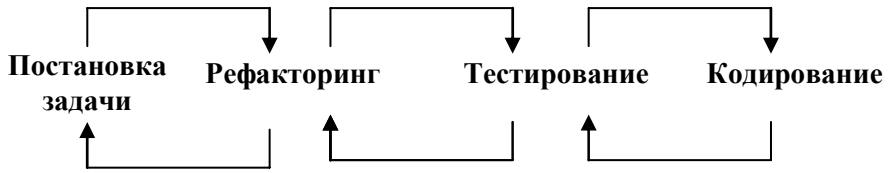

Рис. 11. Циклы в разработки программного продукта в гибких технологиях

Остановимся более подробно на цикле «*тест–код*». Разработчик находится в этом цикле до тех пор, пока задача не будет реализована полностью (разработанный код полностью протестирован). После этого делается попытка переработать программу, чтобы упростить код. Переработка программы начинается с внесения изменений в дизайн.

Поскольку циклы «*дизайн–тест–код*» непродолжительны, а заказчик часто получает работающие версии программного продукта, обратная связь осуществляется непрерывно. При такой технологии проектирования обратная связь служит для контроля, что проектирование и кодирование продвигаются в нужном направлении. Так как изменения на каждом цикле малы, решения, от которых приходится отказываться, невелики, в результате чего можно быстро реагировать на изменения с наименьшими затратами.

Основные методики, на которых базируется проектирование в гибких технологиях:

- простота дизайна;
- обязательный выбор метафоры;
- использование CRC-карточек;
- использование пробных решений для уменьшения риска;
- использование технологических прототипов;
- использование принципа постепенного расширения функциональности;
- рефакторинг;
- постоянное интегрирование кода.

4.1. Базовые методики проектирования

4.1.1. Простота дизайна

Стратегия проектирования в гибких технологиях предусматривает, что программный продукт всегда должен обладать наиболее простым дизайном, при котором срабатывает текущий набор тестов. На формирование стратегии проектирования в гибких технологиях оказали влияние все четыре ключевые ценности.

- *Взаимодействие*. Сложный дизайн описать сложнее, чем простой. Следовательно, необходимо создать стратегию дизайна, элементы которого насколько возможно просто будут описывать внутреннее строение программного продукта тому, кто будет изучать этот дизайн.
- *Простота*. Стратегия построения дизайна должна позволять создавать простые дизайны, однако при этом, сама стратегия также должна быть простой.
- *Обратная связь*. Одна из проблем, с которой приходится сталкиваться в процессе проектирования, заключается в том, что нельзя заранее точно сказать, правильно ли выполнено проектирование. Простой дизайн решает эту проблему благодаря тому, что можно быстро написать код и посмотреть результат.
- *Храбрость*. Стратегия простого дизайна основывается на храбрости разработчиков, так как они в любой момент должны быть готовы остановиться, имея лишь часть дизайна, приступить к кодированию и быть уверенными в том, что в случае необходимости в дальнейшем дизайн может измениться.

Какой дизайн является простым? К. Бек [3] отвечает на этот вопрос следующим образом:

- программный продукт должен четко выражать намерения разработчика;
- программный продукт не должен содержать дублирующегося кода;
- все тесты должны корректно срабатывать;
- программный продукт должен состоять из минимального количества классов, и методов, при этом необходимо выбирать их имена так, чтобы их было удобно использовать совместно.

Простой дизайн не предполагает, что программный код будет тривиальным или очень маленьким по объему. Дизайн может быть достаточно сложным, но при этом самым простым из всех возможных. Вместе с тем он должен позволять реализовать функциональность в заданном объеме, но не содержать неиспользуемой функциональности.

Дизайн должен быть достаточно простым, чтобы он был понятен людям, которые в дальнейшем будут сопровождать программный продукт и, возможно, модифицировать код. «Чем сложнее увидеть дизайн в коде, тем

сложнее сохранить его и тем быстрее он разрушается», – говорит один из приверженцев гибких технологий Мартин Фаулер в [5]. Это не значит, что любой человек, не знакомый с предметной областью и не работающий в команде, должен взглянуть на код и сразу все понять. Однако члены команды, которые некоторое время работали над проектом, и даже новые члены вашей команды не должны прикладывать особенных усилий, чтобы понять разработанный вами код.

Как правило, приступая к работе, разработчики не ставят перед собой задачу создать наиболее сложный дизайн из всех возможных. Проблема состоит в том, что разработчики начинают работу с желанием использовать только понятный дизайн, но со временем обстоятельства заставляют их все усложнять. У одних разработчиков отсутствует привычка сначала обдумать проблему, а затем попытаться найти самое простое решение, другие предпочитают сложные технические решения по привычке.

Чтобы сохранить простоту дизайна разработчики должны:

- научиться думать просто;
- иметь настойчивость, чтобы понимать код и модифицировать его;
- соблюдать дисциплину, чтобы регулярно выполнять рефакторинг.

Зачастую опыт, окружающие, а также личные склонности и увлечения мешают разработчику сформировать эти качества. Можно сказать, что концепция простого дизайна противоречит закоренелым инстинктам и привычкам программистов. Поэтому реальное внедрение практики простого дизайна зачастую оказывается непростой задачей, которая стоит перед техническим лидером и другими членами команды.

Рассмотрим следующую модельную ситуацию. Допустим, что в данный момент продукт, с точки зрения поддерживаемой функциональности, находится в состоянии, которое соответствует некоторой точке *A* (см. рис. 12). В течение итерации I_n требуется довести функциональность продукта до состояния, соответствующего некоторой точке *B*. Причем команде разработчиков известно, что на итерации I_{n+1} потребуется довести функциональность продукта до состояния, соответствующего некоторой точке *C*.

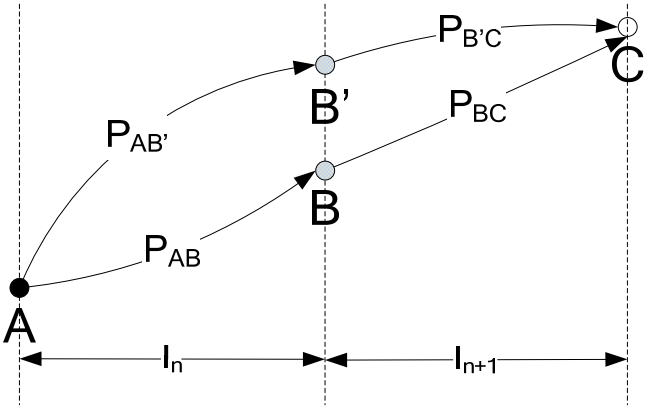

Рис. 12. Простой дизайн с точки зрения трудозатрат

В данном случае команда разработчиков может достаточно точно оценивать состояние продукта в точке *A* (точка закрашена черным цветом) и делать вполне вероятные предположения о состоянии продукта в точке *B* (точка закрашена серым цветом). Однако достаточно затруднительно делать сколько-нибудь вероятные предположения о состоянии продукта в точке *C* (точка вообще не закрашена). Такое положение дел обусловлено тем, что разработчики имеют возможность подробно исследовать текущее состояние продукта, который уже находится в точке *A*, поэтому риск сделать неправильный вывод о состоянии продукта является минимальным. Также, разработчикам известен набор функциональности, который будет добавлен в продукт в течение итерации I_n. Однако любые предположения о состоянии продукта в точке *B* несут в себе неточность, которая обусловлена многочисленными рисками разработки программного обеспечения. Предположения о состоянии продукта в точке *C* несут в себе еще большую неточность, так как во многом опираются на состояние продукта в точке *B*, которое само по себе еще не известно.

Исходя из заведомо неточной информации о состоянии продукта в точках *B* и *C*, разработчики могут предположить, что во время итерации I_n лучше развивать продукт в сторону некоторой точки *B'*, чтобы потом сэкономить во время итерации I_{n+1}. То есть разработчики делают предположение, что затраты на переход из точки *B* в *C* значительно больше затрат на переход из точки *B'* в *C*. Другими словами, разработчики предполагают, что $P_{BC} >> P_{B'C}$. Важно ответить, что продукт в точке *B'* будет содержать некоторую *подготовку* для перехода в точку *C*. При этом дополнительная гибкость или сложность продукта не будет приносить никакой пользы после завершения итерации I_n. В случае, если по предположениям разработчиков, затраты на движение по траектории $A \rightarrow B' \rightarrow C$ меньше затрат на движение по траектории $A \rightarrow B \rightarrow C$, в качестве цели для итерации I_n может быть выбрана точка *B'*.

С точки зрения гибких технологий разработки программного обеспечения и практики простого дизайна, такая стратегия является неправильной. Потому, что в точке *B'* продукт будет обладать более сложным дизайном, чем это реально необходимо, в результате чего значительно возрастут риски. При этом достаточно вероятными являются следующие варианты развития событий:

- разработчики неправильно оценили состояние продукта в точках *B* и *C* и трудозатраты на переход из *B'* в *C* эквивалентны или больше трудозатрат на переход из *B* в *C*. При этом вполне может оказаться, что $P_{AB'} + P_{B'C} >> P_{AB} + P_{BC}$;
- после завершения итерации I_n могут незначительно измениться приоритеты и, в качестве цели для итерации I_{n+1}, может быть выбрана некоторая точка *C'*. При этом также может оказаться, что $P_{AB'} + P_{B'C'} >> P_{AB} + P_{BC'}$;

- после завершения итерации I_n может значительно измениться представление о развитии продукта и вместо траектории $A \rightarrow B \rightarrow C$ может быть выбрана траектория $A \rightarrow B \rightarrow D \rightarrow C$. При этом также может оказаться, что $P_{AB'} + P_{B'D} + P_{DC} >> P_{AB} + P_{BD} + P_{DC}$.

Поэтому, с точки зрения гибких технологий разработки программного обеспечения, продукт *всегда* должен обладать максимально простым дизайном, достаточным для реализации текущей функциональности. Гибкие технологии рекомендуют выполнять усложнение дизайна только тогда, когда без этого невозможно достижение ближайших целей проекта.

4.1.2. Выбор системной метафоры

Архитектура в проектах, разработанных с использованием гибких технологий, менее формальна, чем в традиционных методологиях. В гибких технологиях этап разработки архитектуры начинается с *выбора метафоры*. Метафора представляет мощный способ описать сложное понятие из незнакомой предметной области. Метафора обеспечивает глобальное представление архитектуры проекта.

Метафора – это простая общедоступная и общеизвестная история, которая коротко описывает логику работы программного продукта. Метафора играет роль концептуальной основы, определяющей ключевые объекты и их интерфейсы. Она помогает каждому члену команды, как заказчику, так и разработчикам, понимать базовые структуру системы. Однако основное предназначение метафоры заключается не в том, чтобы облегчить программистам понимание архитектуры системы, а в том, чтобы усовершенствовать общение в команде и сформировать общее представление о системе. Метафоры особенно полезны, когда разработчиков из одной предметной области привлекают к проекту из другой области.

Метафора может быть упрощенным представлением какой-либо концепции или характерного свойства. Метафоры не задают точных представлений, а предоставляют контекст для обсуждения проблем и их решения. Метафора описывает, на что похож определяемый объект или свойство. По мере того, как идет работа над проектом, метафора развивается и обрастает деталями. В процессе работы может оказаться, что выбранная метафора оказалась ошибочной. В этом случае также следует от нее отказаться и выбрать другую метафору.

В качестве метафоры может быть предложен *контрпример*. Контрпример описывает черты, которые нежелательно иметь в новом программном продукте.

Приведем несколько советов, которые могут оказаться полезными при выборе метафоры.

1. ***Для записи метафоры рекомендуется использовать терминологию, хорошо сочетающуюся с предметной областью, для которой разрабатывается программный продукт.***

Например, при работе над проектом *C3* для автомобильной компании *Daimler Chrysler* для определения правил обработки платежей команда

использовала производственную метафору конвейера, так как такие понятия, как конвейер и деталь в автомобильной компании понимают все. Использование в этом проекте водопроводной трубы с вентилями в качестве метафоры вряд ли оказалось бы эффективным. Не следует тратить слишком много времени на поиски идеальной метафоры. Совершенно необязательно добиваться, чтобы метафора была идеальной. Отдельные части метафоры могут быть написаны в разном стиле и включать пояснения, показывающие, насколько то, что желательно получить, отличается от того, что описано в метафоре. Роль метафоры – обеспечить понимание заказчиком и разработчиком, каким должен быть программный продукт.

2. Используйте метафору для создания пространства имен.

Для понимания общего дизайна системы и исключения дублирующего кода чрезвычайно важно правильно именовать объекты. Если разработчик в состоянии правильно предугадать, как может быть назван существующий объект, это ведет к экономии времени. Поэтому в первую очередь при выборе метафоры необходимо создать систему имен (объектов, классов и методов) так, чтобы каждый член команды мог пользоваться ею без специальных знаний.

3. Следует помнить об ограничениях, связанных с метафорой.

Если метафора перестает играть позитивную роль, от нее нужно отказаться. Между тем, что является частью метафоры, и тем, что представляет собой требование, имеется разница. Часть метафоры в дальнейшем может перейти в пожелание заказчика.

Выбор метафоры - один из самых сложных, но эффективных процессов для установления взаимопонимания с заказчиком на этапе проектирования. Многие считают, что эту методику невозможно внедрить именно так, как это рекомендуется.

4.1.3. Использование CRC-карточек

Использование для проектирования CRC-карточек (*Class, Responsibilities and Collaboration Cards* – Класс, Обязанности, Взаимодействие) является фундаментальным принципом многих гибких методологий, например экстремального программирования. Являясь простейшими моделями, они позволяют скрыть маловажные детали, удобны для коллективного проектирования и способствуют более тесному общению членов команды.

Использование CRC-карточек позволяет описать программную систему как совокупность объектов и отношений между ними, а не в виде совокупности процедур и функций. Каждая CRC-карточка представляет собой экземпляр объекта.

Класс объекта на карточке записывается сверху, слева перечисляются обязанности класса, а справа – взаимодействия. Обязанности класса записываются в виде произвольно составленного текста. Как правило, каждая обязанность описывается одним предложением или, в крайнем случае, одним

абзацем. Пример CRC-карточки для класса ***Датчик температуры***, взаимодействующего с классами ***Ремонт*** и ***Резерв***, приведен на рис. 13.

В принципе число обязанностей класса может быть произвольным, но их не должно быть слишком много. На практике, очень хорошо структурированные классы имеет ровно одну обязанность.

Датчик температуры	
Обязанности **– Измерить температуру** **– Подать сигнал тревоги, если значение температуры превышает заданный уровень**	**Взаимодействия** – ***Ремонт*** – ***Резерв***

Рис. 13. CRC– карточка для класса «Датчик температуры»

CRC-карточки позволяют большому количеству людей (в идеале – всей команде) участвовать в процессе дизайна, а чем больше людей разрабатывают дизайн, тем больше может быть получено интересных идей. Обычно сеанс быстрого проектирования (CRC-сессия) выглядит следующим образом: каждый из участников воспроизводит элемент поведения системы, озвучивая, каким объектам и какие сообщения он передает. Таким образом проверяется сообщение за сообщением. При таком моделировании процесса сразу же выявляются слабые места и проблемы и могут быть предложены альтернативные варианты дизайна.

4.1.4. Пробные решения

При принятии любого технического решения возможен риск. Если при разработке дизайна необходимо решить сложный технический вопрос или требуется обосновать то или иное технологическое решение, разработчикам рекомендуется для уменьшения рисков создавать пробные решения. Например, можно быстро написать программу, которая воспроизводит исследуемую проблему, игнорируя все остальные факторы.

Большинство пробных решений не предназначены для последующего использования. Цель их создания – уменьшить риск принятия неправильного технического решения или получить более точную оценку времени, необходимого для реализации пожелания пользователя.

4.1.5. Использование технологических прототипов

При использовании новых технологий или непроверенных возможностей известных технологий необходимо опробовать их как можно раньше, чтобы уменьшить риск оказаться перед решением новой технической проблемы за несколько дней до выпуска очередной версии. Такой подход, возможно, поможет команде заранее понять бесперспективность определенной технологии, но в любом случае время, затраченное на создание технологических прототипов, окупится в будущем. Такие прототипы должны быть максимально простыми и решать только необходимые задачи.

4.1.6. Методика постепенного расширения функциональности

У разработчиков всегда имеется искушение добавить функциональность в разрабатываемый программный продукт, как только они ее увидят. Основной довод, которым разработчики при этом руководствуются, заключается в том, что программный продукт в этом случае будет намного лучше.

Гибкие технологии разработки программного обеспечения исходят из принципа, что только 10% от ожидаемого в действительности понадобится в первоначальном виде, т.е. 90% времени будет потрачено впустую. Преждевременная дополнительная функциональность замедляет прогресс и съедает ресурсы, поэтому лучше сконцентрироваться на решении текущих проблем.

4.2. Гибкое моделирование

Гибкое моделирование (Agile Modeling – AM) – это систематизирующая, практическая методология для эффективного моделирования и документирования программных продуктов. Гибкое моделирование позволяет определить разумный объем моделирования, когда его достаточно, чтобы изучить и документировать программный продукт, но не настолько много, чтобы тяжесть моделей замедлила развитие проекта.

> *Моделирование схоже с планированием – наибольшую ценность представляет процесс моделирования, а не сама модель.*
>
> *Скотт Амблер*

Гибкое моделирование не является законченным процессом разработки программных продуктов, так как оно не охватывает такие проблемы, как непосредственно программирование, тестирование, управление проектом, сопровождение программных продуктов и т.п. Гибкое моделирование – это дополнение к существующим методикам, оно не является отдельной методологией, поэтому его необходимо использовать совместно с другими технологиями разработки программных продуктов.

> *Основная задача гибкого моделирования – это моделирование, дополнительная – создание документации.*

Методики гибкого моделирования могут использоваться для повышения эффективности процесса моделирования в проектах, не ориентированных на гибкий подход к разработке программного обеспечения, но *наибольший эффект* от гибких моделей можно получить, если AM использовать *совместно с* другими практиками гибких технологий разработки программного обеспечения.

Основой для успешного гибкого моделирования, так же как для других практик гибких технологий, является эффективное общение между всеми участниками проекта – заказчиками, разработчиками, менеджерами. Стремление достичь максимально простого из возможных решений, соответствующих всем требованиям на определенном этапе разработки, постоянная обратная связь между участниками проекта, смелость принимать и отстаивать принятые решения, а также способность выбрать верный вариант решения, независимо от того, кто его предложил также влияют на успешность внедрения гибкого моделирования.

Существует несколько распространенных заблуждений относительно моделирования при использовании гибких технологий:

- в гибких технологиях нет места моделированию;
- при моделировании в гибких технологиях не создается никакой документации;

- рефакторинг и гибкое моделирование не совместимы.
-

Такие заблуждения обычно возникают за счет непонимания гибких технологий. Остановимся на этих вопросах более подробно.

Моделирование является частью гибких технологий

Карточки с пожеланиями заказчика, так же как и карточки CRC, являются очень важными артефактами гибких технологий разработки программного обеспечения. Пожелания заказчика обеспечивают высокоуровневый набор требований к программному продукту, а карточки CRC используются для концептуального моделирования и для проектирования. Как пожелания пользователя, так и карточки CRC являются моделями, следовательно, уже поэтому моделирование является частью гибких технологий. Если пожелания пользователя или карточки CRC не подходят для описания какой-нибудь ситуации, разработчики, работающие с использованием гибких технологий, могут создавать какие-нибудь другие схемы, которые также являются моделями.

Документация является важной частью гибких технологий

В гибких технологиях документацию принято разделять на внутреннюю и внешнюю. В гибких технологиях постоянное устное общение между членами команды, методики «Программирование в паре» и «Совместное владение кодом» снижают необходимость создания внутренней документации. В гибких технологиях роль важной документации для разработчиков играют тесты и сам программный код, который в результате использования методики рефакторинга является ясным и понятным. Даже если среда разработки легко позволяет встроить документацию в код, следует руководствоваться правилом, что документацию нужно создавать не тогда, когда это делается легко, а когда необходимо.

Внешняя документация обычно создается в конце жизненного цикла, когда программный продукт становится относительно стабильным. В это время создается следующая документация:

- ***Документация по системе***. Данный документ представляет собой краткое описание реализованного бизнес-процесса, высокоуровневых требований к системе, архитектуры системы. Также данный документ может содержать краткий обзор важнейших проектных решений и проектных моделей.
- ***Документация по эксплуатации.*** Этот тип документации описывает правила взаимодействия разработанного программного продукта с другими системными и прикладными программами, базами данных и файлами, содержит ссылки на процедуры резервирования, краткое описание требований работоспособности, надежности и нагрузочной способности программного продукта, советы по устранению неисправностей.

- ***Документация по обслуживанию.*** Данная документация включает в себя учебные материалы для группы сопровождения программного продукта, справочный и руководящий материал по устранению неисправностей.
- ***Пользовательская документация.*** Этот тип документации включает в себя руководства и инструкции пользователей, учебный и справочный материал.

Рефакторинг и гибкое моделирование

Рефакторинг является основной методикой для реструктурирования программного кода организованным образом. Основная идея рефакторинга заключается в том, что разработчик вносит небольшие изменения в код для реализации новых требований и сохранения максимально простого дизайна. При этом полученный в результате рефакторинга код может не соответствовать проектной модели, по которой создавался первоначальный код.

Эта проблема не имеет ничего общего с типом проектной модели. С таким же успехом в качестве модели могли быть использованы диаграмма классов UML, карточки CRC или другой тип модели. Однако гибкое моделирование в этой ситуации дает рекомендации, как решить проблему несоответствия. Например, методика гибкого моделирования «Совершенствуйте только в случае необходимости» рекомендует не хранить такие артефакты, как проектная модель и код, если они не синхронизированы.

В случае, если нужно произвести глобальный реинжиниринг системы, необходимо собрать всю команду и обсудить с ней возникшую проблему. Возможно, потребуется выполнить повторное моделирование, а затем приступить к реинжинирингу. В настоящее время существуют инструментальные средства (CASE-средства) обратного моделирования и конструирования. Так, например, CASE-средства позволяют импортировать исходный объектно-ориентированный код, написанный на Java или C++, и создавать диаграммы классов или диаграммы последовательности UML.

5. Планирование и управление проектом

Начиная работу над проектом, разработчик, как правило, не может располагать абсолютно полной, исчерпывающей информацией о том, каким должен быть разрабатываемый программный продукт.

Большинство традиционных методов планирования программного продукта основаны на следующих предположениях:

- прежде чем приступить к планированию, необходимо собрать *все требования* заказчика;
- планированием должны заниматься определенные члены команды, так как оно требует специальных навыков и знаний;
- сначала необходимо разработать план, согласовать его с заказчиком, а только потом передать его для реализации программного проекта разработчикам;
- планирование – весьма дорогостоящая процедура, поэтому планировать следует как можно реже, а все остальное время тщательно отслеживать план.
- гибкие технологии предполагают, что ответы на большую часть вопросов будут найдены в процессе работы над проектом. При этом подразумевается, что, получив в свое распоряжение рабочую версию продукта, заказчик сможет точнее понять, что ему нужно, и скорректировать свои требования. Разработчик же, получив от заказчика новые пожелания и уточнения реализованных требований, смогут без особых осложнений внести в программный продукт необходимые изменения и реализовать новую версию.

В гибких технологиях планирование проекта (*Planning Game*) – это ключевая методика, используемая для создания плана проекта. Планирование основывается на следующих предположениях:

- в начале работы над проектом требования заказчика не могут быть полными и точными, в ходе работы над проектом требования заказчика будут изменяться;
- в процессе планирования должны принимать непосредственное участие все имеющие отношение к проекту люди, каждому из которых назначена определенная роль; в процессе планирования участники проекта действуют строго в рамках полномочий, определенных этой ролью;
- все заинтересованные стороны должны фактически постоянно поддерживать взаимосвязь;
- планирование должно выполняться очень часто и требовать небольших затрат.

Основная цель планирования – быстро сформировать приблизительный план работы и постоянно обновлять его по мере того, как уточняются требования к продукту.

Планирование в гибких технологиях – это процесс предположения, как будет выглядеть процесс разработки программного продукта для заказчика. При этом планирование осуществляется в следующем порядке: сначала быстро, с минимальными затратами формируется общий план, затем он постоянно пересматривается с целью формирования более конкретных целей для более коротких сроков: (месяцев, недель, дней). Ниже приводятся основные цели, которые достигаются в процессе планирования:

- определение объема работ и приоритетов;
- оценка затрат и графиков работ;
- появление у каждого заинтересованного лица ощущения того, что проект действительно может быть реализован;
- согласование деятельности всех участников проекта;
- определение действий участников проекта при возникновении нештатных ситуаций.

Рассмотрим принципы, которые влияют на планирование, Некоторые из них являются базовыми для гибких технологий, другие относятся конкретно к планированию:

- необходимо с высокой степенью детализации планировать только то, что требуется для реализации очередного этапа;
- ответственность за выполнение любой работы может быть *только принята*, но не может быть назначена сверху;
- предположительные оценки трудозатрат должны выполняться лицом, ответственным за реализацию;
- при планировании необходимо игнорировать взаимосвязи между частями проекта; это позволит планировать так, чтобы в первую очередь можно было бы реализовывать наиболее приоритетные требования.

Согласно гибким технологиям, каждые несколько месяцев команда разработчиков должна выпускать новую версию программного продукта. Работа над каждой версией разбивается на отдельные итерации, которые длятся от одной до трех недель. Итерация, в свою очередь, делится на отдельные задачи, каждая из которых должна разрабатываться несколько дней.

Разработка программного продукта имеет *циклический характер*, при этом при планировании необходимо синхронизировать два разных жизненных цикла: бизнес-цикл и цикл разработки программного обеспечения.

Бизнес-цикл состоит из следующих работ:

- методологии производство, выпуск и установка программного продукта;
- обучение пользователя;
- выставление счетов.

Каждый оборот такого цикла, продолжающийся от одного до нескольких месяцев, называется *выпуском версии программного продукта*.

Цикл разработки программного обеспечения всегда короче бизнес-цикла, поскольку, чем короче цикл разработки, тем проще вносить изменения в проект. Если бизнес-цикл сократится, то вместе с ним должен сократиться и цикл разработки ПО. Для того, чтобы удовлетворить этому требованию, версия разбивается на несколько небольших итераций, ограниченных по срокам. Каждая итерация содержит в себе все элементы полного цикла разработки программного обеспечения: планирование, проектирование, кодирование, интеграция, поставка, обучение, документирование. Результатом итерации должна быть *полностью оттестированный*, готовый к работе программный продукт.

Одна из целей планирования – установить, какие задачи должны быть реализованы в первую очередь. Нельзя случайно выбирать задачи из общего списка задач, надеясь, что они окажутся самыми приоритетными. Гибкие технологии предполагают, что в начале каждой итерации заказчик определяет наиболее важные, с его точки зрения, задачи, которые должны быть реализованы в следующей итерации.

Планирование программного проекта должно выполняться в *три* этапа:

1. Начальное планирование.
2. Планирование выпуска версии программного продукта.
3. Планирование итерации.

Начальное планирование часто выполняется до того, как будет сформирована команда разработчиков, которая будет работать над проектом. В начальном планировании может участвовать всего один разработчик и один заказчик. Этим начальное планирование отличается от планирования выпуска версии, которое осуществляется в присутствии всех разработчиков и представителей заказчика.

Начальное планирование рекомендуется начинать как можно раньше, когда проект находится в стадии начального обдумывания, в то время как планирование выпуска версии необходимо выполнять только тогда, когда изучены все основные подходы к решению проблемы и разработчик обладает ресурсами, необходимыми для развертывания работы над выпуском версии. В процессе начального планирования ни о каких потенциальных сроках выпуска первой версии продукта не может быть даже речи.

5.1. Распределение ролей при планировании

Планирование в гибких технологиях осуществляется совместными усилиями заказчиков (принимают решения о том, что должен делать программный продукт) и разработчиков отвечают за реализацию программного продукта). Такой подход к планированию оказывается достаточно эффективным, поскольку в данном случае заказчик отвечает за принятие бизнес-решений (приоритеты, сроки, объемы), а команда разработчиков отвечает за принятие технических решений (оценку трудозатрат). Решения, принятые одной стороной, должны стать *базой для принятия решений* другой стороной. Если это правило не выполняется, процесс распадается. Если заказчик получает слишком большие полномочия, он начинает диктовать разработчикам значения всех четырех переменных, которыми можно варьировать для достижения цели (качество, объем работ, цена, время). Когда разработчикам предоставляется слишком большая свобода, они начинают использовать новые технологии, новые инструментальные средства разработки программного обеспечения, новые языки программирования, руководствуясь, в основном, соображениями, что они очень интересны и современны.

Рон Джеффриз [1] следующим образом определяет права заказчика

Права заказчика

1. Заказчик имеет право на составление общего плана, чтобы знать, что будет выполнено, когда и с какими затратами.

2. Заказчик имеет право получить максимальный результат от каждой недели программирования.

3. Заказчик имеет право, используя приемочные тесты, регулярно контролировать ход выполнения проекта на работающей версии программного продукта.

4. Заказчик имеет право поменять свое мнение, заменить какие-либо функции и приоритеты, не неся при этом чрезмерных дополнительных издержек.

5. Заказчик имеет право получать информацию о любых неожиданных изменениях в ходе работ, что позволить им вовремя уменьшить функционал и сохранить первоначальные сроки поставки программного продукта.

6. Заказчик имеет право прекратить проект в любое время и получить при этом работающую версию программного продукта, разработка функциональности которого оплачена на текущий день.

Заказчик должен принимать решения в следующих областях:

- определять объем работ – какую часть проблем достаточно решить для того, чтобы программный продукт можно было эксплуатировать в реальных производственных условиях;

- определять сроки выпуска версий;
- определять состав версий;
- назначать приоритеты, т.е. определять, какие из функций должны быть реализованы в первую очередь.

Основная обязанность заказчика – принятие бизнес-решений. Имеется существенная разница между определением требований и принятием бизнес-решений. Чтобы приступить к процедуре принятия бизнес-решений, заказчик должен, как минимум, сформировать набор пожеланий и назначить каждому из них приоритет. Заказчик не может принимать бизнес-решения до тех пор, пока не узнает, сколько будет стоить реализация каждого пожелания. Точно так же заказчик не может определить объем затрат, связанных с реализацией того или иного пожелания.

Таким образом, заказчик не может принимать решения в вакууме. Он снабжает разработчика сведениями, на основании которых последний сможет формировать обоснованные технические решения. Разработчики, в свою очередь, должны сформировать набор технических решений, которые должны стать исходным материалом при формировании бизнес-решений.

Разработчики должны принимать решения в следующих областях:

- формировать оценку времени, которое ориентировочно потребуется для реализации каждой User Story;
- оценивать затраты, связанные с выбором той или иной технологии, например, системы управления базами данных, на основе которой будет выполняться реализация;
- определять, как будет организована команда разработчиков и как будет организован процесс работы в команде;
- распределять задачи между членами команды;
- оценивать риск, связанный с реализацией каждого пожелания заказчика;
- определять порядок, в котором будут реализованы User Stories, являющиеся частью текущей итерации.

Права разработчика в [1] определяются следующим образом:

Права разработчика

1. Разработчик имеет право знать, что от него требуется, и каков приоритет каждой задачи.

2. Разработчик имеет право в любое время производить качественный программный продукт.

3. Разработчик имеет право просить и получать помощь от коллег, начальства и заказчика.

4. Разработчик имеет право самостоятельно оценивать объем трудозатрат по задачам, а потом вносить в свои оценки изменения. Данное право не дает разработчику права преднамеренно устанавливать неправильные оценки или переоценивать задачу с целью получить возможность работать вполсилы

5. Разработчик имеет право брать на себя ответственность, а не следовать жестким предписаниям.

Разработчик не несет ответственности за то, что реализованные пожелания оказались менее полезными, чем предполагал заказчик. Разработчик обязан информировать заказчика о возможных последствиях в случае, если заказчик прелагает исключить ту или иную функциональность. Разработчик не может самостоятельно добавить в программный продукт какую-либо функцию, это может сделать только заказчик. Если разработчик не понимает того или иного требования заказчика, он должен уточнить его у заказчика. Если у разработчика появляется свободное время, он должен узнать у заказчика, что делать дальше. Если у разработчика не хватает времени , он должен спросить у заказчика, реализацию какого пожелания можно отложить. Каждый раз, когда разработчик сомневается, он должен обратиться за разъяснением к заказчику.

Если разработчик видит изъян в пожеланиях заказчика, он должен как можно более вежливо указать ему на это и постараться обсудить с заказчиком этот изъян.

Чтобы точно оценить трудозатраты, связанные с реализацией пожеланий заказчика, разработчик должен хорошо понимать, что хочет заказчик. Если разработчик затрудняется дать точную оценку, он должен оценить трудозатраты хотя бы приблизительно и попросить заказчика предоставить ему *дополнительное время*, в течение которого он более подробно исследует проблему и сформирует точную оценку.

Заказчик – одна из самых важных ролей в команде. Каждый заказчик обладает сильными и слабыми сторонами. Некоторые заказчики описывают пожелания с чрезмерным количеством подробностей. Другие формулируют пожелания очень приблизительно, возлагая детальную проработку на разработчика. Иногда разработчик знаком с предметной областью в большей степени, чем заказчик. В некоторых случаях заказчик представляет интересы

конкретных конечных пользователей программного продукта, в других же случаях конечный пользователь является чисто гипотетическим субъектом.

Хороший заказчик должен:

- досконально знать предметную область, для которой разрабатывается программный продукт;
- понимать, какие преимущества даст разрабатываемый программный продукт в этой предметной области;
- настроиться на постоянное создание небольших промежуточных результатов, в которых реализовано, возможно, мало новых функций;
- принимать решения относительно того, какие функции должны быть реализованы в первую очередь;
- никогда не отменять сроки поставки программного продукта;
- доверять оценкам трудозатрат, которые делают разработчики, несмотря на то, что последние могут ошибаться;
- уметь взять на себя основную ответственность за успех или провал проекта;

В гибких технологиях процесс планирования намеренно абстрагируется для двух участников – заказчика и разработчика (команды разработчиков). В действительности, абстрактную роль заказчика могут играть несколько человек. К ним относятся конечные пользователи разрабатываемого программного продукта, рассказчики (члены команды, формулирующие требования с помощью историй пользователя (*User Story*)), приемщики (рассказчики или лица, действующие от их имени, которые выполняют приемочные тесты), финансисты (обеспечивают ресурсы для проекта), планировщики (составляют расписание выпуска версий системы, устанавливают степень реализации функциональных возможностей продукта, определяют необходимость выпуска новой версии).

5.2. Планирование версий

Для каждой версии программного продукта заказчик отбирает список пожеланий, которые должны быть реализованы в официально публикуемой версии продукта.

Планирование версий состоит из трех этапов:

- ***исследование*** (*exploration*) – на этом этапе необходимо определить, что должен делать программный продукт;
- ***подтверждение*** (*exploration*) – на этом этапе необходимо решить, какое подмножество требований заказчика необходимо удовлетворить в следующей версии программного продукта;
- ***управление*** (*exploration*) – на этом этапе необходимо управлять процессом разработки по мере того, как реальность вносит свои коррективы в план.

Смена этапов осуществляется циклически.

Выпуск версий обычно осуществляется один раз в течение нескольких месяцев. Если выпускать версии чаще, они будут мало отличаться друг от друга. Если же выпускать версии реже, можно отстать от конкурентов.

Этап исследования

Этап исследования включает в себя следующие действия:

- написание пожеланий (User Story) заказчиком;
- оценивание пожеланий разработчиком;
- разделение пожеланий заказчиком.

Написание пожеланий заказчиком

В гибких технологиях единицей функциональности программного продукта принято считать пожелание заказчика. Эти пожелания должны быть понятны как заказчику, так и разработчику. Пожелания записываются на специальных карточках из бумаги или картона и содержат номер пожелания, сведения об авторе (фамилия, имя), дату, краткую формулировку пожелания и текст пожелания. Каждое пожелание должно быть небольшим, чтобы разработчики могли реализовать за одну итерацию несколько (5 – 6) таких пожеланий. Считается, что чем короче пожелание, тем лучше, поскольку оно должно передавать только общее представление о том, что нужно заказчику. Пожелания должны быть независимыми друг от друга, что дает возможность реализовывать их в любой последовательности.

Оценивание пожеланий разработчиком

После того, как пожелания составлены, разработчики должны оценить трудозатраты на реализацию каждого пожелания и записать их, используя *специальные единицы измерения* в правом верхнем углу карточки. Пример

карточки с пожеланием и проставленной оценкой трудозатрат приведен на рис. 14.

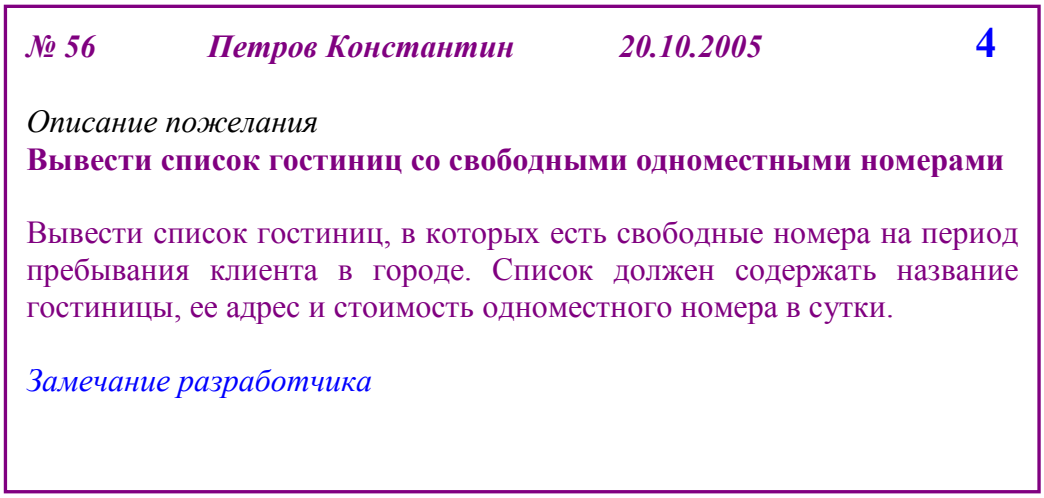

Рис. 14. Пример карточки с пожеланием и оценкой трудозатрат

В качестве единиц измерения труда в гибких технологиях [3] принято использовать *идеальное время* – период времени, в течение которого разработчик работает только над одной задачей, не отвлекаясь на любую другую деятельность с производительностью, близкой к максимальной. На практике часто в качестве единицы времени используется *идеальная человеко-неделя или идеальная половина человека-недели*. Вторая оценка более реальная, так как есть множество препятствий, в том числе переписка по электронной почте, совещания, помощь членам команды, выходные дни, болезнь, отпуска, которые не позволяют использовать идеальное время в качестве единицы для оценки трудоемкости.

Каждая оценка должна быть обдумана, проверена и подкреплена опытом и дополнительными соображениями. Самый простой и эффективный способ определения объема трудозатрат – сравнить подобное пожелание с пожеланиями, которые разработчик реализовывал ранее. Если ранее разработчик не сталкивался с похожей задачей, он должен провести дополнительное обследование (провести дополнительное обсуждение пожелания с заказчиком, выполнить предварительное программирование и т.п.).

Процесс формирования пожеланий и их оценка является итеративным и требует многократных встреч заказчика с разработчиками. Формирование пожеланий и оценка трудозатрат на их реализацию – это общий труд заказчика и разработчиков. В некоторых случаях разработчик может записать в специальном поле карточки замечание к пожеланию заказчика.

Следует особо отметить, что оценка трудозатрат – это не обещание выполнить реализацию пожелания абсолютно точно в указанный на карточке срок. Поэтому при оценке трудозатрат нужно уделять внимание следующим ключевым моментам:

- на карточке требуется записывать приблизительную оценку времени;
- при оценке трудозатрат не следует углубляться в детали проектирования или кодирования, однако не следует их полностью игнорировать;
- следует ограничивать время, требуемое для оценки одного пожелания.

Оценка трудозатрат в гибких технологиях является, возможно, самым трудным процессом для тех команд, у кого недостаточно опыта. Вместе с тем это конструктивный процесс, в ходе которого оценки становятся все точнее. Чем больше он выполняется, тем больше получается реальных оценок, тем точнее команда может оценить трудозатраты. Поэтому не нужно ждать хороших результатов от первоначальных оценок, поскольку со временем они будут становиться все точнее и точнее. Чтобы механизм оценок трудозатрат заработал в полной мере, необходимо обязательно фиксировать реальные трудозатраты на реализацию конкретных пожеланий с той же степенью подробностей, что и при планировании.

Пожелания пользователя должны реализовываться по принципу «все или ничего». Если объем пожелания слишком велик (трудоемкость реализации больше 5 единиц), заказчика просят разбить его на части. В этом случае в правом верхнем углу карточки записывается слово «разделить». *Деление пожеланий* пользователя является задачей заказчика. Разработчик лишь может давать рекомендации, как лучше всего выполнить разделение, но в конечном итоге решающее слово остается за заказчиком.

Лучший способ разделить пожелание заказчика – постараться создать два или больше независимых пожелания. Это довольно просто сделать, если заказчик в пожелании записал больше одного действия, которые могут выполняться независимо друг от друга. Такое решение не только уменьшает трудозатраты на реализацию одного пожелания, но и сводит к минимуму зависимости между пожеланиями.

Если же на карточке записано одно пожелание, но оно слишком велико, следует убрать *декорации*, т.е. те добавления в пожелание, которые придают ему дополнительную ценность, но без которых можно обойтись. Убрать декорации можно путем их исключения из реализации или путем создания нового пожелания с наименьшим приоритетом. Например, пожелание из примера, приведенного на рис. 14, первоначально могло бы содержать требование вывода списка гостиниц с их логотипами.

Этап подтверждения

На этом этапе заказчик должен определить объем работ и дату выхода следующей версии, а разработчики должны со всей уверенностью подтвердить, что они в состоянии выполнить намеченный объем работ к заданному сроку. Этап подтверждения включает в себя четыре шага.

- ***Сортировка пожеланий в соответствии с ценностью.*** В соответствии с их ценностью заказчик разделяет пожелания на три категории:

  обязательные – без реализации этих пожеланий программный продукт не сможет функционировать или будет бесполезен заказчику;

  желательные;

  необязательные – реализация этих пожеланий может быть пока отложена.

Порядок реализации пожеланий для заказчика определяется их приоритетом.

- ***Сортировка пожеланий в соответствии с риском.*** Отсортированные заказчиком пожелания разработчики разделяют на три категории в соответствии с их риском реализации:

  пожелания, которые можно оценить с высокой степенью точности;

  пожелания, которые можно оценить с приемлемой степенью точности;

  пожелания, которые практически невозможно оценить.

Разработчики предпочитают начинать реализацию с пожеланий, имеющих наибольший риск. При таком подходе, если в процессе реализации возникнут проблемы, у разработчиков будет достаточно времени на их решение.

- **Определение скорости реализации.**

  Разработчики сообщают заказчику, сколько единиц времени команде требуется для реализации запланированных пожеланий.

- ***Определение объема работ*** – заказчик выбирает набор карточек для очередной версии. Существует две основных стратегии определения объема работ:

  заказчик устанавливает дату завершения работы над версией и выбирает набор пожеланий в соответствии с их оценкой и скоростью работы над проектом;

  заказчик выбирает набор пожеланий в соответствии с их оценкой, на основании чего затем устанавливается дата завершения работы.

Сложно подобрать пожелания так, чтобы суммарное количество единиц труда, требуемых для реализации отобранных пожеланий, точно соответствовало размеру итерации. Например, три самых приоритетных пожелания требуют для реализации 24 единицы труда, в то время как вычисленный размер итерации составляет 26 дней. В подобной ситуации рекомендуется лучше оставить небольшой запас времени, чем сорвать сроки

реализации итерации или некачественно выполнить работу. Если в течение нескольких подряд итераций планируется сделать меньше, чем размер итерации, на следующую итерацию можно запланировать объем работ, превышающих размер итерации.

Каким образом разрешаются коллизии между последовательностью реализации пожеланий, предлагаемой заказчиком, и той, последовательностью, которую предлагают разработчики? Если разработчики предвидят большие риски, заказчик должен прислушиваться к их аргументам. Опасения разработчиков могут вызываться несколькими причинами, среди которых наиболее часто встречаются следующие:

- разработчики не уверены, что смогут точно оценить объем трудозатрат для реализации данного пожелания;
- реализация некоторого пожелания связана с использованием программного обеспечения, которое разрабатывалось третьими лицами, в связи с чем ему нельзя полностью доверять;
- разработчики не знают, как добиться заданной производительности программного продукта;
- разработчики не знают, каким образом выполнить проектирование программного обеспечения, чтобы в будущем реализация какого-нибудь из незапланированных пожеланий не привела к переработке всего кода.

В любом случае последовательность реализации пожеланий определяет заказчик. Задача разработчиков – открыто предупредить о наличии всех рисков, а не принимать решение за заказчика.

Этап управления

Существует несколько причин, которые могут заставить команду вернуться к процессу планирования версий системы. Основные из них – это *смена приоритетов в пожеланиях заказчика* и *изменение скорости работы разработчиков*. По сравнению с традиционными методологиями, в гибких технологиях заказчик до начала работ не должен детально описывать то, какой программный продукт он хочет получить. Появление новых пожеланий – заранее прогнозируемый и оправданный процесс.

Этап управления включает в себя следующие действия:

- *Итерация* – в начале каждой итерации, которая длится в течение от одной до трех недель, заказчик выбирает несколько наиболее ценных для него пожеланий, которые будут реализованы в рамках данной итерации. В результате реализации пожеланий самой первой итерации должен получиться работающий от начала до конца программный продукт, пусть даже в самом зачаточном состоянии.
- *Регенерация* – если разработчики приходят к выводу, что они переоценили собственную скорость, они согласовывают с заказчиком, какой набор наиболее важных пожеланий следует

сохранить в рамках текущей версии. При определении этого набора учитываются пересмотренные скорость и оценки.

- *Новое пожелание* – если в середине работы над очередной версией заказчик приходит к выводу, что в версию необходимо добавить новое пожелание, он может его написать. Разработчики оценивают новое пожелание, после чего заказчик убирает из оставшегося набора пожелания с эквивалентной суммарной оценкой и добавляет в план новое пожелание.
- *Переоценка* – если разработчики приходят к выводу, что план больше не соответствует точной картине разработки, они могут заново оценить оставшиеся пожелания и заново определить объем работ и скорость разработки.

После того, как пожелания распределены между отдельными итерациями, заказчик может определить приблизительную дату выпуска первой работоспособной версии продукта. Эта дата зависит от количества итераций, в ходе которых планируется реализовать все пожелания из категории обязательных. Заказчик также может определить даты выпуска наиболее важных *демо-версий*.

На практике возникают ситуации, которые требуют существенной переработки плана. К таким ситуациям относятся следующие:

- отложенных «на потом» пожеланий становится слишком много;
- существенно изменяется скорость работы команды.

В этих случаях приходится считать, что план становится недействительным и необходимо разработать новый план.

Переработка плана начинается с того, что разработчики заново оценивают объем трудозатрат. К этому времени у разработчиков появился конкретный опыт работы над данным проектом, поэтому в процессе работы над планом они уже могут оперировать реальными показателями, взятыми из прошлых пожеланий, и тем самым более точно оценивать объем будущей работы. Особенно важно повторно переценить трудозатраты на реализацию пожеланий, которые планировались в начале работы, так как первые планы всегда получаются наименее точными.

Как только разработчики уточнят трудозатраты, заказчик снова просматривает все пожелания и сортирует их в соответствии с требуемым порядком реализации. Далее процесс планирования выполняется как обычно.

Как показывает практика, план выпуска версий приходится переделывать каждые три-четыре итерации. Особенно большим изменениям подвергается первый план выпуска версий, так как когда он составляется, никакой статистики и реализованных пожеланий еще не существует. Первый план служит некоторой отправной точкой, от которой можно отслеживать ход реализации проекта, накапливать статистические данные по проекту.

Существует две стратегии планирования версий проекта в зависимости от того, насколько частыми могут быть выпуски версий. В некоторых случаях, например, при создании приложений, работающих в режиме «тонкого» клиента, в которых обновление версии осуществляется только на

сервере, выпуски версий можно совмещать с выпуском итераций. Такая стратегия позволяет разработчику очень быстро реагировать на изменение требований заказчика, а заказчику дает возможность максимально быстро оценивать результаты реализации своих требований. Однако при таком подходе заказчик может потерять общее представление о разрабатываемом проекте, концентрируясь на решении краткосрочных проблем. В связи с этим рекомендуется не пренебрегать долгосрочным планированием даже в тех случаях, когда разработчики выпускают новую версию каждые две-три недели.

При втором подходе к планированию версии выпускаются довольно редко, например, раз в год. Обычно это бывает в тех случаях, когда новая версия программного продукта полностью заменяет предыдущую, например, при выпуске «коробочного» продукта. Для того чтобы версии могли выпускаться редко, желательно выполнение одного из двух условий:

1. старая и новая версии некоторое время существуют одновременно;
2. имеются «дружественно» настроенные пользователи продукта, которые готовы эксплуатировать новую версию в режиме эксперимента и высказывать замечания и предложения по ее улучшению.

5.3. Планирование итераций

Следующим шагом после планирования выпуска версий является деление версии на серию коротких итераций. Продолжительность итерации определяется многими факторами, но, как правило, составляет две-три недели.

Планирование итерации выполняется в основном разработчиками в начале каждой итерации. Этот процесс может занимать от нескольких часов до двух дней. Планирование итерации состоит из тех же этапов, что и планирование выпуска версии, и напоминает планирование выпуска версии в миниатюре.

Этап исследования

На этапе исследования выполняются следующие работы:

- *написание задачи* – выбор пожеланий заказчика для итерации и преобразование их в задачи.
- *разделение задачи/комбинация задач*.

Формально считается, что итерация начинается с совещания по планированию итерации, на котором обсуждается, что было сделано в процессе предыдущей итерации, и что должно быть реализовано в рамках следующей итерации. При этом заказчик учитывает мнение разработчиков и опыт, приобретенный ими во время реализации предыдущей итерации. Затем разработчики определяют общий технический подход, который будет использоваться для реализации пожеланий заказчика в рамках следующей итерации и решают, каким образом пожелания заказчика можно превратить в набор конкретных задач, реализация которых занимает не более одного-трех дней работы. Пожелания заказчика обрабатываются в порядке уменьшения приоритета. Если несколько пожеланий содержат схожие фрагменты, можно выделить одну задачу для реализации нескольких пожеланий. Иногда в итерацию включаются задачи, которые не связаны напрямую с каким-либо пожеланием заказчика, например, может потребоваться задача для работы с новой версией системного программного обеспечения.

Если пожелание нельзя разделить на задачи, которые можно реализовать в течение несколько дней, ее следует разделить на более мелкие задачи. Если выполнение нескольких задач потребует всего несколько часов, можно скомбинировать их в одну большую задачу.

При планировании итераций важно учитывать зависимости между пожеланиями заказчика. Пожелание заказчика не может быть включено в итерацию, если оно зависит от пожелания, еще не включенного в план.

Большая часть времени команды будет уходить на реализацию требований заказчика. Однако имеются еще и *технические задачи*, о которых заказчики, как правило, ничего не знают, но которые вместе с тем должны быть выполнены. К таким задачам можно отнести, например, переустановку системного и инструментального программного обеспечения, рефакторинг сложного фрагмента программного кода, перепроектирование

какой-нибудь части программного обеспечения. Эти задачи также должны быть включены в итерацию с высоким приоритетом.

Этап подтверждения

На этапе подтверждения выполняются следующие шаги.

- *Выбор задачи* – разработчик принимает на себя ответственность за выполнение задачи. Обычно разработчики выбирают такие задачи, которыми они хотят заниматься. Это поддерживает в них увлеченность и заинтересованность в работе. При этом некоторые разработчики выбирают задачи одного типа, другие же, наоборот, берут разные задачи, так как хотят расширить свои знания и опыт. Некоторые руководители считают, что при подобном подходе имеется риск, что для решения некоторых задач не найдется желающих. В действительности все разработчики настолько отличаются друг от друга в плане опыта и предпочтений, что вероятность возникновения такой ситуации незначительна. В худшем случае, для задач, которые ни у кого не пользуются спросом, руководству необходимо найти дополнительную мотивацию.
- *Оценка задачи* – инструктор (роль инструктора подробно рассматривается в теме 6) оценивает количество единиц идеального времени, необходимых для реализации каждой из задач. Задачи, трудоемкость которых превышает некоторый порог, принятый в команде, необходимо разделить. При планировании не следует явно учитывать фактор парного программирования, поскольку этот фактор учитывается при определении фактора нагрузки.
- *Определение фактора нагрузки* – процента времени, которое фактически тратится на разработку. Фактор нагрузки определяется как отношение числа идеальных рабочих дней к общему числу календарных дней. Каждый разработчик выбирает для итерации свой собственный фактор нагрузки, исходя из имеющегося опыта. Для новых членов команды и инструктора, который вынужден значительную часть рабочего времени тратить на совещания с руководством проекта, количество идеальных рабочих дней может составлять 2-3 дня в течение трехнедельной итерации. Для всех остальных членов команды это значение не должно превышать 7-8 дней. Если для какого-то разработчика это значение больше, это означает, что он слишком мало времени тратит на помощь своим коллегам.
- *Балансировка* – каждый разработчик умножает суммарную величину идеального времени, необходимого для реализации своих задач, на фактор нагрузки. Разработчики, которые при этом оказываются перегруженными, должны отказаться от части своих задач. Если перегруженной оказывается вся команда, необходимо

сбалансировать ситуацию на этапе управления (см. шаг «регенерация»).

Один из самых важных принципов гибких технологий заключается в том, что запланированные однажды сроки не должны изменяться на протяжении всего времени работы – меняться может только объем работ, который следует сделать за запланированное время.

Перенос сроков – одна из самых вредных привычек, от которой затем чрезвычайно трудно избавиться.

Этап управления

Этап управления включает в себя следующие шаги.

- *Реализация задачи*. Разработчик берет карточку с описанием задачи, находит себе партнера, пишет тесты для задачи (модульные тесты), выполняет кодирование и запускает тесты в автоматическом режиме, после чего новый код интегрируется в программный продукт.
- *Отслеживание прогресса* – каждые два-три дня один из членов команды узнает у каждого из разработчиков, сколько времени израсходовано на реализацию каждой из запланированных задач и какой резерв времени еще остался.
- *Регенерация* – оказание помощи разработчикам, которые не укладываются в план, путем:
  - уменьшения объема работ для некоторых из задач;
  - обращения к заказчику с просьбой уменьшить объем работ для некоторых из пожеланий;
  - отказа от решения наименее важных задач;
  - получения более квалифицированной или большей по объему помощи со стороны других членов команды;
  - обращения к заказчику с просьбой отложить реализацию некоторых пожеланий до следующей итерации.
- *Проверка пожелания* – как только все задачи для некоторого пожелания реализованы и функциональные (приемочные) тесты готовы, производится тестирование программного продукта с использованием этих тестов для того, чтобы убедиться в том, что пожелание реализовано правильно.

Одним из способов синхронизации действий членов команды, работающей по гибким технологиям, является проведение *ежедневных совещаний*. Поскольку совещания проводятся ежедневно, команда может действовать неслаженно не больше одного рабочего дня. Для того, чтобы эти совещания отнимали минимум рабочего времени, они проводятся стоя.

На совещании должны присутствовать все члены команды разработчиков, поэтому имеются все условия для обсуждения текущего состояния плана итерации и имеющихся технических проблем. Заметим, что цель собрания – обсуждение проблемы, а не ее решение. Разработчики могут помочь друг другу решить проблемы после окончания совещания.

Рассмотрим основные различия между планированием версий и планированием итераций.

Во-первых, это различие состоит в том, что план итерации может быть более гибким, чем план выпуска версии. Если прошла одна из трех недель итерации и процесс реализации идет слишком медленно, возможно, имеет смысл остановиться на один день и выполнить совместную переработку кода для того, чтобы обеспечить дальнейший прогресс. Как правило, после нескольких таких ситуаций у разработчиков не складывается впечатление, что весь проект разваливается на части. Однако если заказчики наблюдают подобные ситуации довольно часто, это заставляет их нервничать.

Различие между планированием в рамках итерации и между итерациями – это расширение принципа разделения решений между заказчиком и разработчиками. На определенном уровне детализации существуют изменения, которые не должны беспокоить заказчиков, разработчики лучше заказчиков знают, как осуществлять управление своим рабочим временем.

Еще одно отличие между планированием итерации и планированием версии заключается в том, что разработчик самостоятельно выбирает задачу *перед* тем, как производится оценка трудоемкости ее реализации. Ответственность за реализацию пожеланий заказчика принимает на себя команда, поэтому оценки трудоемкости в этом случае выполняются всей командой коллективно. Ответственность за реализацию задач берут на себя отдельные разработчики, поэтому они должны оценивать эти задачи самостоятельно в индивидуальном порядке.

Планирование итераций отличается от планирования версий также тем, что в итерацию могут включаться задачи, напрямую не связанные с требованиями заказчика. К таким задачам можно отнести, например, задачу, связанную с доработкой или модификацией инструментальных средств, используемых для интеграции разрабатываемого программного продукта. Если трудоемкость решения этой задачи значительна, то она, как отдельная задача, включается в график работ и ей назначается приоритет наравне с другими задачами.

Планирование итераций для небольших проектов, выполняемых двумя-тремя разработчиками, может не выполняться, однако оно необходимо для координации работы команды из десяти программистов.

В заключение остановимся более подробно на планировании первой итерации и создании условий для начала работы над проектом.

Первая итерация – особенная, она определяет ядро программного продукта, на котором будут построены все остальные итерации. Выпуск первой итерации служит для заказчика подтверждением того, что гибкие технологии работают, а разработчикам – подтверждением их возможностей.

Средства, выделенные на первую итерацию, должны быть истрачены на реализацию основных возможностей программного продукта. Первая итерация должна содержать несколько самых ценных пожеланий заказчика, сосредоточенных вокруг основных функций. При этом от пожеланий заказчика на первой итерации не должны зависеть пожелания заказчика вне этой итерации. Стоимость реализации пожеланий заказчика, включенных в первую итерацию, не должна превышать выделенного на данную итерацию бюджета.

До начала работы над проектом по разработке программного обеспечения необходимо выполнить ряд работ, включающих инсталляцию среды разработки, системы управления базами данных и т.п. Важно, чтобы все эти программные средства начали успешно работать до начала первой итерации. Включение в первую итерацию задач по установке системных и инструментальных средств является большой ошибкой, так как первая итерация предназначена для демонстрации основных функций, присущих разрабатываемому программному продукту, а не отсутствия таковых.

5.4. Менеджмент и управление человеческим фактором

Разработка любого программного продукта в гибких технологиях осуществляется группой людей, объединенных в команду. Команда состоит из людей, компетентных в том виде деятельности, которым они собираются заниматься. Члены команды коллективно отвечают за созданный командой программный продукт.

В команде разработчиков, так же как, например, во многих спортивных командах, каждый участник берет на себя исполнение определенной роли. Некоторые из этих ролей предусматривают действия в индивидуальном порядке, другие созданы для исправления ошибок других и управления их действиями. Подбор этих ролей основан на опыте и, как правило, определяется правилами игры. Роли в гибких технологиях имеют неформальное значение. Они служат руководством: что нужно делать и как вести себя при выполнении той или иной работы.

Допустим в команде разработчиков были выделены некоторые роли, которые играли разработчики в ранее выполненных проектах. Если в новом проекте задействованы люди, которые не соответствуют этим ролям, следует *заменить роли*, а не пытаться изменить людей. Например, если в вашем проекте роль, предполагающая склонность человека к большому риску, то не следует назначать на эту роль скрупулезного, педантичного человека, который не предпринимает никаких действий, не взвесив все «за» и «против». Вместо этого нужно найти либо другое разделение ролей, либо подыскать другую кандидатуру.

5.5. Распределение ролей в команде

Рассмотрим роли, которые хорошо зарекомендовали себя во многих проектах. Данное разделение ролей не является обязательным и носит чисто рекомендательный характер.

Наставник

Хороший наставник нужен в любом деле, не только в процессе разработки ПО. Многие практики гибких технологий – программирование в парах, использование метафоры, коллективное владение кодом – проще показать на деле, чем прочитать о них в литературе. Наставник или инструктор (*coach*) – это опытный член команды разработчиков, отвечающий за весь процесс разработки. Задача наставника состоит в том, чтобы команда была настроена на выигрыш. Он должен сконцентрировать свое внимание на общей производительности команды.

Наставник должен поощрять команду, когда дела идут хорошо. Наставник должен выводить из состава команды тех ее членов, которые систематически нарушают правила гибких технологий или не приносят ощутимой пользы. Наставник должен воспитывать и обучать членов команды, которые работают недостаточно хорошо.

Наставник должен отслеживать задачи, которым команда разработчиков не уделяет достаточно внимания, и предотвращать кризисные моменты в проекте. Если же наставнику не удалось предотвратить кризис, ему предоставляется решающее слово при принятии решения. Вместе с тем наставник не должен излишне опекать разработчиков, лишать их самостоятельности при принятии решений, что, в конечном счете, может привести к снижению производительности труда и ухудшению морального климата в команде.

Основные обязанности наставника:

- быть доступным в качестве партнера по разработке, особенно для новичков, которые только начинают брать на себя ответственность за решение сложных технических задач;
- видеть долгосрочные цели переработки кода и стимулировать мелкомасштабную переработку для того, чтобы частично способствовать достижению этих целей;
- помогать программистам индивидуальными техническими навыками, например, тестированием или рефакторингом;
- предоставлять информацию о ходе процесса разработки менеджерам высшего звена.

Опытный наставник, хорошо знакомый с членами команды, которая практически самостоятельно справляется с работой в соответствии с гибкими технологиями, может постоянно не присутствовать в команде. Такой подход лучше всего использовать тогда, когда наставник был рядом с командой в самом начале работы, когда он хорошо знаком с проектом, знает об особенностях использования практик в этом проекте. Однако даже в этом

случае наставник должен посещать команду достаточно часто, так как иногда весьма серьезные проблемы остаются незаметными для людей, которые постоянно работают над проектом. Даже если наставник не присутствует в команде постоянно, он должен быть всегда доступен для ответов на вопросы и решения возникающих проблем.

Работа наставника – тяжелая работа. Для большинства команд наставник должен отдавать своему занятию все свое время. Поэтому, как правило, член команды не может быть одновременно наставником и играть какую-либо другую роль.

Если руководство компании не может принять на работу опытного наставника, в качестве наставника может выступать хороший лидер, уверенный в гибких технологиях и обладающий достаточным опытом разработки программного обеспечения. Роль наставника теряет свою значимость по мере того, как команда совершенствует свое мастерство.

Контролер

Контролер или ревизор (*tracker*) обеспечивает обратную связь предварительных оценок разработчиков с требованиями заказчика. Он сравнивает запланированные показатели с реальными их значениями с целью внесения поправок для формирования последующих оценок.

Контролер должен следить за общей картиной разработки и информировать команду об изменениях, вносимых в план. После выполнения половины работы над очередной итерацией контролер должен быть способен сказать команде, удастся ли успешно завершить работу над итерацией, если следовать ранее определенным курсом, или следует что-то изменить. После прохождения первых двух итераций контролер должен быть способен сказать команде, сможет ли она в срок выпустить очередную версию без существенных изменений или потребуется пересмотреть план. Также в зону ответственности контролера входит обсуждение с заказчиком любых значительных изменений в плане.

Контролер осуществляет также контроль результатов функционального тестирования. Он ведет журнал, в который заносятся обнаруженный дефект, ответственный за его исправление и тесты для идентификации каждого из дефектов.

Контролер должен обладать умением собирать необходимую информацию о ходе выполнения проекта, не беспокоя разработчиков больше, чем это необходимо.

Консультант

В рамках гибких технологий разработки программного обеспечения обычно отсутствует узкая специализация занятых на проекте программистов. Каждый член команды работает в паре с любым из других членов, партнеры в парах часто меняются, и каждый член команды при желании может взять на себя ответственность за решение любой задачи. Это является существенным преимуществом гибких технологий, но это является также и

слабым местом, так как время от времени команде могут потребоваться глубокие технические знания в некоторой области и она может нуждаться в консультациях специалистов по определенным техническим вопросам.

В рамках гибких технологий задача команды состоит в том, чтобы получить от консультанта все необходимые знания и попытаться самостоятельно решить проблему. Для этих целей один или двое членов команды, выделенные для взаимодействия с консультантом, будут решать возникшую проблему вместе с ним, задавать ему множество вопросов и тщательно оценивать все предлагаемые консультантом решения. В итоге последнее слово при решении данной проблемы останется за членами команды, а не за консультантом.

Дипломат

Дипломат (*facilitator*) – это член команды разработчиков, обладающий лучшими способностями общения с людьми. Его роль – упростить взаимодействие между командой разработчиков и заказчиками, в особенности на совместных совещаниях, посвященных планированию версий и итераций, на которых присутствие представителей заказчика обязательно. Опытные дипломаты понимают, как с максимальной выгодой использовать каждое совещание обеспечивают максимальную насыщенность повестки дня каждого совещания

Деятельность дипломата не ограничивается только собраниями. Каждый раз, когда возникают конфликты между командой разработчиков и заказчиком, в игру вступает дипломат. Его роль – найти решение, выгодное для обеих сторон, и доступно объяснить конфликтующим сторонам, что необходимо сделать для разрешения конфликта.

Дипломат должен всегда соблюдать нейтралитет и никогда не высказывать свою точку зрения.

Архитектор

Роль архитектора (*architect*) в гибких технологиях так же необходима, как и в традиционных процессах разработки программных продуктов, но она существенно изменилась. Вместо того, чтобы разрабатывать идеальную архитектуру системы и пытаться ей следовать, архитекторы в гибких технологиях создают и перерабатывают архитектуру по необходимости.

Программист

Программист (programmer) в гибких технологиях – это не то же самое, что программист, работающий в любой другой технологии. В гибких технологиях усилия программиста по разработке программного кода распределяются несколько по-другому. Работа программиста не заканчивается, когда разработанный им программный модуль начинает работать. Если программа уже работает, но при этом некий важный компонент взаимодействия не завершен, значит, он должен продолжить работу. Это может быть создание тестов, которые демонстрируют жизненно

важные аспекты разрабатываемого программного продукта, разделение программы на меньшие фрагменты или объединение слишком маленьких частей программы в более крупные, совершенствование системы именования таким образом, чтобы имена более точно отражали намерения программистов. Такая деятельность не является стремлением к совершенству, а желание разработать программное обеспечение, которое будет как можно более полезным для заказчика.

Как член команды, программист должен обладать навыками, которые не требуются или, по крайней мере, не являются настолько важными в рамках других стилей разработки. Это программирование в паре, стремление к простоте, умение перерабатывать код, разрабатывать модульные тесты, свободно общаться с людьми и координировать с ними свою деятельность, поддерживать принцип коллективного владения кодом.

Тестировщик

Большая часть обязанностей, связанных с тестированием, при использовании гибких технологий лежит на плечах программистов, поэтому роль человека, выполняющего тестирование, фокусируется на заказчике. Тестировщик (tester) помогает заказчику в составлении функциональных тестов. Если функциональные тесты не являются частью интеграционного пакета, тестировщик отвечает за регулярный запуск функциональных тестов и публикацию результатов тестирования. Он осуществляет также регулярную проверку правильности работы инструментальных средств, используемых для тестирования.

5.6. Организация рабочего времени в команде

Многие люди, занимающиеся информационными технологиями, сталкиваются с систематической перегрузкой и сверхурочными работами. В последние годы, в связи с появлением Интернета, эта проблема стала еще более актуальной. Чрезмерная сверхурочная работа ведет к неудачам и разочарованиям. Если человек работает слишком много, он сгорает на работе, даже если ему работа очень нравится.

Как правило, люди работают слишком много потому, что:

- они пытаются уйти от чего-то другого, с чем они не могут или не хотят иметь дело;
- сверхурочная работа – это признак особого статуса;
- они думают, что сверхурочная работа позволить им догнать или обогнать график;
- у них нет выбора, если они хотят сохранить свою работу.

На самом деле, работа во внеурочное время – это признак серьёзных проблем в проекте. Если проблема кроется в какой-либо другой области жизни, необходимо устранить эту проблему. Если сверхурочная работа является предметом гордости в организации, необходимо настоять на пересмотре приоритетов. Если сверхурочная работа позволяет продвинуться по службе, необходимо изменить систему стимулирования в организации.

Одна из базовых практик гибких технологий разработки программного обеспечения – 40-часовая рабочая неделя. Конечно, 40 часов – это приблизительное значение. Разные люди способны плодотворно работать в течение различного времени. Один человек не может концентрировать свое внимание дольше, чем в течение 35 рабочих часов, другой способен эффективно работать в течение 45 часов в неделю. Но никто не способен работать по 60 часов в неделю на протяжении нескольких недель и при этом оставаться свежим, творческим, внимательным и уверенным в своих силах.

Для того, чтобы люди работали продуктивно, необходимо не только ограничить рабочую неделю, но и установить гибкий график работы: время начала и окончания работы может варьироваться, должен быть обязательный обеденный перерыв. Кроме обеденного перерыва, члены команды должны иметь возможность сделать кратковременный (несколько минут) перерыв, чтобы выпить чашку чая или кофе или просто передохнуть.

Как показывает опыт, производственные проблемы фактически никогда не удается решить, заставляя людей работать сверхурочно. Сверхурочную работу можно использовать для устранения незначительного отставания от плана, однако зачастую это тоже не помогает. Уставшие от работы люди совершают больше ошибок, их продуктивность снижается. Правильно выбранный баланс между рабочим и нерабочим временем – это единственный фактор, который позволяет успешно работать в течение длительного периода времени.

В компаниях, в которых сверхурочная работа является нормой, очень сложно внедрить 40-часовую рабочую неделю. Для начала нужно определить, что является нормой для вашей команды. Команда не может работать слаженно, если ее члены обладают разной выносливостью. Поэтому очень важно добиться того, чтобы все члены команды согласились работать в рамках приблизительно одного и того же графика.

Одним из реальных путей перехода на нормальный график работы является отказ от компромиссов в ходе планирования. Если во время планирования произошла переоценка собственных сил, необходимо признать свою ошибку, исправить ее и в дальнейшем подходить к планированию более обоснованно.

И, наконец, для перехода на 40- часовую рабочую неделю требуется определенная смелость. Если руководство требует, чтобы команда систематически работала сверхурочно, необходимо проанализировать ситуацию и, если необходимость сверхурочной работы возникла не по вине команды, необходимо найти в себе силы сказать «нет».

5.7. Организация общения между членами команды

Одна из основных ценностей гибких технологий – общение. Никто не может знать о разрабатываемом программном продукте абсолютно все. Это значит, что время от времени каждый из членов команды ощущает необходимость в получении дополнительной информации. Самый простой способ получения информации – общение. Программирование в паре, совещание по выпуску итерации и по выпуску версии – все это различные формы общения членов команды между собой. Однако самым эффективным способом синхронизации работы членов команды над проектом являются *экспресс-совещания*. Экспресс-совещания соединяют между собой тех, кто нуждается в информации, и тех, кто этой информацией владеет.

Экспресс-совещание – это короткое (не более пятнадцати минут) ежедневное совещание всех членов команды с целью обмена информацией о текущем состоянии дел. Поскольку они проводятся ежедневно, команда будет действовать неслаженно максимум один рабочий день. Экспресс-совещания предоставляют членам команды возможность обсуждения того, что уже выполнено на текущий день и что осталось сделать. Экспресс-совещания позволяют:

- узнать о проблемных участках проекта;
- определить, кто может оказать помощь в их решении;
- узнать о неожиданностях, к которым следует подготовиться;
- убедиться в том, что день начинается правильно.

Цель совещания – не решать проблемы, а акцентировать на них внимание членов команды. Разработчики могут помочь друг другу решить проблемы после окончания совещания.

Чтобы экспресс-совещания были короткими, их проводят стоя. Это значит, что все участники совещания стоят. Каждый из присутствующих по очереди рассказывает, какую работу он выполнил вчера, каких добился результатов и что планирует сделать сегодня. Рассказывать в первую очередь нужно о том, что может непосредственно повлиять на деятельность остальных членов команды. Любые длительные дискуссии не поощряются, вплоть до формального запрета. Рекомендуется задавать только те вопросы, на которые можно ответить очень коротко. Любые более длительные обсуждения должны проводится после экспресс-совещания, причем участвовать в них будут только заинтересованные лица.

Если люди, владеющие информацией, ведут себя излишне сдержанно, следует попробовать понять, что им мешает раскрыться. Как правило, меньше всего заинтересованы в проведении экспресс-совещаний именно те, кто не расположен в распространении информации. Необходимо попытаться стимулировать таких людей.

Чтобы воспитать привычку говорить коротко, зачастую приходится прерывать слишком многословных коллег. Однако если в сообщаемой информации заинтересовано значительное количество человек, следует предоставить выступающему немного больше времени, чем полагается.

Место проведения экспресс-совещаний не должно располагаться далеко от рабочего места программистов. Если для проведения экспресс-совещаний у команды нет места, необходимо обратиться к менеджеру, с просьбой реорганизовать рабочее пространство.

5.8. Перепланирование при изменении команды

Рассмотрим, каким образом изменения в команде влияют на выполнение плана работ. Если в команде появляются новые сотрудники, необходимо дать им время (одну – две итерации) на то, чтобы они смогли освоиться на новом месте. В это время они могут:

- программировать в паре с наиболее опытными программистами;
- изучать код и тесты;
- общаться с заказчиком.

На этой стадии обычно нет никаких изменений в скорости работы команды. Время, потраченное на введение новичков в курс дела, компенсируется свежими идеями, которые новички привносят в проект. Поэтому во время планирования в первое время следует исходить из принципа, что за неделю можно сделать столько же, сколько было сделано за предыдущую. Только через некоторое время можно будет увеличить объем работ, учитывая потенциал новых членов команды.

Если из команды увольняется сотрудник, то в следующей итерации следует уменьшить объем работ на долю, вносимую уходящим сотрудником. Например, если уходит сотрудник из команды, в которой работало пять человек, следует снизить объем работ в итерации на 20%.

Если требуется увеличить размер команды, работающей в гибких технологиях, в несколько раз, рекомендуется выполнить следующее. Выпустить первую версию программного продукта, после чего разделить команду пополам на две равные команды. Каждая из образовавшихся команд получает от заказчика набор определенных пожеланий. Затем обе команды вырастают до исходного количества разработчиков. Достаточно повторить описанную процедуру несколько раз, чтобы получить требуемый размер команды.

Что касается планирования, то каждой команде, получившейся в результате первого деления, следует запланировать половину того объема работ, который они выполняли до деления. Затем на основе анализа полученных результатов, можно перейти к более точному планированию.

5.9. Организация рабочего места разработчика

Идеологи гибких технологий разработки программного обеспечения утверждают, что без хорошо организованного места работы программный проект не может быть выполнен успешно. Различие между хорошим и плохим рабочим местом для команды значительно и подчас критично.

Правильная организация рабочих помещений – это в любом случае непростая задача. В этой области существует большое количество конфликтующих ограничений. Планировщики рабочих помещений стараются потратить как можно меньше средств и при этом обеспечить как можно большую гибкость. Люди, использующие эти помещения, желают работать в тесной связке с остальной командой, но в то же время они нуждаются в некотором отделенном от остальных людей индивидуальном пространстве, из которого они могли бы, например, звонить по личным вопросам.

Гибкие технологии предполагают коллективную разработку программных продуктов. Члены команды должны видеть друг друга, слышать возникающие у коллег вопросы, участвовать в обсуждениях, в которые они могли бы принести пользу.

Общепринятая структура помещений для программистов, в которых все помещение разбито на индивидуальные ячейки, плохо соответствует идеологии гибких технологий. Для гибких технологий лучшим является план помещения, в котором центральное обширное общее пространство обрамлено по периметру небольшими отгороженными ячейками. Члены команды могут хранить в этих ячейках свои личные вещи, вести личные телефонные переговоры или проводить в них время в одиночестве, когда они не хотят, чтобы их отвлекали. Индивидуальные ячейки могут быть как оснащены компьютерами, так и не иметь компьютеров.

Самые большие, высокоскоростные компьютеры следует разместить в середине центрального общего пространства. За этими компьютерами размещаются программисты, работающие в паре. Если имеется возможность, рядом с общим рабочим пространством можно выделить небольшое помещение для отдыха, поместив туда кофеварку и чайник. Здесь разработчики могут отдохнуть в течение рабочего дня.

Если корпоративный взгляд на планировку рабочего пространства конфликтует со взглядом команды, команда, как правило, побеждает. Установление контроля над физической средой – это первый шаг в направлении установления контроля над общим стилем работы команды.

5.10. Управление работой членов команды

Для управлении работой членов команды над проектом используются базовые приемы управления: поэтапная поставка программного продукта, быстрая и надежная обратная связь, тесная взаимосвязь требований к программному продукту, использование квалифицированных специалистов для решения конкретных задач.

Для гибких технологий характерно скорее децентрализованное принятие решений, чем централизованный контроль. Принципы менеджмента в гибких технологиях:

Распределенная ответственность.

В соответствии с этим принципом обязанность менеджера состоит не в том, чтобы назначить ту или иную работу какому-нибудь члену команды, а в том, чтобы указать команде на необходимость выполнения этой работы.

Качественная работа.

Взаимодействие между менеджером и членами команды должно основываться на доверии, так как программисты хотят выполнять свою работу качественно.

Постепенное изменение.

Этот принцип исходит из предположения, что управление процессом разработки программных продуктов осуществляется в течение всего времени работы над проектом.

Локальная адаптация.

Менеджеры должны взять на себя ответственность за адаптацию гибких технологий к локальным условиям, в случае, если основные принципы гибких технологий конфликтуют с общими принципами, принятыми в компании.

Путешествие налегке.

Этот принцип предполагает, что менеджер не создает чрезмерной нагрузки на участников проекта (например, длительные совещания с обязательным присутствием всех членов команды или продолжительные доклады о текущем состоянии проекта). Любая просьба, которую менеджер может адресовать члену команды, должна быть выполнена без больших затрат времени.

Откровенные оценки.

Любые метрики, собираемые менеджером, должны быть реальными и понятными для тех, чья работа подвергается анализу и оценке.

Одной из причин, по которым менеджер может вмешаться в процесс разработки, является изменение в составе команды. Если член команды не справляется со своими обязанностями, менеджер вынужден пойти на непопулярное решение – уволить сотрудника.

В случае если работа команды требует изменений, менеджер не должен решать, что и как надо изменить, а должен всего лишь указать на необходимость изменений. Команда сама должна откорректировать свои действия, а менеджер должен доложить команде о том, как изменились метрики в результате выполненных изменений.

Имеется еще одна причина для вмешательства менеджера. Это закрытие проекта, так как в большинстве случаев команда не сможет отказаться от продолжения работы над проектом по собственной воле.

В гибких технологиях роль менеджера в явном виде, как правило, отсутствует. Вместо этой роли имеются две роли: наставник и контролер. Техническое руководство осуществляет наставник. Основная задача наставника – помогать членам команды принимать правильные решения. Слежение за ходом работ над проектом выполняет контролер.

5.11. Организация работы на этапе внедрения программного продукта

К концу работы над первой версией разработчик готов приступить к эксплуатации программного продукта в реальных условиях. На завершающих стадиях работы над любой версией следует сократить цикл обратной связи. Вместо трехнедельных итераций рекомендуется перейти на итерации продолжительностью в одну неделю.

Как правило, для сертификации программного продукта (определения, готов ли продукт к использованию в реальных производственных условиях) требуется разработать новые тесты для того, чтобы убедиться в готовности системы к реальной работе. Именно на этой стадии применяется параллельное тестирование.

После внедрения очередной версии программного продукта начинается стадия сопровождения программного продукта. Начиная с этого времени, команда должна одновременно работать над реализацией новой функциональности и поддерживать установленную у заказчика версию в рабочем состоянии.

Завершающая стадия работы над проектом – самое время для оптимизации производительности программного продукта. Это объясняется тем, что в это время команда разработчиков обладает наиболее полными знаниями о предметной области, внедренными в дизайн программного продукта, имеет наиболее реалистичную оценку производственной нагрузки на продукт, получает в свое распоряжение реальную производственную аппаратную платформу. На завершающей стадии работы над проектом можно выполнить крупномасштабный рефакторинг кода, попробовать новую технологию, которую предполагается использовать в следующей версии продукта или в новом проекте, перевести программный продукт на другую платформу.

Разработка системы, которая уже функционирует в условиях реального производства, существенно отличается от разработки системы, которая еще не эксплуатируется на реальном предприятии. Команда разработчиков должна быть готова в любой момент прервать дальнейшую разработку для того, чтобы решить проблемы, возникшие при эксплуатации продукта, работающего на реальных данных

Внедрение программного продукта в штатную эксплуатацию влияет на скорость разработки. При планировании очередных итераций необходимо учитывать издержки, связанные с сопровождением продукта. Возможно, менеджеру проекта придется изменить состав команды, организовать службу технической поддержки, чтобы большая часть разработчиков не отрывалась слишком часто от текущей разработки для решения проблем, возникающих при эксплуатации программного продукта.

Рекомендуется организовать работу команды таким образом, чтобы со временем все разработчики поработали в службе технической поддержки, так как существуют вещи, которым можно научиться, только осуществляя

техническую поддержку программного продукта и о которых нельзя узнать другим путем. С другой стороны, в целом техническая поддержка – утомительное, менее интересное занятие, чем разработка.

Идеологи гибких технологий рекомендуют интегрировать новые разработанные фрагменты программного кода в работающей в реальных условиях программный продукт по мере их разработки. Нельзя оставлять готовый код в бездействующем состоянии дольше, чем в течение одной итерации. Если команда будет поддерживать код, эксплуатируемый у заказчика, и код, находящийся в разработке, приблизительно в синхронизированном состоянии, она будет раньше узнавать об интеграционных проблемах.

Каким бы не был удачным проект, наступает время, когда он прекращает свое существование. Это может произойти по двум причинам:

- У заказчика нет больше новых пожеланий. В этом случае необходимо написать небольшой документ – описание программного продукта, который может потребоваться через несколько лет, если в продукт будут вноситься изменения.
- Программный продукт перестал удовлетворять заказчика, количество дефектов превысило допустимую величину, добавление новых функций и исправление дефектов становится экономически невыгодно. В этом случае работа над проектом прекращается.

6. Обзор гибких методологий разработки программного обеспечения

6.1. Экстремальное программирование

История экстремального программирования началась в середине 90-х годов. В 1995 году промышленная компания Daimler Chrysler, выпускающая автомобили, начала работу над проектом системы контроля платежей служащим компании – Chrysler Comprehensive Compensation System (C3). Действуя в соответствии с общепринятой практикой реализации подобных проектов, Daimler Chrysler заключила контракт с партнерской компанией, в соответствии с которым за реализацию проекта бралась объединенная команда разработчиков из обеих организаций. Разработчики из партнерской компании придерживались технологии разработки, ориентированной на использование графического интерфейса и игнорировавшей автоматизацию тестирования. В результате была создана система, которая имела невыразительный графический интерфейс и для большинства служащих неправильно вычисляла зарплату. Время работы системы было очень велико: для создания месячной платежной ведомости ей понадобилось бы свыше трех месяцев.

Получив столь плачевный результат, руководство корпорации обратилось с просьбой спасти проект к всемирно известному специалисту в области разработки программного обеспечения Кенту Беку (Kent Beck). Анализируя проект C3, Кент Бек обнаружил плохо продуманный код, тесты, которые нельзя было запустить повторно, руководство, потерявшее уверенность в своем проекте. Он предложил выбросить весь написанный к этому времени программный код и начать полномасштабный XP-проект.

Прежний контракт был разорван, и Daimler Chrysler почти наполовину обновила свою команду разработчиков. Начиная с этого времени, команда стала работать по правилам экстремального программирования. Были распределены обязанности в команде, спланирован порядок реализации функций, установлены правила тестирования, опробовано и принято в качестве стандарта парное программирование. Результат был ошеломляющим: к концу 33-й недели была получена система, в которой уже было можно отлаживать производительность и проводить параллельное тестирование. В настоящее время система C3 ежемесячно рассчитывает зарплату более чем 86 тыс. рабочих компании.

6.1.1. Описание экстремального программирования

Экстремальное программирование основано на следующих ключевых ценностях:

- **Коммуникация.** Дисциплина XP нацелена на обеспечение непрерывной, постоянно осуществляемой коммуникации между участниками проекта. В команде разработчиков должна быть создана культура общения, которая предполагает обсуждение всех возникающих проблем и сложностей непосредственно в момент их обнаружения.
- **Простота.** Дисциплина XP предлагает решать задачу самым простым способом из всех возможных.
- **Обратная связь.** Дисциплина XP предлагает сделать все возможное, чтобы начать получать отзывы о разрабатываемом продукте как можно раньше и в дальнейшем получать их как можно чаще.
- **Храбрость.** Дисциплина XP призывает совершать правильные поступки в нужные моменты времени.

Дисциплина XP содержит следующие практики, которые позволяют непротиворечиво следовать ключевым ценностям:

- Игра в планирование – диалог между заказчиками и разработчиками с целью определить текущие задачи, приоритеты и сроки.
- Разработка через тестирование – частью системы являются автоматические тесты, которые позволяют в любой момент убедиться в корректности работы системы.
- Парное программирование – любой код разрабатывается двумя программистами.
- Рефакторинг – непрерывное улучшение кода путем его переработки.
- Простой дизайн – постоянное видоизменение дизайна системы с целью его упрощения и очищения.
- Коллективное владение кодом – все члены команды обладают правом вносить изменения в любой код системы.
- Постоянная интеграция – интеграция продукта выполняется постоянно, по несколько раз в день.
- Заказчик на месте разработки – рядом с разработчиками постоянно находится представитель заказчика, который работает вместе с ними.
- Частые выпуски версий – новые версии продукта выпускаются и внедряются в эксплуатацию как можно чаще.

- 40-часовая рабочая неделя – разработчики не должны работать сверхурочно, так как от этого снижается их производительность и качество их работы.
- Стандарты кодирования – все участники команды должны придерживаться общих стандартов кодирования.
- Метафора системы – простая аналогия, интуитивно понятная всем участникам проекта, которая в доступной форме описывает строение и функционирование системы.

Очевидно, что все вышеперечисленные практики не являются уникальными или оригинальными. Более того, от многих из этих практик уже давно отказались в пользу других, более сложных и более высокоуровневых, так как их слабость стала очевидной. Поэтому может создаться впечатление, что XP является упрощенной методологией разработки программного обеспечения, не вполне подходящей для реальных проектов.

Однако это не так. Дело в том, что в XP перечисленные практики комбинируются таким образом, что недостатки каждой из практик компенсируются достоинствами других практик. Более того, каждая из перечисленных практик используется в экстремальном виде, то есть с максимальной эффективностью. В результате возникает так называемый «синергетический эффект XP», когда величина результата превышает величину суммы составляющих.

6.1.2. Комбинация практик экстремального программирования

Игра в планирование

На первый взгляд, невозможно приступить к работе над проектом, имея на руках только грубый, приблизительный план предстоящей работы. Более того, постоянное обновление плана потребовало бы больших трудозатрат и повлекло бы недовольство заказчиков. Однако эти недостатки компенсируются в XP следующими факторами:

- заказчики выполняют обновление плана самостоятельно, на основании оценок, которые делаются программистами;
- в самом начале проекта имеется план, достаточно четкий для того, чтобы понять какой будет система через два-три года разработки;
- версии продукта выходят достаточно часто, что упрощает постоянное уточнение плана;
- представитель заказчика является членом команды разработки и имеет возможность постепенно уточнять план в зависимости от складывающейся ситуации.

Эти факторы позволяют приступить к работе, имея на руках только упрощенный, грубый план проекта. В течение проекта план будет постоянно пересматриваться и уточняться, в зависимости от текущей ситуации на проекте.

Небольшие версии

Может показаться, что невозможно приступить к эксплуатации системы уже через несколько месяцев разработки. Более того, частый выпуск версий кажется невозможным с точки зрения обеспечения высокого качества продукта. Однако:

- игра в планирование позволяет работать над наиболее важными историями, поэтому даже небольшая система будет способна приносить пользу заказчику;
- непрерывная интеграция значительно сокращает затраты, связанные с формированием очередной полноценной версии;
- разработка через тестирование обеспечивает небольшое количество дефектов в продукте и значительно сокращает затраты на традиционное тестирование;
- простой дизайн позволяет иметь архитектуру настолько простую, насколько это позволяет текущая версия продукта, что значительно упрощает выпуск первых версий продукта.

Эти факторы позволяют выпускать продукт небольшими версиями, запуская его в полноценную эксплуатацию уже через несколько месяцев разработки.

Метафора

На первый взгляд, невозможно приступить к разработке, обладая лишь метафорой архитектуры продукта, потому что в метафоре недостаточно детально отображено строение системы. Кроме того, метафора может оказаться ошибочной. Однако:

- разработка реального кода и тестов обеспечивает быструю обратную связь, которая позволяет уточнять и модифицировать метафору в течение проекта;
- наличие простой метафоры упрощает общение с заказчиком;
- постоянный рефакторинг позволяет итеративно пересматривать представление о выбранной метафоре.

Эти факторы позволяют спокойно приступать к разработке продукта, обладая лишь приблизительной метафорой архитектуры системы.

Простой дизайн

Может показаться, что невозможно разрабатывать систему на основе архитектуры, которой достаточно лишь для решения сегодняшних задач. Очевидным кажется факт, что нельзя проектировать систему не учитывая проблемы, с которыми придется столкнуться в будущем. Однако:

- постоянный рефакторинг кода значительно упрощает изменение и развитие архитектуры продукта;
- наличие простой метафоры позволяет систематично развивать архитектуру продукта;
- парное программирование обеспечивает использование простого дизайна, а не наивного дизайна.

Эти факторы позволяют строить системы на основе простого дизайна, достаточного лишь для решения текущих задач.

Разработка через тестирование

На первый взгляд, разработка через тестирование не приведет не к чему, кроме увеличения трудозатрат. Более того, разработчики презирают тестирование. Однако:

- дизайн системы максимально прост, что упрощает, в свою очередь, написание тестов;
- парное программирование в значительной мере снимает эмоциональный барьер, который обычно возникает у разработчиков во время написания тестов;
- наличие модульных тестов в значительной мере повышает уверенность в работоспособности продукта, что способствует улучшению эмоционального климата как в команде разработчиков;
- наличие модульных тестов в значительной мере повышает уверенность заказчика.

Эти факторы обеспечивают вам возможность постоянно разрабатывать тесты. Более того, разработка через тестирование, скорее всего, будет осуществляться с полной поддержкой заказчика.

Рефакторинг

Может показаться, что невозможно постоянно перерабатывать дизайн системы. Во-первых, это потребует слишком много времени. Во-вторых, в процессе рефакторинга высока вероятность нарушения работоспособности системы. Однако:

- коллективное владение кодом и стандарты кодирования обеспечивают возможность внесения изменений в любую часть системы, когда это необходимо;
- парное программирование снимает эмоциональный барьер, который возникает во время необходимости переработать часть дизайна системы;
- использование максимально простого дизайна значительно сокращает трудозатраты на рефакторинг;
- разработка через тестирование значительно сокращает вероятность внесения дефектов во время рефакторинга;
- непрерывная интеграция обеспечивает оперативное обнаружения дефектов, внесенных во время рефакторинга;
- 40-часовая рабочая неделя обеспечивает высокий эмоциональный тонус команды, что значительно сокращает вероятность внесения дефектов во время рефакторинга.

Эти факторы позволяют вам постоянно осуществлять переработку кода без опасений потери времени и внесения дефектов в систему.

Парное программирование

На первый взгляд, парное программирование должно сокращать производительность команды как минимум в два раза. Более того, парное программирование должно приводить к постоянным конфликтам между разработчиками. Однако:

- стандарты кодирования снижают трения между разработчиками;
- 40-часовая рабочая неделя обеспечивает высокий эмоциональный тонус, что тоже способствует уменьшению вероятности возникновения конфликтов;
- парное программирование обеспечивает написание высококачественного кода и тестов;
- парное программирование обеспечивает обсуждение всех изменений, которые вносятся в дизайн системы.

Эти факторы обеспечивают высокую эффективность парного программирования. Более того, парное программирование в значительной мере повышает качество дизайна и кода продукта.

Коллективное владение кодом

Может показаться, что невозможно разрешить любому члену команды вносить любые изменения в любую часть системы в любое удобное для этого время. Более того, кажется очевидным, что подобное отношение к коду системы должно привести к хаосу и заблокировать разработку. Однако:

- непрерывная интеграция значительно сокращает вероятность возникновения конфликтов, так как все дефекты обнаруживаются уже через несколько часов после их внесения;
- разработка через тестирование и парное программирование значительно сокращают вероятность нарушения работоспособности системы;
- следование общим стандартам кодирования значительно сокращает вероятность возникновения конфликтов между членами команды.

Эти факторы обеспечивают возможность коллективного владения кодом системы. Более того, коллективное владение кодом значительно снижает «фактор грузовика» (*truck factor*).

Непрерывная интеграция

На первый взгляд, непрерывная интеграция системы кажется невозможной, потому что это, обычно, является трудоемким и болезненным процессом. Однако:

- разработка через тестирование обеспечивает простой способ контроля над работоспособностью системы;
- парное программирование значительно сокращает возможность внесения дефектов в систему;
- непрерывный рефакторинг и простой дизайн значительно упрощают интегрирование новой функциональности.

Эти факторы обеспечивают возможность осуществления непрерывной интеграции. Более того, чем реже выполняется интеграция, тем больше она требует трудозатрат и тем вероятнее возникновение конфликтов в системе.

40-часовая рабочая неделя

Может показаться, что невозможно работать всего 40-часов в неделю, потому что это приведет к задержке выпуска продукта. Однако:

- игра в планирование позволяет вам делать наиболее важную работу в первую очередь;
- комбинация игры в планирование и разработки через тестирование сокращает вероятность возникновения авральных ситуаций;
- весь набор предлагаемых в экстремальном программировании практик позволяет работать с максимальной производительностью.

Эти факторы обеспечивают возможность использования 40-часовой рабочей недели. Более того, 40-часовая рабочая неделя обеспечивает полноценный отдых членов команды, что, в свою очередь, способствует благоприятному эмоциональному климату в команде разработчиков.

Заказчик на месте разработки

На первый взгляд, невозможно получить реального представителя заказчика в качестве члена команды разработчиков. Однако:

- представитель заказчика занят разработкой функциональных тестов системы, которые все равно придется разрабатывать заказчику;
- представитель заказчика помогает разработчикам в принятие важных решений по дизайну и функциональности системы;
- представитель заказчика постоянно участвует в игре в планирование, что обеспечивает взаимопонимание между разработчиками и заказчиком.

Эти факторы обеспечивают возможность использования представителя заказчика в качестве члена команды разработчиков. Более того, это в значительной мере способствует успешности проекта и снижения рисков.

Стандарты кодирования

Может показаться, что невозможно убедить команду разработчиков следовать общим стандартам кодирования. Однако:

- стандарты кодирования значительно сокращают трения между членами команды;
- стандарты кодирования способствуют взаимопониманию между разработчиками.

Эти факторы обеспечивают возможность использования общих стандартов кодирования. Более того, стандарты кодирования позволяют значительно увеличить производительность членов команды.

6.2. Scrum

Термин *Scrum*, применительно к разработке программного обеспечения, был впервые использован в работах Хиротаки Такеучи и Икуджиро Нонака, где авторы отметили успешность проектов, в которых участвуют небольшие сплоченные команды без жесткой специализации. Этот термин был заимствован из игры регби, где *scrum* – это команда из восьми человек, каждый член которой в каждый момент играет конкретную роль и взаимодействует с другими членами команды для достижения общей цели. Джеф Сазерленд использовал работу Такеучи и Нонака при создании методологии для корпорации *Easel* в 1993 году, которую по аналогии была названа *Scrum*, а Кен Швабер формализовал процесс для использования в индустрии разработки программного обеспечения.

Методология *Scrum* устанавливает простые и понятные правила управления процессом разработки и позволяет использовать уже существующие практики проектирования и кодирования, итеративно корректируя требования к продукту. Использование этой методологии дает возможность итеративно выявлять и устранять отклонения от желаемого результата на всех этапах разработки программного продукта.

Основой *Scrum* является итеративная разработка. *Scrum* определяет итеративные правила управления проектом, которые призваны обеспечивать достижение максимального эффекта от реализованной функциональности. Также в *Scrum* определяются основные правила взаимодействия участников команды, которые призваны обеспечивать максимально быстрой реакции на существующую ситуацию. Каждая итерация в *Scrum* может быть описана следующим образом: планируем – фиксируем – реализуем – анализируем. За счет фиксирования требований на время одной итерации и изменения длины итерации методология *Scrum* позволяет управлять балансом между гибкостью и предсказуемостью разработки.

6.2.1. Описание Scrum

В методологии *Scrum* формально выделены три роли, для каждой из которых формально определены зоны прав и ответственности: *владелец продукта* (*Product Owner*), *команда* (*Team*) и *скрам-мастер* (*ScrumMaster*). *Владелец продукта* является лицом, ответственным за определение требований к продукту. *Команда* является группой самостоятельных и инициативных разработчиков, ответственных за реализацию проекта. *Скрам-мастер* является лицом, ответственным за решение всех организационных проблем и соблюдение методологии *Scrum*.

В методологии *Scrum* проект разбивается на три фазы: *подготовка* (*Pregame*), *реализация* (*Game*) и *завершение* (*Postgame*). Во время фазы *подготовки* проекта формируется общий план проекта, определяется список основных требований к продукту и разрабатывается высокоуровневая архитектура продукта. Во время фазы *реализации* осуществляется итеративное развитие продукта. Во время фазы *завершения* проекта осуществляются действия, необходимые для подготовки продукта к выходу на рынок.

Фаза реализации разбита на последовательность итераций, которые называются *спринтами* (*Sprint*). В результате каждого спринта в продукте реализуется новый, заметный для владельца продукта, объем функциональности. Важно, что в конце каждого спринта продукт остается в работоспособном состоянии. Каждый спринт начинается с сессии планирования (*Sprint Planning Meeting*), во время которой определяется объем функциональности, которая будет реализована в течение спринта. Ежедневно, в течение спринта, проводится собрание участников проекта, называемое *скрам-сессия* (*Daily Scrum Meeting*), на котором все члены команды сообщают о своих текущих планах и всех трудностях, возникающих в работе. По завершению спринта проводится демонстрационная сессия (*Sprint Review Meeting*), во время которой реализованная функциональность демонстрируется владельцу проекта.

Реализация проекта в методологии *Scrum* управляется тремя документами: *журнал продукта* (*Product Backlog*), *журнал спринта* (*Sprint Backlog*) и *график спринта* (*Burndown Chart*). Журнал продукта является высокоуровневым списком функциональных и технических требований, необходимых для реализации продукта. Журнал спринта является детализированным списком функциональных и технических требований, необходимых для успешного завершения итерации. График спринта показывает ежедневное изменение общего объема работ, оставшегося до завершения итерации.

6.2.2. Роли в Scrum

Четкое разделение ролей исключительно важно для повышения производительности участников проекта и вполне соответствует игровой стратегии, которой следует методология *Scrum* в целом. Разделение прав и ответственности позволяет эффективно принимать решения и не допускать организационных заторов.

Все заинтересованные в проекте лица разделяются на две большие символические группы: «поросята» (*pigs*) и «цыплята» (*chickens*). Такое разделение основано на старой шутке, в которой цыпленок предлагает поросенку открыть ресторан. Поросенок отказывается, узнав, что в ресторане предполагается подавать яичницу с беконом.

Данная шутка достаточно полно описывает ситуацию, которая часто возникает в проектах по разработке программного обеспечения. Группа разработчиков, архитекторов, администраторов и все прочих специалистов непосредственно выполняющих работы по реализации проекта оказывается в неравном положении по отношению к менеджменту проекта. Методология *Scrum* предполагает справедливое разделение прав и обязанностей между разработчиками («поросятами») и менеджментом («цыплятами»). Группа разработчиков получает возможность самостоятельно принимать часть решений, в то время как менеджмент сохраняет вполне достаточный контроль над ходом проекта.

Как уже упоминалось выше, в методологии *Scrum* формально выделены роли *владельца продукта*, *команды и скрам-мастера*, из которых только *команда* относится к поросятам. *Владелец продукта*, *скрам-мастер* и представители менеджмента организации являются цыплятами, так как непосредственно не вовлечены в выполнение работ по реализации проекта.

Владелец продукта

Владелец продукта (Product Owner) это человек, отвечающий за формирование списка требований к функциональности продукта. Обычно владелец продукта является представителем или доверенным лицом заказчика. Для компаний, выпускающих коробочные продукты, владелец продукта является лицом, которое имеет исчерпывающее представление о предметной области и текущем положении на рынке.

Владелец продукта должен составить бизнес план, показывающий ожидаемую доходность и план развития с требованиями, отсортированными по коэффициенту окупаемости инвестиций. Исходя из имеющейся информации, владелец продукта подготавливает список требований, отсортированный по значимости. Весь круг вопросов, связанный с коммерческой эффективностью продукта является зоной ответственности владельца продукта.

Владелец продукта обладает следующими основными правами и обязанностями:

- формирует журнал продукта;

- определяет приоритеты функциональности в соответствии с их коммерческой ценностью;
- определяет дату и содержание выпусков продукта;
- отвечает за коммерческую успешность продукта;
- уточняет функциональность и приоритеты после каждого спринта;
- принимает (или отклоняет) результаты проекта.

Команда

Команда (Scrum Team) – группа, состоящая из пяти–девяти самостоятельных, инициативных разработчиков, включая программистов, архитекторов, тестеров и всех прочих специалистов, непосредственно работающих над реализацией проекта. Команда активно участвует в выборе цели для следующей итерации и обеспечивает достижение этой цели в отведенные сроки и с заявленным качеством. Цель итерации считается достигнутой только в том случае, когда вся заявленная функциональность полностью реализована и протестирована. Все члены команды должны уметь самостоятельно оценивать и планировать свою работу в рамках итерации. В обязанности всех членов команды входит участие в выборе цели итерации и определение результата работы.

Команда обладает следующими основными правами и обязанностями:

- активно участвует в выборе цели следующей итерации;
- имеет право предпринимать все возможные (в рамках проекта) действия ради достижения цели спринта;
- обеспечивает принятый уровень качества продукта;
- самостоятельно организует себя и свою работу;
- демонстрирует результаты работы владельцу проекта.

Скрам-мастер

Скрам-мастер – человек, являющийся посредником между всеми участниками проекта, чьими основными обязанностями является организация процесса и соблюдение методологии *Scrum*. В обязанности скрам-мастера входит обеспечение четкого взаимодействия между всеми участниками проекта. Ответственностью скрам-мастера является, также, обеспечение соблюдения регламента всех сессий, предусмотренных методологией. От действий скрам-мастера во многом зависит производительность команды и успех проекта в целом.

Скрам-мастер обладает следующими основными правами и обязанностями:

- гарантирует высокую производительность работы команды;
- отвечает за обеспечение кооперации между всеми участниками проекта;

- отвечает за преодоление всех возникающих организационных и коммуникационных барьеров;
- защищает команду от всех нежелательных воздействий извне;
- отвечает за соблюдение на проекте всех правил методологии *Scrum*.

6.2.3. Фазы проекта в Scrum

Проект в *Scrum* имеет три основных фазы: подготовка (pregame), разработка (game) и з*авершение (postgame)*. Как уже упоминалось выше, в методологии *Scrum* предполагается, что процессы дизайна, анализа и разработки полностью непредсказуемы. Разделение на фазы, которое используется в методологии *Scrum*, позволяет контролировать эту непредсказуемость и гибко управлять ходом проекта.

Подготовка

Фаза подготовки состоит из двух основных частей: планирование и разработка архитектуры. Основной целью фазы подготовки является формирование высокоуровневого плана проекта, как в организационном, так и в техническом смысле.

На этапе планирования:

- разрабатывается журнал продукта, который содержит исчерпывающий список требований к разрабатываемому продукту;
- определяет дата и состав релизов;
- осуществляется учет возможных рисков;
- осуществляется выбор средств разработки;
- оцениваются затраты на проект, включая стоимость разработки, маркетинга и вывода продукта на рынок;
- подтверждается финансирование проекта в запланированном объеме.

На этапе разработки архитектуры:

- проводится обзор требований, внесенных в журнал проекта;
- архитектура проекта разрабатывается (или уточняется) таким образом, чтобы удовлетворить все требования, внесенные в журнал проекта;
- идентифицируются основные проблемы и сложности, которые могут возникнуть во время реализации проекта.

Разработка

В методологии *Scrum* фаза разработки считается непредсказуемым процессом. В данной методологии не предпринимается никаких попыток формализовать процессы дизайна, анализа и разработки. Вместо этого методология *Scrum* предоставляет возможности адекватного контроля над ходом проекта и своевременного реагирования на текущие результаты проекта. Процессы дизайна, анализа и разработки рассматриваются как «черный ящик» и разделяются на относительно короткие итерации.

Каждая итерация, которая называется *спринт* (*Sprint*), имеет ограничение по времени от двух до четырех недель. Этого времени должно быть достаточно для реализации такого объема функциональности, который

значительно влияет на состояние продукта и заметен с точки зрения владельца продукта. После завершения каждого спринта продукт должен оставаться в работоспособном состоянии. С другой стороны, длительность спринта должна быть достаточно небольшой, чтобы команда могла эффективно работать, не тратя значительные ресурсы на отчеты и документирование хода процесса разработки.

Разделение процесса разработки на спринты позволяет владельцу продукта контролировать и влиять процесс разработки. Более того, в течение проекта могут появляться новые требования к продукту и изменяться ситуация на рынке, в результате чего могут изменяться цели и задачи проекта в целом. Однако во время спринта журнал продукта замораживается в целях обеспечения высокой производительности команды и сокращения организационных издержек.

Каждый спринт включает в себя следующие основные этапы:

- сессия планирования спринта (*Sprint Planning Meeting*);
- основная итерация;
- демонстрационная сессия (*Sprint Review Meeting*).

Сессия планирования спринта предназначена для формирования журнала спринта, а демонстрационная сессия предназначена демонстрации результатов спринта. Более подробно сессии будут описаны ниже.

В течение спринта команда может искать советов, информации и любых других видов поддержки у других участников проекта. С другой стороны, никто из участников проекта не имеет право каким-либо образом манипулировать действиями команды во время спринта.

План спринта может оказаться неадекватным и команда может прийти к выводу, что она не сможет реализовать все включенные требования. Также возможна ситуация, в которой команда может получить возможность реализовать требования, не включенные в текущий спринт. В этом случае команда имеет право, проконсультировавшись с владельцем проекта, включить или исключить некоторые требования из текущего спринта.

Спринт может быть прерван в скрам-мастером в следующих случаях:

- техническое решение, принятое на этапе планирования спринта, оказалось неработоспособным;
- резко изменилось положение на рынке;
- из спринта исключено слишком много требований, и его результаты будут ничтожны с точки зрения владельца проекта;
- работоспособность команды заблокирована по организационным причинам.

В случае прерывания спринта назначается планирование нового спринта.

Члены команды, в течение спринта, имеют только две административных обязанности:

- посещать ежедневную скрам-сессию;
- поддерживать журнал спринта в адекватном состоянии и обеспечивать доступ к нему всем желающим участникам проекта.

Итерация считается успешно завершенной, если полностью реализована вся запланированная функциональность и в журнале спринта не осталось незавершенных задач. В этом случае проводятся демонстрационная сессия и анализ производительности команды, после чего начинается следующая итерация.

Завершение

Когда команда менеджмента чувствует, что требования к продукту удовлетворены на достаточном уровне и продукт готов к выходу на рынок, начинается стадия завершения. На этой стадии осуществляются такие действия, как доведение пользовательской документации (например, типографическая подготовка документации к публикации), сертификация и подготовка маркетинговых материалов.

6.2.4. Сессии

Методология *Scrum* предполагает регулярное проведение формальных сессий или собраний, таких как сессия планирования спринта, ежедневная скрам-сессия и демонстрационная сессия. Все сессии имеют формальное ограничение по времени, что предотвращает возникновение серий затяжных совещаний, которые могут блокировать работоспособность участников проекта. Кроме этого, для каждой сессии формально определены права и обязанности всех участников проекта, что призвано предотвращать возникновение организационных и личностных конфликтов.

Сессия планирования спринта

Сессия планирования спринта проводится в начале каждого спринта. Данная сессия обычно имеет ограничение по времени в восемь часов и состоит из двух сегментов по четыре часа каждый. Первый сегмент предназначен для определения цели спринта на основе журнала продукта, а второй сегмент предназначен для формирования журнала спринта.

На сессии планирования спринта присутствуют скрам-мастер, владелец продукта и команда. На сессию планирования спринта также могут быть приглашены дополнительные участники, которые могут предоставить вспомогательную коммерческую или техническую информацию. Однако дополнительные участники не влияют на исход сессии и могут быть удалены после того, как вспомогательная информация будет предоставлена.

Во время первого сегмента сессии команда указывает набор требований, которые она могла бы, гипотетически, реализовать в течение спринта. Команда также имеет право давать рекомендации относительно включения требований в спринт, но право включения требования в планируемый спринт полностью принадлежит владельцу продукта. Однако за командой сохраняется право контролировать объем работ, назначенных на планируемый спринт.

Как указано выше, все участники сессии действуют в условиях весьма ограниченного времени, которого может оказаться недостаточно для детального анализа требований. В этом случае данный анализ выполняется в течение спринта. Поэтому, в случае нечеткого или недостаточно детализованного журнала продукта, вполне может возникнуть ситуация, в которой команда будет не в состоянии реализовать все требования, назначенные на спринт.

Второй сегмент сессии происходит непосредственно после первого сегмента. Результатом второго сегмента является список конкретных действий, которые необходимо выполнить для реализации требований, назначенных на спринт. Этот список называется журналом спринта. Журнал спринта может быть неполным, но должен быть достаточно подробным для того, чтобы команда сразу могла начать работу. Журнал спринта может дополняться или уточняться командой в течение спринта.

Владелец продукта обязан присутствовать второго сегмента сессии и отвечать на все вопросы, возникающие у команды. Однако в течение второго сегмента команда действует полностью самостоятельно. Остальные участники сессии могут только предоставлять информацию по просьбе команды.

Ежедневная скрам-сессия

В течение основной итерации ежедневно проводится так называемая *скрам-сессия* (*Daily Scrum Meeting*). Данная сессия имеет ограничение по времени в 15 минут, независимо от размеров команды. Основной целью ежедневной скрам-сессии является повышение дисциплины и фокусировка команды на целях спринта.

Рекомендуется проводить скрам-сессию в одном и том же месте в одно и тоже время. На скрам-сессии в обязательном порядке должны присутствовать все члены команды, поэтому рекомендуется выбирать место и время скрам-сессии таким образом, чтобы оно было максимально удобно всем членам команды. Если кто-то из членов команды не может присутствовать на скрам-сессии лично, он должен участвовать по телефону или попросить другого члена команды выступить за него. Скрам-мастер начинает скрам-сессию точно в назначенное время, независимо от того, сколько членов команды присутствует.

Во время скрам-сессии скрам-мастер последовательно опрашивает всех членов команды. Каждый член команды обязан ответить на следующие вопросы:

- что он сделал для проекта со времени прошлой скрам-сессии;
- что он собирается сделать для проекта до следующей скрам-сессии;
- что мешает ему работать над проектом максимально эффективно.

Во время скрам-сессии члены команды не должны углубляться в обсуждение дизайна, ошибок и прочих технических или организационных проблем. Скрам-мастер отвечает за управление процессом блиц-опроса. В каждый момент скрам-сессии говорит не более одного человека. Все возможные дискуссии осуществляются после скрам-сессии (по договоренности участников проекта).

Представители менеджмента могут посещать скрам-сессии, но их роль исключительно пассивная. Цыплята не могут активно участвовать в скрам-сессии. Более того, цыплята не имеют права по собственному желанию вступать в дискуссии, давать советы или каким-либо другим образом влиять на поведение членов команды.

Скрам-мастер несет ответственность за проведение скрам-сессий в установленном порядке. В случае если присутствие представителей менеджмента каким-либо образом повышает напряженность обстановки, скрам-мастер может ограничить присутствие цыплят на скрам-сессиях. В случае если кто-либо из цыплят регулярно нарушает правило проведения скрам-сессий, скрам-мастер имеет возможность лишить данного человека

права присутствия на скрам-сессиях. В случаях же, когда регламент регулярно нарушает кто-либо из членов команды, скрам-мастер вправе исключить данного человека из команды.

Демонстрационная сессия

Демонстрационная сессия проводится в конце каждого спринта. Данная сессия обычно имеет ограничение по времени в четыре часа. Целью демонстрационной сессии является демонстрация реализованной функциональности владельцу проекта и другим представителям менеджмента.

В течение демонстрационной сессии команда может демонстрировать только ту функциональность, которая уже полностью реализована. Понимание словосочетания «полностью реализованная функциональность» зависит от специфики проекта, но методология *Scrum* предполагает здесь достаточно высокий уровень завершенности, когда функциональность реализована, интегрирована, протестирована и не нарушает общую работоспособность продукта. Во избежание заблуждений среди представителей менеджмента, запрещается демонстрировать функциональность, которая на данный момент реализована лишь частично.

Демонстрационная сессия начинается с выступления одного из членов команды, который описывает цели текущего спринта и приводит список требований к продукту, которые были реализованы. Далее члены команды совместно представляют результаты спринта.

В конце презентации все представители менеджмента последовательно опрашиваются на предмет любых замечаний и предложений по поводу текущего спринта и всего проекта в целом. Представители менеджмента имеют право свободно высказываться по всем аспектам проекта, включая отсутствие необходимой функциональности, качества продукта в целом и т.д.

По итогам презентации и обсуждения, владелец продукта может инициировать дискуссию по добавлению в журнал продукта новых требований и изменению приоритетов.

Ответственностью скрам-мастера является выбор участников демонстрационной сессии, решение организационных вопросов, связанных с проведением сессии и назначение даты следующей демонстрационной сессии.

6.2.5. Документы

Методология *Scrum* требует наличия всего трех типов формальных документов: журнала продукта, журнала спринта и графика спринта. Наличие данных документов в достаточной степени обеспечивает понимание текущих процессов всеми участниками проекта, не требуя неоправданных затрат ресурсов.

Журнал продукта

Журнал продукта содержит список требований к продукту, отсортированных по значимости. Журнал продукта должен включать функциональные и технические требования, необходимые для реализации продукта. Самые приоритетные из требований должны быть достаточно детально прописаны, чтобы команда могла их проанализировать и предоставить грубую оценку трудозатрат, необходимых для их реализации. Журнал продукта может дополняться, изменяться и уточняться в течение всего проекта. Своевременная детализация журнала продукта необходима для обеспечения команды достаточным объемом требований, подготовленных для анализа и реализации, и является областью ответственности владельца продукта.

Журнал спринта

Журнал спринта содержит детализированный список задач, выполнение которых необходимо для реализации требований, которые включены в текущий спринт. Рекомендуется проводить разбивку на задачи таким образом, чтобы выполнение одной задачи занимало не больше двух дней. После завершения детализации, оценка трудозатрат по журналу спринта сравнивается с первичной оценкой в журнале продукта. Если существует значительное расхождение, то команда договаривается с владельцем продукта об объеме работ, который должен быть выполнен в течение итерации, и о том, какой объем работ будет перенесен на следующую итерацию.

График спринта

График спринта показывает ежедневное изменение общего объема работ, оставшегося до окончания итерации. Этот график позволяет всем участникам проекта осуществлять анализ текущей ситуации и своевременно реагировать на отклонения.

Во время сессии планирования команда формирует журнал спринта, который содержит список задач, выполнение которых необходимо для успешного завершения итерации. Сумма оценок трудозатрат по всем задачам в журнале спринта является общим объемом работы, который необходимо выполнить за итерацию. После завершения каждой задачи скрам-мастер пересчитывает оставшийся объем работ и отмечает это на графике спринта. Итерация считается успешной, если в журнале спринта не осталось незавершенных задач и график спринта используется как вспомогательный

инструмент, позволяющий корректировать работу для завершения итерации вовремя, с полностью реализованной функциональностью и требуемым качеством.

6.3. Dynamic Systems Development Method

Методология *Dynamic Systems Development Method* (далее *DSDM*) создана в 1994 году консорциумом из 17 британских компаний. На данный момент членами консорциума DSDM являются тысячи компаний по всему миру. Фреймворк методологии DSDM включает в себя большую часть современного знания об управлении проектами. Изначально методология DSDM была предназначена исключительно для проектов по разработке программного обеспечения, однако в настоящий момент DSDM используется и в других отраслях. Основными достоинствами фреймворка DSDM являются простота, расширяемость и доказанная высокая эффективность. При этом важно отметить, что, как и многие другие гибкие методологии, фреймворк DSDM не предназначен для управления любыми типами проектов.

Основным недостатком методологии DSDM является относительно высокий барьер входа. Внедрение фреймворка DSDM не является быстрой и дешевой процедурой. Более того, внедрение DSDM может потребовать значительных изменений в культуре организации.

Методология DSDM позиционируется как наиболее зрелая методология гибкой разработки программного обеспечения. Многие другие гибкие методологии сфокусированы на *программировании*, в то время как методология DSDM сфокусирована на организации процесса производства программного обеспечения. В этом смысле методология DSDM имеет много общих черт с методологией Scrum.

6.3.1. Принципы DSDM

Перечисленные ниже принципы являются неотъемлемой частью фреймворка DSDM. Нарушение любого из этих принципов является нарушением философии DSDM и может привести к значительному росту рисков.

Активное вовлечение пользователей в процесс разработки

Данный принцип является наиболее важным в методологии DSDM, поскольку вовлечение пользователей в процесс разработки значительно сокращает количество ошибок в проектных решениях, что в свою очередь значительно сокращает совокупную стоимость проекта. Вместо работы с большой аудиторией пользователей, DSDM предлагает *непрерывно* взаимодействовать с небольшой, специально отобранной группой пользователей.

Команда разработчиков может принимать решения

Бюрократические препоны, такие как обязательное согласование даже мелких и рутинных решений, могут значительно затруднить продвижение проекта. Фреймворк DSDM приветствует сокращение издержек подобного рода. Одной из мер сокращения бюрократических издержек является

частичная передача членам команды права принимать решения в следующих областях:

- выбор функциональности на текущую итерацию;
- назначение приоритетов требованиям;
- любые технические детали реализации.

Частая поставка результатов

Частый выпуск версий гарантирует, что все ошибки будут обнаружены быстро. Чем раньше обнаружена ошибка, тем проще ее исправить. Этот принцип относится как к выпуску работоспособных версий, так и к поставке документов содержащих анализ требований, модели данных и т.п.

Критерием приемлемости результатов является их соответствие бизнесу

Основной целью проекта, в соответствии с методологией DSDM, является как можно более быстрая поставка программного обеспечения, способного удовлетворять потребности бизнеса. При этом методология DSDM не призывает изготавливать программное обеспечение «на коленках». Данная методология призывает сначала удовлетворить потребности бизнеса, а потом уже выполнить такие *обязательные* действия как, например, рефакторинг.

Итеративная и инкрементальная разработка

Для того, чтобы сложность управления проектом всегда оставалась на приемлемом уровне, проект разбивается на несколько итераций. В течение каждой итерации в продукт добавляется новая функциональность, до тех пор, пока потребности бизнеса не будут полностью удовлетворены. Этот принцип требует признания того факта, что любое программный продукт будет изменяться в будущем. Этот принцип может быть принят в самом начале проекта. То есть список требований к проекту тоже строиться итеративно и может изменяться. Причем чем меньше итерации, тем проще реагировать на изменения.

Любые действия могут быть отменены

Любое действие, выполненное в процессе разработки, может быть отменено в будущем. Разработчики часто не любят откатывать изменения, сделанные в продукте, потому что боятся потерять результаты сделанной работы. Однако методология DSDM пропагандирует разработку *маленькими шагами*, что гарантирует, в худшем случае, незначительность потери.

Стабильность высокоуровневых требований

Чрезмерная свобода в изменении требований может значительно увеличить риски. Для ограничения возможностей изменения требований в процессе разработки, методология DSDM предлагает *заморозить* основные высокоуровневые требования к продукту. Набор основных высокоуровневых требований к продукту выбирается и согласуется на ранних стадиях проекта.

Тестирование выполняется постоянно и непрерывно

Многие традиционные методологии разработки программного обеспечения откладывают тестирование на поздние стадии жизненного цикла проекта. Методология DSDM призывает начать тестирование как можно раньше. Вплоть до тестирования проектных документов на специальных семинарах с заинтересованными лицами.

Взаимодействие и кооперация

Методология DSDM поощряет взаимодействие между техническим персоналом и менеджментом. Взаимодействие и кооперация между всеми заинтересованными лицами являются одними из основных факторов успеха проекта.

6.3.2. Роли

Методология DSDM определяет набор стандартных ролей. Каждая роль имеет свою собственную зону ответственности. Каждому члену команды сопоставляется одна из ролей. Ниже перечислены основные роли, определяемые методологией DSDM.

Менеджер проекта

Менеджер проекта (*Project Manager*) обеспечивает общее руководство проектом.

Провидец

Провидец (*Visionary*) является движущей силой проекта. *Провидец* следит за соответствием проекта коммерческим целям и задачам. *Провидцем* часто является топ-менеджер, который инициировал проект. Данная роль предполагает возможность частичной занятости.

Чемпион проекта

Чемпион проекта (*Project Champion* или *Executive Sponsor*) обладает возможностями и обязанностями по распоряжению ресурсами и фондами, которые необходимы данному проекту. *Чемпион проекта* несет ответственность за принятие любых решений, связанных с проектом. Данная роль предполагает возможность частичной занятости.

Лидер команды

Лидер команды (*Team Leader*) руководит командой разработчиков и обеспечивает эффективность ее работы.

Технический координатор

Технический координатор (*Technical Co-ordinator*) отвечает за разработку архитектуры продукта. *Технический координатор* также отвечает общее техническое состояние проекта.

Разработчик

Разработчик (*Developer*) участвует в анализе требований, моделировании и проектировании. Очевидно, что основной обязанностью *разработчика* является *программирование*.

Тестировщик

Тестировщик (*Tester*) отвечает техническое тестирование продукта.

Представительный пользователь

Представительный пользователь (*Ambassador User*) представляет пользователей продукта. *Представительный пользователь* отвечает за то, чтобы разработчики вовремя получали обратную связь со стороны пользователей.

Пользователь-консультант

Пользователем-консультантом (*Advisor User*) может быть любой пользователь, представляющий значительную точку зрения на продукт. *Пользователь-консультант* привносит в проект знание по некоторому аспекту использования разрабатываемого продукта. Данная роль предполагает возможность частичной занятости.

Секретарь

Секретарь (*Scribe*) отвечает за протоколирование всех соглашений и решений, принятых во время семинаров.

Посредник

Посредник (*Facilitator*) отвечает за проведение семинаров. *Посредник* также отвечает за эффективность коммуникации между всеми членами команды.

6.3.3. Разделение на команды

Методология DSDM рекомендует создавать команды небольшого размера, до шести человек (без учета руководящих персон вроде *провидца* и *чемпиона проекта*). При этом над одним проектом может работать несколько команд. Известны примеры проектов, выполненных по методологии DSDM, в которых участвовало до 150 человек. Команды могут отвечать как за реализацию некоторого набора функциональности (например, отдельная команда может отвечать за реализацию системы базы данных), так и за некоторые виды деятельности (например, в проекте могут быть две команды, отвечающие за *разработку*, и одна команда, отвечающая за *тестирование*). На рис. 15 приведена типичная структура команды разработчиков (участники проекта, имеющие полную занятость, изображены в закрашенных прямоугольниках).

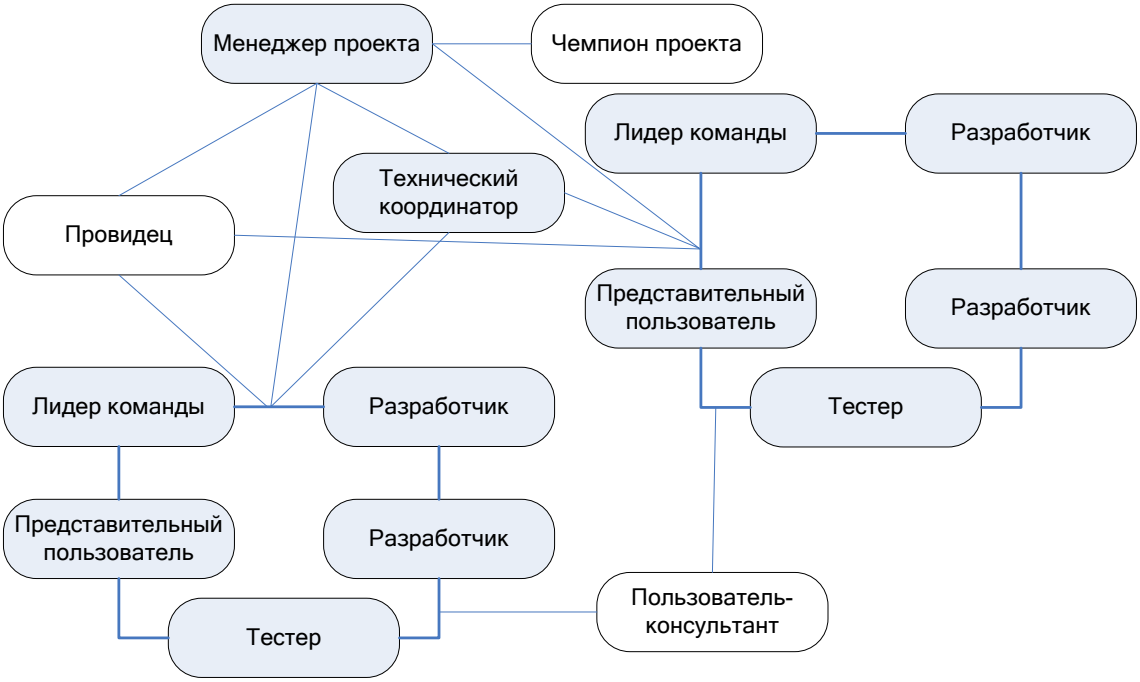

Рис. 15. Типичная структура команды в DSDM

6.3.4. Фазы проекта

Процесс разработки в методологии DSDM состоит из семи основных фаз. Каждая фаза имеет свои ключевые задачи и при этом может включать дополнительные задачи (особенно в случаях, когда DSDM комбинируется с другими методологиями). Методология DSDM также не специфицирует какие технические задачи должны выполняться на каждой итерации, что позволяет адаптировать DSDM к разным проектам и организациям. Ход типичного проекта по методологии DSDM изображен на рис. 16.

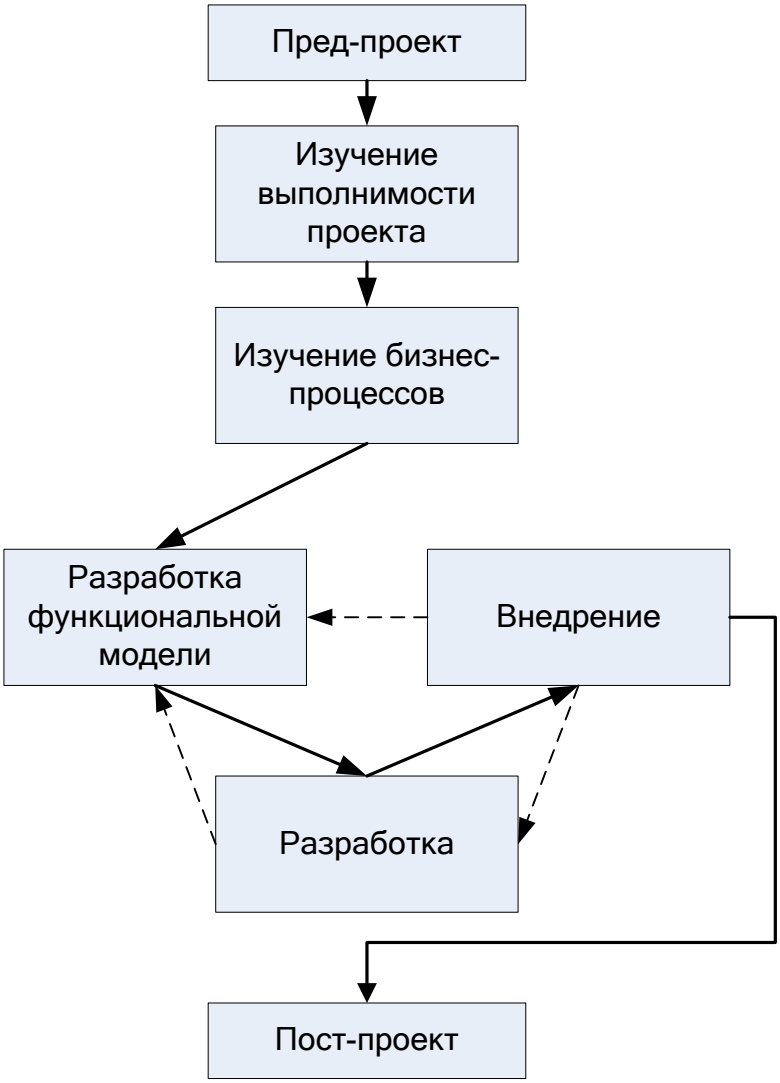

Рис. 16. Ход типичного проекта в DSDM

Пред-проект

Фаза *пред-проекта* (*Pre-Project*) включает в себя обсуждение возможных проектов. На этой стадии предпринимается решение о реализации проекта *вообще*.

Изучение выполнимости проекта

На фазе *изучения выполнимости проекта* (*The Feasibility Study*) анализируется возможность выполнения проекта. Исследуются возможности реализации программной системы, которая способна удовлетворить потребности бизнеса. На стадии *изучения выполнимости проекта* принимается решение, что проект *можно реализовать* (с учетом текущих ограничений времени и ресурсов).

Изучение бизнес-процессов

На фазе *изучения бизнес-процессов* (*The Business Study*) исследуются коммерческие аспекты проекта:

- имеет ли проект коммерческую ценность?
- каков будет состав участников проекта?

- каков общий план реализации проекта?
- какие технологии будут использоваться?

Итерация разработки функциональной модели

Во время *итерации разработки функциональной модели* (*Functional Model Iteration*) разрабатывается и анализируется функциональный прототип системы. Функциональный прототип показывает, какими функциями должна обладать система и как она должна их выполнять. Функциональный прототип позволяет ответить на следующие вопросы:

- насколько удобно будет пользоваться готовой системой?
- будет ли готовая система обладать необходимым уровнем производительности?
- каков наилучший путь реализации готовой системы?

Проект может включать в себя несколько итераций разработки функциональной модели.

Итерация разработки

Во время *итерации разработки* (*Design and Build Iteration*) продукт проектируется и разрабатывается. Причем разработка сразу ведется на должном уровне качества. Обычный проект содержит несколько *итераций разработки*.

Внедрение

На фазе *внедрения* (*Implementation*) продукт подготавливается к выпуску, разрабатывается пользовательская документация, проводится обучение пользователей и т.д. Проект может включать в себя несколько итераций внедрений.

Пост-проект

На фазе *пост-проекта* (*Post-Project*) осуществляется анализ реализованной системы и принимается решение о возможном расширении функциональности продукта. Данный анализ выполняется по прошествии некоторого времени с момента технического завершения проекта.

6.3.5. Ключевые практики

Таймбоксы

Вместо контрольных точек в методологии DSDM используются так называемые таймбоксы (от англ. *timebox*). Таймбокс является коротким интервалом времени (обычно не длиннее шести недель), в течение которого должен быть выполнен некоторый набор задач. Таймбоксы должны обладать по возможности меньшей длительностью, потому, что прогнозирование и планирование затрудняется с увеличением рассматриваемого периода времени. В конце таймбокса обычно осуществляется поставка продукта. Таймбоксы обладают следующими свойствами:

- таймбоксы могут различаться по длительности (в рамках одного проекта);
- возможны параллельные таймбоксы;
- в рамках таймбокса может выполняться несколько фаз DSDM;
- таймбоксы могут быть вложены друг в друга.

Принцип назначения приоритетов MoSCoW

Предполагается, что проекты DSDM должны выполняться вовремя и в соответствии с запланированным бюджетом. Однако при этом часть функциональности может быть не реализована. Поэтому важным является принцип назначение приоритетом, которым руководствуются при анализе требований. Для назначения приоритетов в методологии DSDM используется принцип MoSCoW:

- **must have** – требования, помещенные в эту категорию, должны быть обязательно удовлетворены, иначе система будет бесполезна;
- **should have** – требования, помещенные в эту категорию, важны для системы в целом и несут в себе значительную ценность для пользователя. Однако они могут быть (временно) опущены исходя из соображений экономии времени и ресурсов;
- **could have** – требования, помещенные в эту категорию, расширяют функциональность системы, но могут быть реализованы в следующем таймбоксе без особых потерь;
- **want to have** – требования, помещенные в эту категорию, имеют ценность только для узких групп пользователей и не влияют на проект в целом.

Список требования, классифицированных по принципу MoSCoW, является удобным источником задач на протяжении всего проекта.

Прототипирование

Прототипирование удовлетворяет двум основным принципам методологии DSDM – частой поставке результатов и инкрементальной разработке. Прототипирование используется для изучения технических сложностей, с которыми придется столкнуться при реализации запланированной функциональности. Также прототипирование позволяет быстро получить отзывы от пользователей. В методологии DSDM различаются следующие виды прототипов:

- **бизнес-прототип** – позволяет оценить разрабатываемую функциональность с коммерческой точки зрения;
- **usability-прототип** – позволяет оценить удобство пользовательского интерфейса;
- **performance-прототип** – позволяет оценить производительность системы на заданном уровне загрузки;
- **технологический прототип** – позволяет технически оценить возможный вариант реализации новой функциональности.

Семинары

Семинары используются во многих методологиях разработки программного обеспечения, поскольку они позволяют установить тесный контакт между всеми заинтересованными лицами. Однако семинары не столь эффективны в случае больших или гетерогенных команд. Методология DSDM пытается решить эти проблемы, предлагая эффективный вариант выбора состава участников каждого семинара. На каждый семинар приглашаются только действительно заинтересованные лица или их представители. Также предусмотрен вариант привлечения незаинтересованных арбитров в случае обсуждения *острых* тем.

6.3.6. DSDM как гибкая методология разработки ПО

Принципы методологии DSDM хорошо сочетаются с основными принципами манифеста гибких технологий разработки программного обеспечения. В методологии DSDM высоко ценится индивидуальность всех заинтересованных сторон. Создание работающего программного обеспечение, способного удовлетворять потребности бизнеса, является в методологии DSDM основным критерием успеха проекта. Сотрудничество и кооперация всех заинтересованных сторон является в методологии DSDM одним из ключевых способов достижения успеха в проекте. Методология DSDM также изначально предполагает возможность изменения требований к продукту на любой стадии проекта.

Поэтому методология DSDM, несмотря на некоторую формальность предлагаемого процесса разработки, полностью соответствует основным принципам гибких технологий разработки программного обеспечения. Более того, методология DSDM предполагает тесную интеграцию с другими гибкими методологиями. Например, методология экстремального программирования может быть тесно интегрирована с методологией DSDM в рамках одного проекта. В результате чего появляется возможность использовать все основные преимущества экстремального программирования в области проектирования и реализации системы совместно с основными преимуществами методологии DSDM в области управления требованиями и проектом в целом.

6.4. Feature Driven Development

Методология *Feature Driven Development* (далее *FDD*) была разработана Джефом Де Люка в 1997 году для реализации проекта в одном из крупных банков Сингапура. В отличие от других гибких методологий разработки программного обеспечения, таких как экстремальное программирование, методология *FDD* обладает большим формализмом, но при этом позволяет полнее контролировать процесс разработки.

Проект по методологии *FDD* состоит из двух основных фаз:

- получение списка всех функциональных возможностей, которыми должен обладать продукт;
- последовательная реализация функциональных возможностей.

Получение списка всех функциональных возможностей является одним из наиболее важных процессов в методологии *FDD*. От качества выполнения данного процесса во многом зависит успех всего проекта. Функциональные возможности разрабатываемой системы описываются на языке, который понятен как разработчиками, так и заказчикам. Список функциональных возможностей позволяет заказчикам сформировать достаточно полное представление о будущих характеристиках разрабатываемого продукта. Также список функциональных возможностей является источником гранулярных заданий для разработки, благодаря чему появляется возможность использовать итеративный процесс разработки.

Во время каждой итерации реализуется некоторый (небольшой) набор взаимосвязанных функциональных возможностей. Итерация обычно длится от одной до трех недель. Результатом каждой итерации является работающая версия продукта, которая может быть передана заказчику.

Методология *FDD* обладает следующими преимуществами:

- возможность управления проектами до 500 человек;
- возможность управления проектами по разработке критически важных систем;
- возможность использования менее квалифицированных разработчиков;
- возможность внедрения в организациях, которые формально требуют использование каскадных моделей жизненного цикла проекта.

6.4.1. Роли

Методология *FDD* предполагает наличие шести основных ролей:

- менеджер проекта;
- главный архитектор;
- менеджер разработки;
- ведущий программист;
- владелец класса;
- эксперт предметной области.

Каждый участник проекта может выполнять одну или несколько ролей. Например, ведущий программист может быть также владельцем класса.

Менеджер проекта

Менеджер проекта является административным лидером проекта и отвечает за управление бюджетом проекта и распределение ресурсов. На плечах *менеджера проекта* также лежит представление отчетов о состоянии проекта руководству организации.

Главный архитектор

Главный архитектор отвечает за общий дизайн системы. Во время коллективных сессий, посвященных проектированию системы, *главный архитектор* выступает в ролях эксперта и посредника. Также *главному*

архитектору принадлежит исключительное право принятия решений в спорных ситуациях.

Менеджер разработки

Менеджер разработки отвечает за управление процессами разработки. Обязанностью менеджера разработки является обеспечение эффективного использования всех ресурсов проекта. В частности, *менеджер разработки* выступает в роли арбитра в организационных спорах между главными программистами (например, споры о разделение ресурсов проекта между функциональными командами).

Ведущие программисты

Ведущими программистами назначаются опытные разработчики, которые хотя бы однажды прошли через все этапы жизненного цикла проекта. *Ведущие программисты* участвуют в процессах высокоуровневого анализа требований и проектирования. *Ведущие программисты* также возглавляют небольшие *функциональные команды* (от трёх до шести человек) и руководят локальными процессами анализа требований, проектирования и реализации. *Ведущие программисты* взаимодействуют друг с другом для решения повседневных локальных технических и организационных проблем.

Владельцы классов

Владельцем класса является разработчик, который отвечает за проектирование и реализацию некоторого конкретного класса. В методологии *FDD* не используется распространенная практика коллективного владения кодом. Индивидуальное владение классами имеет следующие преимущества:

- благодаря полностью самостоятельной разработке части системы, разработчики часто испытывают чувство гордости, что является дополнительным мотивирующим фактором;
- дизайн и реализация класса получаются более целостными, так как они спроектированы и запрограммированы одним человеком.

При реализации функциональных возможностей, которые затрагивают несколько классов, соответствующие *владельцы классов* объединяются в *функциональные команды* и работают под началом ведущего программиста.

Эксперт предметной области

Эксперты должны обладать глубоким знанием *предметной области*. *Эксперты* играют роль основных источников подробной информации обо всех аспектах предметной области системы. Эксперты предметной области постоянно взаимодействуют с другими участниками проекта и активно участвуют в таких процессах как анализ требований, проектирование и тестирование разрабатываемого продукта.

6.4.2. Структура проекта

Структура проекта в методологии FDD состоит из пяти основных процессов:

- разработка общей модели;
- построение списка функциональных возможностей;
- планирование реализации функциональных возможностей;
- проектирование функциональности;
- реализация функциональности.

В результате последовательного выполнения трех первых процессов разрабатывается *общая модель* продукта. Последние два процесса выполняются итеративно для каждой небольшой группы взаимосвязанных функциональных возможностей. Каждый процесс предполагает решение некоторых основных задач, которые будут описаны ниже. Общая структура проекта в методологии FDD приведена на рис. 17.

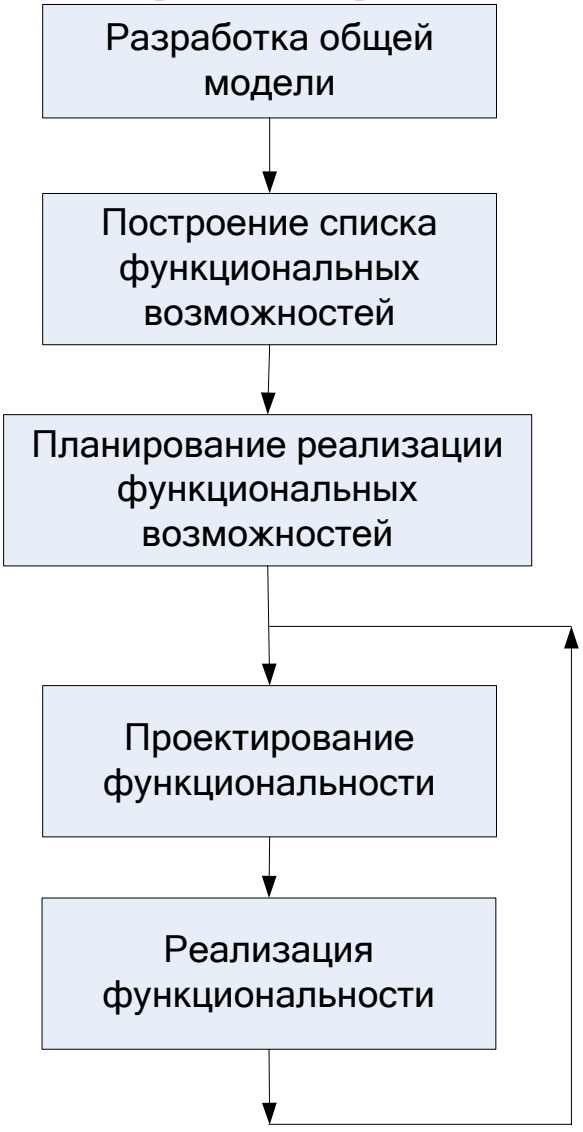

Рис. 17. Структура проекта в методологии FDD

6.4.3. Разработка общей модели

Проект в методологии FDD начинается с процесса *разработки общей модели* (*Develop Overall Model*). Результатом процесса *разработки общей модели* является общая модель предметной области и разрабатываемой системы.

Процесс *разработки общей модели* включает в себя следующие задачи:

- формирование команды моделирования;
- анализ предметной области;
- изучение проектной документации;
- разработка модели;
- детализация модели;
- описание модели.

Критерии начала процесса

Найдены эксперты предметной области. Назначены *главный архитектор* и *ведущие программисты*.

Формирование команды моделирования

Команда моделирование включает в себя постоянных членов, таких как эксперты предметной области и ведущие программисты. Другие члены команды разработчиков периодически включаются в команду моделирования, чтобы у них был шанс поучаствовать в процессе моделирования.

Анализ предметной области

Эксперты проводит обзор предметной области, в которой предстоит осуществлять моделирование. Во время анализа могут обсуждаться любые аспекты предметной области, в том числе и не имеющие прямого отношения к разрабатываемому продукту.

Изучение проектной документации

Команда моделирования изучает имеющиеся проектные документы, такие как список требований, модели данных, руководства пользователей и т.п.

Разработка модели

Формируются небольшие группы (не более трёх человек). Каждая из этих групп разрабатывает собственный вариант модели некоторой предметной области. Затем представитель каждой группы презентует разработанную в данной группе модель всей команде моделирования. Окончательная модель либо выбирается из предложенных группами моделей, либо конструируется на основе нескольких предложенных группами моделей.

Детализация модели

Общая модель расширяется по результатам итеративного выполнения предыдущей задачи.

Описание модели

Описываются основные аспекты разработанной общей модели. Также описываются основные варианты альтернативных моделей, которые могут пригодиться в течение проекта.

Верификация

Внутренняя верификация результатов процесса разработки общей модели осуществляется благодаря активному участию экспертов предметной области. Внешняя верификация результатов процесса осуществляется благодаря согласованию результатов моделирования с заказчиком.

Критерии завершения процесса

Для завершения процесса команда моделирования должна предоставить детальное описание разработанной общей модели. Результаты процесса должны быть согласованны с менеджером проекта и главным архитектором.

6.4.4. Построение списка функциональных возможностей

Результатом данного процесса является *построение списка функциональных возможностей* (*Build Feature List*), которые должна предоставлять разрабатываемая система. Процесс *построения списка функциональности* включает в себя следующие задачи:

- формирование команды для разработки списка функциональности;
- разработка списка функциональности.

Критерии начала процесса

Успешно завершен предыдущий процесс *разработки общей модели*.

Формирование команды для разработки списка функциональных возможностей

В команду включаются главные программисты, состоявшие в команде моделирования.

Разработка списка функциональности

Вся функциональность системы разбивается на предметные области, которые получаются путем декомпозиции общей предметной области. Для каждой предметной области системы описываются все вероятные варианты использования. Затем для каждого варианта использования составляется список функциональных возможностей, которыми должна обладать система. Каждая функциональная возможность записывается в виде предложения, имеющего смысл с точки зрения конечного пользователя системы. Например, «Проверить пароль пользователя». Реализация каждой

функциональной возможности должна занимать не более двух недель. В противном случае данная функциональная возможность должна быть разделена на части.

Верификация

Внутренняя верификация осуществляется благодаря тому, что все члены команды по разработке списка функциональных возможностей принимали активное участие в процессе разработки общей модели. Внешняя верификация результатов процесса осуществляется через согласование списка функциональных возможностей с экспертами предметной области и заказчиком.

Критерии завершения процесса

Результатом процесса является список функциональных возможностей, который содержит:

- список предметных областей продукта;
- список вариантов использования для каждой предметной области;
- список функциональных возможностей, необходимых для поддержки каждого из вариантов использования.

6.4.5. Планирование реализации функциональности

Результатом процесса *планирования реализации функциональности* (*Plan By Feature*) является создание общего плана разработки продукта. Общий план разработки продукта определяет порядок, в котором будут реализовываться функциональные возможности. Данный план учитывает зависимости между функциональными возможностями, сложность их реализации и загрузку команды.

Процесс планирования реализации функциональности включает в себя следующие задачи:

- формирование команды планирования;
- определения порядка разработки;
- назначение ответственности ведущих программистов за реализацию конкретных вариантов использования;
- назначение владельцев классов.

Критерии начала процесса

Успешно завершен предыдущий процесс *построения списка функциональных возможностей*.

Формирование команды планирования

В команду включаются менеджер по разработке и главные программисты.

Определения порядка разработки

Команда планирования назначает, с точностью до месяца, дату завершения реализации каждого варианта использования. Идентификация вариантов использования и дат завершения их реализации основаны на следующих основных факторах:

- учитываются зависимости между функциональными возможностями на уровне участвующих классов;
- осуществляется балансировка нагрузки между владельцами участвующих классов;
- учитывается сложность реализации тех или иных функциональных возможностей;
- функциональные возможности, реализация которых связана со значительным риском, имеют приоритет перед другими возможностями;
- учитываются разнообразные внешние факторы, такие как выпуск промежуточных версий, проведение публичных презентаций и т.п.

Назначение ответственности главных программистов за реализацию конкретных вариантов использования

Команда планирования должна назначить ответственность главных программистов за реализацию конкретных вариантов использования. Распределение ответственности основывается на следующих основных факторах:

- учитывается порядок разработки;
- зависимостями между функциональными возможностями на уровне участвующих классов;
- осуществляется балансировка нагрузки между владельцами участвующих классов (главные программисты тоже являются владельцами классов);
- учитывается сложность реализации тех или иных функциональных возможностей.

Назначение владельцев классов

Команда планирования должна назначить ответственность разработчиков за реализацию конкретных классов. Каждый разработчик может отвечать за разработку нескольких классов. Распределение ответственности основывается на следующих основных факторах:

- учитывается балансировка нагрузки между разработчиками;
- учитывается сложность реализации тех или иных классов;
- учитывается порядок разработки.

Верификация

Верификация результатов планирования осуществляется непосредственно в процессе планирования при активном участии ведущих программистов, менеджера разработки и менеджера проекта.

Критерии завершения процесса

Результатом процесса является общий план разработки, который содержит:

- варианты использования и даты завершения их реализации;
- ответственность главных программистов за реализацию конкретных вариантов использования;
- ответственность разработчиков за реализацию конкретных классов.

6.4.6. Проектирование функциональности

Целью процесса *проектирования функциональности* (*Design By Feature*) является разработка архитектуры, необходимой для реализации небольшой группы взаимосвязанных функциональных возможностей. Ответственность за реализацию функциональных возможностей распределена между главными программистами. Главные программист выбирает одну или несколько взаимосвязанных функциональных возможностей для реализации в течение данной итерации.

Критерии начала процесса

Успешно завершен предыдущий процесс *планирования*.

Формирование функциональной команды

Ведущий программист идентифицирует классы, которые связаны с реализацией выбранного набора функциональных возможностей. Владельцы данных классов включаются в команду по разработке данной функциональной возможности.

Анализ предметной области

Эксперт проводит обзор предметной области, с которой связан выбранный набор функциональных возможностей. Выполнение анализа предметной области является необязательной задачей и зависит от сложности реализации выбранных функциональных возможностей, сложности общей предметной области и т.п.

Изучение проектной документации

Члены функциональной команды изучают всю доступную проектную документацию относительно выбранных функциональных возможностей.

Моделирование взаимодействия

Члены функциональной команды моделируют, в том или ином виде, взаимодействие объектов, вовлеченных в реализацию разрабатываемых функциональных возможностей. В модели учитываются все взаимодействующие объекты и последовательность сообщений, которыми они обмениваются.

Детализация общей модели

Ведущий программист уточняет общую модель, добавляя или изменяя классы и методы, опираясь на результаты моделирования функциональных возможностей.

Написание заготовки для реализации

Опираясь на уточненную общую модель, каждый разработчик делает изменения в интерфейсах принадлежащих ему классов (не затрагивая реализацию классов). Затем ведущий программист формирует API для разрабатываемых функциональных возможностей.

Верификация

Верификация дизайна разрабатываемых функциональных возможностей осуществляется членами функциональной команды с возможным привлечением других участников проекта.

Критерии завершения процесса

Результатом процесса проектирования функциональности является полностью разработанный и верифицированный проект реализации набора взаимосвязанных функциональных возможностей, который содержит:

- пояснительную записку, которая обобщает и описывает дизайн, необходимый для реализации новых функциональных возможностей;
- все необходимые вспомогательные документы (включая результаты моделирования);
- описание основных альтернативных вариантов дизайна;
- обновленную общую модель;
- календарный план реализации набора новых функциональных возможностей.

6.4.7. Реализация функциональности

Целью процесса *реализации функциональности* (*Build By Feature*) является полная реализация набора взаимосвязанных функциональных возможностей, ценных с точки зрения пользователя. Разработчики вносят изменения в классы, владельцами которых они являются. Для вновь написанного кода разрабатываются модульные тесты. Также проводится инспекция всех изменений внесенных в код системы.

Критерии начала процесса

Для разрабатываемой функциональной возможности успешно завершен процесс проектирования.

Разработка

Разработчики вносят все необходимые изменения в классы, владельцами которых они являются.

Инспекция кода

Проводится полная инспекция всего внесенного кода с участием членов функциональной команды и, возможно, других участников проекта.

Разработка модульных тестов

Разработчики проектируют и реализуют модульные тесты для классов, владельцами которых они являются. По решению главного программиста, могут быть также разработаны функциональные тесты, которые специфицируют основные варианты использования новых функциональных возможностей.

Интеграция

После исчерпывающего модульного и функционального тестирования и успешной инспекции кода новые функциональные возможности интегрируется в разрабатываемый продукт.

Верификация

Верификация осуществляется благодаря исчерпывающему модульному и функциональному тестированию и инспекциям кода.

Критерии завершения процесса

Результатом процесса реализации набора взаимосвязанных функциональных возможностей является следующее:

- реализован набор взаимосвязанных функциональных возможностей, ценных с точки зрения пользователя;
- весь новый код протестирован и проинспектирован;
- новые функциональные возможности интегрированы в систему.

Список литературы

Основная литература по модулю

1. Бек К. Экстремальное программирование. – Спб.: Питер, 2002.
2. Мартин Р., Ньюкирк Д., Косс Р. Быстрая разработка программ. Принципы, примеры, практика. – М.: Издательский дом “Вильямс”, 2004.
3. Ауер К., Миллер Р. Экстремальное программирование: постановка процесса. С первых шагов и до победного конца. – Спб.: Питер, 2004.
4. Бек К. Экстремальное программирование: разработка через тестирование. – Спб.: Питер, 2003.
5. Бек К. Фаулер М. Экстремальное программирование: планирование. – Спб.: Питер, 2003.
6. Астелс Д., Миллер Г., Новак М. Практическое руководство по экстремальному программированию. – М.: Издательский дом “Вильямс”, 2002.

Дополнительная литература

7. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2001.
8. Рефакторинг: улучшение существующего кода / М. Фаулер, К. Бек, Д. Брант и др. – СПб.: Символ-Плюс, 2004.
9. Амблер С. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки. – Спб.: Питер, 2004.
10. Демарко Т., Листер Т. Человеческий фактор: успешные проекты и команды. – М.: Символ-Плюс, 2005.
11. Макконнелл С. Совершенный код. – СПб.: Питер, 2007.
12. Гецци К., Мехди Дж., Мандриоли Д. Основы инженерии программного обеспечения. 2-е издание. – Санкт-Петербург: БХВ-Петербург, 2005.
13. Крачтен Ф. Введение в Rational Unified Process. 2-е издание. – М.: Издательский дом "Вильямс", 2002.
14. Паук М., Куртис Б., Хриссис М. Б. Модель зрелости процессов разработки программного обеспечения. – М.: Интерфейс-Пресс, 2002.