Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Макаренко Елена Николаевна
Должность: Ректор
Дата подписания: 09.04.2021 18:46:49
Уникальный программный ключ:
c098bc0c1041cb2a4cf926cf171d6715d99a6ae00adc8e27b55cbe1e2dbd7c78

Министерство образования и науки Российской Федерации
Ростовский государственный экономический университет (РИНХ)

А.И. Долженко, С.А. Глушенко

Программная инженерия

Учебное пособие

Ростов-на-Дону
2017

УДК 004.4(075)
ББК 32.97
Д 64

Долженко, А.И.

Д 64 Программная инженерия : учебное пособие / А.И. Долженко, С.А. Глушенко. – Ростов н/Д : Издательско-полиграфический комплекс РГЭУ (РИНХ), 2017. – 128 с.
ISBN 978-5-7972-2388-7

Учебное пособие посвящено вопросам теории и инструментальной поддержке программной инженерии. Рассматривается общий подход к программной инженерии, модели жизненного цикла и методологии создания программного обеспечения. Детально анализируется методология компании Microsoft – Microsoft Solution Framwork, используемая в проектах создания программных решений. Приводится описание гибкой методологии разработки программного обеспечения Agile и Scrum. Анализируются вопросы управления требованиями к программному обеспечению, конфигурационного управления и обеспечения качества программных продуктов. Приводятся сведения по архитектуре и функциональным возможностям программного инструментария Microsoft Team Foundation Server и Visual Studio для управления жизненным циклом программных приложений.

Предназначено для бакалавров и магистрантов, обучающихся по направлениям 09.03.01 «Программная инженерия», 09.03.02 «Информационные системы и технологии», 09.03.03 «Прикладная информатика», 080500 «Бизнес-информатика».

УДК 004.4(075)
ББК 32.97

Рецензенты:
Е.Н. Ефимов, д.э.н., профессор кафедры информационных технологий и защиты информации РГЭУ (РИНХ);
С.М. Щербаков, д.э.н., профессор кафедры информационных систем и прикладной информатики РГЭУ (РИНХ)

Утверждено в качестве учебного пособия редакционно-издательским советом Ростовского государственного экономического университета (РИНХ).

ISBN 978-5-7972-2388-7

© РГЭУ (РИНХ), 2017
© Долженко А.И., Глушенко С.А., 2017

ОГЛАВЛЕНИЕ

Введение 5
1. Введение в программную инженерию 7
Назначение программной инженерии 7
Программное обеспечение 9
Процесс создания программного обеспечения 9
Классические модели процесса 11
Ключевые термины 16
Краткие итоги 16
Вопросы для самопроверки 17
2. Рабочий продукт, дисциплина обязательств, проект 18
Рабочий продукт 18
Дисциплина обязательств 20
Проект 22
Ключевые термины 25
Краткие итоги 25
Вопросы для самопроверки 26
3. Управление жизненным циклом приложений 27
Архитектурное проектирование 28
Разработка приложения 30
Тестирование приложения 34
Ключевые термины 36
Краткие итоги 37
Вопросы для самопроверки 37
4. Методология MSF 38
Основные принципы 39
Модель жизненного цикла 41
Фаза выработки концепции 42
Фаза планирования 43
Фаза разработки 44
Фаза стабилизации 44
Фаза внедрения 45
Модель команды 45
Ключевые термины 49
Краткие итоги 50
Вопросы для самопроверки 50
5. Гибкие технологии разработки ПО 52
Принципы и значение гибкой разработки 52
Методология гибкой разработки SCRUM 57
Рабочие элементы 57
Организация команды 59

Жизненный цикл проекта ПО ... 60
Управление невыполненной работой ... 62
Ключевые термины ... 67
Краткие итоги ... 68
Вопросы для самопроверки ... 69
6. Управление требованиями ... 70
Требования к программному обеспечению ... 70
Процесс управления требованиями ... 70
Виды и свойства требований ... 72
Варианты формализации требований ... 74
Цикл работы с требованиями ... 77
Ключевые термины ... 81
Краткие итоги ... 82
Вопросы для самопроверки ... 82
7. Конфигурационное управление ... 84
Единицы конфигурационного управления ... 84
Управление версиями ... 86
Системы контроля версий ... 87
Основы Git ... 90
Управление сборками ... 94
Понятие baseline ... 95
Ключевые термины ... 96
Краткие итоги ... 96
Вопросы для самопроверки ... 97
8. Обеспечение качества программных продуктов ... 98
Управление качеством ... 98
Качество программного обеспечения ... 102
Тестирование ... 103
Устранение ошибок ... 108
Рефакторинг ... 110
Ключевые термины ... 112
Краткие итоги ... 113
Вопросы для самопроверки ... 113
9. Архитектура и функциональные возможности TFS ... 115
Развертывание Team Foundation Server ... 116
Шаблоны командных проектов ... 119
Ключевые термины ... 122
Краткие итоги ... 122
Вопросы для самопроверки ... 123
Заключение ... 124
Библиографический список ... 125

ВВЕДЕНИЕ

Программная инженерия (Software engineering) ориентирована на систематизацию и повышение эффективности процессов проектирования, тестирования и оценки качества программных систем. В создании программных систем принимают участие команды разработчиков, которые включают специалистов различных квалификаций (бизнес-аналитики, проектировщики, программисты, тестировщики). Для получения качественного программного продукта необходимо эффективно организовать процесс его создания, управлять процессами и командой разработчиков на основе существующих методологий и стандартов.

Рациональная организация процессов разработки программных систем описывается в стандартах (международных, государственных, корпоративных), которые часто называют методологиями разработки ПО. Методологии создания ПО обычно создаются ведущими производителями программных систем и их сообществами с учетом особенностей программных продуктов, а также сферы внедрения. Методологии описывают подходы к организации рациональной стратегии и возможному набору процессов создания ПО.

В настоящее время все большее распространение получают гибкие методологии разработки программного обеспечения, где основное внимание сосредоточено на создании качественного продукта. При этом акцент делается на организацию эффективного управления командой. Самоорганизация и целеустремленность команды разработчиков позволяют создавать высококачественные программные продукты в сжатые сроки.

Инструменты управления жизненным циклом программных систем во многом способствуют успешности программных проектов. Компания Microsoft предоставляет разработчикам гибкий инструментарий для управления жизненным циклом приложений – ALM Visual Studio и Team Foundation Server. Традиционные средства разработки программ в Visual Studio дополнены средствами архитектурного проектирования и тестирования. Инструментарий Team Foundation Server позволяет формировать и от-

слеживать требования к программной системе, связывать их с задачами и реализацией, распределять между членами команды, проводить построение программного продукта, управлять тестированием, проводить контроль версий, предоставлять средства коммуникации с членами команды и заказчиками, подготавливать многочисленные отчеты.

Современные студенты, будущие разработчики программного обеспечения, менеджеры программных проектов, тестировщики, верификаторы, контролеры качества должны владеть знаниями в области теории и прикладных методов проектирования. Кроме того, студенты должны знать методы управления командами разработчиков, методы планирования и оценивания качества выполняемых работ и организовывать выполнение проектов в заданные сроки в соответствии с бюджетом проекта.

Данное учебное пособие посвящено систематическому описанию накопленных знаний в области программной инженерии, отражает аспекты теории и практики создания программных систем командами разработчиков с использованием программного инструментария управления жизненным циклом приложений.

1. ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ

Назначение программной инженерии

Программная инженерия (ПИ) представляет собой область исследования, которая ориентирована на создание качественного *программного обеспечения* (ПО) [6, 16]. Методы и подходы программной инженерии применяются для формирования эффективных подходов к созданию программных продуктов, их сопровождению и развитию. Разработка современных *программных продуктов* (ППр) проводится, как правило, командой в соответствии с заданной целью и набором требований. Процесс создания ППр ограничен по времени и имеет определенный бюджет. При проведении работ по созданию ППр необходимо выполнять следующие этапы и виды деятельности [5]:

- анализ предметной области, для которой создается ППр;
- разработку требований к программному продукту на основе пожеланий (ожиданий) заказчика и других заинтересованных лиц;
- планирование этапов разработки ПО;
- проектирование ПО, включающее разработку архитектуры и функциональных моделей системы;
- создание прототипов ППр для апробации проектных решений и получения уверенности в возможности реализации ПО в соответствии с заданными требованиями на основе доступных технологических решений;
- разработку эффективного кода ПО в соответствии с заданными нормами, шаблонами и правилами;
- тестирование ПО на различных фазах разработки;
- конфигурационное управление программным продуктом на этапах разработки и сопровождения;
- проектный менеджмент процесса разработки;
- создание различной документации (проектной, пользовательской и пр.).

Современные подходы к организации процесса разработки ППр, в подавляющем большинстве случаев, используют итера-

тивно-инкрементальную модель [4, 6], при которой требуемая функциональность создается поэтапно, постепенно наращивая реализацию требований к программной системе. При этом менеджеры и все заинтересованные лица могут оценить степень соответствия разрабатываемого ППр заявленным требованиям и при необходимости внести изменения в процесс разработки.

Методы и подходы программной инженерии предполагают создание ПО с учетом обеспечения возможности эффективного сопровождения программного продукта на протяжении всего жизненного цикла ППр. Создаваемый код должен использовать доступные библиотеки и ранее разработанные и протестированные компоненты, а также предполагать разумную степень повторного использования и интеграции с другими системами.

Из всего вышеизложенного можно отметить, что *программная инженерия* как вид деятельности представляет собой систематический, измеряемый подход к разработке, функционированию и сопровождению программного обеспечения, а также специальную область знаний (научную дисциплину) по исследованию этих подходов [9, 16].

Следует отметить, что проекты создания ППр, как правило, не начинаются с нуля, а базируются на предыдущем положительном опыте, полученном данной или другими командами разработчиков ПО. Такой опыт разработок многократно проверяется, обобщается и специальным образом оформляется для повторного использования, например, программные решения публикуются в репозитории GitHub [27], который является хостингом для ИТ-проектов и их совместной разработки. В настоящее время существуют методы и лучшие практики для работы с требованиями, архитектурные шаблоны, шаблоны проектирования и тестирования.

Программная инженерия включает стандарты и всемирно признанные методологии процессов создания программного обеспечения (например, MSF [31], RUP [34], CMMI [22], Scrum [35]).

Программная инженерия как специальная область знаний возникла в конце 60-х годов прошлого века. В 1968 г. на конференции NATO Software Engineering в г. Гармиш (ФРГ) [36] обсуждались вопросы, связанные с улучшением процессов создания и развития программного обеспечения, то есть вопросы, которые

сейчас относятся к области знаний «программная инженерия». Программная инженерия охватывает тематику организации и улучшения процессов разработки программного обеспечения, управления проектными командами, разработки и внедрения программных средств поддержки жизненного цикла разработки ПО.

Программное обеспечение

Программная инженерия оперирует таким понятием, как *программное обеспечение* ЭВМ. Под программным обеспечением понимается совокупность программ системы обработки информации и эксплуатационной документации. ПО является сложной динамической системой и характеризуется следующими свойствами [15]:

– *сложностью* программных компонентов, которая зависит от алгоритмов обработки информации, размеров кода, количества пользователей, объема обрабатываемых данных, требований по быстродействию;
– *согласованностью,* которая предполагает взаимодействие большого числа компонентов через стандартизированные и нестандартизированные интерфейсы;
– *изменяемостью,* которая проявляется в частом изменении требований к ППр, как при разработке, так и при эксплуатации;
– *нематериальность* – в отличие от материальных объектов ПО легко поддается тиражированию, созданию различных версий и копий.

Процесс создания программного обеспечения

В программной инженерии важным элементом является *процесс* создания программного обеспечения, который включает различные виды деятельности, методы, методики и этапы, используемые для разработки и усовершенствования ПО и связанных с ним элементов (проектных планов, документации, программного кода, тестов, пользовательской документации).

Для разработки ПО используют различные процессы, которые отражают взгляды команд разработчиков, используемых программных средств управления жизненным циклом программ. Например, в системе Microsoft Visual Team System имеются шаб-

лоны процессов на базе стандартов CMMI [18, 22], а также шаблоны для гибкой разработки Agile и Scrum [18, 20, 35].

Команды разработчиков, как правило, используют корпоративные процессы разработки ПО, которые адаптируют стандартные процессы под создаваемые проектные решения. В корпоративных процессах описывают следующее [9]:

– информацию, правила использования, документацию и инсталляционные пакеты средств разработки, используемых в проектах компании (систем версионного контроля, средств контроля ошибок, средств программирования – различных IDE, СУБД и т.д.);
– описание лучших практик разработки – проектного управления, правил и процедур работы с заказчиком и т.д.;
– шаблонов проектных документов – результатов обследований и анализ предметной области, требований к проектируемой системе, технических заданий, спецификаций, тестовых планов и т.д. и пр.

Наличие тех или иных корпоративных процессов разработки и сопровождения программного обеспечения в организации отражает уровень её зрелости и возможности в плане разработки качественного программного продукта.

Для организаций, занимающихся разработкой программного обеспечения, важным является совершенствование процессов создания ПО. Для этого необходимо проводить изменения существующих процессов с целью повышения качества разрабатываемых ППр, улучшения потребительских характеристик (благожелательности и удобства пользовательского интерфейса), повышения надежности, реактивности и быстродействия, снижения совокупной стоимости владения программным продуктом.

Совершенствование процессов разработки ПО диктуется быстрыми темпами развития информационных технологий (программные и аппаратные платформы, специализированные библиотеки программ – фреймворки, развитие языков программирования, новые технологии работы с данными) и высокой конкуренции на рынке программных продуктов.

Для улучшения процессов разработки ПО целесообразно переходить на новые средства разработки ПО, технологии и язы-

ки программирования, проводить совершенствование процессов в программных проектах, сертифицировать организации на соответствие международным стандартам (CMM/CMMI, ISO 9000 [22, 30]).

Работы по улучшению процессов разработки ПО рекомендуют проводить непрерывно, малыми шагами, для того чтобы не нарушать непрерыв-ности процессов создания ППр в организации.

Классические модели процесса

Модель процесса разработки программного продукта определяет последовательность действий команды разработчиков при создании ППр, этапы разработки, роли членов команды, подходы к управлению командой и всем процессом в целом [5, 7, 15].

В каждой организации используют модели разработки ПО, которые в наибольшей степени соответствуют достижению максимальной эффективности программных проектов.

Существуют классические модели процессов разработки ПО, на которых строятся корпоративные модели процессов. В любой модели процессов выделяют фазы работ и виды деятельности.

Фаза процесса разработки представляет собой определенный этап процесса, характеризующийся началом, концом и результатом. На фазе процесса создания ПО выполняются работы по реализации некоторых требований к системе. Результаты выполнения работ на определенной фазе могут быть представлены заказчику для оценки реализованных требований и, возможно, оплаты за выполненную работу. Выделение фаз в процессе создания ПО позволяет оформить и оценить промежуточные результаты программного проекта, проводить мониторинг продвижения проекта. В качестве фаз можуг быть выделены такие работы, как: разработка технического задания, реализация определенных требований к программной системе, разработка кода для ряда функций, тестирование ПО, выпуск бета-версии.

Вид деятельности представляет собой определенный тип работы, выполняемый в процессе разработки ПО. Различные виды деятельности требуют разных компетенция и выполняются, как правило, разными специалистами. Так управление проектом

выполняет менеджер проекта, разработку кода программ проводит программист, а тестирование относится к компетенции тестировщика. В программном проекте некоторые виды деятельности могут совмещаться отдельными членами команды, например, проектирование и разработка кода. В то же время ряд видов деятельности не рекомендуют поручать одному и тому же члену команды. Например, виды деятельности тестирование и разработка кода крайне нежелательно совмещать (за исключением модульного тестирования, которое обязан делать программист при разработке кода)

На одной фазе могут выполняться различные виды деятельности. В то же время определенный вид деятельности может реализовываться на разных фазах. Так, тестирование необходимо проводить на фазах анализа предметной области, проектирования архитектуры и компонентов программной системы, кодирования модулей и при комплексной сдаче системы заказчику.

Для сложного программного обеспечения используются модели процесса создания ПО, в которых присутствует отделение фаз от видов деятельности, что позволяет облегчить управление процессом разработки ПО.

Одной из первых моделей процесса разработки ПО была *водопадная модель*, предложенная в 1970 г. Винстоном Ройсом. В этой модели были выделены различные шаги разработки программного обеспечения: разработка системных требований, разработка функциональных и нефункциональных требований к ПО, детальный анализ предметной области, проектирование архитектуры и компонентов системы, кодирование программных модулей, тестирование, использование, то есть эксплуатация программной системы (рис. 1.1).

В данной модели имеется ограничение возможности возвратов на предыдущие этапы работ. Допускается возврат только на один шаг назад, то есть с фазы «кодирование» можно вернуться только на фазу «проектирование».

В рамках водопадной модели было предусмотрено прототипирование. При разработке прототипа системы проверялись основные проектные решения: архитектура, функциональность, дизайн. Если прототип удовлетворял основным пожеланиям и тре-

бованиям заказчика и показывал возможность реализации программной системы, то переходили к разработке полного функционала системы.

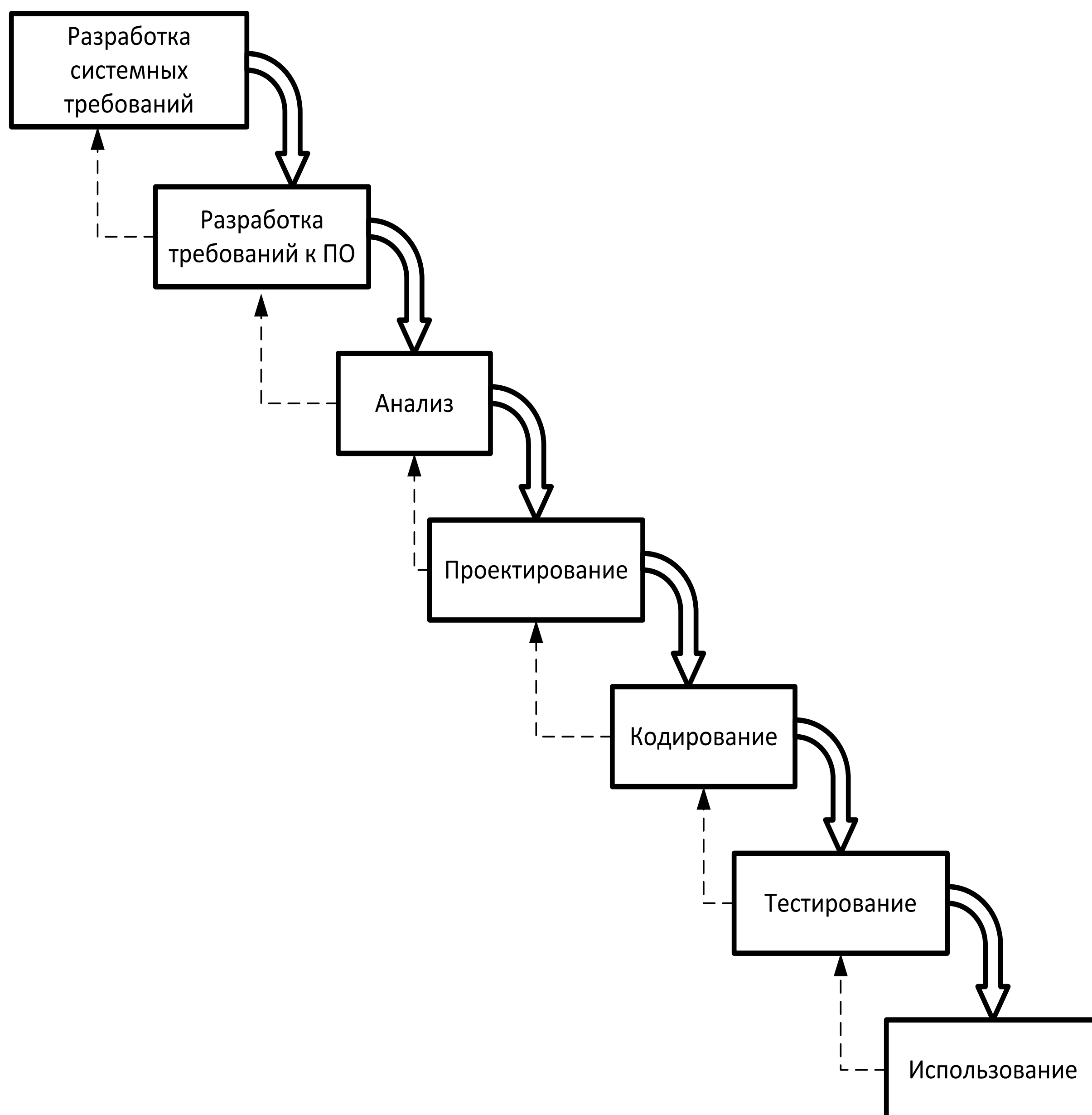

***Рисунок 1.1** – Водопадная модель жизненного цикла ПО*

В 70–80 гг. прошлого века водопадная модель жизненного цикла ПО активно использовалась в разработке ПО в силу своей простоты. В дальнейшем были выявлены ряд недостатков модели, которые препятствовали эффективной реализации программных проектов. Недостатками водопадной модели являются [5]:

– отождествление фаз и видов деятельности, что влечет потерю гибкости разработки;
– требование полного окончания работ на определенной фазе перед переходом на следующую фазу;
– трудности внесения изменений в ходе разработки программной сис-темы вследствие ограничений на возврат к предыдущим этапам работ;
– интеграция всех результатов разработки происходит в конце, вследствие чего интеграционные проблемы дают о себе знать слишком поздно;
– наличие рисков несоответствия ожиданий заказчика и предъявляемого разработчиками созданного программного обеспечения, так как разработка программной системы проводится на основании требований и спецификаций, которые не подлежат изменению в процессе создания программного продукта.

Следует отменить, что водопадная модель жизненного цикла программного обеспечения не потеряла своей актуальности и может эффективно применяться для небольших проектов или при разработке типовых систем. С ее помощью удобно отслеживать разработку и осуществлять поэтапный контроль за проектом. Водопадная модель вошла в качестве составной части в другие модели и методологии.

Спиральная модель была предложена Бэри Боемом в 1988 г. для преодоления недостатков водопадной модели, прежде всего, для лучшего управления рисками [6]. Согласно этой модели разработка продукта осуществляется по спирали, каждый виток которой является определенной фазой разработки (рис. 1.2).

В отличие от водопадной модели в спиральной нет предопределенного и обязательного набора витков, каждый виток может стать последним при разработке системы, при его завершении составляются планы следующего витка. Наконец, виток является именно фазой, а не видом деятельности, как в водопадной модели, в его рамках может осуществляться много различных видов деятельности, то есть модель является двумерной.

Рисунок 1.2 *– Спиральная модель жизненного цикла ПО*

Последовательность витков может быть такой: на первом витке принимается решение о целесообразности создания ПО, на следующем – определяются системные требования, потом осуществляется проектирование системы и т.д. Витки могут иметь и иные значения.

Каждый виток имеет следующую структуру (секторы):

– определение целей, ограничений и альтернатив проекта;
– оценка альтернатив, оценка и разрешение рисков; возможно использование прототипирования (в том числе создание серии прототипов), моделирование системы, визуальное моделирование и анализ спецификаций; фокусировка на самых рисковых частях проекта;
– разработка и тестирование – здесь возможна водопадная модель или использование иных моделей и методов разработки ПО;
– планирование следующих итераций – анализируются результаты, планы и ресурсы на последующую разработку, принимается (или не принимается) решение о новом витке; анализируется, имеет ли смысл продолжать разрабатывать систему или нет; разработку можно и приостановить, например, из-за сбоев в финансировании; спиральная модель позволяет сделать это корректно.

Отдельная спираль может соответствовать разработке некоторого программного компонента или внесению очередных изменений в продукт. Таким образом, у модели может появиться третье измерение.

Спиральную модель нецелесообразно применять в проектах с небольшой степенью риска, с ограниченным бюджетом для небольших проектов. Кроме того, отсутствие хороших средств прототипирования может также сделать неудобным использование спиральной модели.

Спиральная модель не нашла широкого применения в индустрии и важна, скорее, в историко-методологическом плане: она является первой итеративной моделью, имеет красивую метафору – спираль, – и, подобно водопадной модели, использовалась в дальнейшем при создании других моделей процесса и методологий разработки ПО.

Ключевые термины

Программная инженерия – область исследования, ориентированная на создание качественного программного обеспечения.

Программное обеспечение – совокупность программ системы обработки информации и эксплуатационной документации.

Процесс создания программного обеспечения – процесс, который включает различные виды деятельности, методы, методики и этапы, используемые для разработки и усовершенствования.

Фаза процесса разработки ПО представляет собой определенный этап процесса, характеризующийся началом, концом и результатом.

Вид деятельности представляет собой определенный тип работы, выполняемый в процессе разработки ПО.

Водопадная модель – модель разработки ПО, при которой этапы работ выполняются последовательно без возврата на предыдущие этапы.

Спиральная модель – модель, при которой разработка продукта осуществляется по спирали, каждый виток которой является определенной фазой разработки.

Краткие итоги

Программная инженерия представляет собой систематический, измеряемый подход к разработке, функционированию и со-

провождению программного обеспечения, а также специальную область знаний по исследованию этих подходов. Программная инженерия включает методы и подходы для формирования эффективных способов создания программных продуктов, их сопровождения и развития.

Программное обеспечение представляет собой совокупность программ системы обработки информации и эксплуатационной документации и характеризуется следующими свойствами: сложностью, согласованностью, изменяемостью и нематериальностью. Процесс создания программного обеспечения включает различные виды деятельности, методы, методики и этапы, используемые для разработки и усовершенствования ПО. Командная разработка является основой создания современных программных продуктов. Модель процесса разработки программного продукта определяет последовательность действий команды разработчиков при создании ППр, этапы разработки, роли членов команды, подходы к управлению командой и всем процессом в целом. Классическими моделями жизненного цикла программного обеспечения являются водопадная и спиральная модели.

Вопросы для самопроверки

1. Охарактеризуйте процесс разработки программного обеспечения.
2. В чем проявляется совершенствование процесса разработки программного обеспечения?
3. Для чего используется модель процесса разработки программного обеспечения?
4. Дайте определение фаз и видов деятельности в процессе разработки ПО.
5. Опишите водопадную модель процесса разработки ПО.
6. Опишите спиральную модель процесса разработки ПО.

2. РАБОЧИЙ ПРОДУКТ, ДИСЦИПЛИНА ОБЯЗАТЕЛЬСТВ, ПРОЕКТ

Рабочий продукт

Одним из существенных условий для управляемости процесса разработки ПО является наличие отдельно оформленных результатов работы – как в окончательной поставке, так и промежуточных. Эти отдельные результаты в составе общих результатов работ помогают идентифицировать, планировать и оценивать различные части результата. Промежуточные результаты помогают менеджерам разных уровней отслеживать процесс воплощения проекта, заказчик получает возможность ознакомиться с результатами задолго до окончания проекта. Более того, сами участники проекта в своей ежедневной работе получают простой и эффективный способ обмена рабочей информацией – обмен результатами.

Таким результатом является *рабочий продукт* – любой артефакт, произведенный в процессе разработки ПО, например, файл или набор файлов, документы, составные части продукта, сервисы, процессы, спецификации, счета и т.д. (рис. 2.1) [9].

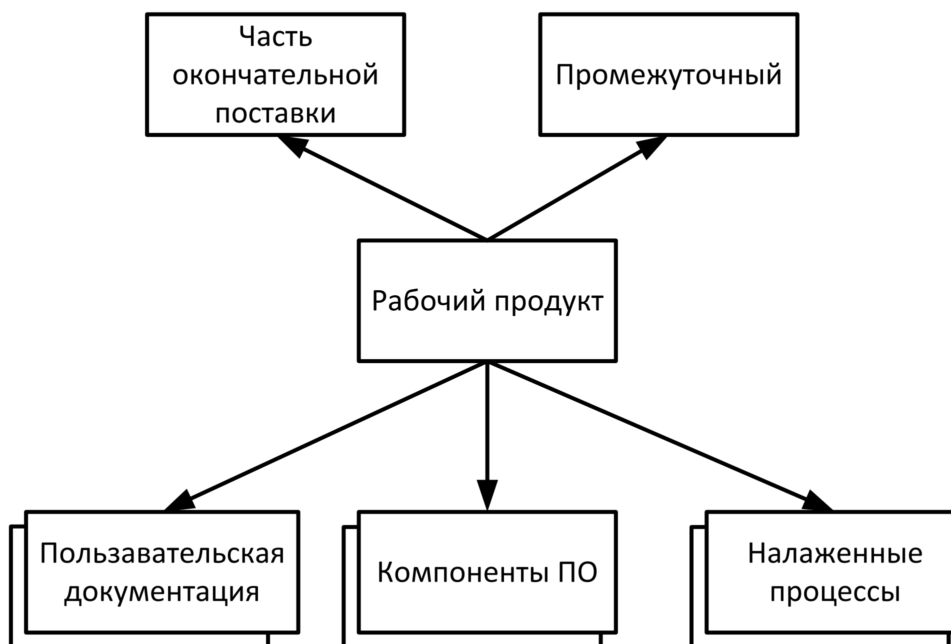

***Рисунок 2.1** – Состав рабочего продукта*

Ключевая разница между рабочим продуктом и компонентом ПО заключается в том, что первый необязательно материален и осязаем, хотя может быть таковым. Нематериальный рабочий продукт – это, как правило, некоторый налаженный процесс – промышленный процесс производства какой-либо продукции, учебный процесс в университете (на факультете, на кафедре) и т.д.

Важно отметить, что рабочий продукт совсем необязательно является составной частью итоговой поставки. Например, налаженный процесс тестирования системы не поставляется заказчику вместе с самой системой. Умение управлять проектами (не только в области программирования) во многом связано с искусством определять нужные рабочие продукты, настаивать на их создании и в их терминах вести приемку промежуточных этапов работы, организовывать синхронизацию различных рабочих групп и отдельных специалистов.

Многие методологии включают в себя описание специфичных рабочих продуктов, используемых в процессе – CMMI [22], MSF [31], RUP [34] и др. Например, в MSF это программный код, диаграммы приложений и классов, план итераций, модульный тест и др. Для каждого из них точно описано содержание, ответственные за разработку, место в процессе и др. аспекты.

Компонента ПО, созданная в проекте одним разработчиком и предоставленная для использования другому разработчику, оказывается рабочим продуктом. Ее надо минимально протестировать, поправить имена интерфейсных классов и методов, быть может, убрать лишнее, не имеющее отношение к функциональности данной компоненты, лучше разделить области видимости объектов (public и private), и т.д., то есть проделать некоторую дополнительную работу, которую, быть может, разработчик и не стал делать, если бы продолжал использовать компоненту только сам. Объем этих дополнительных работ существенно возрастает, если компонента должна быть представлена для использования в разработке, например, в другом центре разработки. Итак, изготовление хороших промежуточных рабочих продуктов очень важно для успешности проекта, но требует дополнительной работы от их авторов. Работать одному, не предоставляя рабочих продуктов – легче и для многих предпочтительнее. Но работа в ко-

манде требует накладных издержек, в том числе и в виде трат на создание промежуточных рабочих продуктов. Конечно, качество этих продуктов и трудозатраты на их изготовление сильно варьируются в зависимости от ситуации, но тут важно понимать сам принцип.

Промежуточный рабочий продукт должен обязательно иметь ясную цель и конкретных пользователей, чтобы минимизировать накладные расходы на его создание (рис. 2.2).

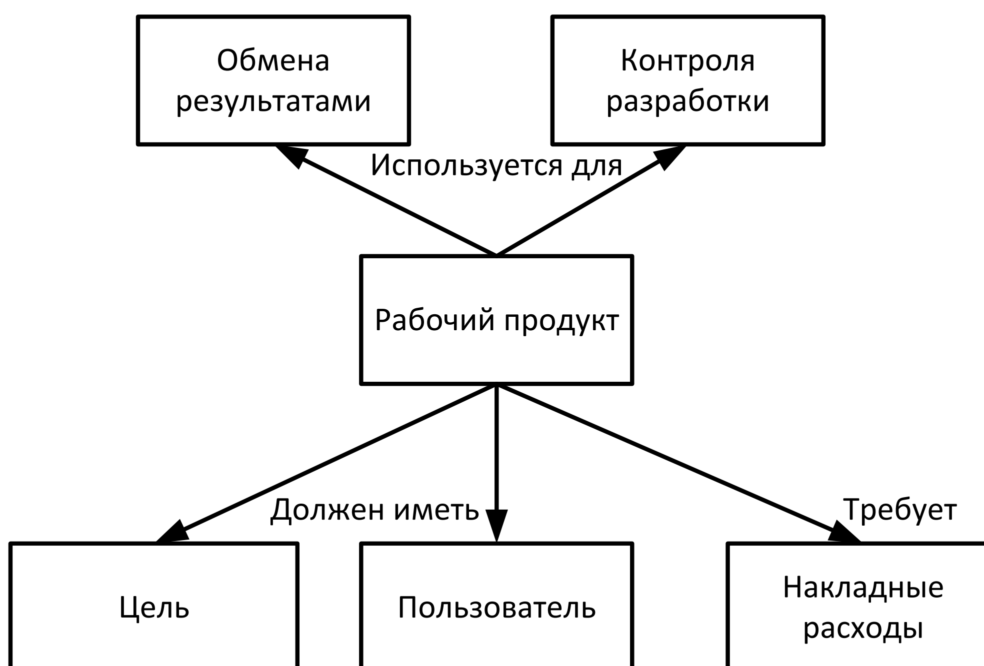

***Рисунок 2.2** – Рабочий продукт*

Дисциплина обязательств

В основе разделения обязанностей в бизнесе и промышленном производстве лежит определенная деловая этика, форма отношений – *дисциплина обязательств*. Она широко используется на практике и является одной из возможных форм социального взаимоотношения между людьми. При внесении в бизнес и промышленность иных моделей человеческих отношений – семейных, сексуальных, дружеских и т.д. часто наносит делам серьезный урон, порождает конфликтность, понижает эффективность.

Основой этой формы отношений являются обязательства, которые:

– даются добровольно;

– не даются легко – работа, ресурсы, расписание должны быть тщательно учтены;
– между сторонами включает в себя то, *что* будет сделано, *кем* и в какие *сроки*;
– открыто и публично сформулированы (то есть это не «тайное знание»).

Кроме того:

– ответственная сторона стремится выполнить обязательства, даже если нужна помощь;
– до наступления срока выполнения обязательств, как только становится очевидно, что работа не может быть закончена в срок, обсуждаются новые обязательства.

Отметим, что дисциплина обязательств не является каким-то сводом правил, законов, она отличается также от корпоративной культуры. Это – определенный групповой психический феномен, существующий в обществе современных людей. Приведенные выше пункты не являются исчерпывающим описанием этого феномена, но лишь проявляют и обозначают его, так сказать, вызывают нужные воспоминания.

Дисциплина обязательств, несмотря на очевидность, порой, не просто реализуется на практике, например, в творческих областях человеческой деятельности, в области обучения и т.д. Существуют отдельные люди, которым эта дисциплина внутренне чужда вне зависимости от их рода деятельности.

С другой стороны, люди, освоившие эту дисциплину, часто стремятся применять ее в других областях жизни и человеческих отношений, что оказывается не всегда оправданным. Подчеркнем, что данная дисциплина является далеко не единственной моделью отношений между людьми. В качестве примера можно рассмотреть отношения в семье или дружбу, что, с очевидностью, не могут быть выражены дисциплиной обязательств. Так, вместо точности и пунктуальности в этих отношениях важно эмоционально-чувственное сопереживание, без которого они невозможны.

Дисциплине обязательств уделяется много внимания в рамках MSF, поскольку там в модели команды нет лидера, начальника. Эта дисциплина реализована также в Scrum: Scrum-команда

имеет много свобод, и в силу этого – большую ответственность. Регламентируются также правила действий, когда обязательства не могут быть выполнены такой командой.

Проект

Проект – это уникальная деятельность, имеющая начало и конец во времени, направленная на достижение определённого результата/цели, создание определённого, уникального продукта или услуги, при заданных ограничениях по ресурсам и срокам, а также требованиям к качеству и допустимому уровню риска.

В частности, разработка программного обеспечения является преимущественно проектной областью.

Необходимо различать проекты промышленные и проекты творческие. У них разные принципы управления. Сложность промышленных проектов – в большом количестве разных организаций, компаний и относительной уникальности самих работ. Пример – строительство многоэтажного дома. Сюда же относятся различные международные проекты и не только промышленные – образовательные, культурные и пр. Задача в управлении такими проектами – это все охватить, все проконтролировать, ничего не забыть, все свести воедино, добиться движения, причем движения согласованного.

Творческие проекты характеризуются абсолютной новизной идеи – новый сервис, абсолютно новый программный продукт, какого еще не было на рынке, проекты в области искусства и науки. Любой начинающий бизнес, как правило, является таким вот творческим проектом. Причем новизна подобных проектах не только абсолютная – такого еще не было. Такое, может, уже и было, но только не с нами, командой проекта. То есть присутствует огромный объем относительной новизны для самих людей, которые воплощают этот проект.

Проекты по разработке программного обеспечения находятся между двумя этими полюсами, занимая в данном пространстве различное положение. Часто они сложны потому, что объемны и находятся на стыке различных дисциплин – того целевого бизнеса, куда должен встроиться программный продукт, и сложного, нетривиального программирования. Часто сюда добавляется еще

разработка уникального электронно-механического оборудования. С другой стороны, поскольку программирование активно продвигается в разные сферы человеческой деятельности, то происходит это путем создания абсолютно новых, уникальных продуктов, и их разработка и продвижение обладают всеми чертами творческих проектов.

Управление проектами – область деятельности, в ходе которой, в рамках определенных проектов, определяются и достигаются четкие цели при нахождении компромисса между объемом работ, ресурсами (такими, как: время, деньги, труд, материалы, энергия, пространство и др.), временем, качеством и рисками.

Отметим несколько важных аспектов управления проектами.

- *Заинтересованные стороны* – это люди/стороны, которые не участвуют непосредственно в проекте, но влияют на него и или заинтересованы в его результатах. Это могут быть будущие пользователи системы (например, в ситуации, когда они и заказчик – это не одно и то же), высшее руководство компании-разработчика и т.д. Идентификация всех stakeholders и грамотная работа с ними – важная составляющая успешного проектного менеджмента.
- *Границы проекта*. Это очень важное понятие для программных проектов ввиду изменчивости требований. Часто бывает, что разработчики начинают создавать одну систему, а после постепенно она превращается в другую. Причем для менеджеров по продажам, а также заказчика ничего радикально не произошло, а с точки зрения внутреннего устройства ПО, технологий, алгоритмов реализации, архитектуры – все радикально меняется. За подобными тенденциями должен следить и грамотно с ними разбираться проектный менеджмент.
- *Компромиссы* – важнейший аспект управления программными проектами в силу согласовываемости ПО. Важно не потерять все согласуемые параметры и стороны и найти приемлемый компромисс.

При разработке программных проектов, следуя MSF 3.1 [31], важны следующие области управления.

Таблица 2.1 *– Области управления*

Область управления проектами	Описание
Планирование и мониторинг проекта, контроль за изменениями в проекте	Интеграция и синхронизация планов проекта; организация процедур и систем управления и мониторинга проектных изменений
Управление рамками проекта	Определение и распределение объема работы (рамок проекта); управление компромиссными решениями в проекте
Управление календарным графиком проекта	Составление календарного графика исходя из оценок трудозатрат, упорядочение задач, соотнесение доступных ресурсов с задачами, применение статистических методов, поддержка календарного графика
Управление стоимостью	Оценки стоимости исходя из оценок временных затрат; отчетность о ходе проекта и его анализ; анализ затратных рисков; функционально-стоимостной анализ
Управление персоналом	Планирование ресурсов; формирование проектной команды; разрешение конфликтов; планирование и управление подготовкой
Управление коммуникацией	Коммуникационное планирование (между проектной группой, заказчиком/спонсором, потребителями/пользователями, др. заинтересованными лицами); отчетность о ходе проекта
Управление рисками	Организация процесса управления рисками в команде и содействие ему; обеспечение документооборота управления рисками
Управление снабжением	Анализ цен поставщиков услуг и/или аппаратного/программного обеспечения; выбор поставщиков и субподрядчиков; составление контрактов; договора; заказы на поставку и платежные требования
Управление качеством	Планирование качества, определение применяемых стандартов, документирование критериев качества и процессов его измерения

Ключевые термины

Рабочий продукт – любой артефакт, произведенный в процессе разработки ПО.

Дисциплина обязательств – форма отношений, определяющая разделение обязанностей в бизнесе и промышленном производстве.

Проект – уникальная деятельность, имеющая начало и конец во времени, направленная на достижение определённого результата/цели при заданных ограничениях по ресурсам и срокам.

Управление проектами – область деятельности, в ходе которой, в рамках определенных проектов, определяются и достигаются четкие цели при нахождении компромисса между объемом работ, ресурсами, временем, качеством и рисками.

Заинтересованные стороны – это люди/стороны, которые не участвуют непосредственно в проекте, но влияют на него и или заинтересованы в его результатах.

Краткие итоги

При управлении разработкой ПО рабочий продукт представляет любой артефакт, произведенный в процессе разработки ПО. Это может быть файл или набор файлов, документы, составные части продукта, сервисы, процессы, спецификации, счета. Рабочий продукт может характеризовать промежуточные этапы разработки или часть окончательной поставки. Дисциплина обязательств в проектной команде определяет принятые при разработке процедуры распределения обязанностей и ответственности за выполнением работ. Разработка ПО является проектной деятельностью, а проект представляет собой уникальную деятельность, имеющую начало и конец во времени, направленную на достижение определённого результата/цели, создание определённого, уникального продукта или услуги, при заданных ограничениях по ресурсам и срокам, а также требованиям к качеству и допустимому уровню риска. При создании программных проектов необходимо учитывать мнение всех заинтересованных лиц (stakeholders), а не только заказчиков и разработчиков.

Вопросы для самопроверки

1. Понятие рабочего продукта процесса разработки программного обеспечения.
2. Поясните разницу между рабочим продуктом и компонентой ПО.
3. Промежуточные рабочие продукты процесса разработки программного обеспечения.
4. Дисциплина обязательств в процессе разработки программного обеспечения. Основные формы отношений при проектировании ПО.
5. Проект разработки программного обеспечения. Определение, характеристики.
6. Управление программными проектами.
7. Поясните содержание понятия *заинтересованные лица*.
8. Поясните содержание понятия *границы проекта*.
9. Области управления программными проектами.

3. УПРАВЛЕНИЕ ЖИЗНЕННЫМ ЦИКЛОМ ПРИЛОЖЕНИЙ

Управление жизненным циклом приложений (application lifecycle management – ALM) – это концепция управления программным проектом на всех этапах его жизни [3, 5, 6, 23].

Решения для управления жизненным циклом приложений, ориентированы на поддержку проектов по разработке ПО, координацию членов команды разработчиков, управление процессами создания ПО такими как планирование, управление изменениями, определение и управление требованиями, архитектурой, конфигурациями ПО, автоматизацией сборок и развертывание, управление качеством. Помимо основных возможностей решения для ALM включают в себя отслеживание связей между элементами жизненного цикла, определение процесса разработки ПО, составление различных отчетов.

Важными преимуществами решений для ALM являются возможности координировать членов команды разработчиков, процессы разработки, информацию и инструментарий, задействованные в проекте. Поскольку не существует универсального решения, подходящего всем, целесообразно сосредоточиться на следующих принципах при внедрении управления жизненным циклом:

- планировании в реальном времени;
- трассировке в жизненном цикле для связанных артефактов;
- обеспечении возможности для взаимодействия заинтересованных в проекте лиц;
- применении бизнес-аналитики для разработки;
- непрерывном улучшении процесса разработки.

Для реализации концепции управления жизненным циклом приложений компания Microsoft предлагает решение на основе Visual Studio и Team Foundation Server (TFS) [5, 23, 37]. Технологии ALM в Visual Studio позволяют разработчикам контролировать жизненный цикл создания ПО, сокращая время разработки, устраняя издержки и внедряя непрерывный цикл реализации бизнес-ценностей.

Управление жизненным циклом приложения в Visual Studio базируется на следующих принципах:

- продуктивности (productivity);
- интеграции (integration);
- расширяемости (extensibility).

Продуктивность обеспечивается возможностью совместной работы и управлением сложностью продукта. Все элементы проекта (требования, задачи, тестовые случаи, ошибки, исходный код и построения) и отчеты централизованно управляются через TFS. Инструменты визуального моделирования архитектуры, возможности управления качеством кода, инструменты тестирования позволяют управлять сложностью продукта.

Интеграция обеспечивается возможностями Visual Studio по предоставлении всем участникам проекта информации о состоянии дел, что упрощает коммуникацию между членами команды и обеспечивает прозрачность хода процесса проектирования.

Расширяемость обеспечивается API-интерфейсом служб TFS и интегрированной средой разработки (integrated development environment – IDE). API-интерфейс служб TFS позволяет создавать собственные инструменты и расширять существующие, а IDE – конечным пользователям и сторонним разработчикам добавлять инструменты с дополнительными функциями.

При создании программного продукта необходимо вначале спроектировать архитектуру, что возлагается на архитектора программного продукта. На основе архитектуры осуществляется разработка, что является предназначением разработчика программного продукта. Созданный продукт необходимо тестировать на его соответствие требованиям заказчика, что осуществляет тестировщик. Visual Studio и TFS обеспечивают совместную командную работу архитектора, разработчика и тестировщика, предоставляя им необходимый инструментарий и функциональные возможности для выполнения требуемых работ.

Архитектурное проектирование

В Visual Studio для архитектурного проектирования используются инструменты визуального проектирования на основе языка UML, которые предназначены для следующего:

- визуализации архитектурных аспектов проектируемой системы;

– создания моделей структуры и поведения системы;
– разработки шаблонов для проектирования системы;
– документирования принятых решений.

Диаграммы UML позволяют визуально описывать приложение, наглядно представлять архитектуру и документировать требования к приложению.

Архитектурные инструменты в Visual Studio Ultimate позволяют создавать шесть видов схем и документ ориентированных графов:

– схему классов UML;
– схему последовательностей UML;
– схему вариантов использования UML;
– схему активности UML;
– схему компонентов UML;
– схему слоев.

Схемы (диаграммы) классов UML описывают объекты в прикладной системе. Диаграммы классов отражают иерархию внутри приложения или системы и связи между ними.

Схемы (диаграммы) последовательностей UML показывают взаимодействие между различными объектами. Они используются для демонстрации взаимодействия между классами, компонентами, подсистемами или субъектами.

Схемы (диаграммы) вариантов использования UML определяют функциональность системы и описывают с точки зрения пользователей их возможные действия с программным продуктом. Данные диаграммы определяют связи между функциональными требованиями, пользователями и основными компонентами системы.

Схемы (диаграммы) активности UML описывают бизнес-процесс или программный процесс в виде потока работ через последовательные действия. Диаграммы активности используются для моделирования логики в конкретном варианте использования или для моделирования подробностей бизнес-логики.

Схемы (диаграммы) компонентов UML описывают распределение программных составляющих приложения, позволяя наглядно отобразить на высоком уровне структуру компонентов

и служб. С помощью этих схем можно визуализировать компоненты и другие системы, показывая связи между ними. В качестве компонентов могут выступать исполнительные модули, DLL-библиотеки и другие системы.

Схемы (диаграммы) слоев используются для описания логической архитектуры системы. Диаграммы слоев могут использоваться для проверки того, что разработанный код отвечает высокоуровневому проекту на схеме слоев. Диаграммы позволяют проверять архитектуру приложения на соответствие базе кода.

Документ ориентированных графов позволяет создать граф зависимостей, отображающий отношения между компонентами архитектурных артефактов. Графы зависимостей обеспечивают визуальные способы проверки кода, анализа зависимостей между файлами.

Разработка приложения

Основным средством разработки в Visual Studio является интегрированная среда разработки (IDE). IDE-среда интегрирована со средствами модульного тестирования и обеспечивает возможности выявления неэффективного, небезопасного или плохо написанного кода, управление изменениями и модульное тестирование как кода, так и базы данных.

Важнй инструмент разработчика программного обеспечения – модульное тестирование, которое реализуется в среде Unit Test Framework. Назначением модульных тестов является проверка того, что код работает правильно с точки зрения программиста. Модульные тесты формируются на более низком уровне, чем другие виды тестирования, и проверяют, работают ли лежащие в их основе функции так, как ожидается. Для модульного тестирования используется метод прозрачного ящика, для которого требуется знание внутренних структур кода.

Модульные тесты помогают обнаружить проблемы проектирования и реализации. Кроме того, модульный тест является хорошей документацией по использованию проектируемой системы. Хотя модульное тестирование требует дополнительного программирования, но его применение окупается за счет сокращения затрат на отладку приложения.

Модульные тесты являются важным элементом регрессионного тестирования. Регрессионное тестирование представляет собой повторное тестирование части программы после внесения в неё изменений или дополнений. Цель регрессионного тестирования – выявление ошибок, которые могут появиться при внесении изменений в программу

В Visual Studio имеется *функция «Анализ покрытия кода»*, которая проводит мониторинг того, какие строки кода исполнялись в ходе модульного тестирования. Результатом анализа покрытия кода является выявление областей кода, которые не покрыты тестами.

Важный аспект создания качественного программного продукта – соблюдение разработчиками правил и стандартов организации в написания кода. В Visual Studio имеются *функции анализа кода*, которые позволяют проанализировать код, найти типичные ошибки, нарушения стандартов и предложить меры по устранению ошибок и нарушений. Наборы правил анализа кода поставляются с Visual Studio. Разработчики могут настроить свои проекты на определенный набор правил, а также добавить свои специфичные правила анализа кода.

В процессе анализа кода используются метрики кода, которые дают количественные оценки различных характеристик кода. Метрики позволяют определить сложность кода и его изолированные области, которые могут привести к проблемам при сопровождении приложения. В Visual Studio используются следующие метрики кода:

- сложность организации циклов – определяет число разных путей кода;
- глубина наследования – определяет число уровней в иерархии наследования объектов;
- объединение классов – определяет число классов, на которые есть ссылки;
- строки кода – определяет количество строк кода в исполняемом методе;
- индекс удобства поддержки – оценивает простоту обслуживания кода.

Для анализа производительности и эффективности использования ресурсов приложением в Visual Studio имеются инструменты профилирования. *Профилирование* представляет собой процесс наблюдения и записи показателей о поведении приложения. Инструментарий профилирования (профилировщики) позволяет обнаружить у приложения проблемы с производительностью. Такие проблемы, как правило, связаны с кодом, который выполняется медленно, неэффективно или чрезмерно использует системную память. Профилирование обычно используется для выявления участков кода, которые в ходе выполнения приложения выполняются часто или долгое время.

Профилировщики бывают с выборкой и инструментированием. Профилировщики с выборкой делают периодические снимки выполняющегося приложения и записывают его состояние. Профилировщики с инструментированием добавляют маркеры отслеживания в начало и конец каждой исследуемой функции. В процессе работы профилировщика маркеры активизируются, когда поток исполнения программы входит в исследуемые функции и выходит из них. Профилировщик записывает данные о приложении и о том, какие маркеры были затронуты в ходе исполнения приложения. В Visual Studio поддерживается профилирование с выборкой и с инструментированием. Для анализа производительности необходимо:

- создать новый сеанс производительности;
- с помощью Обозревателя производительности задать свойства сеансы;
- запустить сеанс, выполняя приложение и профилировщик;
- проанализировать данные в отчетах по производительности.

Большинство корпоративных приложений работает с базами данных, что определяет необходимость разработки и тестирования приложений совместно с базами данных командой проекта. В Visual Studio имеется инструментарий создания баз данных и развертывания изменений в них. Для этого используется автономная разработка схем баз данных, которая позволяет вносить изменения в схемы без подключения к производственной базе данных. После внесения изменений в среду разработки Visual Studio позволяет протестировать их в самой среде разработки и/или выде-

ленной среде тестирования. Кроме того, Visual Studio позволяет сгенерировать псевдореальные данные для проведения тестов. При положительных результатах тестирования Visual Studio позволяет сгенерировать сценарии для обновления производственной базы данных. Цикл разработки базы данных приложения состоит из следующих шагов:

- перевода схемы базы данных в автономный режим;
- итеративной разработки приложения с базой данных;
- тестирования схемы базы данных;
- построения и развертывания базы данных и приложения.

Для совершенствования процесса отладки приложений в Visual Studio имеется функция интеллектуального отслеживания работы программы *IntelliTrace*. Функция *IntelliTrace* конфигурируется с помощью следующих разделов:

- Общие (General);
- Дополнительно (Advanced);
- События IntelliTrace (IntelliTrace Events);
- Модули (Modules).

Раздел *Общие* позволяет включить и отключить функцию *IntelliTrace* и задать запись только событий или дополнительной информации, включающей события, данные диагностики, вызовы и отслеживание на уровне методов. В разделе *Дополнительно* задается расположение для генерируемого файла журнала и его максимальный размер. В разделе *События IntelliTrace* перечисляются все события диагностики, которые будут собираться в ходе отладки приложения. Раздел *Модули* определяет список модулей, для которых необходимо собирать данные в процессе отладки приложения. При записи событий, происходящих в приложении, происходит их перехват при работе приложения и информация о событиях фиксируется в журнале. При отладке с *IntelliTrace* можно приостановить интерактивный сеанс отладки и просмотреть события или вызовы. Имеется возможность остановки выполнения приложения и пошаговое движение назад и вперед в интерактивном сеансе отладки, а также воспроизведение записанного сеанса отладки.

Тестирование приложения

Тестирование приложения является необходимым этапом управления жизненным циклом приложения [5, 11, 13, 16]. Тестирование выполняет разработчик в процессе создания кода приложения, а также тестировщик при проверке качества разрабатываемого программного продукта.

Visual Studio Ultimate предоставляет разработчику инструмент для создания и использования модульных тестов, нагрузочных тестов и веб-тестов производительности, а также тестов пользовательского интерфейса.

Модульные тесты представляют собой низкоуровневые программные тесты, написанные на языках Visual C#, Visual Basic или Visual C++. Они позволяют быстро проверить наличие логических ошибок в методах классов. Основной их целью является проверка того, что код приложения работает так, как ожидает разработчик.

Нагрузочные тесты используются для исследования работоспособности приложения путем моделирования множества пользователей, которые работают с программой одновременно. Система позволяет использовать большое количество виртуальных пользователей на локальных и удаленных компьютерах при выполнении нагрузочного теста. В Visual Studio Ultimate имеется три встроенных шаблона нагрузки: постоянный, пошаговый и с учетом эталона. Цели нагрузочного теста определяют выбор шаблона нагрузки.

Тесты пользовательского интерфейса позволяют автоматически сформировать код теста путем записи действий пользователя при работе с приложением и впоследствии выполнять эти тесты автоматически.

Архитектура средств тестирования в Visual Studio приведена на рис. 3.1.

Центральное место занимает платформа модульного тестирования. Доступ к средствам тестирования может осуществляться из обозревателя тестов, командной строки и при построении приложения через TeamBuild. Платформа тестирования обеспечивает подключение плагинов тестирования. В составе Visual Studio установлены плагины MS-Test Managed и MS-Test Native. Плагины сторонних разработчиков (NUnit, xUnit.net, MbUnit и другие)

можно подключить к платформе юнит-тестирования. Средства модульного тестирования ориентированы на использование разработчиками в процессе написания кода.

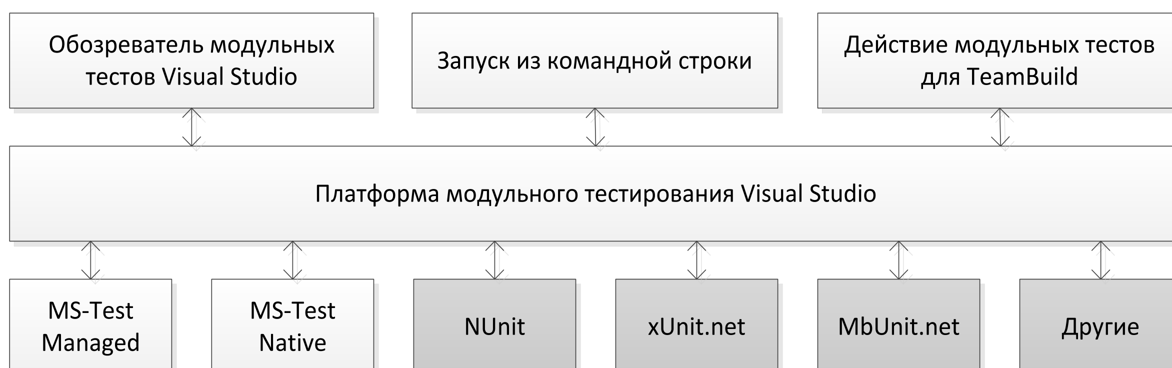

Рисунок 3.1 *– Архитектура инструментов тестирования*

Для тестировщиков в Visual Studio имеется специализированный инструмент – Microsoft Test Manager, который позволяет создавать планы тестирования, формировать, добавлять и удалять тестовые случаи, определять и управлять физическими и виртуальными тестовыми средами, выполнять ручные и автоматические тесты.

Создание эффективной среды тестирования сложных приложений выполняется с помощью лаборатории тестирования – Lab Management, которая интегрирована с Team Foundation Server и представляет собой диспетчер виртуальной среды. Лаборатория тестирования предоставляет следующие возможности:

- создание, управление и освобождение сред, состоящих из одной или более виртуальных машин;
- автоматическое развертывание построений в виртуальных средах;
- выполнение ручных и автоматических тестов в виртуальных средах;
- использование снимков, позволяющих быстро вернуть среды к заданному состоянию;
- сетевую изоляцию виртуальных сред.

Администрирование виртуальной среды Lab Management производит диспетчер Microsoft System Center Virtual Machine Manager (SCVMM), с помощью которого производят необходимые настройки виртуальной тестовой лаборатории.

В Visual Studio и Team Foundation Server добавлены средства взаимодействия с пользователями разработанных программных продуктов, которое по запросу позволяет сформировать отзыв пользователя в базе данных для последующей обработки командой проекта.

Ключевые термины

Управление жизненным циклом приложений – концепция управления программным проектом на всех этапах его жизни.

Схемы классов UML – описывают объекты в прикладной системе.

Схемы последовательностей UML – показывают взаимодействие между различными объектами.

Схемы вариантов использования UML – определяют функциональность системы и описывают с точки зрения пользователей их возможные действия с программным продуктом.

Схемы активности UML – описывают бизнес-процесс или программный процесс в виде потока работ через последовательные действия

Схемы компонентов UML – описывают распределение программных составляющих приложения, позволяя наглядно отобразить на высоком уровне структуру компонентов и служб.

Схемы слоев – используются для описания логической архитектуры системы.

IDE – интегрированная среда разработки Visual Studio.

Модульные тесты – тесты, используемые для проверки правильности работы методов и классов.

Функция анализа покрытия кода – программа, которая проводит мониторинг того, какие строки кода исполнялись в ходе модульного тестирования.

Функция анализа кода – программа, которая позволяет проанализировать код, найти типичные ошибки, нарушения стандартов и предложить меры по устранению ошибок и нарушений.

Профилирование – процесс наблюдения и записи показателей о поведении приложения.

Метрика кода – количественный показатель, который оценивает различные характеристики кода

IntelliTrace – функция интеллектуального отслеживания работы программы

Краткие итоги

Visual Studio и Team Foundation Server представляют решение компании Microsoft по управление жизненным циклом приложений. Управление жизненным циклом приложения в Visual Studio базируется на принципах продуктивности, интеграции и расширяемости. В Visual Studio для архитектурного проектирования используются инструменты визуального моделирования на основе языка UML. Основным средством разработки в Visual Studio является интегрированная среда разработки IDE, которая интегрирована со средствами модульного тестирования. Visual Studio Ultimate представляет разработчику инструмент для создания и использования модульных тестов, нагрузочных тестов и веб-тестов производительности, а также тестов пользовательского интерфейса. Инструментами тестировщика при создании приложений является специализированный инструмент – Microsoft Test Manager. Создание эффективной среды тестирования сложных приложений выполняется с помощью лаборатории тестирования – Lab Management.

Вопросы для самопроверки

1. Что определяет понятие «управление жизненным циклом приложений»?
2. Назовите принципы управления жизненным циклом приложения в Visual Studio.
3. Для чего предназначены инструменты визуального проектирования в Visual Studio?
4. Какие виды схем архитектурного проектирования можно подготовить в Visual Studio?
5. Для чего предназначена функция «Анализ покрытия кода» в Visual Studio?
6. Поясните назначение профилировщика в Visual Studio.
7. Поясните назначение программы IntelliTrace.
8. Какие метрики кода используются в Visual Studio?
9. Какое назначение нагрузочных тестов в Visual Studio?
10. Какие возможности предоставляет лаборатория тестирования – Lab Management?

Упражнения

1. Проведите анализ подходов архитектурного проектирования ПО.
2. Проведите анализ подходов в формировании требований к ПО.
3. Проведите анализ средств разработки ПО.

4. МЕТОДОЛОГИЯ MSF

В 1990-х годах компания Microsoft, стремясь достичь максимальной отдачи от реализации заказных IT-решений и в целях улучшения работы с субподрядчиками, обобщила свой опыт по разработке, внедрению, сопровождению и консалтингу ПО, создав методологию MSF. В 2002 г. вышла версия MSF 3.1, состоявшая из пяти документов-руководств [5, 31]:

– модели процессов,
– модели команды,
– модели управления проектами,
– дисциплины управления рисками,
– управления подготовкой.

IT-решение понимается как скоординированная поставка набора элементов (таких, как: программные средства, документация, обучение и сопровождение), необходимых для удовлетворения бизнес-потребности конкретного заказчика. Причем под его разработкой понималось создание ПО, обучение пользователей и полная передача продукта команде сопровождения. Задача наладки полноценного сопровождения IT-решения – важная составляющая успешности проекта.

Основными новшествами MSF являются следующие:

1. Акцент на внедрении IT-решения.
2. Модель процесса, объединяющая спиральную и водопадную модели.
3. Особая организация команды – не иерархическая, а как группа равных, но выполняющих разные функции (роли) работников.
4. Техника управления компромиссами.

Ниже мы рассмотрим эти положения более детально.

В 2005 г. MSF претерпела значительные изменения. Версия MSF 4.0. стала составной частью продукта Visual Studio Team System (VSTS) и разделилась на две ветки – MSF for Agile и MSF for CMMI [18, 19, 20]. При этом, если версии до 3.x были именно методологиями (там были изложены принципы, MSF свободно распространялась в виде Word-документов, которые были также

переведены на русский язык), то теперь MSF превратилась в шаблоны процесса для VSTS. Эти шаблоны имеют описание в виде html-документов (Word-документов уже нет) и определяют типы ролей, их ответственность, действия в рамках этой ответственности, а также все входные и выходные артефакты этих деятельностей и другие формализованные атрибуты процесса разработки. Кроме этого «человеческого» описания MSF for Agile и MSF for CMMI имеют XML-настройки, которые позволяют в точности следовать предложенным выше описаниям, используя VSTS. При этом на процесс накладываются достаточно жесткие ограничения, деятельность разработчиков сопровождается набором автоматических действий – все это задано в шаблонах. Данные шаблоны можно частично использовать (например, без некоторых ролей), а также изменять (VSTS представляет обширные средства настойки шаблонов). Версия MSF 4.2 продолжила направление версии MSF 4.0.

Можно считать, что фактически версии MSF 4.x являются продуктами другого класса, чем MSF 3.x. MSF 3.x были нацелены на разработку заказных IT-решений, MSF 4.0 – на разработку произвольного ПО. Формально документация этих версий несильно пересекается и содержит для 3.x в большей степени общие принципы, а для 4.x – формальные атрибуты в терминах VSTS. В некотором смысле можно сказать, что MSF 4.x является реализацией MSF 3.x для продукта VSTS.

Основные принципы

Перечислим основные принципы MSF.

1. *Единое видение проекта.* Успех коллективной работы над проектом немыслим без наличия у членов проектной группы и заказчика единого видения (shared vision), т.е. четкого и, самое главное, одинакового понимания целей и задач проекта. Как проектная группа, так и заказчик изначально имеют собственные предположения о том, что должно быть достигнуто в ходе работы над проектом. Лишь наличие единого видения способно внести ясность и обеспечить движение всех заинтересованных в проекте сторон к общей цели. Формирование единого видения и последующее следование

ему являются столь важными, что модель процессов MSF выделяет для этой цели специальную фазу – «Выработка концепции», которая заканчивается соответствующей вехой.

2. *Гибкость – готовность к переменам.* Традиционная дисциплина управления проектами и каскадная модель исходят из того, что все требования могут быть четко сформулированы в начале работы над проектом, и далее они не будут существенно изменяться. В противоположность этому MSF основывается на принципе непрерывной изменяемости условий проекта при неизменной эффективности управленческой деятельности.
3. *Концентрация на бизнес-приоритетах.* Независимо от того, нацелен ли разрабатываемый продукт на организации или индивидуумов, он должен удовлетворить определенные нужды потребителей и принести в некоторой форме выгоду или отдачу. В отношении индивидуумов это может означать, например, эмоциональное удовлетворение – как в случае компьютерных игр. Что же касается организаций, то неизменным целевым фактором продукта является бизнес-отдача (business value). Обычно продукт не может приносить отдачу до того, как он полностью внедрен. Поэтому модель процессов MSF включает в свой жизненный цикл не только разработку продукта, но и его внедрение.
4. *Поощрение свободного общения.* Исторически многие организации строили свою деятельность на основе сведения информированности сотрудников к минимуму, необходимому для исполнения работы (need-to-know). Зачастую такой подход приводит к недоразумениям и снижает шансы команды на достижение успеха. Модель процессов MSF предполагает открытый и честный обмен информацией как внутри команды, так и с ключевыми заинтересованными лицами. Свободный обмен информацией не только сокращает риск возникновения недоразумений, недопонимания и неоправданных затрат, но и обеспечивает максимальный вклад всех участников проектной группы в снижение существующей в проекте неопределенности. По этой причине модель процессов MSF предлагает проведение анализа хода работы над

проектом в определенных временных точках. Документирование результатов делает ясным прогресс, достигнутый в работе над проектом - как для проектной команды, так и для заказчика и других заинтересованных в проекте сторон.

Модель жизненного цикла

Модель жизненного цикла сочетает в себе свойства двух стандартных моделей: каскадной и спиральной (рис. 4.1).

В основе методологии MSF лежит *итеративный интегрированный подход* к созданию и внедрению решений, базирующийся на *фазах* и *вехах*.

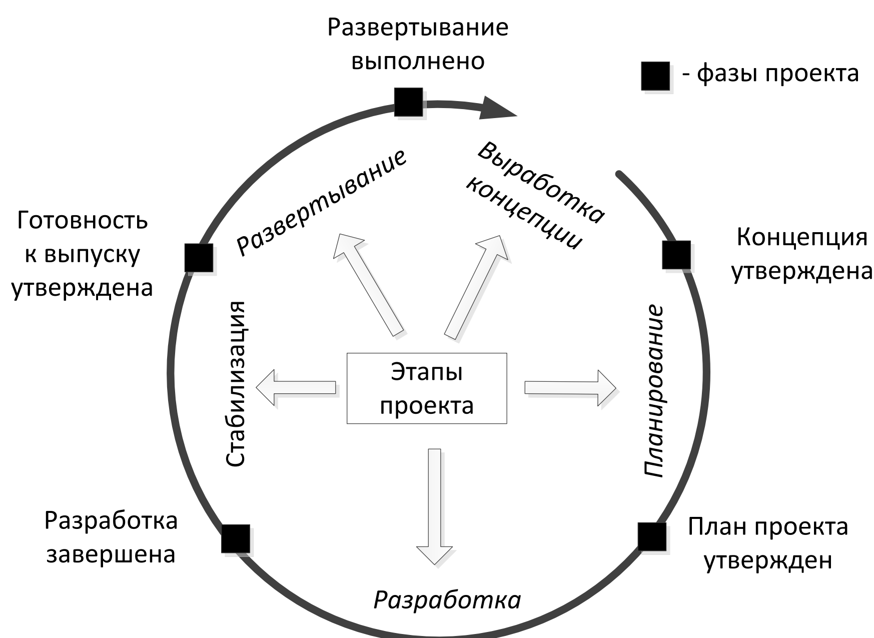

Рисунок 4.1 *– Модель жизненного цикла MSF*

Итеративность подхода предусматривает поэтапное создание всех элементов проекта: программного кода, документации, дизайна, планов. Реализацию проекта рекомендуется начинать с построения, тестирования и внедрения базовой функциональности системы. Затем к решению добавляются все новые и новые возможности. Такой подход к процессу разработки подразумевает достаточную гибкость в ведении документации. Проектные документы должны изменяться по мере эволюции проекта. Их пересмотр не прекращается до конца проекта и производится по-

сле каждой итерации. Такой подход существенно отличается от принципов ведения документации в каскадной модели, где процесс разработки начинается лишь после того, как готовы и зафиксированы все требования и спецификации.

Интеграция в рамках одного проекта процедур разработки и внедрения системы позволяет более полно сосредоточиться на нуждах Заказчика (даже если разработка решения прошла удачно, заказчики не увидят отдачи до тех пор, пока система не запущена в эксплуатацию), и улучшить взаимодействие с командой сопровождения.

Фазы проекта определяют последовательно решаемые задачи, а вехи – ключевые точки проекта, характеризующие достижение какого-либо существенного результата.

В MSF используются два вида вех: главные и промежуточные. Они имеют следующие характеристики:

- *главные вехи* служат точками перехода от одной фазы к другой и определяют изменения в текущих задачах ролевых кластеров проектной команды; в MSF главные вехи являются в достаточной степени универсальными для применения в любом ИТ проекте;
- *промежуточные вехи* показывают достижение определенного прогресса в исполнении фазы проекта и расчленяют большие сегменты работы на меньшие, обозримые и управляемые участки; промежуточные вехи могут варьироваться в зависимости от характера проекта.

Фаза выработки концепции

Цель фазы – создание и сплочение проектной группы на основе выработки единого видения проекта.

Основные выполняемые задачи:

- создание ядра проектной группы;
- подготовка документа общего описания (Видение) и рамок проекта (vision / scope document). Видение (vision) – это ничем не ограничиваемое представление о том, каким должно быть решение. Рамки (scope) – определение того, что из предложенного этим видением будет реализовано в условиях существующих проектных ограничений;

– определение и оценка главных рисков проекта;
– выявление и первичный анализ бизнес-требований (детально эти требования рассматриваются во время фазы планирования).

Результаты выполнения фазы фиксируются в ряде документов:
– общее описание и рамки проекта;
– документ оценки рисков;
– описание структуры проекта.

Фаза планирования

Цель фазы – разработка планов проекта.

Основные выполняемые задачи:

1. Подготовка функциональной спецификации на систему включает в себя анализ и документирование проектных требований (выделяются: бизнес-требования, потребительские требования, эксплуатационные требования и системные требования, относящиеся к решению в целом). Задача предусматривает последовательное выполнение следующих работ:
– выявления типов пользователей системы;
– выявления сценариев использования, в которых моделируется выполнение какой-либо операции определенным типом пользователя;
– выделения последовательностей специфических действий, называемых примерами пользования (use cases), которые необходимо выполнить пользователю для осуществления операции;
– проектирования (дизайн системы). В MSF выделяются три уровня процесса проектирования: концептуальный дизайн (conceptual design), логический дизайн (logical design) и физический дизайн (physical design).

Концептуальный дизайн – описание всего, что нужно включить в конечный продукт. В это описание не входит информация о способе реализации решения. Концептуальный дизайн включает только подробные сведения о функциональности предлагаемого решения, взаимодействии с существующей технологической инфраструктурой, о пользовательском интерфейсе и предполагаемых рабочих характеристиках системы.

Логический дизайн – описание состава, организации и взаимодействия элементов, из которых состоит программное решение.

Физический дизайн – описание программного решения в терминах разработчика системы. Включает все необходимые детали для реализации: технологии, организацию, структуру и взаимосвязи элементов, которые будут использованы при создании программного решения.

Результаты процесса проектирования документируются в функциональной спецификации.

2. Подготовка рабочих планов.

На основе разработанных спецификаций каждый из руководителей ролевых кластеров проектной группы подготавливает планы, относящиеся к его роли (план внедрения, план тестирования, план эксплуатации, план мер безопасности, план обучения и пр.), и принимает участие в командных сессиях планирования, где все планы синхронизируются и представляются вместе в виде сводного плана проекта.

3. Оценка проектных затрат и сроков разработки различных составляющих проекта.

Результаты фазы оформляются в *базовой версии проекта* путем создания следующих документов:

- функциональной спецификации;
- плана управления рисками;
- сводного плана и сводного календарного графика проекта.

Фаза разработки

Цель фазы – создание компонент-решения (включая как документацию, так и программный код).

Результаты фазы:

- исходный и исполнимый код приложений;
- скрипты установки и конфигурирования;
- окончательная функциональная спецификация;
- материалы поддержки решения;
- спецификации и сценарии тестов.

Фаза стабилизации

Цель фазы – тестирование и отладка разработанного решения в реалистичной модели производственной среды.

Основные выполняемые задачи:
- выявление, приоритезация и устранение ошибок;
- пилотное внедрение решения.

Результаты:
- окончательный продукт;
- документация выпуска;
- материалы поддержки решения;
- результаты и инструментарий тестирования;
- исходный и исполнимый код приложений;
- проектная документация.

Фаза внедрения

Цель фазы – установка и отладка системы в реальных условиях эксплуатации, передача системы персоналу поддержки и сопровождения, получение окончательного одобрения результатов проекта со стороны Заказчика.

Результатами этой фазы являются:
- информационные системы эксплуатации и поддержки;
- работающие процедуры и процессы;
- базы знаний, отчеты, журналы протоколов;
- версии проектных документов, массивы данных и программный код, разработанные во время проекта;
- отчет о завершении проекта;
- окончательные версии всех проектных документов;
- показатели удовлетворенности Заказчика и потребителей.

Модель команды

Основные принципы. Главной особенностью модели команды в MSF является то, что она «плоская», то есть не имеет официального лидера. Все отвечают за проект в равной степени, уровень заинтересованности каждого в результате очень высок, а коммуникации внутри группы четкие, ясные, дружественные и ответственные. Конечно, далеко не каждая команда способна так работать – собственно, начальники для того и нужны, чтобы нести основной груз ответственности за проект и во многом освободить от него других. Демократия в команде возможна при высоком уровне осознанности и заинтересованности каждого, а

также в ситуации равности в профессиональном уровне (пусть и в разных областях – см. различные ролевые кластеры в команде, о которых речь пойдет ниже). С другой стороны, в реальном проекте, в рамках данной модели команды, можно варьировать степень ответственности, в том числе вплоть до выделения, при необходимости, лидера.

Одной из особенностей отношений внутри команды является высокая культура дисциплины обязательств:

– готовность работников принимать на себя обязательства перед другими;
– четкое определение тех обязательств, которые они на себя берут;
– стремление прилагать должные усилия к выполнению своих обязательств;
– готовность честно и незамедлительно информировать об угрозах выполнению своих обязательств.

Ролевые кластеры. MSF основан на постулате о семи качественных целях, достижение которых определяет успешность проекта. Эти цели обусловливают модель проектной группы и образуют *ролевые кластеры* (или просто *роли*) в проекте. В каждом ролевом кластере может присутствовать по одному или нескольку специалистов, некоторые роли можно соединять одному участнику проекта. Каждый ролевой кластер представляет уникальную точку зрения на проект, и в то же время никто из членов проектной группы в одиночку не в состоянии успешно представлять все возможные взгляды, отражающие качественно различные цели. Для разрешения этой дилеммы команда соратников (команда равных), работающая над проектом, должна иметь четкую форму отчетности перед заинтересованными сторонами при распределенной ответственности за достижение общего успеха. В MSF следующие ролевые кластеры.

– *Управление продуктом*. Основная задача этого ролевого кластера – обеспечить, чтобы заказчик остался довольным в результате выполнения проекта. Этот ролевой кластер действует по отношению к проектной группе как представитель заказчика и зачастую формируется из сотрудников органи-

зации-заказчика. Он представляет бизнес-сторону проекта и обеспечивает его согласованность со стратегическими целями заказчика. В него же входит контроль за полным пониманием интересов бизнеса при принятии ключевых проектных решений.

- *Управление программой* обеспечивает управленческие функции – отслеживание планов и их выполнение, ответственность за бюджет, ресурсы проекта, разрешение проблем и трудностей процесса, создание условий, при которых команда может работать эффективно, испытывая минимум бюрократических преград.
- *Разработка*. Этот ролевой кластер занимается собственно программированием ПО.
- *Тестирование* – отвечает за тестирование ПО.
- *Удовлетворение потребителя*. Дизайн удобного пользовательского интерфейса и обеспечение удобства эксплуатации ПО (эргономики), обучение пользователей работе с ПО, создание пользовательской документации.
- *Управление выпуском*. Непосредственно ответственен за беспрепятственное внедрение проекта и его функционирование, берет на себя связь между разработкой решения, его внедрением и последующим сопровождением, обеспечивая информированность членов проектной группы о последствиях их решений.
- *Архитектура*[1]. Организация и выполнение высокоуровневого проектирования решения, создание функциональной спецификации ПО и управление этой спецификацией в процессе разработки, определение рамок проекта и ключевых компромиссных решений.

Масштабирование команды MSF. Наличие 7 ролевых кластеров не означает, что команда должна состоять строго из 7 человек. Один сотрудник может объединять несколько ролей. При этом некоторые роли нельзя объединять. В таблице 4.1 ниже представлены рекомендации MSF относительно совмещения ро-

[1] Этот ролевой кластер появился в версиях MSF 4.x. До этого данная ответственность входила в ролевой кластер «Управление программой».

лей в рамках одним членом команды. «+» означает, что совмещение возможно, «+–» –что совмещение возможно, но нежелательно, «–» означает, что совмещение не рекомендуется.

В частности, нельзя совмещать разработку и тестирование, поскольку, как обсуждалось выше, необходимо, чтобы у тестировщиков был сформирован свой, независимый взгляд на систему, базирующийся на изучении требований.

Таблица 4.1 *– Рекомендации MSF по совмещению ролей в команде проекта*

	Управление продуктом	Управление программой	Разработка	Тестирование	Удовлетворение потребителя	Управление выпуском	Архитектура
Управление продуктом		–	–	+	+	+–	–
Управление программой	–		–	+–	+–	+	+
Разработка	–	–		–	–	–	+
Тестирование	+	+–	–		+	+	+–
Удовлетворение потребителя	+	+–	–	+		+–	+–
Управление выпуском	+ –	+	–	+	+–		+
Архитектура	–	+	+	+–	+–	+	

Модель проектной группы MSF предлагает разбиение больших команд (более 10 человек) на малые многопрофильные ***группы направлений***. Эти малые коллективы работают параллельно, регулярно синхронизируя свои усилия, каждая из групп устроена на основе модели кластеров. Это компактные мини-команды, образующие матричную организационную структуру. В них входят по одному или по нескольку членов из разных ролевых кластеров. Такие команды имеют четко определенную задачу и ответственны за все относящиеся к ней вопросы, начиная от проектирования и составления календарного графика. Например, может быть сформирована специальная группа проектирования и разработки сервисов печати.

Кроме того, когда ролевому кластеру требуется много ресурсов, формируются так называемые ***функциональные группы***,

которые затем объединяются в ролевые кластеры. Они создаются в больших проектах, когда необходимо сгруппировать работников внутри ролевых кластеров по их областям компетенции. Например, в Майкрософт группа управления продуктом обычно включает специалистов по планированию продукта и специалистов по маркетингу. Как первая, так и вторая сферы деятельности относятся к управлению продуктом: одна из них сосредоточивается на выявлении качеств продукта, действительно интересующих заказчика, а вторая – на информировании потенциальных потребителей о преимуществах продукта.

Аналогично, в команде разработчиков возможна группировка сотрудников в соответствии с назначением разрабатываемых ими модулей: интерфейс пользователя, бизнес-логика или объекты данных. Часто программистов разделяют на разработчиков библиотек и разработчиков решения. Программисты библиотек обычно используют низкоуровневый язык C и создают повторно используемые компоненты, которые могут пригодиться всему предприятию. Создатели же решения обычно соединяют эти компоненты и работают с языками более высокого уровня, такими, как, например, Microsoft Visual Basic.

Часто функциональные группы имеют внутреннюю иерархическую структуру. Например, менеджеры программы могут быть подотчетны ведущим менеджерам программы, которые, в свою очередь отчитываются перед главным менеджером программы. Подобные структуры могут также появляться внутри областей компетенций. Но важно помнить, что эти иерархии не должны затенять модель команды MSF на уровне проекта в целом.

Ключевые термины

IT решение – скоординированная поставка набора элементов, необходимых для удовлетворения бизнес-потребности конкретного заказчика.

Итеративный подход при разработке ПО – предусматривает поэтапное создание всех элементов проекта.

Фазы проекта – определяют последовательно решаемые задачи.

Вехи проекта – ключевые точки проекта, характеризующие достижение какого-либо существенного результата.

Главные вехи – служат точками перехода от одной фазы проекта к другой.

Промежуточные вехи – показывают достижение определенного прогресса в исполнении фазы проекта.

Ролевые кластеры в MSF – представляет уникальную точку зрения на проект, определяющую необходимые роли для выполнения поставленных задач на определенной фазе проекта.

Краткие итоги

При разработке программных продуктов компания Microsoft использует понятие IT – решение, под которым понимается как скоординированная поставка набора элементов (таких, как: программные средства, документация, обучение и сопровождение), необходимых для удовлетворения бизнес-потребности конкретного заказчика. Методология компании Microsoft MSF является основой систем управления жизненным циклом приложений, реализованных в виде программного продукта Visual Studio Foundation Server. В MSF имеются шаблоны для формальных (MSF for CMMI) и гибких (MSF for Agile) методологи проектирования ПО. MSF использует модель жизненного цикла, которая сочетает в себе свойства двух стандартных моделей: каскадной и спиральной. В модели MSF определены следующие фазы: выработки концепции, планирования, разработки, стабилизации и внедрения. Модель команд MSF базируется на следующих ролевых кластерах: управлении продуктом, управлении программой, разработке, тестировании, удовлетворении потребителя, управлении выпуском и архитектуре. Модель команд MSF может масштабироваться для различных по сложности и объему проектов создания ПО.

Вопросы для самопроверки

1. История развития методологии MSF.
2. Поясните содержание понятия ИТ-решение в методологии Microsoft.
3. Основные принципы MSF.
4. Поясните принцип MSF «*Единое видение проекта*».
5. Поясните принцип MSF «*Гибкость – готовность к переменам*».
6. Поясните принцип MSF «*Концентрация на бизнес-приоритетах*».
7. Поясните принцип MSF «*Поощрение свободного общения*».
8. Поясните основные принципы модели команд MSF.

9. Поясните особенности отношений внутри команды MSF.
10. Ролевые кластеры.
11. Назначение кластера *Управление продуктом.*
12. Назначение кластера *Управление программой.*
13. Назначение кластера *Разработка и Тестирование.*
14. Назначение кластера *Удовлетворение потребителя.*
15. Назначение кластера *Управление выпуском.*
16. Назначение кластера *Архитектура.*
17. Масштабирование команды MSF.
18. Модель процесса в MSF.

5. ГИБКИЕ ТЕХНОЛОГИИ РАЗРАБОТКИ ПО

Гибкая методология разработки программного обеспечения ориентирована на использование итеративного подхода, при котором программный продукт создается постепенно, небольшими шагами, включающими реализацию определенного набора требований [4, 6, 12]. При этом предполагается, что требования могут изменяться. Команды, использующие гибкие методологии, формируются из универсальных разработчиков, которые выполняют различные задачи в процессе создания программного продукта.

При использовании гибких методологий минимизация рисков осуществляется путём сведения разработки к серии коротких циклов, называемых *итерациями*, продолжительностью 2–3 недели. Итерация представляет собой набор задач, запланированных на выполнение определенный период. В каждой итерации создается работоспособный вариант программной системы, в которой реализуются наиболее приоритетные (для данной итерации) требования заказчика. На каждой итерации выполняются все задачи, необходимые для создания работоспособного программного обеспечения: планирование, анализ требований, проектирование, кодирование, тестирование и документирование. Хотя отдельная итерация, как правило, недостаточна для выпуска новой версии продукта, подразумевается, что текущий программный продукт готов к выпуску в конце каждой итерации. По окончании каждой итерации команда выполняет переоценку приоритетов требований к программному продукту, возможно, вносит коррективы в разработку системы.

Принципы и значение гибкой разработки

Для методологии гибкой разработки декларированы ключевые постулаты, позволяющие командам достигать высокой производительности:

- люди и их взаимодействие;
- доставка работающего программного обеспечения;
- сотрудничество с заказчиком;
- реакция на изменение.

Люди и взаимодействие. Люди – важнейшая составная часть успеха. Отдельные члены команды и хорошие коммуникации важны для высокопроизводительных команд. Для содействия коммуникации гибкие методы предполагают частые обсуждения результатов работы и внесение изменений в решения. Обсуждения могут проводиться ежедневно длительностью несколько минут и по завершении каждой итерации с анализом результатов работ и ретроспективой. Для эффективных коммуникаций при проведении собраний участники команд должны придерживаться следующих ключевых правил поведения:

- уважать мнение каждого участника команды;
- быть правдивым при любом общении;
- прозрачность всех данных, действий и решений;
- уверенность, что каждый участник поддержит команду;
- приверженность команде и ее целям.

Для создания высокопроизводительных команд в гибких методологиях кроме эффективной команды и хороших коммуникаций необходим совершенный программный инструментарий.

Работающее программное обеспечение важнее всеобъемлющей документации. Все гибкие методологии выделяют необходимость доставки заказчику небольших фрагментов работающего программного обеспечения через заданные интервалы. Программное обеспечение, как правило, должно пройти уровень модульного тестирования, тестирования на уровне системы. При этом объем документации должен быть минимальным. В процессе проектирования команда должна поддерживать в актуальном состоянии короткий документ, содержащий обоснования решения и описание структуры.

Сотрудничество с заказчиком важнее формальных договоренностей по контракту. Чтобы проект успешно завершился, необходимо регулярное и частое общение с заказчиком. Заказчик должен регулярно участвовать в обсуждении принимаемых решений по программному обеспечению, высказывать свои пожелания и замечания. Вовлечение заказчика в процесс разработки программного обеспечения необходимо для создания качественного продукта.

Оперативное реагирование на изменения важнее следования плану. Способность реагирования на изменения во многом определяет успех программного проекта. В процессе создания программного продукта очень часто изменяются требования заказчика. Заказчики очень часто точно не знают, чего хотят, до тех пор, пока не увидят работающее программное обеспечение. Гибкие методологии ищут обратную связь от заказчиков в процессе создания программного продукта. Оперативное реагирование на изменения необходимо для создания продукта, который удовлетворит заказчика и обеспечит ценность для бизнеса.

Постулаты гибкой разработки поддерживаются 12 принципами [6, 8]. В конкретных методологиях гибкой разработки определены процессы и правила, которые в большей или меньшей степени соответствуют этим принципам. Гибкие методологии создания программных продуктов основываются на следующих принципах [8]:

1. *Высшим приоритетом считать удовлетворение пожеланий заказчика* посредством поставки полезного программного обеспечения в сжатые сроки с последующим непрерывным обновлением. Гибкие методики подразумевают быструю поставку начальной версии и частые обновления. Целью команды является поставка работоспособной версии в течение нескольких недель с момента начала проекта. В дальнейшем программные системы с постепенно расширяющейся функциональностью должны поставляться каждые несколько недель. Заказчик может начать промышленную эксплуатацию системы, если посчитает: она достаточно функциональна. Заказчик может просто ознакомиться с текущей версией программного обеспечения, представить свой отзыв с замечаниями.
2. *Не игнорировать изменение требований*, пусть даже на поздних этапах разработки. Гибкие процессы позволяют учитывать изменения для обеспечения конкурентных преимуществ заказчика. Команды, использующие гибкие методики, стремятся сделать структуру программы качественной, с минимальным влиянием изменений на систему в целом.

3. *Поставлять новые работающие версии ПО часто*, с интервалом от одной недели до двух месяцев, отдавая предпочтение меньшим срокам. При этом ставится цель поставить программу, удовлетворяющую потребностям пользователя, с минимумом сопроводительной документации.
4. *Заказчики и разработчики должны работать совместно* на протяжении всего проекта. Считается, что для успешного проекта заказчики, разработчики и все заинтересованные лица должны общаться часто и помногу для целенаправленного совершенствования программного продукта.
5. *Проекты должны воплощать в жизнь целеустремленные люди*. Создавайте команде проекта нормальные условия работы, обеспечьте необходимую поддержку и верьте, что члены команды доведут дело до конца.
6. *Самый эффективный и продуктивный метод передачи информации команде разработчиков и обмена мнениями внутри неё – разговор лицом к лицу.* В гибких проектах основной способ коммуникации – простое человеческое общение. Письменные документы создаются и обновляются постепенно по мере разработки ПО и только в случае необходимости.
7. *Работающая программа – основной показатель прогресса в проекте.* О приближении гибкого проекта к завершению судят по тому, насколько имеющаяся в данный момент программа отвечает требованиям заказчика.
8. *Гибкие процессы способствуют долгосрочной разработке.* Заказчики, разработчики и пользователи должны быть в состоянии поддерживать неизменный темп сколь угодно долго.
9. *Непрестанное внимание к техническому совершенству и качественному проектированию* повышает отдачу от гибких технологий. Члены гибкой команды стремятся создавать качественный код, регулярно проводя рефакторинг.
10. *Простота* – искусство достигать большего, делая меньше. Члены команды решают текущие задачи максимально просто и качественно. Если в будущем возникнет какая-либо проблема, то в качественный код имеется возможность внести изменения без больших затрат.

11. *Самые лучшие архитектуры, требования и проекты выдают самоорганизующиеся команды.* В гибких командах задачи поручаются не отдельным членам, а команде в целом. Команда сама решает, как лучше всего реализовать требования заказчика. Члены команды совместно работают над всеми аспектами проекта. Каждому участнику разрешено вносить свой вклад в общее дело. Нет такого члена команды, который единолично отвечал бы за архитектуру, требования или тесты.
12. *Команда должна регулярно задумываться над тем, как стать ещё более эффективной*, а затем соответственно корректировать и подстраивать свое поведение. Гибкая команда постоянно корректирует свою организацию, правила, соглашения и взаимоотношения.

Вышеприведенным принципам в определенной степени соответствует ряд методологий разработки программного обеспечения:

- *Agile Modeling* – набор понятий, принципов и приёмов (практик), позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки программного обеспечения [1];
- *Agile Unified Process* (AUP) упрощенная версия IBM Rational Unified Process (RUP), которая описывает простое и понятное приближение (модель) для создания программного обеспечения для бизнес-приложений [37];
- *OpenUP* – это итеративно-инкрементальный метод разработки программного обеспечения. Позиционируется как лёгкий и гибкий вариант RUP [33];
- *Agile Data Method* – группа итеративных методов разработки программного обеспечения, в которых требования и решения достигаются в рамках сотрудничества разных кросс-функциональных команд [21];
- *DSDM* – методика разработки динамических систем, основанная на концепции быстрой разработки приложений (Rapid Application Development, RAD). Представляет собой итеративный и инкрементный подход, который придаёт особое значение продолжительному участию в процессе пользователя/потребителя [24];

- *Extreme programming (XP)* – экстремальное программирование [8];
- *Adaptive software development (ADD)* – адаптивная разработка программ [9];
- *Feature driven development* (FDD) – разработка ориентированная на постепенное добавление функциональности [25];
- *Getting Real* – итеративный подход без функциональных спецификаций, использующийся для веб-приложений [26];
- MSF fog Agile Software Development – гибкая методология разработки ПО компании Microsoft [18, 32];
- Scrum устанавливает правила управления процессом разработки и позволяет использовать уже существующие практики кодирования, корректируя требования или внося тактические изменения [20, 35].

Следует отметить, что в чистом виде методологии гибкого программирования редко используются командами разработчиков. Как правило, успешные команды применяют полезные приемы и свойства нескольких процессов, подстраивая их под конкретное представление команды о гибкости процесса разработки.

Методология гибкой разработки SCRUM

Методология Scrum представляет собой итеративный процесс разработки программного обеспечения [5, 20, 39]. При такой разработке для программного продукта создается много последовательных выпусков, в которых постепенно добавляется требуемая функциональность. Итеративный подход позволяет по завершении текущей итерации продемонстрировать заказчику работоспособный программный продукт, возможно с ограниченной функциональностью, получить отзыв, замечания и дополнительные требования, которые будут учтены в следующих итерациях. Основными артефактами в методологии Scrum являются рабочие элементы, отчеты, книги и панели мониторинга [20].

Рабочие элементы

Рабочие элементы используются для отслеживания, наблюдения за состоянием хода разработки ПО и создания отчетов. *Рабочий элемент* – это запись, которая создается в Visual Studio Team Foundation Server для задания определения, назначения,

приоритета и состояния элемента работы. Для шаблона Microsoft Visual Studio Scrum 2.2 определяет следующие типы рабочих элементов:

– Bug – ошибка;
– Task – задача;
– Product BackLog Item – невыполненная работа по продукту;
– Epic – возможности;
– Feature –функция;
– Impediment – препятствие;
– Test Case – тестовый случай.

На рис. 5.1 приведено меню Рабочие элементы.

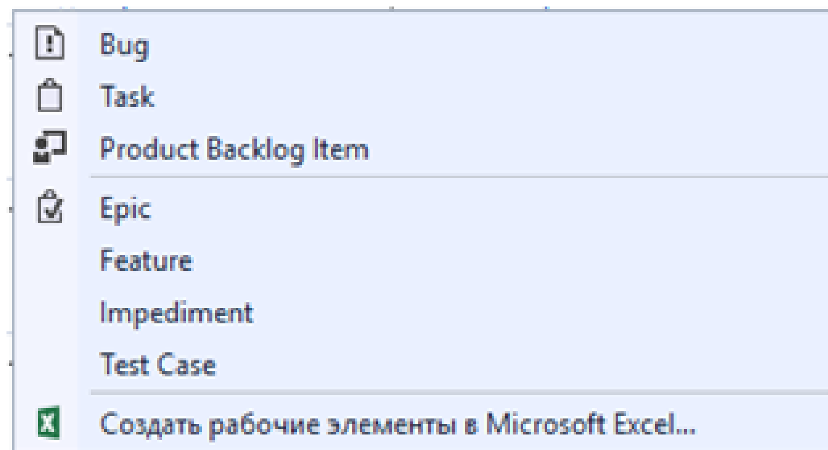

Рисунок 5.1 *– Меню Рабочие элементы*

В методологии Scrum пользовательские требования, которые определяют функциональность продукта, задаются *элементами задела работы продукта* (Product Backlog Item – PBI). Элементы задела работы продукта, которые будем называть «*элементы работы – ЭР*», представляют собой краткое описание функций продукта и оформляются в произвольной форме в виде кратких заметок. Вначале задаются наиболее важные и понятные всем пользовательские требования – ЭР. Элементы работы могут детализироваться в виде задач. В процессе создания программного продукта ЭР могут уточняться, добавляться или удаляться из списка требований.

Цикл выпуска продукта в Scrum состоит из ряда итераций, которые называются *спринтами*. Спринт имеет фиксированную

длительность, как правило, 1–4 недели. Элементы работы, включенные в очередной спринт, не подлежат изменению до его окончания. Хотя спринт завершается подготовкой работоспособного программного продукта, его текущей функциональности может быть недостаточно для оформления выпуска, имеющего ценность для заказчика. Поэтому выпуск работоспособного программного продукта, который заказчик может использовать, включает, как правило, несколько спринтов.

Организация команды

Организация команды в методологии Scrum определяет три роли [6]:

- владелец продукта (Product owner);
- руководитель (ScrumMaster);
- члены команды (Team members).

Владелец продукта отвечает за всё, что связано с потребительскими качествами программного продукта. Он определяет пользовательские требования – ЭР, анализирует их реализацию, обладает правом изменения требований, контролирует качество продукта. Он может быть представителем заказчика в команде или членом команды разработчиков, который представляет интересы заказчика. Владелец продукта выполняет следующие основные задачи:

- определение и приоритезацию требований/функций, то есть элементов работ и задач;
- планирование спринтов и выпусков;
- тестирование требований/функций.

Руководитель отвечает за состояние и координацию проекта, продуктивность команды и устранение препятствий, мешающих проекту. В обязанности руководителя входит:

- проведение ежедневных Scrum-собраний;
- привлечение сотрудников вне команды:
- стимулирование эффективного общения членов команды;
- определение размера команды.

Члены команды отвечают за разработку программного продукта высокого качества. Они должны обладать навыками в проектировании и архитектуре программного продукта, бизнес-

анализе, программировании, тестировании, настройке баз данных и проектировании пользовательского интерфейса. Члены команды участвуют в планировании спринтов. Команда может включать опытных разработчиков и новичков, которые в процессе работы должны совершенствоваться при обмене знаниями с другими членами команды. Члены команды отвечают за следующие задачи в проекте:

- обязательное выполнение элементов работ, включенных в текущий спринт;
- акцент на взаимосвязанных задачах спринта;
- совершенствование команды.

Жизненный цикл проекта ПО

Инструментальная и методическая поддержка гибкого подхода к созданию программных продуктов Scrum, реализованная в Visual Studio, позволяет управлять жизненным циклом проекта ПО (рис. 5.2) [20].

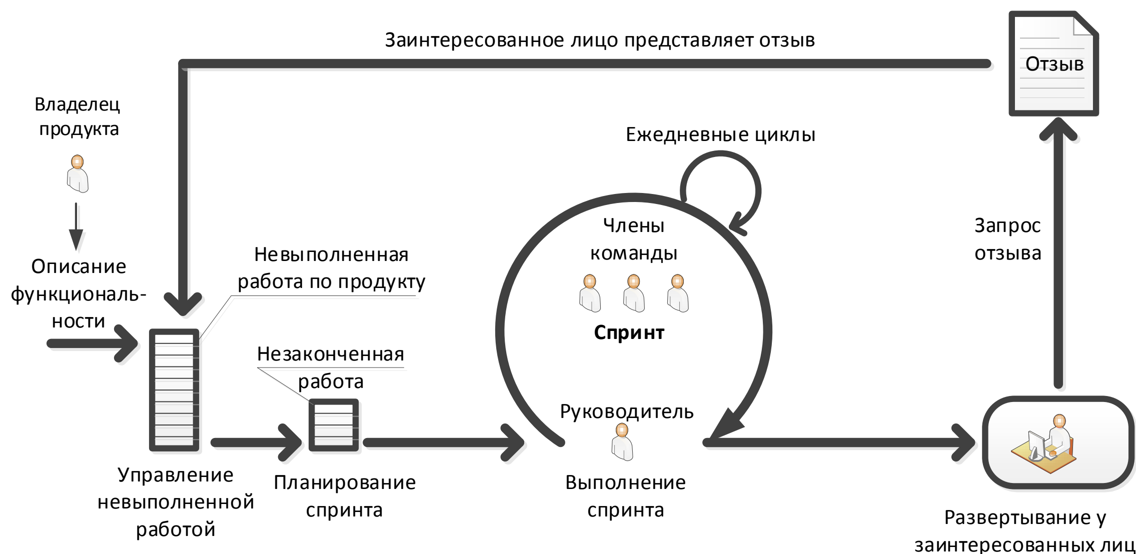

***Рисунок 5.2** – Жизненный цикл проекта ПО*

В начале проектирования владелец продукта и заказчик формируют концепцию программного продукта, которая показывает, для кого предназначен продукт, какие преимущества получат пользователи и какие существуют конкуренты. Концепция продукта связывается с областью проекта и ограничениями. Об-

ласть проекта определяет масштаб работ, а ограничения – условия, которыми будут руководствоваться для первых спринтов и выпусков. Далее владелец продукта создает список всех потенциальных функций продукта – «Невыполненная работа по продукту» (Product Backlog). *Невыполненная работа по продукту*, которую в дальнейшем будем называть *невыполненная работа – НВР*, содержит список *элементов работы* – пользовательских описаний функциональности.

Управление *невыполненной работой* по проекту сводится к поддержанию *элементов работ* в актуальном состоянии. Отдельные элементы работ списка НВР могут добавляться или удаляться в процессе создания ПО. Это является результатом того, что команда получает дополнительную информацию о новых требованиях заказчика к проектируемому программному продукту, а заказчик выясняет, как реализуются его ожидания.

Для элементов работ владелец продукта совместно с командой проекта расставляет приоритеты. При планировании спринта в него включают наиболее значимые, с точки зрения владельца продукта, пользовательские требования – ЭР, которые характеризуются наибольшей потребительской ценностью. Выбранные элементы работ перемещают в список «*Незаконченная работа*» (Sprint Backlog). Список *Незаконченная работа* (НЗР) отражает состав работ планируемого спринта. Список НЗР является результатом процесса планирования спринта.

Координацией работ в спринте занимается руководитель спринта (ScrumMaster). Он организовывает процесс приоритезации задач спринта, распределения задач между членами команды. Руководитель спринта проводит собрание по планированию работ, ежедневные собрания для краткого обсуждения результатов работы и проблем, обзорные собрания в конце спринта и выпуска.

Ежедневные Scrum-собрания имеют продолжительность 15–30 минут. Целью таких собраний является выявление проблем, которые тормозят процесс разработки, и определить действия по их нейтрализации. Для простых проблем принимается решение по их устранению, а сложные проблемы откладываются на последующие спринты. В ходе ежедневного Scrum-собрания руководитель задает темп спринта, акцентирует команду на

наиболее важных элементах списка невыполненных работ. Каждый член команды сообщает, что было сделано вчера, что будет делать сегодня и какие имеются препятствия в работе. Если на ежедневном Scrum-собрании возникают вопросы, для решения которых необходимы специалисты, которых в команде нет, тогда руководитель берет на себя анализ и возможные пути разрешения данного вопроса.

Результатом спринта является работоспособное ПО, возможно, обладающее только частью необходимых функций программного продукта. Выпуск спринта может быть развернут у заинтересованных лиц для предварительного анализа соответствия ожиданиям заказчика. Результатом отзыва является формирование «Отзыв» (FeedBack), что может привести к изменению содержания списка *Элемент задела работы продукта*. После выполнения всех работ по программному продукту, то есть обнуления списка требований в списке *Элемент задела работы продукта* подготавливается финальный выпуск программного продукта.

Управление невыполненной работой

Список *Невыполненная работа по продукту* является одним из ключевых артефактов в методологии Scrum. Успех Scrum-команды во многом определяется качественным содержанием данного списка. Список НВР обычно включает пользовательские описания функциональности - *элементы работы*, а также может включать нефункциональные требования. Для создания списка НВР в TFS могут применяться различные клиентские сервисы (рис 5.3):

- командный обозреватель Visual Studio;
- веб доступ через Team Web Access;
- Microsoft Office Excel;
- Microsoft Office Project.

Владелец продукта на основе требований и пожеланий клиентов формирует список функций продукта в виде элементов задела работы продукта, которые помещает в список *Невыполненная работа по продукту*. При создании нового элемента невыполненной работы для него устанавливается состояние «Новый» (рис. 5.4).

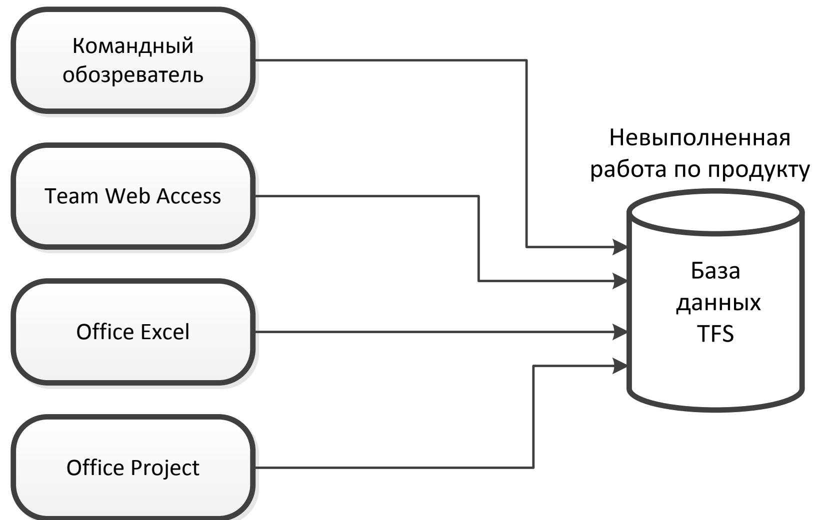

Рисунок 5.3 *– Клиентские сервисы TFS*

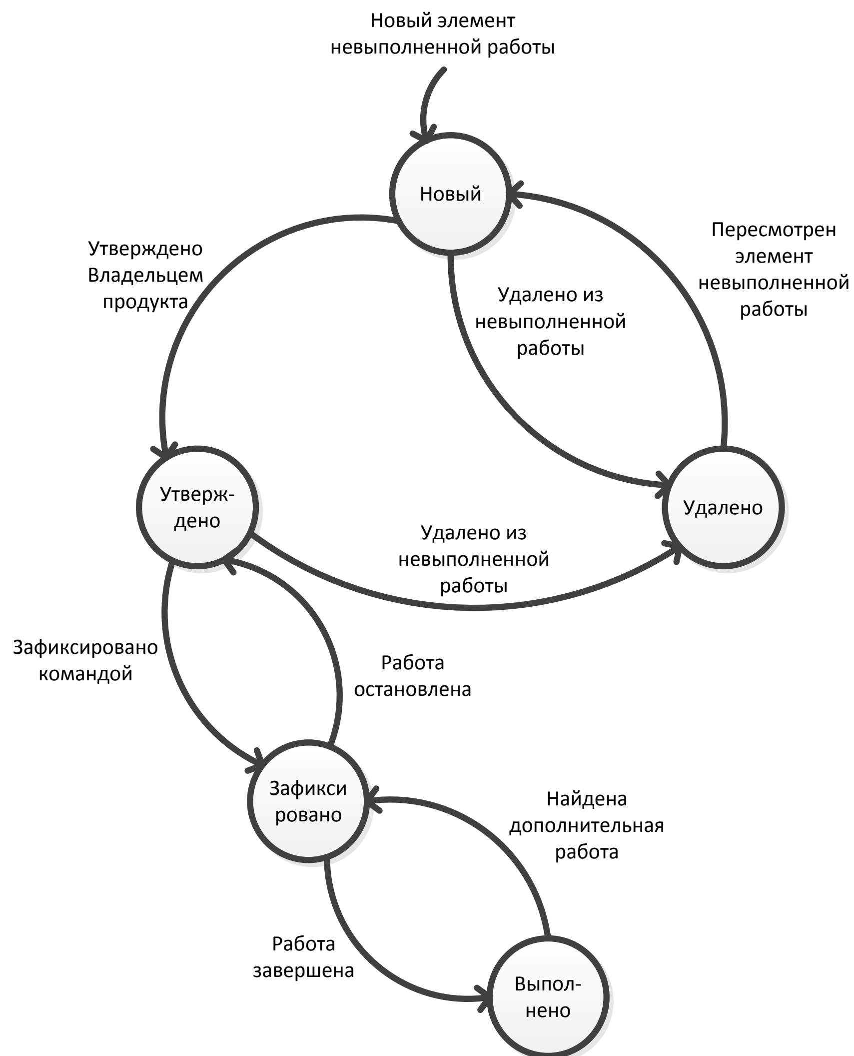

Рисунок 5.4 *– Рабочий процесс элемента невыполненной работы*

После установки элементу работы приоритета его состояние изменяют на «Утверждено». На собрании по планированию спринта команда просматривает наиболее приоритетные элементы работы и выбирает те, которые будут выполняться в текущем спринте. Для элементов невыполненной работы, которые попали в текущий спринт, устанавливается состояние «Зафиксировано». Это означает тот факт, что рабочие элементы спринта не подлежат изменению до конца спринта. При завершении работы по элементу его состояние устанавливается «Выполнено». Если для элемента работы, находящегося в состоянии «Выполнено», выявляется дополнительная работа, то этот элемент может быть переведен в состояние «Зафиксировано». Для элемента работы, находящегося в состоянии «Зафиксировано», при возникновении проблем, препятствующих его завершению в спринте, может быть работа приостановлена и установлено состояние «Утверждено». Элемент невыполненной работы может быть удален из списка *Невыполненная работа по продукту* по решению Владельца продукта. Это может произойти как из состояния «Новый», так и состояния «Утверждено», что соответствует установке для элемента работы состояния «Удалено». В результате пересмотра элемента работы, имеющего состояние «Удалено», для него вновь возможен перевод в состояние «Новый».

Список *Невыполненная работа по продукту* является главным документом для Scrum-команды. На основе данного списка команда создает другие рабочие элементы, составляющие спринты и выпуски. Для *Элементов невыполненной работы* команда создает задачи и тестовые случаи. Задачи детализируют элемент работы и определяют конкретную реализацию требований пользователя. Тестовые случий необходимы для проверки соответствия функциональности кода требованиям пользователя. Если тестовый случай не проходит, то создается рабочий элемент «Ошибка». При блокировании задачи из-за невозможности её выполнения в текущем спринте создают рабочий элемент «Препятствие». Scrum-команда может создавать вспомогательные рабочие элементы (ошибки и препятствия) в отношении элементов, на которые они влияют (задачи и тестовые случаи), и связывать эти элементы (рис. 5.5).

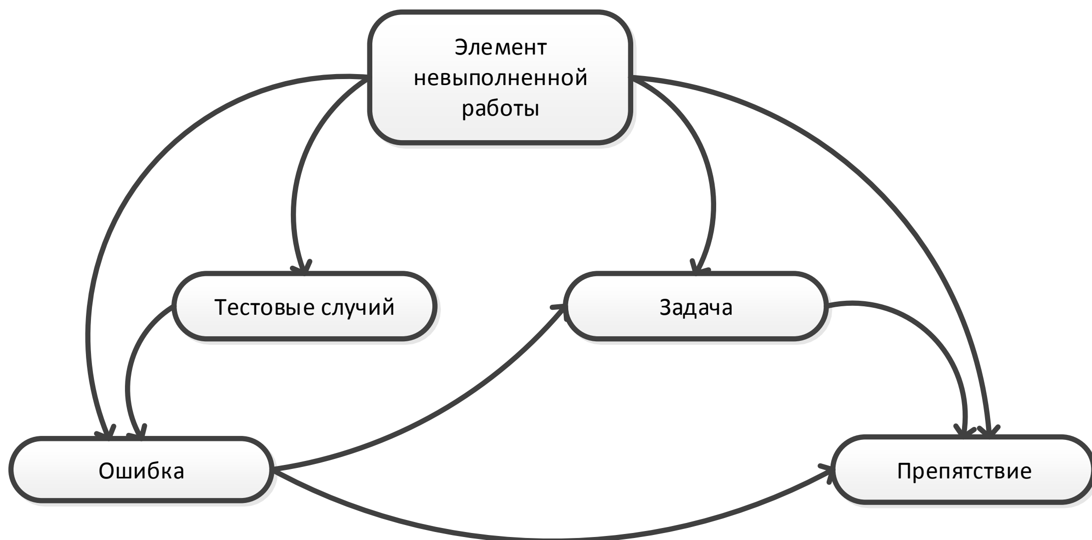

***Рисунок 5.5** – Связь между рабочими элементами*

Для отслеживания хода выполнения проекта можно создавать отчеты, отражающие наиболее важные данные для текущего проекта. На рис.5.6 приведен график выполнения работ по проекту. На рис. 5.7 приведен график скорости работ команды по спринту. На рис. 5.8 приведена панель мониторинга работ по спринту.

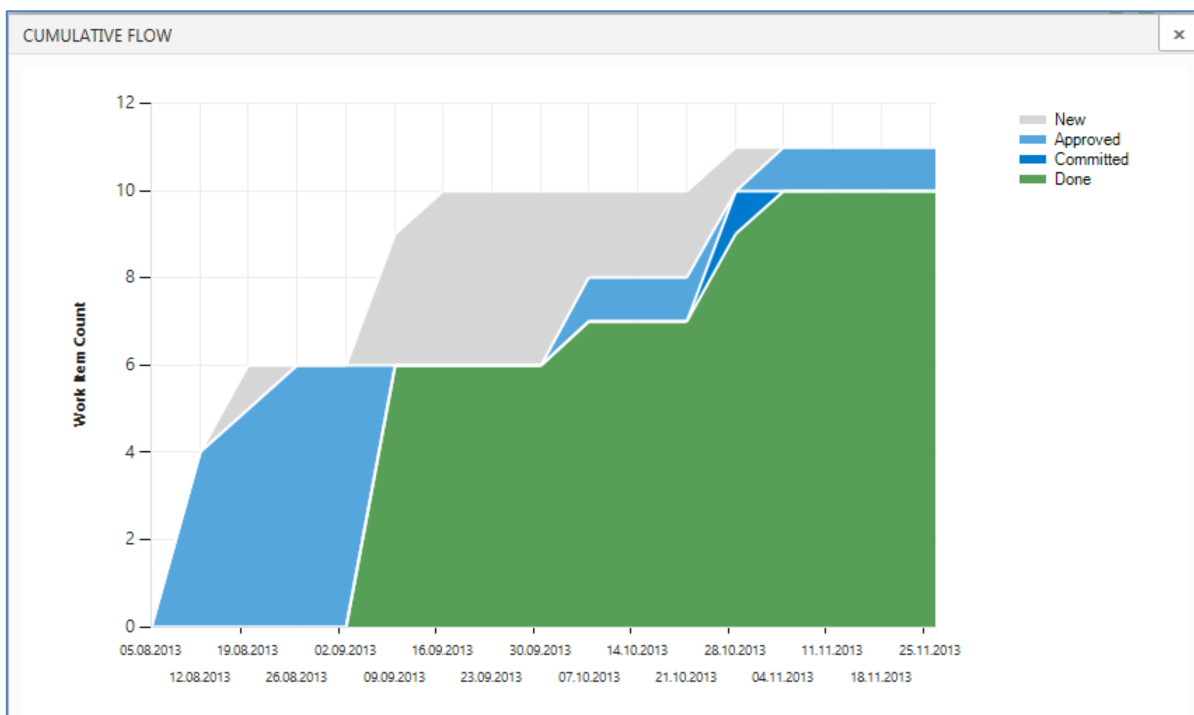

***Рисунок 5.6** – Выполнение работ по проекту*

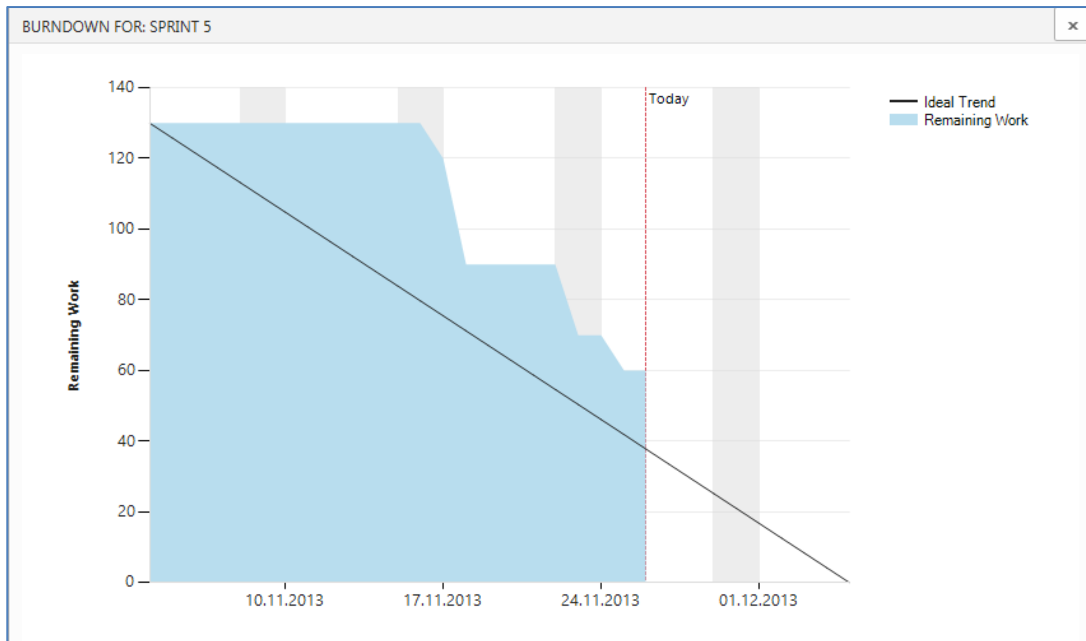

***Рисунок 5.7** – Скорость выполнения работ по спринту*

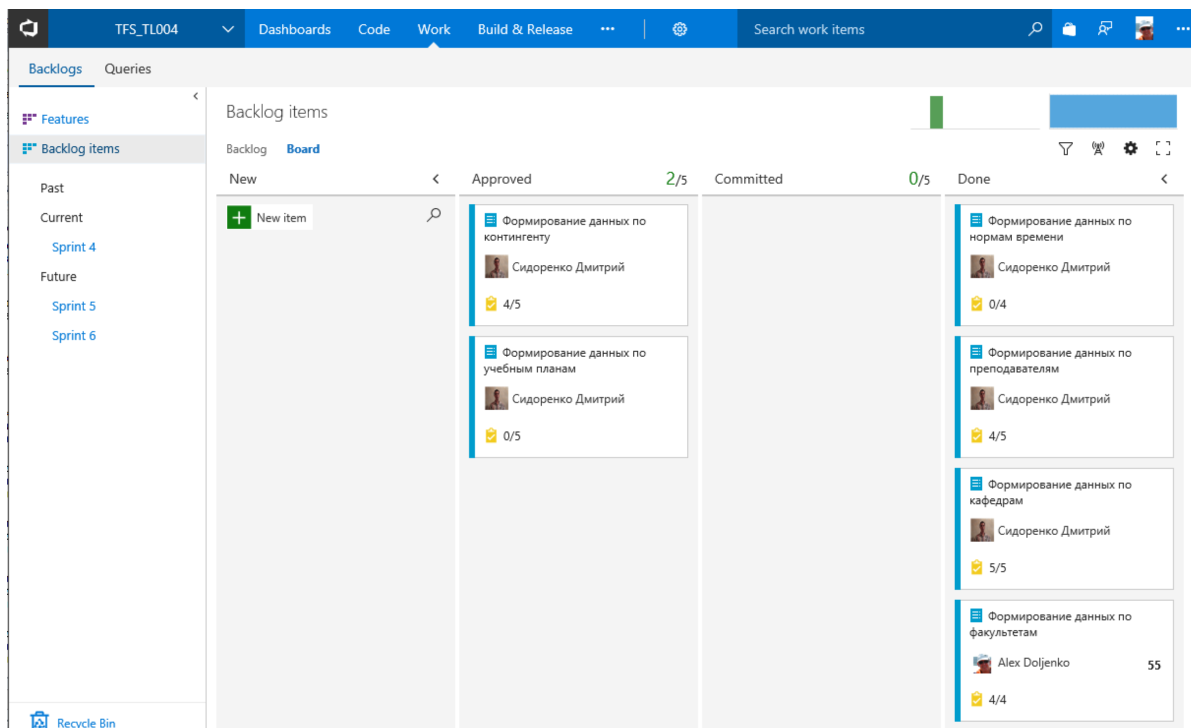

***Рисунок 5.8** – Панель мониторинга работ по спринту*

В процессе создания ПО можно пользоваться стандартными отчетами или создавать собственные отчеты. Отчеты можно создавать, настраивать и просматривать с помощью Excel, Project или служб Reporting Services SQL Server.

Методология Scrum имеет следующие положительные стороны:

- пользователи начинают видеть систему спустя всего несколько недель и могут выявлять проблемы на ранних стадиях разработки программного продукта;
- интеграция технических компонентов происходит в ходе каждого спринта, и поэтому проблемы проекта (если они возникают) выявляются практически сразу;
- в каждом спринте команда фокусируется на контроле качества;
- гибкая работа с изменениями в проекте на уровне спринта.

Ключевые термины

Гибкая методология разработки программного обеспечения – методология, ориентированная на использование итеративного подхода, при котором программный продукт создается постепенно, небольшими шагами, включающими реализацию определенного набора требований.

Итерация – набор задач, запланированных на выполнение определенный период.

Методология Scrum – гибкая методология разработки программного обеспечения, которая представляет собой итеративный процесс.

Рабочий элемент – запись, которая создается в Visual Studio Team Foundation Server для задания определения, назначения, приоритета и состояния элемента работы.

Элементы задела работы продукта – список пользовательских требований, которые определяют функциональность продукта, в методологии Scrum.

Элемент работы – краткое описание функций продукта, в методологии Scrum.

Спринт – набор задач, запланированных на выполнение определенный период времени, в методологии Scrum.

Владелец продукта – член Scrum-команды, который отвечает за всё, что связано с потребительскими качествами программного продукта.

Руководитель – член Scrum-команды, который отвечает за состояние и координацию проекта, продуктивность команды и устранение препятствий, мешающих проекту.

Член команды – член Scrum-команды, который наравне с другими членами команды отвечает за разработку программного продукта высокого качества.

Невыполненная работа по продукту – список элементов работы (пользовательских описаний функциональности), которые необходимо выполнить при создании программного продукта.

Незаконченная работа – список элементов работы планируемого спринта.

Краткие итоги

Гибкая методология разработки программного обеспечения ориентирована на использование итеративного подхода, при котором программный продукт создается постепенно. Программный продукт создается за несколько итераций, включающих реализацию определенного набора требований. Итерации имеют длительность 2–3 недели. Результатом итерации является промежуточный вариант работоспособного программного обеспечения. Для методологии гибкой разработки декларированы ключевые постулаты и принципы. Принципам гибкой разработки ПО, в определенной степени, соответствуют ряд методологий. Методология Scrum представляет собой итеративный процесс разработки программного обеспечения. Рабочие элементы используются для отслеживания, наблюдения за состоянием хода разработки ПО и создания отчетов. Организация команды в методологии Scrum определяет роли владельца продукта, руководителя и членов команды. Жизненный цикл проекта ПО включает формирование и управление невыполненной работой, планирование и выполнение спринта, развертывание ПО у заинтересованных лиц, получение отзывов для улучшения программного продукта. При управлении рабочим процессом элементов невыполненной работы по продукту постоянно отслеживаются связи и изменяется состояние этих элементов.

Вопросы для самопроверки

1. Поясните понятие «гибкая методология разработки программного обеспечения».
2. Какие компетенции необходимы для команды разработчиков, использующих гибкие методологии?
3. Как управляют рисками в гибких методологиях разработки ПО?
4. Какие задачи выполняются на итерациях в методологии гибкой разработки?
5. Назовите ключевые ценности методологий гибкой разработки ПО.
6. Назовите основные принципы гибкой разработки ПО.
7. Какие существуют методологии, которые соответствуют принципам гибкой разработки ПО?
8. Поясните, как в гибком подходе относятся к документированию и выпуску работоспособного кода.
9. Поясните, как должно быть организовано взаимодействие с заказчиком в гибком подходе к разработке ПО.
10. Поясните, как относятся к изменениям в гибком подходе к разработке ПО.
11. Поясните основные положения методологии Scrum.
12. Какие артефакты характерны для методологии Scrum?
13. Какие рабочие элементы определены в шаблоне Microsoft Visual Studio Scrum 2.2?
14. Поясните назначение Элементов задела работы продукта.
15. Что представляет собой спринт в методологии Scrum?
16. Какие роли определены в организации команды в методологии Scrum?
17. Кто отвечает за качественный выпуск программного продукта в методологии Scrum?
18. Поясните содержание жизненного цикла проекта ПО в методологии Scrum.
19. Поясните содержание рабочего процесса элемента невыполненной работы.
20. Поясните возможные связи между рабочими элементами.

Упражнения

1. Проведите анализ возможностей методологии Agile Unified Process.
2. Проведите анализ возможностей методологии Agile Data Method.
3. Проведите анализ возможностей методологии Feature driven development.
4. Проведите исследование (поиск по Интернету) применимости методологии Scrum для проектирования программных систем.
5. Проанализируйте причины распространения методологии Scrum для создания эффективных программных решений.
6. Проведите исследование (поиск по Интернету) программного инструментария методологии Scrum для платформ, отличных от Microsoft.Net.

6. УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ

Требования к программному обеспечению

Требование – это любое условие, которому должна соответствовать разрабатываемая система или программное средство. Требованием может быть возможность, которой система должна обладать, и ограничение, которому система должна удовлетворять.

В соответствии с Глоссарием терминов программной инженерии IEEE [40], являющимся общепринятым международным стандартным глоссарием, требование – это:

- условия или возможности, необходимые пользователю для решения проблем или достижения целей;
- условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;
- документированное представление условий или возможностей для пунктов 1 и 2.

В соответствии со стандартом разработки требований ISO/IEC 29148, требование – это утверждение, которое идентифицирует эксплуатационные, функциональные параметры, характеристики или ограничения проектирования продукта или процесса, которое однозначно, проверяемо и измеримо. Необходимо для приемки продукта или процесса (потребителем или внутренним руководящим принципом обеспечения качества). Также глоссарий ITILv3 определяет такое понятие, как набор требований – документ, содержащий все требования к продукту, а также к новой или измененной ИТ-услуге.

Процесс управления требованиями

Управление требованиями к программному обеспечению – процесс, включающий идентификацию, выявление, документирование, анализ, отслеживание, приоритизацию требований, достижение соглашения по требованиям и затем управление изменениями и уведомление соответствующих заинтересованных лиц. Управление требованиями – непрерывный процесс на протяжении всего проекта разработки программного обеспечения.

Цель управления требованиями состоит в том, чтобы гарантировать, что организация документирует, проверяет и удовлетворяет потребности и ожидания её клиентов и внутренних или внешних заинтересованных лиц. Управление требованиями начинается с выявления и анализа целей и ограничений клиента. Управление требованиями, далее, включает поддержку требований, интеграцию требований и организацию работы с требованиями и сопутствующей информацией, поставляющейся вместе с требованиями.

Установленная таким образом отслеживаемость требований используется для того, чтобы уведомлять заинтересованных участников об их выполнении, с точки зрения их соответствия, законченности, охвата и последовательности. Отслеживаемость также поддерживает управление изменениями как часть управления требованиями, так как она способствует пониманию того, как изменения воздействуют на требования или связанные с ними элементы, и облегчает внесение этих изменений.

Управление требованиями включает общение между проектной командой и заинтересованными лицами с целью корректировки требований на протяжении всего проекта. Постоянное общение всех участников проекта важно, для того чтобы ни один класс требований не доминировал над другими.

При формулировке требований к программной системе существуют трудности в понимании между заказчиком и программистами. Далеко не очевидно, что та система, которую хочет заказчик, вообще можно сделать. Ошибки и разночтения, которые возникают при выявлении требований к системе, оказываются одними из самых дорогих. Требования – это то исходное понимание задачи разработчиками, которое является основой всей разработки.

Требования к ПО подвержены изменениям:

- меняется ситуация на рынке, для которого предназначалась система;
- в ходе разработки возникают проблемы и трудности, в силу которых итоговая функциональность меняется (видоизменяется, урезается);

– заказчик может менять свое собственное видение системы: то ли он лучше понимает, что же ему на самом деле надо, то ли выясняется, что он что-то упустил с самого начала, то ли выясняется, что разработчики его не так поняли.

Кроме того, требования могут изменяться по ходу разработки.

Виды и свойства требований

Разделим требования на две большие группы – функциональные и нефункциональные.

Функциональные требования являются детальным описанием поведения и сервисов системы, ее функционала. Они определяют то, что система должна уметь делать.

В свою очередь функциональные требования делятся на три основных группы:

– бизнес-требования;
– пользовательские требования;
– функциональные требования.

Бизнес-требования описывают пожелания пользователей с точки зрения задач бизнеса. Примеры бизнес-требований:

– ускорение обработки заказов на 30 %.
– снижение издержек ведения бухгалтерии на 200 тыс. рублей в год.

Источником бизнес-требований, как правило, является отдел маркетинга и высшее руководство. Бизнес-требования описывают критерии успешности проекта с точки зрения спонсоров проекта. Бизнес-требования являются требованиями высшего уровня, остальные требования подчинены им.

Пользовательские требования отражают пожелания непосредственных пользователей системы. Работе с пользовательскими требованиями следует уделять пристальное внимание, так как мнение о системе, сложившееся у непосредственных пользователей, является ключевым фактором успешности внедрения и эксплуатации системы.

Самый нижний уровень требований – это функциональные требования. Данный вид требований детально описывает функции, которые должно выполнять ПО для удовлетворения пользовательских требований. Функциональные требования фактически являются заданиями для архитекторов и программистов.

Нефункциональные требования не являются описанием функций системы. Этот вид требований описывает такие характеристики системы, как: производительность, надежность, особенности поставки (наличие инсталлятора, документации), определенный уровень качества. Сюда же могут относиться требования на средства и процесс разработки системы, требования к переносимости, соответствию стандартам и т.д. Требования этого вида часто относятся ко всей системе в целом. На практике, особенно начинающие специалисты, часто забывают про некоторые важные нефункциональные требования.

Важными нефункциональными требованиями являются:

- *ясность, недвусмысленность* – однозначность понимания требований заказчиком и разработчиками. Часто этого трудно достичь, поскольку конечная формализация требований, выполненная с точки зрения потребностей дальнейшей разработки, трудна для восприятия заказчиком или специалистом предметной области, которые должны проинспектировать правильность формализации;
- *полнота и непротиворечивость;*
- *необходимый уровень детализации*. Требования должны обладать ясно осознаваемым уровнем детализации, стилем описания, способом формализации: либо это описание свойств предметной области, для которой предназначается ПО, либо это техническое задание, которое прилагается к контракту, либо это проектная спецификация, которая должна быть уточнена в дальнейшем, при детальном проектировании. Либо это еще что-нибудь. Важно также ясно видеть и понимать тех, для кого данное описание требований предназначено, иначе не избежать недопонимания и последующих за этим трудностей. Ведь в разработке ПО задействовано много различных специалистов – инженеров, программистов, тестировщиков, представителей заказчика, возможно, будущих пользователей – и все они имеют разное образование, профессиональные навыки и специализацию, часто говорят на разных языках. Здесь также важно, чтобы требования были максимально абстрактны и независимы от реализации;

– *прослеживаемость* – важно видеть то или иное требование в различных моделях, документах, наконец, в коде системы. А то часто возникают вопросы типа «Кто знает, почему мы решили, что такой-то модуль должен работать следующим образом ….?». Прослеживаемость функциональных требований достигается путем их дробления на отдельные, элементарные требования, присвоение им идентификаторов и создание трассировочной модели, которая в идеале должна протягиваться до программного кода. Хочется, например, знать, где нужно изменить код, если данное требование изменилось. На практике полная формальная прослеживаемость труднодостижима, поскольку логика и структура реализации системы могут не совпадать с таковыми для модели требованиями. В итоге одно требование оказывается сильно «размазано» по коду, а тот или иной участок кода может влиять на много требований. Но стремиться к прослеживаемости необходимо, разумно совмещая формальные и неформальные подходы;
– *тестируемость и проверяемость* – необходимо, чтобы существовали способы оттестировать и проверить данное требование. Причем важны оба аспекта, поскольку часто проверить-то заказчик может, а вот тестировать данное требование очень трудно или невозможно ввиду ограниченности доступа (например, по соображениям безопасности) к окружению системы для команды разработчика. Итак, необходимы процедуры проверки – выполнение тестов, проведение инспекций, проведение формальной верификации части требований и пр. Нужно также определять «планку» качества (чем выше качество, тем оно дороже стоит!), а также критерии полноты проверок, чтобы выполняющие их и руководители проекта четко осознавали, что именно проверено, а что еще нет;
– *модифицируемость*. Определяет процедуры внесения изменений в требования.

Варианты формализации требований

Вообще говоря, требования как таковые – это некоторая абстракция. В реальной практике они всегда существуют в виде ка-

кого-то представления – документа, модели, формальной спецификации, списка и т.д. Требования важны как таковые, потому что оседают в виде понимания разработчиками нужд заказчика и будущих пользователей создаваемой системы. Но так как в программном проекте много различных аспектов, видов деятельности и фаз разработки, то это понимание может принимать разные представления. Каждое представление требований выполняет определенную задачу, например, служит «мостом», фиксацией соглашения между разными группами специалистов, или используется для оперативного управления проектом (отслеживается, в какой фазе реализации находится то или иное требование, кто за него отвечает и пр.), или используется для верификации и модельно-ориентированного тестирования. И в первом, и во втором, и в третьем примере мы имеем дело с требованиями, но формализованы они будут по-разному.

Итак, формализация требований в проекте может быть очень разной – это зависит от его величины, принятого процесса разработки, используемых инструментальных средств, а также тех задач, которые решают формализованные требования. Более того, может существовать параллельно несколько формализаций, решающих различные задачи. Рассмотрим варианты.

1. *Неформальная постановка требований в переписке по электронной почте.* Хорошо работает в небольших проектах, при вовлеченности заказчика в разработку (например, команда выполняет субподряд). Хорошо также при таком стиле, когда есть взаимопонимание между заказчиком и командой, то есть лишние формальности не требуются. Однако электронные письма в такой ситуации часто оказываются важными документами – важно уметь вести деловую переписку, подводить итоги, хранить важные письма и пользоваться ими при разногласиях. Важно также вовремя понять, когда такой способ перестает работать и необходимы более формальные подходы.
2. *Требования в виде документа* – описание предметной области и ее свойств, техническое задание как приложение к контракту, функциональная спецификация для разработчиков и т.д.

3. *Требования в виде графа с зависимостями* в одном из средств поддержки требований (IBM Rational RequisitePro, DOORS, Borland CaliberRM). Такое представление удобно при частом изменении требований, при отслеживании выполнения требований, при организации «привязки» к требованиям задач, людей, тестов, кода. Важно также, чтобы была возможность легко создавать такие графы из текстовых документов, и наоборот, создавать презентационные документы по таким графам.
4. *Формальная модель требований* для верификации, модельно-ориентированного тестирования и т.д.

Итак, каждый способ представления требований должен отвечать на следующие вопросы: кто потребитель, пользователь этого представления, как именно, с какой целью это представление используется.

Некоторые ошибки при документировании требований. Перечислим ряд ошибок, встречающихся при составлении технических заданий и иных документов с требованиями.

1. Описание возможных решений вместо требований.
2. Нечеткие требования, которые не допускают однозначной проверки, оставляют недосказанности, имеют оттенок советов, обсуждений, рекомендаций: «Возможно, что имеет смысл реализовать также...», «и т.д.».
3. Игнорирование аудитории, для которой предназначено представление требований. Например, если спецификацию составляет инженер заказчика, то часто встречается переизбыток информации об оборудовании, с которым должна работать программная система, отсутствует глоссарий терминов и определений основных понятий, используются многочисленные синонимы и т.д. Или допущен слишком большой уклон в сторону программирования, что делает данную спецификацию непонятной всем непрограммистам.
4. Пропуск важных аспектов, связанных с нефункциональными требованиями, в частности, информации об окружении системы, о сроках готовности других систем, с которыми должна взаимодействовать данная. Последнее случается, например, когда данная программная система является ча-

стью более крупного проекта. Типичны проблемы при создании программно-аппаратных систем, когда аппаратура не успевает вовремя и ПО невозможно тестировать, а в сроках и требованиях это не предусмотрено...

Цикл работы с требованиями

В своде знаний по программной инженерии SWEBOK определяются следующие виды деятельности при работе с требованиями.

1. *Выделение требований*, нацеленное на выявление всех возможных источников требований и ограничений на работу системы и извлечение требований из этих источников.
2. *Анализ требований*, целью которого обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация.
3. *Описание требований*. В результате этой деятельности требования должны быть оформлены в виде структурированного набора документов и моделей, который может систематически анализироваться, оцениваться с разных позиций и в итоге должен быть утвержден как официальная формулировка требований к системе.
4. *Валидация требований,* которая решает задачу оценки понятности сформулированных требований и их характеристик, необходимых, чтобы разрабатывать ПО на их основе, в первую очередь, непротиворечивости и полноты, а также соответствия корпоративным стандартам на техническую документацию.

Выделение требований. Процесс первичного сбора требований является первым шагом в процессе создания требований. Сбор требований следует начинать с бизнес-требований, так как все остальные виды требований подчинены им.

Основной задачей этапа сбора бизнес-требований является выработка образа продукта. Образ продукта описывает, что сейчас представляет собой продукт и каким он станет впоследствии. Границы проекта показывают, к какой области долгосрочного образа продукта направлен текущий проект. Образ продукта подразумевает определение долгосрочных перспектив развития про-

екта, в то время как границы проекта могут быть определены для конкретной итерации. Следует очень внимательно отнестись к формированию образа и границ проекта. Это позволит избежать или, по крайней мере, запланировать на более позднее время реализацию менее критичных для успеха проекта возможностей ПО (известны ситуации, когда, войдя во вкус клиент просит сделать еще и вот это и это, при этом, не выделяя дополнительных ресурсов). Формирование образа и границ проекта позволяет руководству понять, за что именно они будут платить деньги, а разработчикам понять, чего хочет от них руководство.

Следующим этапом является *сбор пользовательских требований*, другими словами необходимо выяснить чего ожидают от разработчика непосредственные пользователи. Естественно, что пользовательские требования должны быть жестко подчинены бизнес-требованиям.

Практически любая информационная система имеет несколько категорий пользователей. Очень важно проанализировать потребности каждой из таких групп, достигнув компромисса между противоречащими требованиями различных групп пользователей. Необходимо четко определить приоритетность каждой группы и при разрешении противоречий между требованиями в первую очередь заботится об удовлетворении наиболее приоритетных групп.

Наиболее удачным подходом является оформление пользовательских требований в виде вариантов использования.

Основными ошибками, допускаемыми на этапе выделения требований, являются:

- нечеткое определение границ проекта, что приводит к затягиванию сроков проекта, перерасходу бюджета за счет включения в продукт второстепенных возможностей;
- недостаточно четко определенные группы пользователей продукта. В каждой группе не выделены представители, наиболее заинтересованные в продукте, готовые продуктивно сотрудничать с командой разработчиков;
- попытка с первого раза максимально детализировать и проанализировать все требования.

Чтобы не допустить этих ошибок, можно дать несколько рекомендаций:

– разделите пользователей продукта на категории в соответствии с их ролями;
– в каждой группе выберите одного-двух человек в качестве представителей интересов группы. Выбранные люди должны быть лидерами (пусть даже и неформальными) в своей группе, четко представлять бизнес-процессы, в которых участвует группа, и лояльно настроенными по отношению к проекту создания и внедрения разрабатываемой системы;
– не пытайтесь сразу и досконально описать все требования. Это приведет лишь к затягиванию работ. Требования должны прорабатываться и детализироваться в ходе проекта;
– каждому требованию присваивайте приоритетность. Это позволит реализовывать, в первую очередь, самые необходимые требования, откладывая реализацию второстепенных возможностей на более поздний срок. Как правило, достаточно ввести три – четыре градации приоритетности требований.

Анализ требований. Данный этап является самым продолжительным и наиболее важным. Он может состоять из нескольких итераций. Основные задачи этапа: детализация первичных требований, обеспечение непротиворечивости и полноты требований.

В результате анализа разработчики должны получить набор требований, охватывающий все области разработки приложения. Непротиворечивость требований означает однозначное понимание изложенных требований всеми участниками проекта, а также отсутствие противоречий между различными сформулированными требованиями.

Кроме того, каждому требованию необходимо назначить приоритетность и построить матрицу взаимных зависимостей требований. Назначение приоритетности позволяет выделить наиболее критичные для успеха проекта требования и запланировать их реализацию на более ранние сроки по сравнению с менее значимыми требованиями. Обычно шкала приоритетности требо-

ваний включает три- пять ступеней. Матрица зависимостей между требованиями позволяет построить наиболее оптимальный календарный график работ по реализации требований (критический путь реализации проекта). Построение матрицы связей требований преследует еще одну цель – обеспечение возможности анализа влияния вносимых изменений на проект.

При анализе и детализации требований основным приемом работы с участниками проекта является проведение «круглых столов», обсуждений, «мозговых штурмов», демонстраций предварительных версий разрабатываемой системы.

Практические рекомендации, которые позволяют сэкономить время и силы в ходе проведения анализа требований:

- четко очерчивайте круг обсуждаемых на встрече вопросов;
- заранее оцените время, которое потребуется;
- заранее ознакомьте участников обсуждения с повесткой и расписанием предстоящего мероприятия;
- не позволяйте одному из участников подавлять остальных, вовремя останавливайте, чтобы остальные могли высказать свою точку зрения на обсуждаемую проблему.

На круглые столы и обсуждения достаточно пригласить лишь представителей групп пользователей и разработчиков, которые непосредственно участвуют в решении обсуждаемых вопросов. На показы промежуточных версий полезно пригласить всех, с одной стороны, это позволяет пользователям увидеть и «пощупать» систему, а разработчикам – получить живые отклики. Как показывает практика, оптимальное количество участников «круглого стола» [4–8].

Важным моментом проведения обсуждений является их документирование и протоколирование. Это позволяет не держать постоянно в голове все важные детали проекта, а также отследить источники возникновения тех или иных требований. Довольно часто представители пользователей боятся подписывать протоколы обсуждений. Страх связан с ожиданием каких–либо карательных мер. Следует объяснить пользователям, что цель получения их подписи под протоколом не в том, чтобы потом наказать их, а всего лишь в том, чтобы убедиться, что между участниками обсуждения достигнуто взаимопонимание и выработано совместное решение.

Управление изменениями требований. В ходе развития проекта требования неоднократно корректируются и изменяются. Так как требования связаны друг с другом, изменения, вносимые в одно требование, могут повлиять на связанные с ним другие требования. Требования также являются основаниями для включения в продукт тех или иных функциональных возможностей, поэтому изменения в требованиях могут потребовать доработки или переработки продукта. Таким образом, для того чтобы проект не потерял управляемости и был создан в срок в рамках бюджета, необходимо выработать и утвердить процедуру внесения изменений в требования.

Для внесения и обсуждения изменений в требования, как правило, выделяется отдельный вид требований, называемый запросами на изменение. Создавать запросы на изменение могут все заинтересованные участники проекта, однако, до того как запрос не будет просмотрен, проанализирован и утвержден аналитиком требований. При анализе аналитик выявляет степень влияния предлагаемых доработок на требования, сроки реализации и бюджет. Затем внесенные изменения либо утверждаются, либо отвергаются.

Ключевые термины

Требование – это любое условие, которому должна соответствовать разрабатываемая система или программное средство.

Управление требованиями – процесс, включающий идентификацию, выявление, документирование, анализ, отслеживание, приоритезацию требований.

Функциональные требования – описание поведения и сервисов системы, ее функционала.

Бизнес-требования – это описание пожеланий (ожиданий) пользователей с точки зрения задач бизнеса.

Пользовательские требования – это описание пожеланий (ожиданий) непосредственных пользователей системы.

Нефункциональные требования – описывают такие характеристики системы, как: производительность, надежность, особенности поставки (наличие инсталлятора, документации), определенный уровень качества.

Краткие итоги

Важным этапом процесса создания программного обеспечения является работа с требованиями к системе. Требование представляет собой любое условие, которому должно соответствовать разрабатываемое программное средство. Требованием может быть возможность, которой система должна обладать, и ограничение, которому система должна удовлетворять. В процессе создания ПО необходимо управлять требованиями в силу их изменчивости. Управление требованиями представляет собой процесс, включающий идентификацию, выявление, документирование, анализ, отслеживание, приоритизацию требований, достижение соглашения по требованиям и затем управление изменениями и уведомление соответствующих заинтересованных лиц. Управление требованиями – непрерывный процесс на протяжении всего проекта разработки программного обеспечения. Для ПО различают функциональные и нефункциональные требования. Функциональные требования являются детальным описанием поведения и сервисов системы, ее функционала и делятся на бизнес-требования, пользовательские требования и собственно функциональные требования. Бизнес-требования описывают пожелания пользователей с точки зрения задач бизнеса. Пользовательские требования отражают пожелания непосредственных пользователей системы. Функциональные требования детально описывают функции, которые должно выполнять ПО для удовлетворения пользовательских требований. Нефункциональные требования не являются описанием функций системы, а описывают такие характеристики системы, как: производительность, надежность, особенности поставки, определенный уровень качества. Требования могут быть представлены в виде документа, модели, формальной спецификации, списка. При работе с требованиями выделяют следующие виды деятельности: выделение требований, анализ требований, описание требований, валидацию требований.

Вопросы для самопроверки

1. Проблемы формирования требований к ПО.
2. Причины изменчивости требований.

3. Виды требований, предъявляемых к программному обеспечению.
4. Свойства требований, предъявляемых к программному обеспечению.
5. Варианты формализации требований к программному обеспечению.
6. Характерные ошибки при документировании требований.
7. Цикл работы с требованиями.

7. КОНФИГУРАЦИОННОЕ УПРАВЛЕНИЕ

В проекте по разработке программного обеспечения учету и контролю требуют файлы проекта.

Файл – это виртуальная информационная единица, у которой может быть много версий. *Версия файла* – отражает текущее его состояние и одна версия от другой может отличаться несколькими строчками кода или полностью обновленным содержанием.

В программных проектах необходима специальная деятельность по поддержанию файловых активов проекта в актуальном состоянии, которая называется *конфигурационным управлением*.

В конфигурационном управлении выделяют две основные задачи – *управление версиями* и *управление сборками*. Первая отвечает за управление версиями файлов и выполняется в проекте на основе специальных программных пакетов – *средств версионного контроля*. Существует большое количество таких средств – Microsoft Visual SourceSafe, IBM ClearCase, cvn, subversion и др.

Управление сборками – это автоматизированный процесс трансформации исходных текстов ПО в пакет исполняемых модулей, учитывающий многочисленные настройки проекта, компиляции, и интегрируемый с процессом автоматического тестирования. Эта процедура является мощным средством интеграции проекта, основой итеративной разработки.

Единицы конфигурационного управления

Конфигурационное управление имеет дело с меняющимися в процессе проектирования наборами файлов, которые называют *единицами конфигурационного управления,* например:

- пользовательская документация;
- проектная документация;
- исходные тексты ПО;
- пакеты тестов;
- инсталляционные пакеты ПО;
- тестовые отчеты.

Единицы конфигурационного управления характеризуются следующим:

1. *Структура* – набор файлов. Например, пользовательская документация в html должна включать индекс-файл и набор

html-файлов, а также набор вынесенных картинок (gif или jpeg-файлы). Эта структура должна быть хорошо определена и отслеживаться при конфигурационном управлении – что все файлы не потеряны и присутствуют, имеют одинаковую версию, корректные ссылки друг на друга и т.д.

2. *Ответственное лицо* и, возможно, группу тех, кто их разрабатывает, а также более широкую и менее ответственную группу тех, кто пользуется этой информацией. Например, определенной программной компонентой могут в проекте пользоваться многие разработчики, но отвечать за ее разработку, исправление ошибок должен кто-то один.
3. *Практика конфигурационного управления* – кто и в каком режиме, а также в какое место выкладывает новую версию элемента конфигурационного управления в средство управления версиями, правила именования и комментирования элемента в этой версии, дальнейшие манипуляции с ним. Более высокоуровневые правила, связанные, например, с правилами изменения тестов и тестовых пакетов при изменении кода.
4. *Автоматическая процедура контроля целостности элемента* – например, сборка для исходных текстов программ. Это свойство не относится к элементам конфигурационного управления, например, может не быть у документации, тестовых пакетов.

Элементы конфигурационного управления могут образовывать иерархию. Пример представлен на рис. 7.1.

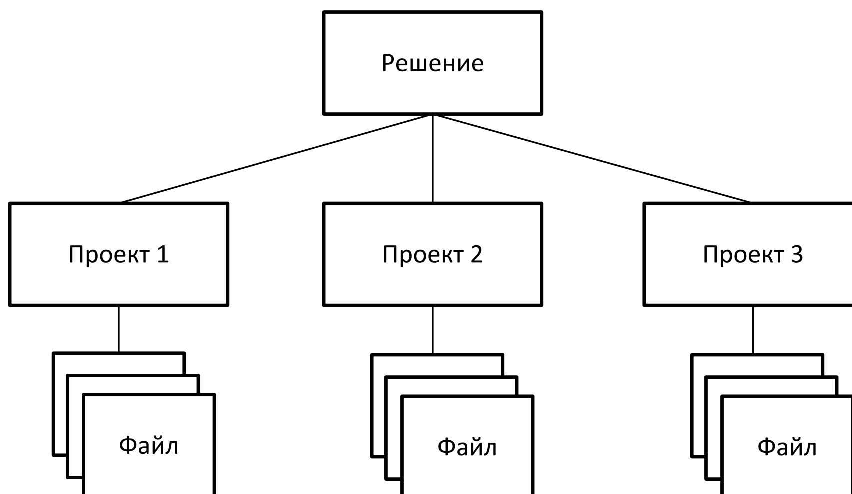

Рисунок 7.1 *– Элементы конфигурационного управления*

Управление версиями

Управление версиями файлов. В процессе разработки ПО создается большое количество файлов. Файлы могут одновременно использоваться различными членами команды. Для согласованной работы над проектом файлы должны быть актуальными. Поэтому необходима постоянная работа по управлению версиями файлов.

Управление версиями составных конфигурационных объектов. Понятие «ветки» проекта. Одновременно может существовать несколько версий системы – и для разных заказчиков, и в одном проекте одного заказчика, но как разные наборы исходных текстов. Для этого при управлении версиями используют разные *ветки*.

Каждая ветка содержит полный образ исходного кода и других артефактов, находящихся в системе контроля версий. Каждая ветвь может развиваться независимо, а может в определенных точках интегрироваться с другими ветвями. В процессе интеграции изменения, произведенные в одной из ветвей, полуавтоматически переносятся в другую. В качестве примера можно рассмотреть следующую структуру разделения проекта на ветки:

– master – ветвь, соответствующая основному направлению развития проекта. По мере созревания именно от этой ветви отходят ветви готовящихся релизов;
– v1.0 – ветвь, соответствующая выпущенному релизу. Внесение изменений в такие ветви запрещено, и они хранят образ кода системы на момент выпуска релиза;
– fix v1.0.1 – ветвь, соответствующая выпущенному пакету исправлений к определенной версии. Подобные ветви ответвляются от исходной версии, а не от основной ветви и замораживаются сразу после выхода пакета исправлений;
– upcoming (V1.1) – ветвь, соответствующая релизу, готовящемуся к выпуску и находящемуся в стадии стабилизации. Для таких ветвей, как правило, действуют более строгие правила и работа в них ведется более формально;
– experiment1 – ветвь, созданная для проверки некоторого технического решения, перехода на новую технологию, или

внесения большого пакета изменений, потенциально нарушающих работоспособность кода на длительное время. Такие ветви, как правило, делаются доступными только для определенного круга разработчиков и удаляются по завершении работ после интеграции с основной веткой.

Системы контроля версий

Система контроля версий (СКВ) – это система, регистрирующая изменения в одном или нескольких файлах, с тем чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. В СКВ целесообразно помещать исходные коды программ, но под версионный контроль можно поместить файлы практически любого типа.

СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. СКВ позволяет восстановить испорченные или потерянные файлы.

СКВ позволяют контролировать версии в локальной базе данных, в которой хранятся все изменения нужных файлов (рис. 7.2).

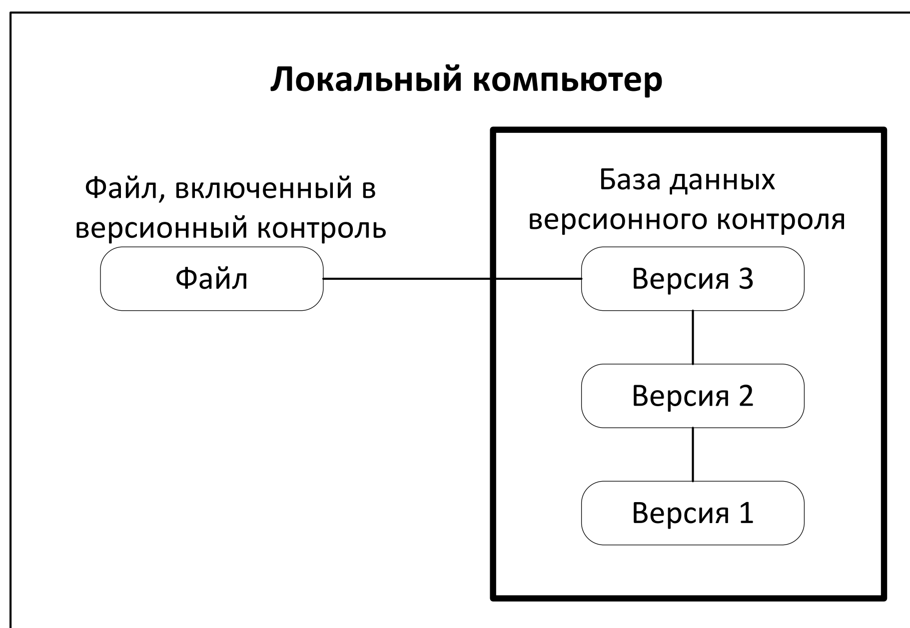

***Рисунок 7.2** – Схема локальной СКВ*

Централизованные системы контроля версий (ЦСКВ) предназначены для совместного использования кода командой разработчиков. В таких системах, например, CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него (рис. 7.3).

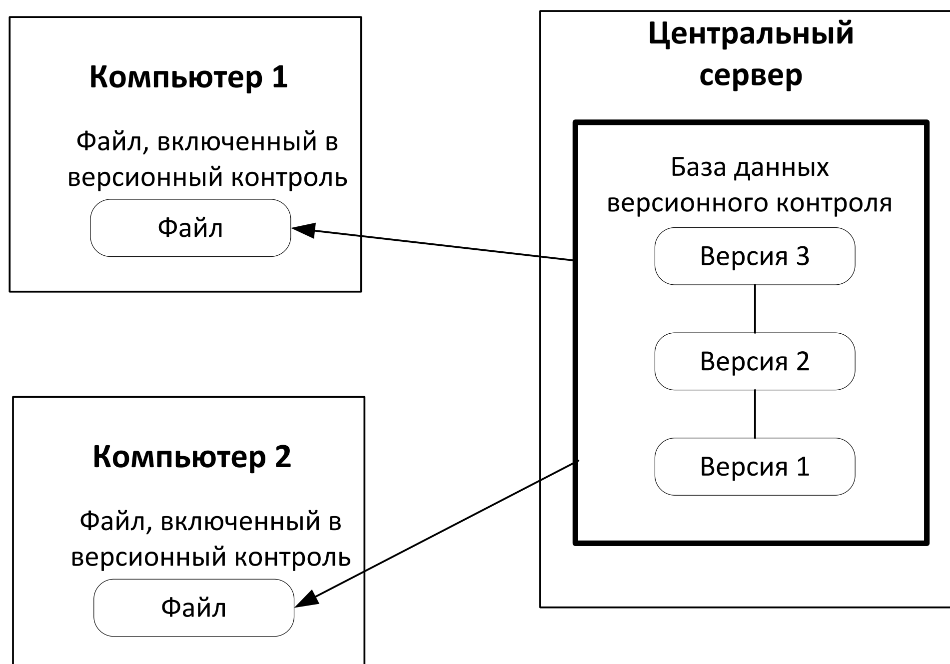

***Рисунок 7.3** – Схема централизованного контроля версий*

Система централизованного контроля версий имеет ряд преимуществ по отношению к локальной СКВ:

- доступ к СКВ нескольких членов команды разработчиков;
- прозрачность состояния проекта для заинтересованных лиц;
- фиксация и мониторинг всех изменений файлов проекта для всех членов команды разработчиков.

В то же время ЦСКВ имеет существанный недостаток – централизованный сервер является критичным ресурсом, и при выходе его из строя все разработчики теряют возможность доступа к информации по версиям файлов проекта.

В *распределённых системах контроля версий* (РСКВ), таких системах как: Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда выходит из строя сервер, через который шла работа, любой клиентский репо-

зиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных (рисунок 7.4).

В Visual Studio Team Foundation Server, начиная с версии 2013 и Visual Studio OnLine, имеется возможность использовать две системы управления версиями: TFVS и Git.

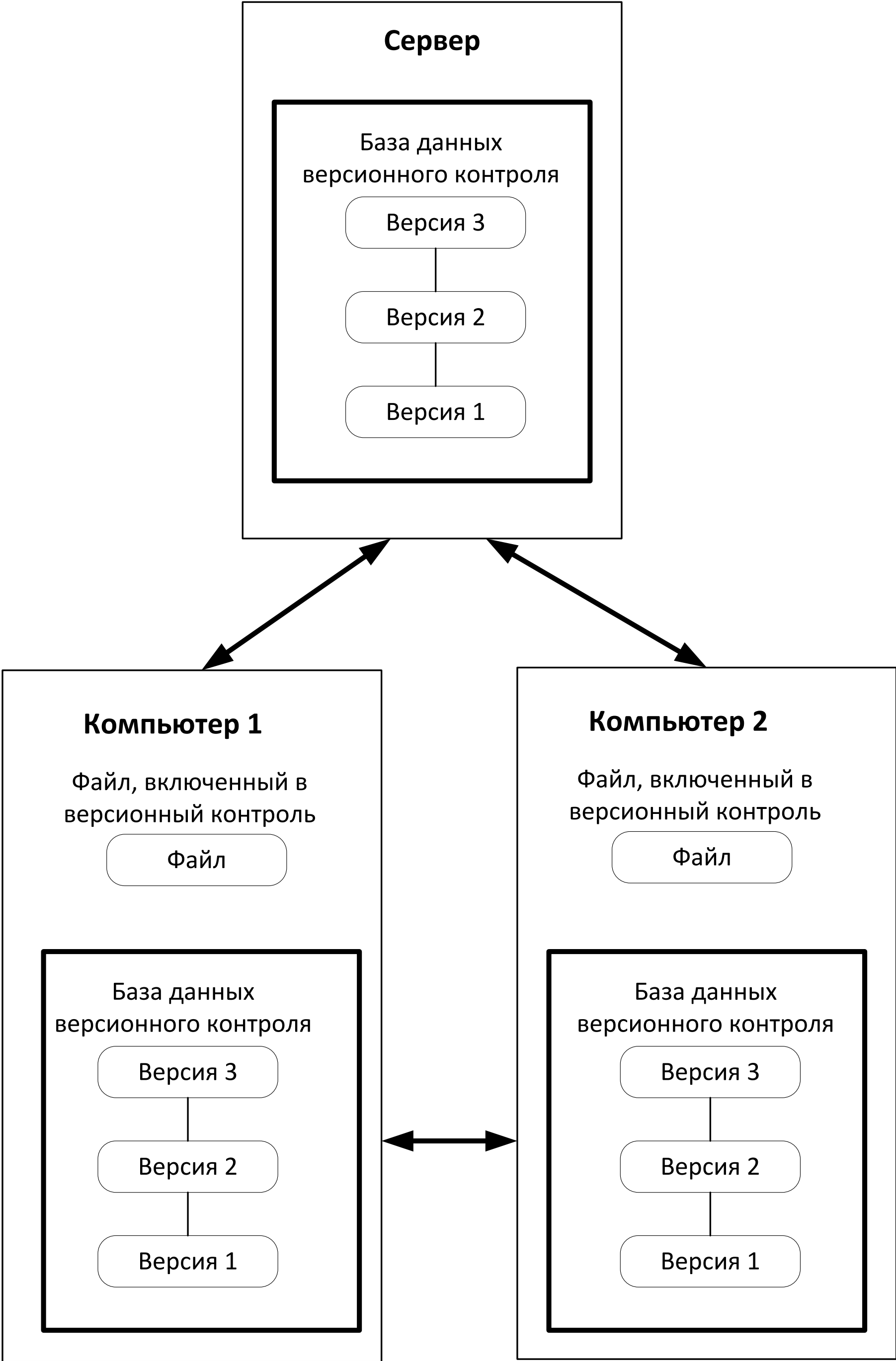

Рисунок 7.4 *– Схема распределённой системы контроля версий*

TFVS представляет решения по контролю версий исходного кода корпоративного уровня: отслеживание изменений исходного кода, атомарные возвраты, ветвление, объединение и отложенные изменения исходного кода, политики и защиту кода.

Основы Git

Главное отличие Git от любых других СКВ (например, Subversion и ей подобных) – это то, как Git хранит информацию об изменениях данных. Большинство систем хранит информацию как список изменений (патчей, например Δ1, Δ2) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar) формируют хранимые данные как набор файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке 7.5.

Рисунок 7.5 *– Хранение данных как изменения к базовой версии для каждого файла*

В Git хранимые данные представляют собой набор слепков небольшой файловой системы. При фиксации текущей версии проекта, Git сохраняет все файлы проекта на текущий момент. Если файл не менялся, то в Git он не сохраняется заново, а делается ссылка на ранее сохранённый файл (рис. 7.6).

Это важное отличие Git'а от практически всех других систем контроля версий. Git больше похож на небольшую файловую систему с мощными инструментами, работающими поверх неё.

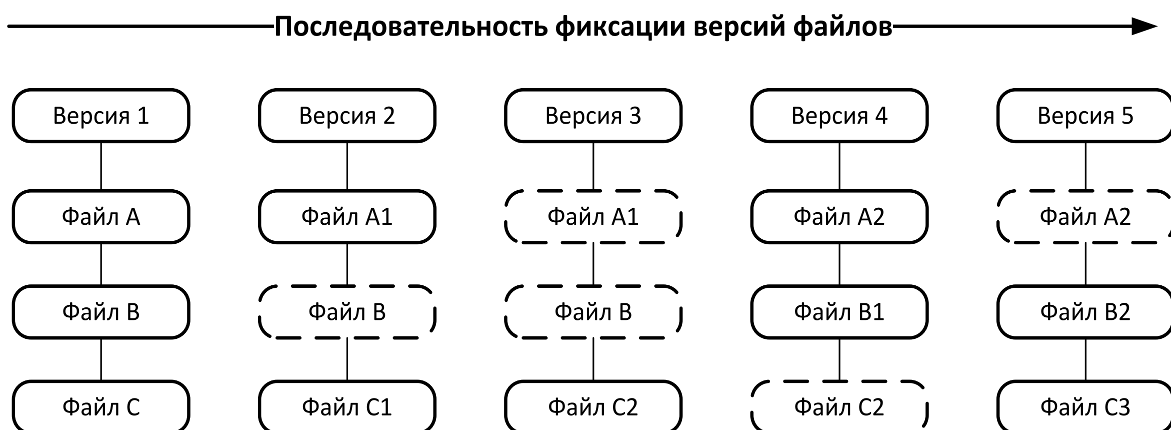

Рисунок 7.6 *– Git хранит данные как слепки состояний проекта во времени*

Для совершения большинства операций в Git необходимы только локальные файлы и ресурсы. К примеру, чтобы показать историю проекта, Git не нужно скачивать её с сервера, он просто читает её прямо из локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу в локальном репозитории, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к серверу. Пользователь может работать локально (of-line), фиксируя изменения в локальном репозитории, а при подключении к сети передать изменения в центральный репозиторий на сервере.

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

При фиксации изменений в базу данных Git данные только добавляются, а не изменяются. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в

любой другой СКВ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

В Git файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. «Зафиксированный» значит, что файл уже сохранён в вашей локальной базе. К «изменённым» относятся файлы, которые пользователь изменил, но ещё не зафиксировал эти изменения. «Подготовленные» файлы – это изменённые файлы, отмеченные для включения в следующую фиксацию файлов (commit).

Таким образом, в проектах, использующих Git, есть три части: каталог Git (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area) (рис. 7.7).

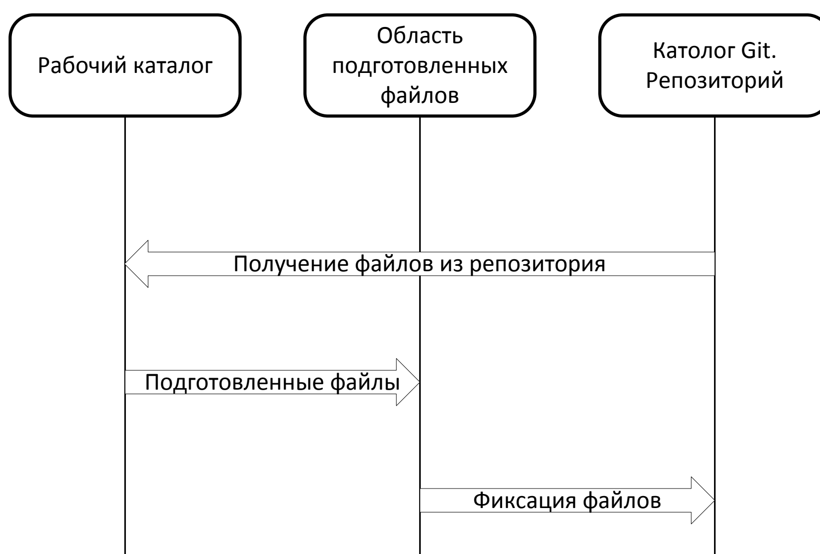

Рисунок 7.7 *– Рабочий каталог, область подготовленных файлов, каталог Git*

Каталог Git – это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог – это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git и помещаются на диск, для того чтобы вы их просматривали и редактировали.

Область подготовленных файлов – это обычный файл, хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующую фиксацию файлов. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Стандартный рабочий процесс с использованием Git выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете фиксацию файлов, которая берёт подготовленные файлы из индекса и помещает их в каталог Git на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из базы данных, но не был подготовлен, то он считается изменённым.

Главной веткой проекта в Git является ветка master. В процессе разработки можно создавать новые ветки, которые впоследствии могут быть объединены с главной веткой (рис. 7.8).

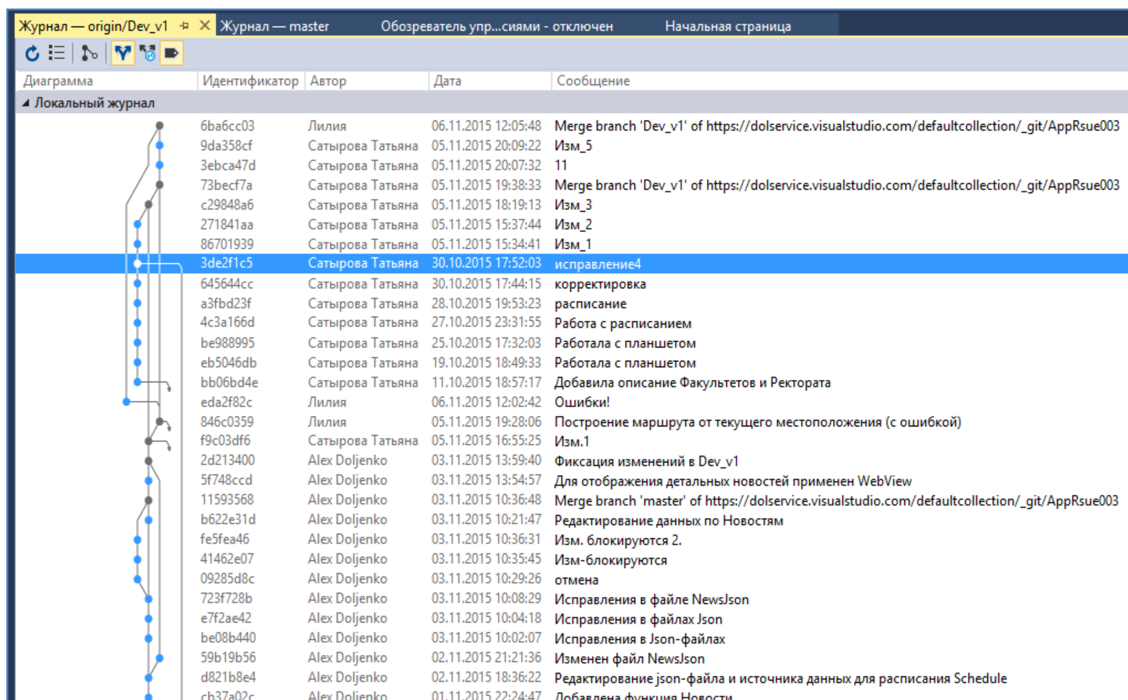

***Рисунок 7.8** – История проект в Git*

Управление сборками

Процедура компиляции и создания exe, dll файлов по исходникам проекта является важным рабочим процессом проекта разработки программного обеспечения. Данная процедура многократно в день выполняется каждым разработчиком на его собственном компьютере, с его собственной версией проекта. Имеются определенные особенности этого процесса:

- набор подпроектов, собираемых разработчиком; он может собирать не весь проект, а только какую-то его часть; другая часть либо им не используется вовсе, либо не пересобирается очень давно, а по факту она давно изменилась;
- отличаются параметры компиляции.

При этом если не собирать регулярно итоговую версию проекта, то общая интеграция может выявить много разных проблем:

- несоответствие друг другу различных частей проекта;
- наличие специфических ошибок, возникших из-за того, что отдельные проекты разрабатывались без учета параметров компиляции (в частности, переход в Visual Studio с debug на release версию часто сопровождается появлением многочисленных проблем).

Процедуры у сборки проекта часто автоматизируют и выполняют не из среды разработки, а из специального скрипта – build-скрипта. Этот скрипт используется тогда, когда разработчику требуется полная сборка всего проекта. А также он используется в процедуре *непрерывной интеграции* (continues integration), то есть регулярной сборке всего проекта (как правило – каждую ночь). Как правило, процедура непрерывной интеграции включает в себя и регрессионное тестирование, и часто – создание инсталляционных пакетов. Общая схема автоматизированной сборки представлена на рис. 7.9.

Рисунок 7.9 *– Общая схема автоматизированной сборки*

Тестировщики должны тестировать по возможности итоговую и целостную версию продукта, так что результаты регулярной сборки оказываются очень востребованы. Кроме того, наличие базовой, актуальной, целостной версии продукта позволяет организовать разработку в итеративно-инкрементальном стиле, то есть на основе внесения изменений. Такой стиль разработки называется baseline-метод.

Понятие baseline

Baseline – это базовая, последняя целостная версия некоторого продукта разработки, например, документации, программного кода и т.д. Подразумевается, что разработка идет не сплошным потоком, а с фиксацией промежуточных результатов в виде текущей официальной версии разрабатываемого актива. Принятие такой версии сопровождается дополнительными действиями по оформлению, сглаживанию, тестированию, включению только законченных фрагментов и т.д. Это результат можно посмотреть, отдать тестировщику, передать заказчику и т.д. Baseline служит хорошим средством синхронизации групповой работы.

Baseline может быть совсем простой – веткой в средстве управления версиями, где разработчики хранят текущую версию своих исходных кодов. Единственным требованием в этом случае может быть лишь общая компилируемость проекта.

Baseline может также поддерживаться непрерывной интеграцией. Важно, что Baseline (особенно в случае с программными активами) не должна устанавливаться слишком рано. Сначала нужно написать какое-то количество кода, чтобы было что интегрировать. Кроме того, вначале много внимания уделяется разработке основных архитектурных решений, и целостная версия оказывается невостребованной. Но начиная с какого-то момента она просто необходима. Какой этот момент – решать членам команды. Наконец, существуют проекты, где автоматическая сборка не нужна вовсе – это простые проекты, разрабатываемые небольшим количеством участников, где нет большого количества исходных текстов программ, проектов, сложных параметров компиляции.

Ключевые термины

Файл – это виртуальная информационная единица, у которой может быть много версий.

Версия файла – отражает текущее его состояние.

Управление версиями – деятельность по сохранению, отслеживанию и получению файлов в процессе разработки программного обеспечения.

Управление сборками – это автоматизированный процесс трансформации исходных текстов ПО в пакет исполняемых модулей.

Система контроля версий – это система, регистрирующая изменения в одном или нескольких файлах, с тем чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов.

Baseline – это базовая, последняя целостная версия файлов некоторого продукта разработки.

Краткие итоги

Конфигурационное управление ориентировано на решение задач управление версиями файлов и управление сборками проекта. Управление версиями файлов выполняется средствами версионного контроля. Управление сборками представляет собой автоматизированный процесс трансформации исходных текстов ПО в пакет исполняемых модулей, учитывающий многочисленные настройки проекта, компиляции, и интегрируемый с процессом автоматического тестирования. Конфигурационное управление работает с единицами конфигурационного управления, такими как: пользовательская документация, проектная документация, исходные тексты ПО, пакеты тестов, инсталляционные пакеты ПО и тестовые отчеты. Система контроля версий регистрирует изменения в одном или нескольких файлах, с тем чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. В неё целесообразно помещать исходные коды программ. Системы контроля версий бывают локальные, централизованные и распределенные. В настоящее время широкое распространение имеет распределенная система контроля версий Git, которая хранит данные как набор файлов и измене-

ний, сделанных для каждого из этих файлов во времени. Управление сборками автоматизирует процедуру компиляции и создания exe, dll файлов по исходникам проекта и является важным рабочим процессом проекта разработки программного обеспечения. В процессе разработки сложных программных систем используется понятие «базовая линия», которая представляет собой последнюю целостную версию некоторого продукта разработки, например, документации, программного кода и т.д., при этом подразумевается, что разработка идет не сплошным потоком, а с фиксацией промежуточных результатов в виде текущей официальной версии разрабатываемого актива.

Вопросы для самопроверки

1. Назначение конфигурационного управления программными продуктами.
2. Единицы конфигурационного управления.
3. Основные характеристики единицы конфигурационного управления.
4. Управление версиями файлов.
5. Управления сборками программной системы.
6. Понятие базовой линии (baseline).
7. Управление конфигурацией программной системы.
8. Функции процесса Управления конфигурацией.
9. Управление релизами программного продукта.
10. Классификация релизов по масштабу.
11. Классификация релизов по способу реализации.
12. Функции процесса управления релизами.

8. ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПРОГРАММНЫХ ПРОДУКТОВ

Управление качеством

Стандартизация в современном бизнесе и промышленности. Развитие мирового рынка привело к тому, что многие товары и услуги стали распространяться по всему миру, стали развиваться глобальные сервисы, в частности, телекоммуникационные, банковские. Для того чтобы устранить технические барьеры в промышленности, торговле и бизнесе, которые возникли вследствие того, что в разных странах для одних и тех же технологий и товаров действовали разнородные стандарты, стали создаваться национальные и международные комитеты по стандартизации. Самыми известными международными комитетами являются следующие.

1. ITU (International Telecommunication Union) – комитет образован в 1865 г. Штаб-квартира находится в Женеве (Швейцария), а ITU является частью ООН. Его основная задача – стандартизация телекоммуникационных протоколов и интерфейсов с целью поддержания и развития глобальной мировой телекоммуникационной сети. Самыми известными стандартами ITU являются:

- ISDN (цифровая телефонная связь, объединяющая телефонные сервисы и передачу данных);
- ADSL (широко известная модемная технология, позволяющая использовать телефонную линию для выхода в Интернет, не блокируя при этом обычного телефонного сервиса);
- OSI (модель открытого 7-уровневого сетевого протокола, на которой базируются все современные стандартные сетевые интерфейсы и протоколы; также является стандартом ISO);
- языки визуального проектирования телекоммуникационных систем, SDL и MSC, влившиеся позднее в UML.

Многие стандарты ITU переводятся на русский язык и превращаются в российские стандарты в виде ГОСТов.

2. ISO (International Organization for Standardization) – организация создана в 1946 г. Цель – содействие развитию стандарти-

зации, а также смежных видов деятельности в мире с целью обеспечения международного обмена товарами и услугами, способствование и развитие сотрудничества в интеллектуальной, научно-технической и экономической областях. К настоящему времени создано около 17 000 стандартов в самых разных областях промышленности – продовольственные и иные товары, различное оборудование, банковские сервисы и т.д. Вот некоторые стандарты:

- серия стандартов ISO 9000. Направлены на стандартизацию качества товаров и услуг. Определение качества, определение системы поддержки качества на всех жизненных фазах изделия, товара, услуги (проектирование, разработка, коммерциализация, установка и обслуживание), описание процедур по улучшению деятельности компании, промышленного производства;
- ISO/IEC 90003:2004 – адаптация стандартов ISO 9000 к производству ПО в русле обеспечения качества в жизненном цикле ПО;
- ISO 9126:2001 – определение качественного ПО и различных атрибутов, описывающих это качество.

Многие стандарты ISO переводятся на русский язык и превращаются в российские стандарты в виде ГОСТов. Имеется много стандартов в области информационных технологий, а также несколько – в области программной инженерии. На соответствие стандартам ISO существует сертификация. В частности, компании сертифицируются на соответствие стандартам ISO 9000, то есть на качественный процесс разработки ПО.

3. ETSI (European Telecommunications Standards Institute) – организация создана в 1988 г., штаб-квартира находится в г. Софии (Антиполис) (Франция). Является независимой, некоммерческой организацией по стандартизации в телекоммуникационной промышленности (изготовители оборудования и операторы сети) в Европе. Самые известные стандарты – GSM, система профессиональной мобильной радиосвязи TETRA.

Остановимся теперь на ряде комитетов, непосредственно связанных с разработкой ПО.

1. SEI (Software Engineering Institute) – организация создана в 1984 год на базе университета Карнеги-Меллон в г. Питс-

бурге (США). Инициатор и главный спонсор – Министерство обороны США. Основная задача – стандартизация в области программной инженерии, выработка критериев для сертификации надежных и зрелых компаний (что в первую очередь интересует Минобороны США для выполнения его заказов). Самые известные продукты – стандарт CMM, CMMI, разработки в области семейства программных продуктов (product lines). Эти продукты шагнули далеко за пределы военных разработок США, их использование и развитие стали международной деятельностью. Некоторые продукты SEI стандартизованы также ISO. На соответствие CMM/CMMI проводится сертификация.

2. IEEE (Institute of Electrical and Electronics Engineers) – организация создана в 1963 г. Ведет историю с конца XIX века, в контексте промышленной стандартизации в США. IEEE является международной некоммерческой ассоциацией специалистов в области техники, мировой лидер в области разработки стандартов по радиоэлектронике и электротехнике. Штаб-квартира в США, существуют многочисленные подразделения в разных странах, включая Россию. IEEE издаёт третью часть мировой технической литературы, касающейся применения радиоэлектроники, компьютеров, систем управления, электротехники, в том числе (январь 2008) 102 реферируемых научных журнала и 36 отраслевых журналов для специалистов, проводит в год более 300 крупных конференций, принимала участие в разработке около 900 действующих стандартов.
3. OMG (Object Management Group) – организация создана в 1989 г. группой американских IT-компаний (в том числе Hewlett Packard, Sun Microsystems, Canon). В настоящий момент включает около 800 компаний членов. Основное направление – разработка и продвижение объектно ориентированных технологий и стандартов, в том числе для создания платформенно-независимых программных приложений уровня предприятий. Известные стандарты CORBA, UML, MDA.

Все эти комитеты и организации включают программную инженерию в сферу своей деятельности, сотрудничают, выпускают совместные стандарты, используют наработки друг друга и т.д.

Стандартизация качества. С точки зрения тестирования ПО нас интересует в этих стандартах стандартизация качества (как контекст тестирования) – сначала выпускаемой продукции, а потом и процессов по ее разработке. Следует отметить, что качественного результата не создать без качественного процесса. Обеспечение качества является более общим контекстом для тестирования.

Качество продукта или сервиса, предназначенного потребителю, определяется в стандарте ISO 9000:2005 как степень соответствия его характеристик требованиям обязательным или подразумеваемым.

Методы обеспечения качества ПО. Основные способы контроля качества, используемые на практике при разработке ПО:

– *наладка качественного процесса,* другими словами, совершенствование процесса. Для комплексного улучшения процессов в компании компаниями-разработчиками ПО используются стандарты CMM/CMMI, а также по стандартам серии ISO 9000 (с последующей официальной сертификацией);
– *формальные методы* – использование математических формализмов для доказательства корректности, спецификации, проверки формального соответствия, автоматической генерации (доказательство правильности работы программ, проверка на моделях определенных свойств, статический анализ кода по дереву разбора программы, модельно ориентированное тестирование, автоматическая генерация тестов и тестового окружения по формальным спецификациям требований к системе). На практике применяются ограниченно из-за необходимости серьезной математической подготовки пользователей, сложности в освоении, большой работы по развертыванию. Эффективны для систем, имеющих повышенные требования к надежности. Также имеются случаи эффективного использования средств, основанных на этих методах, в руках высококвалифицированных специалистов;
– *исследование и анализ динамических свойств ПО.* Например, широко используется профилирование – исследование использования системой памяти, ее быстродействие и другие характеристики путем запуска и непосредственных наблю-

дений в виде графиков, отчетов. В частности, этот подход используется при распараллеливании программ, при поиске «узких» мест;

– *обеспечение качества кода.* Сюда относится целый комплекс различных мероприятий и методов. Разработка стандартов оформления кода в проекте и контроль за соблюдением этих стандартов. Сюда входят правила на создание идентификаторов переменных, методов и имен классов, на оформление комментариев, правила использования стандартных для проекта библиотек. Регулярный рефакторинг для предотвращения образования из кода «вермишели». Существует тенденция ухудшения структуры кода при внесении в него новой функциональности, исправления ошибок. Различные варианты инспекции кода, например, техника peer code review. Последняя заключается в том, что код каждого участника проекта (выборочно) читается и обсуждается на специальных встречах (code review meetings), и делается это регулярно. Практика показывает, что в целом код улучшается;
– *тестирование.* Самый распространенный способ контроля качества ПО, представленный фактически в каждом программном проекте.

Качество программного обеспечения

Качество программного продукта определяется по нескольким критериям. Качественный программный продукт должен отвечать функциональным и нефункциональным требованиям, в соответствии с которыми он создавался, иметь ценность для бизнеса, отвечать ожиданиям пользователей.

В жизненном цикле управления приложениями качество должно отслеживаться на всех этапах жизненного цикла ПО. Оно начинает формироваться с определения необходимых требований. При задании требований необходимо указывать желаемую функциональность и способы проверки её достижения.

Качественный программный продукт должен обладать высоким потребительским качеством, независимо от области применения: внутреннее использование разработчиком, бизнес, наука и

образование, медицина, коммерческие продажи, социальная сфера, развлечения, веб и др. Для пользователя программный продукт должен удовлетворять определенному уровню его потребностей.

Важным аспектом создания качественного ПО является обеспечение нефункциональных требований, таких, как: удобство в эксплуатации, надежность, производительность, защищенность, удобство сопровождения. Надежность ПО определяет способность без сбоев выполнять заданные функции в заданных условиях и в течение заданного отрезка времени. Производительность характеризуется временем выполнения заданных трансакций или длительных операций. Защищенность определяет степень безопасности системы от повреждений, утраты, несанкционированного доступа и преступной деятельности. Удобство сопровождения определяет легкость, с которой обслуживается продукт в плане простоты исправления дефектов, внесения корректив для соответствия новым требованиям, управления измененной средой.

Управление жизненным циклом программного продукта помогает разработчикам целенаправленно добиваться создания качественного ПО, избегать потерь времени на переделку, повторное проектирование и перепрограммирование ПО.

Тестирование

Тестирование – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением в специально заданных, искусственных условиях.

Ожидаемое поведение программы. Исходной информацией для тестирования является знание о том, как система должна себя вести, то есть требования к ней или к ее отдельной части. Самым распространенным способом тестирования является тестирование методом *черного ящика*, то есть когда реализация системы недоступна тестировщику, а тестируется только ее интерфейс. Часто это закрепляется и организацией коллектива – тестировщики оказываются отдельными сотрудниками и в некоторых компаниях они даже принципиально не общаются с разработчиками, чтобы минимально знать реализационные детали и максимально полно выступить в роли проверяющей инстанции. Существует тестиро-

вание методом *белого ящика*, когда код программ доступен тестировщику и используется в качестве источника информации о системе. Его схема представлена на рис. 8.1.

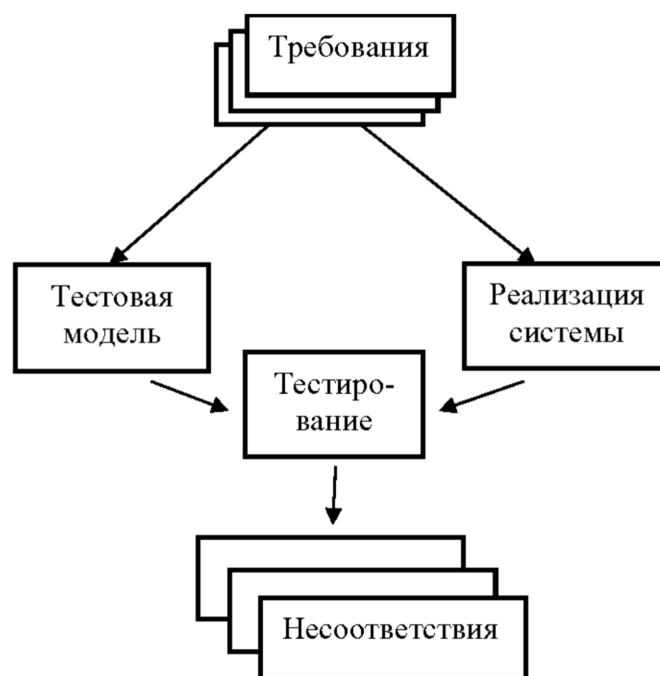

Рисунок 8.1 *– Тестирование методом белого ящика*

На этом рисунке видно, что на основе требований к системе создается реализация и тестовая модель системы. Тестирование есть сопоставление двух этих представлений с целью выявить их несоответствия. Чем независимее друг от друга будут эти представления, тем более информативно их сопоставление. Иначе если тестировщики существенно используют информацию о реализации системы при составлении тестов, то они могут невольно внести в тесты ошибки реализации. Найденное при тестировании несоответствие – это еще не ошибка, поскольку сами тестировщики могли неправильно понять требования, в тестах и средствах тестирования могли быть ошибки.

Данный подход закрепляется также и в организации коллективов программистов – тестировщики, как правило, отделены от разработчиков. Это разные люди, несовместимые роли в MSF.

Тесты могут быть «ручными» и автоматизированными. «Ручной» тест – это последовательность действий тестировщика, которую он (или разработчик) может воспроизвести, и ошибка

произойдет. Как правило, в средствах контроля ошибками такие последовательности действий содержатся в описании ошибки. Автоматический тест – это некоторая программа, которая воздействует на систему и проверяет то или иное ее свойство. Автоматический тест, по сравнению с «ручным», можно легко воспроизводить без участия человека. Можно создавать наборы тестов и прогонять их часто, например, в режиме регрессионного тестирования. Кроме того, автоматические тесты можно генерировать по более высокоуровневым спецификациям, например, по формально описанным требованиям к системе. А, например, тесты для компиляторов можно генерировать по формальному описанию языка программирования.

Таким образом, преимущества автоматических тестов перед «ручными» очевидны. В то же время автоматическому тестированию присущ ряд проблем.

Во-первых, для того, чтобы тесты автоматически запускать, нужны соответствующие программные продукты, которые также являются неотъемлемой частью специально заданных, искусственных условий. Их будем называть *инструментами тестирования*. В их задачу входит запуск теста на системе, «прогон» целого пакета тестов, а также анализ получившихся результатов и их обработка.

Кроме того, немаловажной задачей инструментов тестирования является обеспечение доступа теста к системе через некоторый ее интерфейс. Доступ к системе может оказаться затруднительным, например, в силу политических обстоятельств, когда сторонними разработчиками делается подсистема некоторой стратегической системы, и доступ к этой объемлющей системе у разработчиков сильно ограничен. Или в силу аппаратных ограничений – трудно «залезть» на «железку», где работает целевой код системы.

Кроме того, часто трудно «бесшовно» тестировать систему, оказывая на нее минимальное воздействие и добираясь при этом до всех аспектов ее функционирования. В целом настройка и развертка готовых, сторонних тестовых инструментов часто оказываются дорогостоящей и непростой задачей. Разработка своих собственных тесовых инструментов также непроста.

Во-вторых, часто возникает проблема ресурсов для автоматического тестирования. Особенно при автоматической генерации тестов: часто есть возможность автоматически сгенерировать очень большое количество тестов, так что если их еще выполнять регулярно, в режиме непрерывной интеграции, то не хватит имеющихся системных ресурсов. При этом качество тестирования может оказаться неудовлетворительным – ошибки находятся редко или вообще не находятся. Дело в том, что количество всех возможных состояний программной системы очень велико, и тестирование не может покрыть их все. На практике, в реальных проектах, определяют *критерии тестирования*, которые определяют ту «планку» качества, которую необходимо достичь в этом проекте. Ведь хорошее качество стоит дорого и очевидно, что разное ПО имеет разное качество, например, система управления ядерным реактором и текстовый редактор. На практике часто качество ПО определяется бюджетом проекта по его разработке. Далее, в силу ограниченности ресурсов на тестирование часто целесообразно бывает определить те аспекты ПО, которые наиболее важны – как для общей работоспособности системы, так и для заказчика. Например, при тестировании веб-приложения, предоставляющего услугу по созданию объявлений о продаже недвижимости, такими критериями были:

- правильность переходов сложного мастера – в частности, в связи с возможностью переходов назад;
- целостность введенных пользователем данных о создаваемых объявлениях.

Наконец, кроме ограничения количества тестов их отбора, важным является их прогон на некоторых входных данных. Часто здесь применяют принцип *факторизации* – множество всех возможных входных значений разбивают на значимые с точки зрения тестирования классы и «прогоняют» тесты не на всех возможных входных значениях, а берут по одному набору значений из каждого класса. Например, тестируют некоторую функцию системы на ее граничные значения – очень большие значения параметров, очень маленькие и пр. Часто факторизацию удобно делать, исходя из требований к данной функции, также бывает полезно посмотреть на ее реализацию и «пройтись» тестами по раз-

ным ее логическим веткам (порождаемым, например, условными операторами).

Виды тестирования. Приведем основные следующие виды тестирования [11, 13]:

– *модульное тестирование* – тестируется отдельный модуль, в отрыве от остальной системы. Самый распространенный случай применения – тестирование модуля самим разработчиком, проверка того, что отдельные модули, классы, методы делают действительно то, что от них ожидается. Различные среды разработки широко поддерживают средства модульного тестирования – например, популярная свободно распространяемая библиотека для Visual Studio NUnit, JUnit для Java и т.д. Созданные разработчиком модульные тесты часто включаются в пакет регрессионных тестов и таким образом, могут запускаться многократно;
– *интеграционное тестирование* – два и более компонентов тестируются на совместимость. Это очень важный вид тестирования, поскольку разные компоненты могут создаваться разными людьми, в разное время, на разных технологиях. Этот вид тестирования, безусловно, должен применяться самими программистами, чтобы, как минимум, удостовериться, что все живет вместе в первом приближении. Далее тонкости интеграции могут исследовать тестировщики. Необходимо отметить, что такого рода ошибки – «ошибки на стыках» – непросто обнаруживать и устранять. Во время разработки все компоненты все вместе не готовы, интеграция откладывается, а в конце обнаруживаются трудные ошибки (в том смысле, что их устранение требует существенной работы). Здесь выходом является ранняя интеграция системы и в дальнейшем использование практики постоянной интеграции;
– *системное тестирование* – этот тестирование всей системы в целом, как правило, через ее пользовательский интерфейс. При этом тестировщики, менеджеры и разработчики акцентируются на том, как ПО выглядит и работает в целом, удобно ли оно, удовлетворяет ли оно ожиданиям заказчика. При этом могут открываться различные дефекты, такие как

неудобство в использовании тех или иных функций, забытые или «скудно» понятые требования;

– *регрессионное тестирование* – тестирование системы в процессе ее разработки и сопровождение на нерегресс. То есть проверяется, что изменения системы не ухудшили уже существующей функциональности. Для этого создаются пакеты регрессионных тестов, которые запускаются с определенной периодичностью – например, в пакетном режиме, связанные с процедурой постоянной интеграции;
– *нагрузочное тестирование* – тестирование системы на корректную работу с большими объемами данных. Например, проверка баз данных на корректную обработку большого (предельного) объема записей, исследование поведения серверного ПО при большом количестве клиентских соединений, эксперименты с предельным трафиком для сетевых и телекоммуникационных систем, одновременное открытие большого числа файлов, проектов;
– *стрессовое тестирование* – тестирование системы на устойчивость к непредвиденным ситуациям. Этот вид тестирования нужен далеко не для каждой системы, так как подразумевает высокую планку качества;
– *приемочное тестирование* – тестирование, выполняемое при приемке системы заказчиков. Более того, различные стандарты часто включают в себя наборы приемочных тестов. Например, существует большой пакет тестов, поддерживаемых компанией Sun Microsystems, которые обязательны для прогона для всех новых реализаций Java-машины. Считается, что только после того как все эти тесты успешно проходят, новая реализация вправе называться Java.

Устранение ошибок

Между программистами и тестировщиками необходим специальный интерфейс общения. Ведь ошибок находится много, их исправление требует времени, за их исправление отвечают разработчики, а тестировщики должны удостовериться, что они действительно исправлены. Кроме того, менеджерам нужна статистика по найденным и исправленным ошибкам – это хороший

инструмент контроля проекта. Все это изображено на рис. 8.2. Чтобы справиться с этим потоком информации и обеспечить необходимые в работе, удобные сервисы, существует специальный класс программных средств – *средства контроля ошибок*.

Рисунок 8.2 *– Обработка ошибок*

Как правило, описание ошибки в системе контроля ошибок имеет следующие основные атрибуты:

- ответственного за ее проверку – тестировщика, который ее нашел и который проверяет, что исправления, сделанные разработчиком, действительно устраняют ошибку;
- ответственного за ее исправление – разработчика, которому ошибка отправляется на исправление;
- состояние, например, ошибка найдена, ошибка исправлена, ошибка закрыта, ошибка вновь проявилась и т.д.

Этот список существенно дополняется в различных программных средствах контроля ошибок, но это основные атрибуты.

Использование этих систем давно стало общей практикой в разработке ПО, наравне со средствами версионного контроля и многими иными инструментами. Они включают в себя:

- базу данных для хранения ошибок;
- интерфейс к этой базе данных для внесения новых ошибок и задания их многочисленных атрибутов, для просмотра ошибок на основе различных фильтров – например, все найденные ошибки за последний месяц, все ошибки, за которые отвечает данный разработчик и т.д.;

– сетевой доступ, так как проекты все чаще оказываются распределенными;
– программный интерфейс для возможностей программной интеграции таких систем с другим ПО, поддерживающим разработку ПО (например, со средствами непрерывной интеграции – они могут автоматически вносить в базу данных найденные при автоматическом прогоне тестов ошибки).

Очень важным при работе с ошибками оказываются различные отчеты.

Рефакторинг

Качество кода программного продукта во многом определяется тем, насколько трудно или легко вносить изменения в код, а также тем, насколько доступен код для понимания. Созданное программное приложение может выполнять требуемые функции, но иметь проблемы с внесением изменений или пониманием созданного кода. В этом случае такое ПО нельзя назвать качественным, так как на этапе его сопровождения могут возникнуть проблемы с его модификацией при изменении пользовательских требований.

Для улучшения качества кода программных приложений применяют рефакторинг [2, 17]. По определению М. Фаулера, рефакторинг определяется как «процесс изменения программной системы таким образом, что её внешнее поведение не изменяется, а внутренняя структура улучшается». Следовательно, в процессе проектирования для создания высококачественного программного продукта необходимо не только обеспечить выполнение функциональных требований, но и нефункциональных, в частности, удобства сопровождения, что предполагает возможность и простоту внесения изменений в код, а также возможность легкого понимания созданного кода.

Некачественный дизайн кода определяется по ряду признаков [12]:
– жесткость – характеристика программы, затрудняющей внесение изменений в код;
– хрупкость – свойство программы повреждаться во многих местах при внесении единственного изменения;

– косность характеризуется тем, что код содержит части, которые могли бы оказаться полезными в других системах, но усилия и риски, сопряженные с попыткой отделить эти части кода от оригинальной системы, слишком велики;
– ненужная сложность характеризуется тем, что программа содержит элементы, которые не используются в текущий момент;
– ненужные повторения, которые состоят в повторяющихся фрагментах кода в программе;
– непрозрачность, которая характеризует трудность кода для понимания.

Для создания качественного дизайна кода целесообразно применять некоторые принципы и паттерны проектирования ПО [2, 12].

Принцип единственной обязанности определяет, что у класса должна быть только одна причина для изменения. Из этого принципа следует, что классу целесообразно поручать только одну обязанность.

Принцип открытости/закрытости определяет, что программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации.

Принцип подстановки Лисков определяет, что должна быть возможность вместо базового класса подставить любой его подтип.

Принцип инверсии зависимостей определяет два положения:
– модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций;
– абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Принцип разделения интерфейсов определяет, что клиенты должны знать только об абстрактных интерфейсах, обладающих свойством сцепленности.

Паттерны проектирования предлагают универсальные, проверенные практикой решения. В [2] приведен обширный перечень паттернов. Среди общего списка паттернов следует выделить те, которые целесообразно применять при гибкой разработке

программного обеспечения. Это паттерны Команда, Стратегия, Фасад, Посредник, Одиночка, Фабрика, Компоновщик, Наблюдатель, Абстрактный сервер/адаптер/шлюз, Заместитель и Шлюз, Посетитель и Состояние.

Использование паттернов при разработке позволяет создавать программное обеспечение, которое легко модифицировать и сопровождать.

Следует отметить, что для проведения рефакторинга необходимо иметь надежные тесты, которые обеспечивают контроль соблюдения функциональных требований при улучшении дизайна кода программного обеспечения.

Ключевые термины

ITU (International Telecommunication Union) – комитет по стандартизация телекоммуникационных протоколов и интерфейсов.

ISO (International Organization for Standardization) – организация по содействию развития стандартизации, а также смежных видов деятельности в мире с целью обеспечения международного обмена товарами и услугами.

ETSI (European Telecommunications Standards Institute) – независимая, некоммерческая организация по стандартизации в телекоммуникационной промышленности в Европе.

SEI (Software Engineering Institute) – организация по стандартизации в области программной инженерии, выработке критериев для сертификации надежных и зрелых компаний.

IEEE (Institute of Electrical and Electronics Engineers) – международная некоммерческая ассоциация специалистов в области техники и разработки стандартов по радиоэлектронике и электротехнике.

OMG (Object Management Group) – организация по разработке и продвижению объектно-ориентированных технологий и стандартов.

Тестирование – проверка соответствия между реальным поведением программы и ее ожидаемым поведением в специально заданных, искусственных условиях.

Рефакторинг – процесс изменения программной системы без изменения её внешнего поведения для улучшения внутренней структуры кода.

Краткие итоги

Вопросами стандартизации на международном уровне занимаются такие организации, как: ITU (International Telecommunication Union), ISO (International Organization for Standardization), ETSI (European Telecommunications Standards Institute) и комитетов SEI (Software Engineering Institute), IEEE (Institute of Electrical and Electronics Engineers) и OMG (Object Management Group). Все эти комитеты и организации включают программную инженерию в сферу своей деятельности. Стандартизация качества ПО прежде всего проявляется в вопросах тестирования программного обеспечения и стандартизации процессов его создания. К методам обеспечения качества ПО относятся: наладка качественного процесса разработки, формальные методы доказательства корректности программ, исследование и анализ динамических свойств ПО, обеспечение качества кода и тестирование. Тестирование представляет собой процесс проверки соответствия между реальным поведением программы и ее ожидаемым поведением в специально заданных, искусственных условиях. Существуют различные виды тестирования: модульное, интеграционное, системное, регрессионное, нагрузочное, стрессовое и приемочное. Для улучшения качества кода программных приложений применяют рефакторинг.

Вопросы для самопроверки

1. Назовите стандарты качества ISO.
2. Назовите стандарты телекоммуникационных протоколов и интерфейсов ITU.
3. Национальные и международные комитеты по стандартизации.
4. Стандартизация качества ПО.
5. Качество программного обеспечения.
6. Дайте характеристику методам обеспечения качества ПО.
7. Тестирование программных средств.
8. Виды тестирования программных средств.
9. Поясните назначение модульного тестирования.
10. Поясните назначение интеграционного тестирования.
11. Поясните назначение системного тестирования.
12. Поясните назначение регрессионного тестирования.

13. Поясните назначение нагрузочного тестирования.
14. Поясните назначение стрессового тестирования.
15. Поясните назначение приемочного тестирования.
16. Выявление и исправление ошибок в программах.
17. Для чего применяют рефакторинг кода?
18. Приведите признаки некачественного кода.

9. АРХИТЕКТУРА И ФУНКЦИОНАЛЬНЫЕ ВОЗМОЖНОСТИ TFS

Microsoft Visual Studio *Team Foundation Server* (TFS) предназначен для обеспечения совместной работы команд разработчиков программного обеспечения. Team Foundation Server предоставляет следующие функциональные возможности [37]:

- управление проектами;
- отслеживание рабочих элементов;
- контроль версий;
- управление тестовыми случаями;
- автоматизацию построения;
- отчетность.

Архитектура Team Foundation Server 2012 является трехуровневой сервис-ориентированной (рис. 9.1) [37]. Уровень приложения поддерживается веб-сервером ASP.NET, размещенным в среде IIS. Уровень данных поддерживается сервером баз данных MS SQL Server 2012.

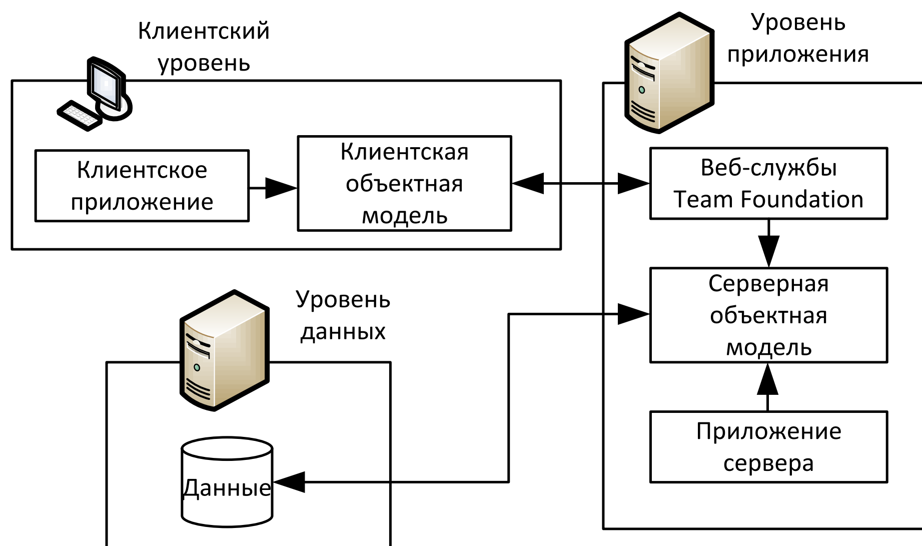

Рисунок 9.1 *– Архитектура Team Foundation Server*

Team Foundation Server представляет с логической точки зрения веб-приложение, состоящее из нескольких веб-служб, выполняющихся на *уровне приложения*. Данные службы реализуют функциональность TFS. В состав веб-служб уровня приложения

входят: управление версиями, служба построения, отслеживания рабочих элементов, службы платформы TFS и лаборатории тестирования Lab Management. Серверная объектная модель является интерфейсом прикладного программирования для TFS. При необходимости расширение функциональности TFS целесообразно строить на базе серверной объектной модели.

Уровень данных состоит из нескольких реляционных баз данных и хранилища данных: базы данных конфигурации сервера, базы данных аналитики, базы данных коллекции командных проектов и хранилища данных. Информация командных проектов сохраняется в реляционных базах данных и через запланированные интервалы времени помещается в хранилище данных.

Клиентский уровень может реализовываться в оболочке Visual Studio и веб-браузере. Клиентский уровень взаимодействует с уровнем приложений через серверную объектную модель и использует веб-службы. Кроме того, клиентский уровень включает интеграцию с Microsoft Office.

Развертывание Team Foundation Server

Для Team Foundation Server можно выполнить развертывание несколькими способами: на одном сервере; на нескольких серверах; в одном домене, рабочей группе или в нескольких доменах [3].

В простейшей серверной топологии для размещения компонентов, составляющих логические уровни Team Foundation, используется один физический сервер (рис. 9.2).

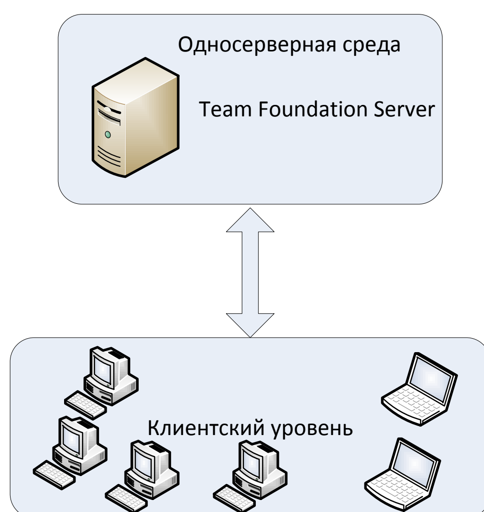

***Рисунок 9.2** – Простейшая серверная топология TFS*

При установке TFS с одним сервером все компоненты (приложение Team Foundation Server, SQL Server, Reporting Services и Widows SharePoint Services) устанавливаются на одном компьютере. Такая конфигурация предполагает выполнение построения (Team Foundation Build) и тестирование либо на сервере, либо на клиентских компьютерах. Общее число пользователей для такой конфигурации, как правило, не более 50.

В простой серверной топологии для размещения компонентов, составляющих логические уровни Team Foundation, также используется один физический сервер (рис. 9.3). Однако в такой технологии учитывается также дополнительная нагрузка на процессорные мощности, создаваемая программным обеспечением для построения и тестирования.

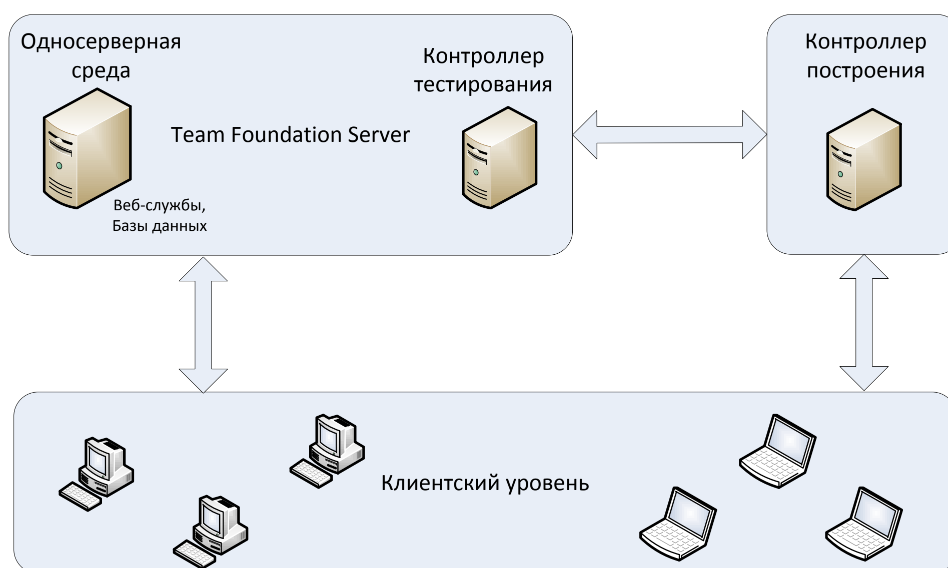

***Рисунок 9.3** – Простая серверная топология TFS*

На рис. 9.3 веб-службы и базы данных для Team Foundation размещаются на одном физическом сервере, но службы построения устанавливаются на отдельный компьютер. К Team Foundation Server можно получить доступ с клиентских компьютеров, принадлежащих к той же рабочей группе или тому же домену. В данной конфигурации контроллер построения для выполнения Team Foundation Build и контроллер тестирования устанавливаются на отдельных компьютерах. Общее число пользователей для такой конфигурации, как правило, не более 100.

В серверной топологии средней сложности используются два или больше серверов для размещения логических компонентов на уровне данных и приложений Team Foundation (рис. 9.4). На рис. 9.4 службы уровня приложений для Team Foundation Server развертываются на одном сервере, а базы данных для TFS устанавливаются на отдельном сервере. На отдельных серверах размещается веб-приложение SharePoint и экземпляр служб отчетов SQL Server. Портал для каждого командного проекта размещается в веб-приложении SharePoint. Сервер Team Foundation Build и тестовые контроллеры команды развертываются на дополнительных серверах. Общее число пользователей для такой конфигурации, как правило, не более 1000.

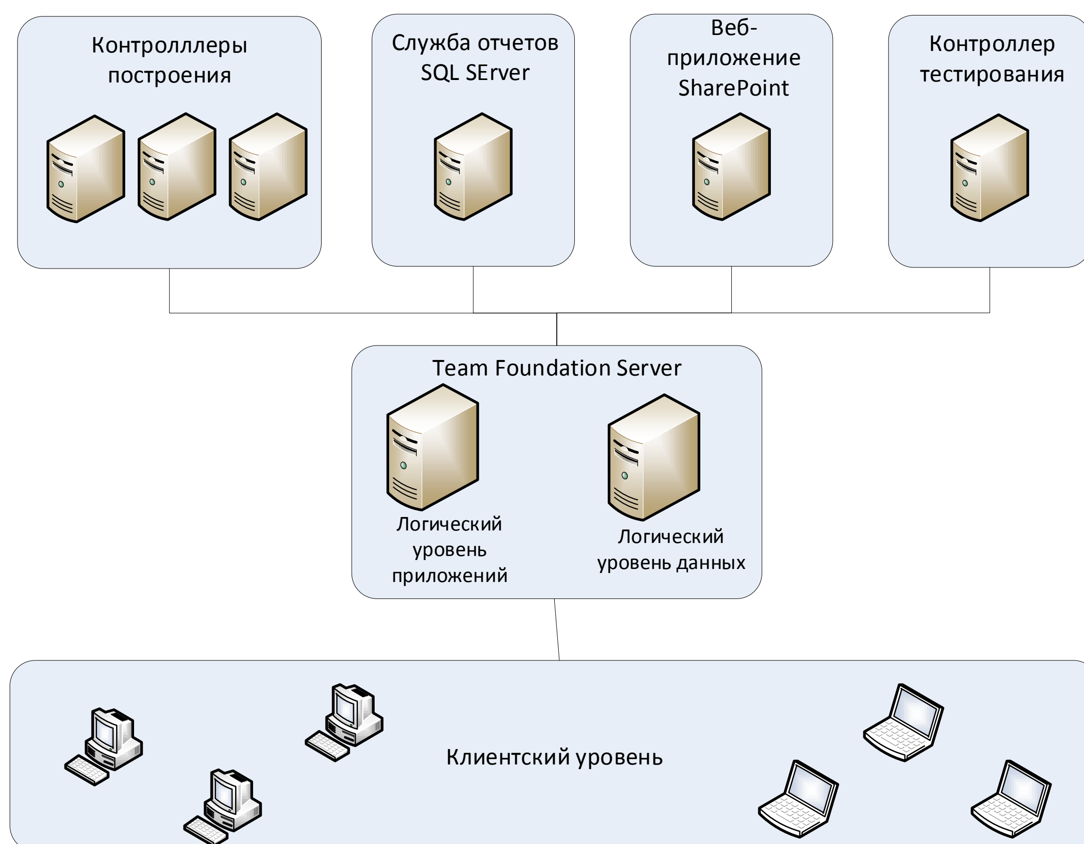

Рисунок 9.4 *– Серверная топология TFS средней сложности*

Для управления командной разработкой Team Foundation Server содержит одну или несколько коллекций командных проектов, каждая из которых может содержать ноль или более проектов.

Шаблоны командных проектов

Командный проект представляет коллекцию рабочих элементов, кода, тестов и построений, которые охватывают все артефакты, используемые в жизненном цикле программного проекта [4]. Командный проект строится на основе шаблона, который представляет набор XML-файлов, содержащих детали того, как должен осуществляться процесс. В TFS 2012 имеются следующие шаблоны проектов:

- *MSF for CMMI Process Improvement 6.0*, который предназначен для больших команд со строго формальным подходом к управлению проектами на основе модели CMM/CMMI;
- *MSF for Agile Software Development 6.0*, который определяет гибкий подход к управлению проектами разработки программного обеспечения;
- *Microsoft Visual Studio Scrum 2.2*, который предназначен для небольших команд (до 7–10 участников), которые используют гибкую методологию и терминологию Scrum.

При создании командного проекта предоставляется возможность настраивания и управления следующими областями проекта:

- отслеживание рабочих элементов позволяет определить начальные типы рабочих элементов, общие и пользовательские запросы, создать начальные рабочие элементы;
- классификации позволяют определить начальные области и итерации проекта;
- веб-сайт на базе SharePoint позволяет управлять библиотекой документов проекта;
- система контроля версий позволяет определять начальные группы безопасности, разрешения, правила возврата версий;
- отчеты формируются в папки, создаваемые на сайте командного проекта;
- группы и разрешения включают группы безопасности TFS и разрешения для каждой группы;
- построения включают стандартные процессы построения для создания новых построений;
- лаборатория включает стандартные процессы лаборатории для использования, а также разрешения лаборатории для каждой группы;

– управление тестированием включает стандартные конфигурации тестирования, переменные, параметры и состояния разрешений.

Рабочими элементами в Team Foundation Server являются: пользовательское описание функциональности, задачи, тестовые случаи, ошибки и препятствия. Этими элементами нужно управлять для выполнения программного проекта. Система отслеживания рабочих элементов позволяет создавать рабочие элементы, отслеживать их состояние, формировать историю их изменения. Все данные по рабочим элементам сохраняются в базе данных TFS.

Team Foundation Server включает централизованную систему контроля версий. Система контроля версий TFS предоставляет следующие возможности:

– атомарные возвраты. Изменения, вносимые в файлы, упаковываются с «набор изменений». Возврат файла в наборе изменений представляет собой неделимую трансакцию. Этим гарантируется согласованность базы кода;
– ассоциация операций возврата с рабочими элементам, что позволяет отслеживать требования от начальной функции до задач и возврата в систему контроля версий кода, реализующего эту функцию;
– ветвление и объединение при разработке кода;
– наборы отложенных изменений позволяют создавать копии кода, не записывая их в главное хранилище системы контроля версий;
– использование подписей позволяет пометить набор файлов в определенной версии с помощью текстовой подписи;
– одновременное извлечение позволяет нескольким членам команды одновременно редактировать файл с последующим объединением изменений;
– отслеживание истории файлов;
– политики возврата, которые предоставляют возможность выполнить код для проверки допустимости возврата;
– примечания при возврате позволяют формировать метаданные о возврате;

– прокси-сервер позволяет оптимизировать работу с региональными центрами разработки.

Построение проекта программного продукта в TFS реализует Сервер Team Foundation Build. Он позволяет стандартизировать инфраструктуру построений для команды проекта. Построение может выполняться в следующих режимах:

– в ручном, при котором построение вручную ставится в очередь построений;
– непрерывной интеграции, которая позволяет выполнять построение при каждом возврате в систему контроля версий;
– в прокрутке построений, при которой возвраты группируются вместе, чтобы в построение включались изменения за определенный период;
– условный возврат, при котором возвраты принимаются, в случае если успешно прошли слияния и построения отправленных изменений;
– расписание, которое позволяет настроить время построения на определенные дни недели.

Разработчик имеет возможность взаимодействовать с ключевыми службами Team Foundation Server посредством:

– командного обозревателя (Team Explorer) Visual Studio 2012, который предоставляет доступ к функциям управления жизненным циклом приложений, включая командные проекты, Мои работы, анализ кода, управление версиями и построениями;
– доступа через веб (Team Web Access), посредством которого предоставляется веб доступ к функциям управления жизненным циклом приложений, включая командные проекты, рабочие группы, управление проектами, управление версиями и построениями;
– Microsoft Excel, посредством которого предоставляется возможность определять и изменять рабочие элементы массивом, а также создавать отчеты на основе запроса рабочего элемента;
– Microsoft Project, посредством которого предоставляется возможность управлять планом проекта, задачами расписа-

ния, ресурсами, формировать календарь проекта, диаграммы Ганта и представления ресурсов;
- консоли администрирования Team Foundation Server, посредством которой осуществляется настройка TFS;
- инструментов командной строки;
- сторонних средств интеграции.

Ключевые термины

Team Foundation Server – сервер для обеспечения совместной работы команд разработчиков программного обеспечения.

Архитектура TFS 2012 – трехуровневая сервис-ориентированная архитектура.

Уровень данных TFS 2012 – архитектурный уровень TFS, который включает несколько реляционных баз данных и хранилище данных.

Клиентский уровень TFS 2012 – архитектурный уровень TFS, который поддерживается в оболочке Visual Studio и веб-браузере.

Уровень приложения TFS 2012 – архитектурный уровень TFS, который состоит из нескольких веб-служб.

Командный проект – коллекция рабочих элементов, кода, тестов и построений, которые охватывают все артефакты, используемые в жизненном цикле программного проекта.

Краткие итоги

Microsoft Visual Studio Team Foundation Server предназначен для обеспечения совместной работы команд разработчиков программного обеспечения и имеет трехуровневую сервис-ориентированную архитектуру. Клиентский уровень реализован в оболочке Visual Studio и веб-браузере. Уровень приложения TFS состоит из нескольких веб-служб. Уровень данных включает несколько реляционных баз данных и хранилище данных. Развертывание TFS можно выполнить на одном сервере, на нескольких серверах, в одном домене, рабочей группе или в нескольких доменах. Командный проект представляет коллекцию рабочих элементов, кода, тестов и построений. Проект строится на основе шаблонов. Рабочими элементами в TFS являются: пользователь-

ское описание функциональности, задачи, тестовые случаи, ошибки и препятствия. TFS включает централизованную систему контроля версий. Построение проекта программного продукта в TFS реализует сервер Team Foundation Build. Разработчик имеет возможность взаимодействовать с ключевыми службами TFS посредством командного обозревателя, доступа через веб, Microsoft Excel, Microsoft Project, консоли администрирования TFS и инструментов командной строки.

Вопросы для самопроверки

1. Поясните назначение и функциональные возможности Microsoft Visual Studio 2012 Team Foundation Server.
2. Дайте характеристику архитектуры TFS.
3. Дайте характеристику простейшей серверной топологии TFS.
4. Дайте характеристику простой серверной топологии TFS.
5. Дайте характеристику серверной топологии TFS средней сложности.
6. Что представляет собой командный проект в TFS?
7. Какие шаблоны проектов имеются в TFS 2012?
8. Какие рабочие элементы определены в Team Foundation Server?
9. Какие возможности представляет система контроля версий TFS?
10. Какие возможности имеет разработчик для взаимодействия с ключевыми службами Team Foundation Server?

Упражнения

1. Проведите анализ системных требований для установки Microsoft Visual Studio 2015 Team Foundation Server.
2. Проведите анализ процесса установки Team Foundation Server.
3. Проведите анализ конфигурирования Team Foundation Server.

ЗАКЛЮЧЕНИЕ

При написании данного учебного пособия ставилась цель – представить основные задачи, решаемые в рамках программной инженерии, осветить методы и методологии, используемые при управлении программными проектами, а также программные средства, поддерживающие управление жизненным циклом программных приложений.

Методы программной инженерии постоянно развиваются, расширяется спектр программных систем управления жизненным циклом ПО, появляются новые подходы интеграции процессов разработки и сопровождения программных систем (DevOps), которые направлены на повышение качества и сокращение сроков создания и модификации приложений.

По мнению авторов, данное учебное пособие может сформировать представление о возможностях программной инженерии для разработки программных систем, а детальное изучение данных вопросов и получение практических навыков являются прерогативой читателей.

Замечания, пожелания и ваше мнение по данному пособию можно направлять по адресу alexdoljenko@mail.ru.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Введение в Agile Modeling [Электронный ресурс]. – Режим доступа: http://www.agilemodeling.com/.
2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно ориентированного программирования. Паттерны проектирования. – СПб.: Питер, 2001.
3. Госсе М., Келлер Б., Кришнамурти А., Вудворт М. Управление жизненным циклом приложений с Visual Studio 2010. Профессиональный подход. – М.: ЭКОМ Паблишерз, 2012.
4. Джеймс А. Хайсмит Адаптивная разработка программного обеспечения: Совместный подход к управлению сложными системами. – Нью-Йорк: Дорсет House, 2000.
5. Долженко А.И. Технологии командной разработки программного обеспечения информационных систем. – М.: Интернет-университет информационных технологий (ИНТУИТ), 2016.
6. Долженко А.И. Технологии программирования: учебник. – Ростов н/Д: Издательско-полиграфический комплекс Ростов. гос. экон. ун-та (РИНХ), 2016.
7. Зараменских Е.П. Управление жизненным циклом информационных систем: монография. – Новосибирск: Издательство ЦРНС, 2014.
8. Кент Бек Экстремальное программирование: разработка через тестирование. – СПб.: Питер, 2003.
9. Match Кознов Д.В. Введение в программную инженерию. Часть I. – СПб.: Изд-во Санкт-Петербургского ун-та, 2005.
10. Кулямин В.В. Технология программирования. Компонентный подход. – М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2007.
11. Левинсон Дж. Тестирование ПО с помощью Visual Studio 2010. – М.: ЭКОМ Паблишерз, 2012.
12. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. – СПб.: Символ-Плюс, 2011.
13. Месарош Дж. Шаблоны тестирования xUnit рефакторинг кода тестов. – М.: ООО «И.Д. Вильямс», 2009.

14. Обзор Microsoft Solutions Framework (MSF) [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/jj161047(v=vs.120).aspx.
15. Орлов С.А. Технологии разработки программного обеспечения: учебное пособие. – СПб.: Питер, 2003.
16. Соммервилл И. Инженерия программного обеспечения. – М.: Издательский дом «Вильямс», 2002.
17. Фаулер М. Рефакторинг: улучшение существующего кода. – СПб.: Символ-Плюс, 2004.
18. Шаблон процесса Agile [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/dd380647.aspx.
19. Шаблон процесса CMMI [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/dd997574.aspx.
20. Шаблон процесса Scrum [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/ff731587.aspx.
21. Agile Data Home Page [Электронный ресурс]. – Режим доступа: http://www.agiledata.org/.
22. CMMI [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/CMMI.
23. DevOps и управление жизненным циклом приложений [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/fda2bad5.aspx.
24. DSDM Altern [Электронный ресурс]. – Режим доступа: http://www.dsdm.org/.
25. Feature Driven Development для веб-разработчиков [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/70424/.
26. Getting Real: The smarter, faster, easier way to build a successful web application. 37signals, 2006.
27. GitHub [Электронный ресурс]. – Режим доступа: https://github.com/.
28. Git and Team Services [Электронный ресурс]. – Режим доступа: https://www.visualstudio.com/ru-ru/docs/git/overview.
29. Git-distributed-is-the-new-centralized [Электронный ресурс]. – Режим доступа: https://git-scm.com/book/ru/v1.
30. ISO-9000 [Электронный ресурс]. – Режим доступа: http://www.pqm-online.com/assets/files/pubs/translations /std/iso-9000-2015-%28rus%29.pdf.

31. Microsoft Solutions Framework [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Microsoft_Solutions_Framework.
32. MSF for Agile Software Development 6.0 [Электронный ресурс]. – Режим доступа: http://msdn.microsoft.com/ru-ru/library/dd380647.aspx.
33. OpenUP – это просто [Электронный ресурс]. – Режим доступа: https://www.ibm.com/developerworks/ru/library/kroll/.
34. Rational Unified Process [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Rational_Unified_Process.
35. Scrum [Электронный ресурс]. – Режим доступа: https://msdn.microsoft.com/ru-ru/library/dd997796(v=vs.100).aspx.
36. SOFTWARE ENGINEERING [Электронный ресурс]. – Режим доступа: http://www.eah-jena.de/~kleine/history/software/nato1968garmisch.pdf.
37. Team Foundation Server architecture [Электронный ресурс]. – Режим доступа: https://www.visualstudio.com/ru-ru/docs/setup-admin/tfs/architecture/architecture.
38. The Agile Unified Process (AUP) [Электронный ресурс]. – Режим доступа: http://www.ambysoft.com/unifiedprocess/agileUP.html.
39. Visual Studio Scrum 2.0 [Электронный ресурс]. – Режим доступа: http://msdn.microsoft.com/ru-ru/library/ff731587.aspx.
40. IEEE Standard 610-90 (Standard Glossary of Software Engineering Terminology).

Учебное издание

Долженко Алексей Иванович, **Глушенко** Сергей Андреевич

Программная инженерия

Учебное пособие

Редактор Т.А. Грузинская
Верстка В.В. Климова

Изд. № 131/3044. Подписано в печать 11.10.2017. Бумага офсетная.
Печать цифровая. Формат 60х84/16. Гарнитура Times, Cambria.
Объем 5 уч.-изд. л.; 8 усл. п. л. Тираж 500 экз. Заказ № 213.

344002, Ростов-на-Дону, Б. Садовая, 69, РГЭУ (РИНХ), к. 152.
Издательско-полиграфический комплекс РГЭУ (РИНХ).