Введение в программную инженерию

Оглавление

Лекция 1. О предмете изучения 3

Программная инженерия 3
Программное обеспечение....5
Литература....6

Лекция 2. Процесс разработки программного обеспечения....7

Процесс 7
Совершенствование процесса 8
Классические модели процесса....9
Литература....13

Лекция 3. Рабочий продукт, дисциплина обязательств, проект....14

Рабочий продукт 14
Дисциплина обязательств 16
Проект 17
Литература....19

Лекция 4. Архитектура ПО 20

Обсуждение 20
Определение....20
Множественность точек зрения 21
Язык UML....24
Литература....28

Лекция 5. Управление требованиями 29

Проблема 29
Виды и свойства требований 30
Варианты формализации требований....31
Цикл работы с требованиями 33
Литература....33

Лекция 6. Конфигурационное управление....34

Проблема 34
Единицы конфигурационного управления 35
Управление версиями....37
Управление сборками....37
Понятие baseline....38
Литература....39

Лекция 7. Тестирование 40

Управление качеством 40
Тестирование....43
Работа с ошибками 47
Литература....48

Лекция 8. Диаграммные техники в работе со знаниями....................49
Метод случаи использования....................49
Итеративный цикл автор/рецензент....................53
Карты памяти....................56
Литература....................60
Лекция 9. MSF....................61
История и текущий статус....................61
Основные принципы....................62
Модель команды....................63
Прочие особенности....................66
Литература....................68
Лекция 10. CMMI....................69
Что такое CMMI?....................69
Уровни зрелости процессов по CMMI....................69
Области усовершенствования....................70
Литература....................71
Лекция 11. «Гибкие» (agile) методы разработки....................72
Общее....................72
Extreme Programming....................72
Scrum....................73
Литература....................75
Лекция 12. Обзор технологии Microsoft Visual Studio Team System (VSTS)....................76
Обзор....................76
Состав продукта....................77
Правила инсталляции....................83
Пакет Team Explorer....................83
Лекция 13. VSTS: управление элементами работ (Work Items)....................86
Определение, свойства, жизненный цикл....................86
Средства использования....................89
Лекция 14. VSTS: конфигурационное управление....................102
Система контроля версий....................102
Автоматические сборки....................117
Лекция 15. VSTS: тестирование....................131
Система отслеживания ошибок....................131
Модульные тесты....................137
Пакеты тестов....................138
Автоматическое тестирование Web-приложений....................142
Лекция 16. VSTS: поддержка различных моделей процесса....................148
Поддержка шаблонов процесса....................148
Обзор существующих шаблонов....................151
MSF for Agile Software Development....................151
Scrum....................152

Лекция 1. О предмете изучения

Программная инженерия

Чем программирование отличается от программой инженерии? Тем, что первое является некоторой абстрактной деятельностью и может происходить во многих различных контекстах. Можно программировать для удовольствия, для того, чтобы научиться (например, на уроках, на семинарах в университете), можно программировать в рамках научных разработок. А можно заниматься промышленным программированием. Как правило, это происходит в команде, и совершенно точно – для заказчика, который платит за работу деньги. При этом необходимо точно понимать, что нужно заказчику, выполнить работу в определенные сроки и результат должен быть нужного качества – того, которое удовлетворит заказчика и за которое он заплатит. Чтобы удовлетворить этим дополнительным требованиям, программирование «обрастает» различными дополнительными видами деятельности: разработкой требований, планированием, тестированием, конфигурационным управлением, проектным менеджментом, созданием различной документации (проектной, пользовательской и пр.).

Разработка программного кода предваряется анализом и проектированием (первое означает создание функциональной модели будущей системы без учета реализации, для осознания программистами требований и ожиданий заказчика; второе означает предварительный макет, эскиз, план системы на бумаге). Трудозатраты на анализ и проектирование, а также форма представления их результатов сильно варьируются от видов проектов и предпочтений разработчиков и заказчиков.

Требуются также специальные усилия по организации процесса разработки. В общем виде это итеративно-инкрементальная модель, когда требуемая функциональность создается порциями, которые менеджеры и заказчик могут оценить, и тем самым есть возможность управления ходом разработки. Однако эта общая модель имеет множество модификаций и вариантов.

Разработку системы также необходимо выполнять с учетом удобств ее дальнейшего сопровождения, повторного использования и интеграции с другими системами. Это значит, что система разбивается на компоненты, удобные в разработке, годные для повторного использования и интеграции. А также имеющие необходимые характеристики по быстродействию. Для этих компонент тщательно прорабатываются интерфейсы. Сама же система документируется на многих уровнях, создаются правила оформления программного кода – то есть оставляются многочисленные семантические следы, помогающие создать и сохранить, поддерживать единую, стройную архитектуру, единообразный стиль, порядок…

Все эти и другие дополнительные виды деятельности, выполняемые в процессе промышленного программирования и необходимые для успешного выполнения заказов и будем называть ***программной инженерией*** (software engineering)[1]. Получается, что так мы обозначаем, во-первых, некоторую ***практическую деятельность***, а во-вторых, специальную ***область знания***. Или другими словами, научную дисциплину. Ведь для облегчения выполнения каждого отдельного проекта, для возможности использовать разнообразный положительный опыт, достигнутый другими командами и

[1] В 70-х годах академиком А.П.Ершовым термин software engineering, переводился на расский язык как «технология программирования». Программная инженерия – более современный, но менее традиционный перевод этого же термина, предложенный в конце 90-х И.В.Поттосиным. В рамках данного курса будем пользоваться именно этим вариантом перевода.

разработчиками, этот самый опыт подвергается осмыслению, обобщению и надлежащему оформлению. Так появляются различные методы и практики (best practices) – тестирования, проектирования, работы над требованиями и пр., архитектурных шаблонов и пр. А также стандарты и методологии, касающиеся всего процесса в целом (например, MSF, RUP, CMMI, Scrum). Вот эти-то обобщения и входят в программную инженерию как в область знания.

Необходимость в программной инженерии как в специальной области знаний были осознаны мировым сообществом в конце 60-х годов прошлого века, более чем на 20 лет позже рождения самого программирования, если считать таковым знаменитый отчет фон Неймана «First Draft of a Report on the EDVAC», обнародованный им в 1945 году. Рождением программной инженерии является 1968 год – конференция NATO Software Engineering, г. Гармиш (ФРГ), которая целиком была посвящена рассмотрению этих вопросов. В сферу программной инженерии попадают все вопросы и темы, связанные с организацией и улучшением процесса разработки ПО, управлением коллектива разработчиков, разработкой и внедрением программных средств поддержки жизненного цикла разработки ПО. Программная инженерия использует достижения информатики, тесно связана с системотехникой, часто предваряется бизнес-реинжинирингом. Немного подробнее об этом контексте программной инженерии.

Информатика (computer science) – это свод теоретических наук, основанных на математике и посвященных формальным основам вычислимости. Сюда относятся математическую логику, теорию грамматик, методы построения компиляторов, математические формальные методы, используемые в верификации и модельном тестировании и т.д. Трудно строго отделить программную инженерию от информатики, но в целом направленность этих дисциплин различна. Программная инженерия нацелена на решение проблем производства, информатика – на разработку формальных, математизированных подходов к программированию.

Системотехника (system engineering) объединяет различные инженерные дисциплины по разработке всевозможных искусственных систем – энергоустановок, телекоммуникационных систем, встроенных систем реального времени и т.д. Очень часто ПО оказывается частью таких систем, выполняя задачу управления соответствующего оборудования. Такие системы называются *программно-аппаратными*, и участвуя в их создании, программисты вынуждены глубоко разбираться в особенностях соответствующей аппаратуры.

Бизнес-реинжиниринг (business reengineering) – в широком смысле обозначает модернизацию бизнеса в определенной компании, внедрение новых практик, поддерживаемых соответствующими, новыми информационными системами. При этом акцент может быть как на внутреннем переустройстве компании так и на разработке нового клиентского сервиса (как правило, эти вопросы взаимосвязаны). Бизнес-реинжиниринг часто предваряет разработку и внедрение информационных систем на предприятии, так как требуется сначала навести определенный порядок в делопроизводстве, а лишь потом закрепить его информационной системой.

Связь программной инженерии (как области практической деятельности) с информатикой, системотехникой и бизнес-реинжинирингом показана на рис. 1.1

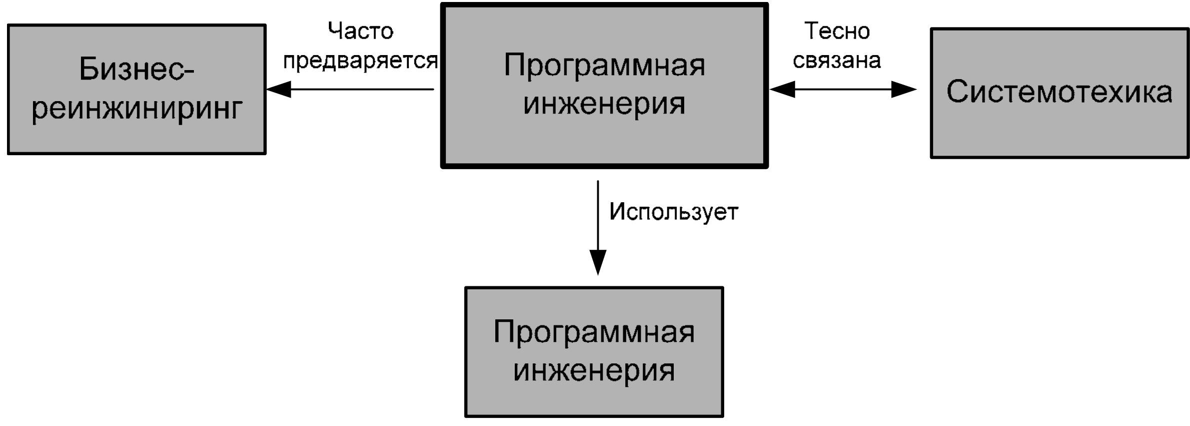

Рис. 1.1

Программное обеспечение

Определение. Будем понимать под ***программным обеспечением*** (ПО) множество развивающихся во времени логических предписаний, с помощью которых некоторый коллектив людей управляет и использует многопроцессорную и распределенную систему вычислительных устройств.

Это определение, данное Харальдом Милсом, известным специалистом в области программной инженерии из компании IBM, заключает в себе следующее.

1. Логические предписания – это не только сами программы, но и различная документация (например, по эксплуатации программ) и шире – определенная система отношений между людьми, использующими эти программы в рамках некоторого процесса деятельности.
2. Современное ПО предназначено, как правило, для одновременной работы с многими пользователями, которые могут быть значительно удалены друг от друга в физическом пространстве. Таким образом, вычислительная среда (персональные компьютеры, сервера и т.д.), в которой ПО функционирует, оказывается распределенной.
3. Задачи решаемые современным ПО, часто требуют различных вычислительных ресурсов в силу различной специализации этих задач, из-за большого объема выполняемой работы, а также из соображений безопасности. Например, появляется сервер базы данных, сервер приложений и пр. Таким образом, вычислительная среда, в которой ПО функционирует, оказывается многопроцессорной.
4. ПО развивается во времени – исправляются ошибки, добавляются новые функции, выпускаются новые версии, меняется его аппаратная база.

Свойства. Таким образом, ПО является сложной динамической системой, включающей в себя технические, психологические и социальные аспекты. ПО заметно отличается от других видов систем, создаваемых (созданных) человеком – механических, социальных, научных и пр., и имеет следующие особенности, выделенные Фредериком Бруксом в его знаменитой статье «Серебрянной пули нет».

1. *Сложность* программных объектов, которая существенно зависит от их размеров. Как правило, большее ПО (большее количество пользователей, больший объем обрабатываемых данных, более жесткие требования по быстродействию и пр.) с аналогичной функциональностью – это другое ПО. Классическая наука строила простые модели сложных явлений, и это удавалось, так как сложность не была характеристической чертой рассматриваемых явлений. (Сравнение программирования именно с наукой, а не с театром, кинематографом, спортом и другими областями человеческой деятельности, оправдано, поскольку оно возникло,

главным образом, из математики, а первые его плоды – компьютеры, языки программы – предназначались для использования при научных расчетах. Кроме того, большинство программистов имеют естественнонаучное, математическое или техническое образование. Таким образом, парадигмы научного мышления широко используются при программировании – явно или неявно.)

2. *Согласованность* – ПО основывается не на объективных посылках (подобно тому, как различные системы в классической науке основываются на постулатах и аксиомах), а должно быть согласовано с большим количеством интерфейсов, с которыми впоследствии оно должно взаимодействовать. Эти интерфейсы плохо поддаются стандартизации, поскольку основываются на многочисленных и плохо формализуемых человеческих соглашениях.
3. *Изменяемость* – ПО легко изменить и, как следствие, требования к нему постоянно меняются в процессе разработки. Это создает много дополнительных трудностей при его разработке и эволюции.
4. *Нематериальность*[2] – ПО невозможно увидеть, оно виртуально. Поэтому, например, трудно воспользоваться технологиями, основанными на предварительном создании чертежей, успешно используемыми в других промышленных областях (например, в строительстве, машиностроении). Там на чертежах в схематичном виде воспроизводятся геометрические формы создаваемых объектов. Когда объект создан, эти формы можно увидеть. А на чем мы основываемся, когда изображаем ПО?

Литература

1. H.Mills. The management of software engineering Part I: Principles of software engineering. // IBM System Journal, Vol. 38, No 2&3, 1999, pp. 289-295.
2. F. Brooks. No Silver Bullet. Information Proceeding of the IFIP 10th World Computing Conference. 1986. pp. 1069-1076. / Русский перевод: Ф. Брукс. Мифический человеко-месяц или как создаются программные системы. СПб Символ, 2000. 298 с.
3. I.Sommerville Software Engineering. 6th edition. Addison-Wesley, 2001. 693 p. / Русский перевод: И.Соммервилл. Инженерия программного обеспечения. Издательский дом “Вильямс”, 2002. 623 с.
4. А.П.Ершов. Технология разработки систем программирования. // А.П.Ершов. Избранные труды. ВО “Наука”. Новосибирск. 1994. С. 230-261.
5. Поттосин И.В. Программная инженерия: содержание, мнения и тенденции. // Программирование. 1997 N 4. С. 26-37.
6. Д.В.Кознов. Введение в программную инженерию. Часть I. Изд-во Санкт-Петербургского ун-та, 2005 г., 43 с.

[2] Здесь мы немного поправили классика – у него была незримость…..

Лекция 2. Процесс разработки программного обеспечения

Без процесса не понять….
Из деловой переписка менеджера программного проекта.

Процесс

Как мы работаем, какова последовательность наших шагов, каковы нормы и правила в поведении и работе, каков регламент отношений между членами команды, как проект взаимодействует с внешним миром и т.д.? Все это вместе мы склонны называть процессом. Его осознание, выстраивание и улучшение - основа любой эффективной групповой деятельности. Не случайно поэтому, что процесс оказался одним из основных понятий программной инженерии.

Центральным объектом изучения программной инженерии является ***процесс*** создания ПО – множество различных видов деятельности, методов, методик и шагов, используемых для разработки и эволюции ПО и связанных с ним продуктов (проектных планов, документации, программного кода, тестов, пользовательской документации и пр.).

Однако на сегодняшний день не существует ***универсального процесса*** разработки ПО – набора методик, правил и предписаний, подходящих для ПО любого вида, для любых компаний, для команд любой национальности. Каждый ***текущий процесс*** разработки, осуществляемый некоторой командой в рамках определенного проекта, имеет большое количество особенностей и индивидуальностей. Однако целесообразно перед началом проекта спланировать процесс работы, определив роли и обязанности в команде, рабочие продукты (промежуточные и финальные), порядок участия в их разработке членов команды и т.д. Будем называть это предварительное описание ***конкретным процессом***, отличая его от плана работ, проектных спецификаций и пр. Например, в системе Microsoft Visual Tem System оказывается шаблон процесса, создаваемый или адаптируемый (в случае использования стандартного) перед началом разработки. В VSTS существуют заготовки для конкретных процессов на базе CMMI, Scrum и др.

В рамках компании возможна и полезна объединение и стандартизация всех текущих процессов, которую будем называть ***стандартным процессом***. Последний, таким образом, оказывается некоторой базой данных, содержащей следующее:

- 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 практик разработки – проектного менеджмента, правил работы с заказчиком и т.д.;
- шаблонов проектных документов – технических заданий, проектных спецификаций, тестовых планов и т.д. и пр.

Также возможна стандартизация процедуры разработки конкретного процесса как «вырезки» из стандартного. Основная идея стандартного процесса – курсирование внутри компании передового опыта, а также унификация средств разработки. Очень уж часто в компаниях различные департаменты и проекты сильно отличаются по зрелости процесса разработки, а также затруднено повторное использование передового опыта. Кроме того, случаются, что компания использует несколько средств параллельных инструментов разработки, например, СУБД средства версионного контроля. Иногда это бывает оправдано (например, таковы требования заказчика), часто это необходимо – например,

Java .NET (большая компетентность оффшорной компании позволяет ей брать более широкий спектр заказов). Но очень часто это произвольный выбор самих разработчиков. В любом случае, такая множественность существенно затрудняет миграцию специалистов из проекта в проект, использование результатов одного проекта в другом и т.д. Однако при организации стандартного процесса необходимо следить, чтобы стандартный процесс не оказался всего лишь формальным, бюрократическим аппаратом. Понятие стандартного процесса введено и подробно описано в подходе CMMI.

Необходимо отметить, что наличие стандартного процесса свидетельствует о наличии «единой воли» в организации, существующей именно на уровне процесса. На уровне продаж, бухгалтерии и др. привычных для всех компаний процессов и активов единство осуществить не трудно. А вот на уровне процессов разработки очень часто каждый проект оказывается сам по себе (особенно в оффшорных проектах) – «текучка» захватывает и изолирует проекты друг от друга очень прочно.

Совершенствование процесса

Определение. Совершенствование процесса (software process improvement) – это деятельность по изменению существующего процесса (как текущего, в рамках одного проекта, так и стандартного, для всей компании) с целью улучшения качества создаваемых продуктов и/или снижения цены и времени их разработки. Причины актуальности этой деятельности для компаний-производителей ПО заключается в следующем.

1. Происходит быстрая смена технологий разработки ПО требует изучения и внедрения новых средств разработки.
2. Наблюдается быстрый рост компаний и их выход на новые рынки, что требует новой организации работ.
3. Имеет место высокая конкуренция, которая требует поиска более эффективных, более экономичных способов разработки.

Что и каким образом можно улучшать.

1. Переход на новые средства разработки, языки программирования и т.д.
2. Улучшение отдельных управленческих и инженерных практик – тестирования, управления требованиями и пр.
3. Полная, комплексная перестройка всех процессов в проекте, департаменте, компании (в соответствии, например, с CMMI).
4. Сертификация компании (CMM/CMMI, ISO 9000 и пр.).

Мы отделили п. 3 от п. 4 потому, что на практике 4 далеко не всегда означает действительную созидательную работу по улучшению процессов разработки ПО, а часто сводится к поддержанию соответствующего документооборота, необходимого для получения сертификации. Сертификат потом используется как средство, козырь в борьбе за заказы.

Главная трудность реального совершенствования процессов в компании заключается в том, что она при этом должна работать и создавать ПО, ее нельзя «закрыть на учет».

Отсюда вытекает идея непрерывного улучшения процесса, так сказать, малыми порциями, чтобы не так болезненно. Это тем более разумно, что новые технологии разработки, появляющиеся на рынке, а также развитие уже существующих нужно постоянно отслеживать. Эта стратегия, в частности, отражена в стандарте совершенствования процессов разработки CMMI.

Pull/Push стратегии. В контексте внедрения инноваций в производственные процессы бизнес-компаний (не обязательно компаний по созданию ПО) существуют две следующие парадигмы.

1. Organization pull – инновации нацелены на решение конкретных проблем компании.
2. Technology push – широкомасштабное внедрение инноваций из стратегических соображений. Вместо конкретных проблем, которые будут решены после внедрения инновации, в этом случае рассматриваются показатели компании (эффективность, производительность, годовой оборот средств, увеличение стоимости акций публичной компании), которые будут увеличены, улучшены после внедрения инновации. При этом предполагается, что будут автоматически решены многочисленные частные проблемы организации, в том числе и те, о которых в данный момент ничего не известно.

Пример использования стратегии organization pull – внедрение новых средств тестирования в ситуации, когда высоки требования по качеству в проекте, либо когда качество программной системы не удовлетворяет заказчика.

Пример использования стратегии technology push – переход компании со средств структурной разработки на объектно-ориентрованные. Еще один пример использования той же стратегии – внедрение стандартов качества ISO 9000 или CMMI. В обоих этих случая компания не решает какую-то одну проблему или ряд проблем – она хочет радикально изменить ситуацию, выйти на новые рубежи и т.д.

Проблемы применения стратегии technology push в том, что требуется глобальная перестройка процесса. Но компанию нельзя "закрыть на реконструкцию" – за это время положение на рынке может оказаться занято конкурентами, акции компании могут упасть и т.д. Таким образом, внедрение инноваций, как правило, происходит параллельно с обычной деятельностью компании, поэтапно, что в случае с technology push сопряжено с большими трудностями и рисками.

Использование стратегии organization pull менее рискованно, вносимые ею изменения в процесс менее глобальны, более локальны. Но и выгод такие инновации приносят меньше, по сравнению с удачными внедрениями в соответствии со стратегией technology push.

Необходимо также отметить, что существуют проблемы, которые невозможно устранить точечными переделками процесса, то есть необходимо применять стратегию technology push. Приведем в качестве примера зашедший в тупик процесс сопровождения и развития семейства программных продуктов – компания терпит большие убытки, сопровождая уже поставленные продукты, инструментальные средства проекта безнадежно устарели и находятся в плачевном состоянии, менеджмент расстроен, все попытки руководства изменить процесс наталкиваются на непонимание коллектива, ссоры и конфликты. Возможно, что в таком случае без "революции" не обойтись.

Еще одно различие обеих стратегий: в случае с organization pull, как правило, возврат инвестиций от внедрения происходит быстрее, чем в случае с technology push.

Классические модели процесса

Определение модели процесса. Процесс создания программного обеспечения не является однородным. Тот или иной метод разработки ПО, как правило, определяет некоторую динамику развертывания тех или иных видов деятельности, то есть, определяет *модель процесса* (process model).

Модель является хорошей абстракцией различных методов разработки ПО, позволяя лаконично, сжато и информативно их представить. Однако, сама идея модели процесса является одной из самых ранних в программной инженерии, когда считалось, что удачная модель – самое главное, что способствует успеху разработки. Позднее пришло осознание, что существует множество других аспектов (принципы управления и разработки, структуру команды и т.д.), которые должны быть определены согласовано друг с другом. И стали развиваться интегральные методологии разработки. Тем не менее существует несколько классических моделей процесса, которые полезны на практике и которые будут рассмотрены ниже.

Фазы и виды деятельности. Говоря о моделях процессов, необходимо различать фазы и виды деятельности.

Фаза (phase) – это определенный этап процесса, имеющий начало, конец и выходной результат. Например, фаза проверки осуществимости проекта, сдачи проекта и т.д. Фазы следуют друг за другом в линейном порядке, характеризуются предоставлением отчетности заказчику и, часто, выплатой денег за выполненную часть работы.

Редко какой заказчик согласится первый раз увидеть результаты только после завершения проекта. С другой стороны, подрядчики предпочитают получать деньги постепенно, по мере того, как выполняются отдельные части работы. Таким образом, появляются фазы, позволяющие создавать и предъявлять промежуточные результаты проекта. Фазы полезны также безотносительно взаимодействия с заказчиком – с их помощью можно синхронизировать деятельность разных рабочих групп, а также отслеживать продвижение проекта. Примерами фаз может служить согласование с заказчиком технического задания, реализация определенной функциональности ПО, этап разработки, оканчивающийся сдачей системы на тестирование или выпуском альфа-версии.

Вид деятельности (activity) – это определенный тип работы, выполняемый в процессе разработки ПО. Разные виды деятельности часто требуют разные профессиональные навыки и выполняются разными специалистами. Например, управление проектом выполняется менеджером проекта, кодирование – программистом, тестирование – тестеровщиком. Есть виды деятельности, которые могут выполняться одними и теми же специалистами – например, кодирование и проектирование (особенно в небольшом проекте) часто выполняют одни и те же люди.

В рамках одной фазы может выполнятся много различных видов деятельности. Кроме того, один вид деятельности может выполняться на разных фазах – например, тестирование: на фазе анализа и проектирования можно писать тесты и налаживать тестовое окружение, при разработке и перед сдачей производить, собственно, само тестирование. На настоящий момент для сложного программного обеспечения используются многомерные модели процесса, в которых отделение фаз от видов деятельности существенно облегчает управление разработкой ПО.

Виды деятельности, фактически, присутствуют, под разными названиями, в каждом методе разработки ПО. В RUP они называются рабочими процессами (work flow), в CMM – ключевыми областями процесса (key process area). Мы будем сохранять традиционные названия, принятые в том или ином методе, чтобы не создавать путаницы.

Водопадная модель была предложена в 1970 году Винстоном Ройсом. Фактически, впервые в процессе разработки ПО были выделены различные шаги разработки и поколеблены примитивные представления о разработке ПО в виде анализа системы и ее кодирования.

Были определены следующие шаги: разработка системных требований, разработка требований к ПО, анализ, проектирование, кодирование, тестирование, использование – см. рис. 2.1.

Еще одним достоинством этой модели явилось ограничение возможности возвратов на произвольный шаг назад, например, от тестирования – к анализу, от разработки – работе над требованиями и т.д. Отмечалось, что такие возвраты могут катастрофически увеличить стоимость проекта и сроки его выполнения. Например, если при тестировании обнаруживаются ошибки проектирования или анализа, то их исправление часто приводит к полной переделке системы. Этой моделью допускались возвраты только на предыдущий шаг, например, от тестирования к кодированию, от кодирования к проектированию и т.д.

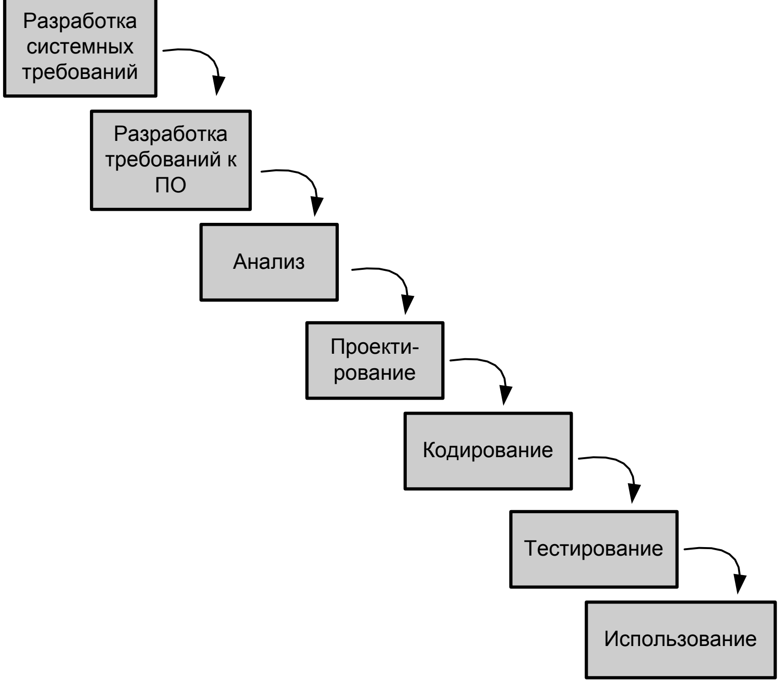

Рис. 2.1.

Наконец, в рамках этой модели было введено прототипирование, то есть предлагалось разрабатывать систему дважды, чтобы уменьшить риски разработки. Первая версия – прототип – позволяет увидеть основные риски и обосновано принять главные архитектурные решения. На создание прототипа отводилось до одной трети времени всей разработки.

В 70-80 годах прошлого века эта модель прочно укоренилась в разработке ПО в силу своей простоты и сходности с моделями разработки иных, не программных систем. В дальнейшем, в связи с развитием программной инженерии и осознанием итеративного характера процесса разработки ПО эта модель активно критиковалась, практически, каждым автором соответствующих статей и учебников. Стало общепринятым мнение, что она не отражает особенностей разработки ПО. Недостатками водопадной модели являются:

- отождествление фаз и видов деятельности, что влечет потерю гибкости разработки, в частности, трудности поддержки итеративного процесса разработки;
- требование полного окончания фазы-деятельности, закрепление результатов в виде подробного исходного документа (технического задания, проектной спецификации); однако опыт разработки ПО показывает, что невозможно

полностью завершить разработку требований, дизайн системы и т.д. – все это подвержено изменениям; и причины тут не только в том, что подвижно окружение проекта, но и в том, что заранее не удается точно определить и сформулировать многие решения, они проясняются и уточняются лишь впоследствии;

- интеграция всех результатов разработки происходит в конце, вследствие чего интеграционные проблемы дают о себе знать слишком поздно;
- пользователи и заказчик не могут ознакомиться с вариантами системой во время разработки, и видят результат только в самом конце; тем самым, они не могут повлиять на процесс создания системы, и поэтому увеличиваются риски непонимания между разработчиками и пользователями/заказчиком;
- модель неустойчива к сбоям в финансировании проекта или перераспределению денежных средств, начатая разработка, фактически, не имеет альтернатив "по ходу дела".

Однако данная модель продолжает использоваться на практике – для небольших проектов или при разработке типовых систем, где итеративность не так востребована. С ее помощью удобно отслеживать разработку и осуществлять поэтапный контроль за проектом. Эта модель также часто используется в оффшорных проектах3 с почасовой оплатой труда. Водопадная модель вошла в качестве составной части в другим модели и методологии, например, в MSF.

Спиральная модель была предложена Бэри Боемом в 1988 году для преодоления недостатков водопадной модели, прежде всего, для лучшего управления рисками. Согласно этой модели разработка продукта осуществляется по спирали, каждый виток которой является определенной фазой разработки. 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.

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), симуляция 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.

---

[3] От английского offshore – вне берега, в расширенном толковании – вне одной страны.

Отдельная спираль может соответствовать разработке некоторой программной компоненты 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.

Литература

1. I.Sommerville Software Engineering. 6th edition. Addison-Wesley, 2001. 693 p. / Русский перевод: И.Соммервилл. Инженерия программного обеспечения. Издательский дом «Вильямс», 2002. 623 с.
2. W.Humphrey. Managing the Software Process. Addison-Wesley, 1989. 494 p.
3. R. W.Zmud. An Examination of «Push-Pull» Theory Applied to Process Innovation in Knowledge Work, Management Science, Vol. 30(6), 1984, pp. 727-738.
4. B. Boehm. A Spiral Model of Software Development and Enhancement. // IEEE Computer, vol.21, #5, May 1988. P. 61-72.
5. W. W. Royce. Managing the Development of Large Software Systems. Proceedings of IEEE WESCON, August 1970. P. 1-9. Переиздана: Proceedings of the 9th International Software Engineering Conference, Computer Society Press, 1987. P. 328-338.
6. Р.Т.Фатрепп, Д.Ф.Шафер, Л.И.Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат. Изд. дом «Вильямс». 2003. 1125 с.
7. Д.В.Кознов. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Изд. СПбГУ, 2004. 170 с.
8. Д.В.Кознов. Введение в программную инженерию. Часть I. Изд-во Санкт-Петербургского ун-та, 2005 г., 43 с.
9. Д.Ахен, А.Клауз, Р.Тернер. CMMI: комплексный подход к совершенствованию процессов. Пер с англ. М.:МФК. 2005. 330 с.
10. CMMI for Development, Version 1.2. *CMMI-DEV* (Version 1.2, August 2006). Carnegie Mellon University Software Engineering Institute (2006). Retrieved on 22 August 2007. http://www.sei.cmu.edu/publications/documents/06.reports.

Лекция 3. Рабочий продукт, дисциплина обязательств, проект

В силу творческого характера программирования, существенной молодости участников разработки ПО, оказываются актуальными некоторые вопросы обычного промышленного производства, ставшие давно общим местом. Прежде всего, это дисциплина обязательств и рабочий продукт. Данные знания, будучи освоенными на практике, чрезвычайно полезны в командной работе. Кроме того, широко применяемые сейчас на практике методологии разработки ПО, поддержанные соответствующим программным инструментарием, активно используют эти понятия, уточняя и конкретизируя их.

Рабочий продукт

Одной из существенных условий для управляемости промышленного процесса 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*** (work product) – любой артефакт, произведенный в процессе разработки ПО, например, файл или набор файлов, документы, составные части продукта, сервисы, процессы, спецификации, счета и т.д.

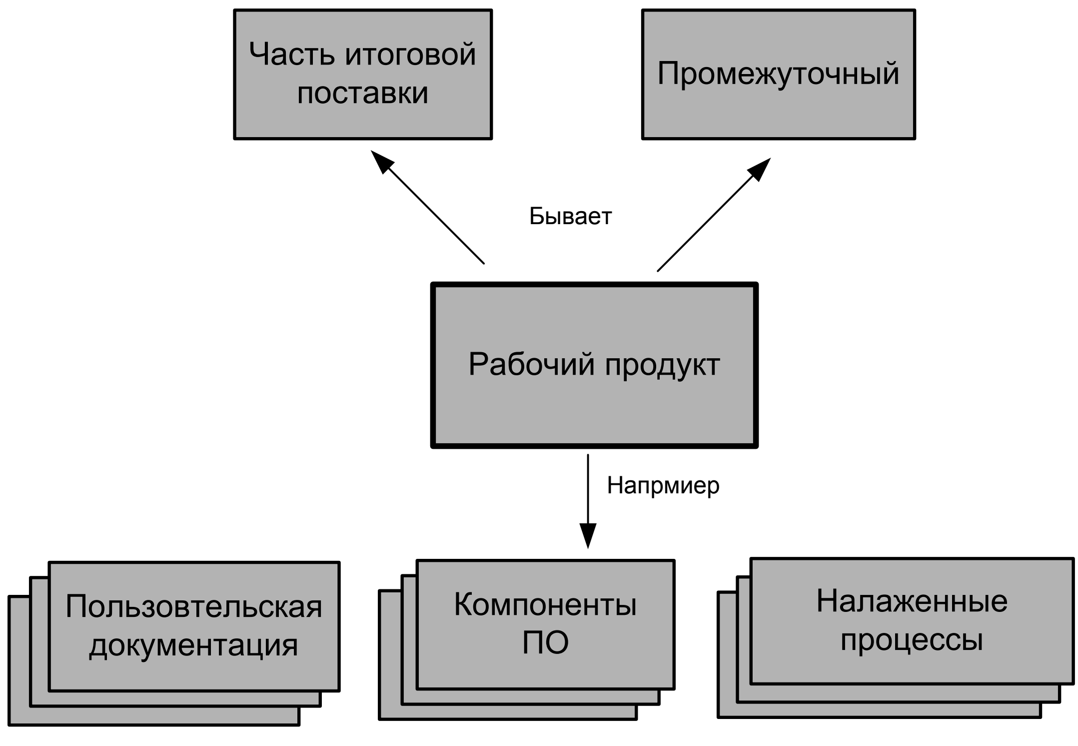

Рис.3.1.

Ключевая разница между рабочим продуктом и компонентой ПО заключается в том, что первый необязательно материален и осязаем (not to be engineered), 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 не обязательно 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, MSF, RUP и др. Например, в MSF это программный код, диаграммы приложений и классов (application diagrams и class diagrams), план итераций (iteration plan), модульный тест (unit test) и др. Для каждого из них точно описано содержание, ответственные за разработку, место в процессе и др. аспекты.

Остановимся чуть детальнее на промежуточных рабочих продуктах. Компонента ПО, созданная в проекте одним разработчиком и предоставленная для использования другому разработчику, оказывается рабочим продуктом. Ее надо минимально протестрировать, поправить имена интерфейсных классов и методов, быть может, убрать лишнее, не имеющее отношение к функциональность данной компоненты, по-лучше разделить 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 эти дополнительных работ существенно возрастает, если компонента должна быть представлена для использования в разработке, например, в другой центр разработки (например, иностранным партнерам, что является частой ситуацией в оффшорной разработке). 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 команде 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.

Итак, подытожим, что промежуточный рабочий продукт должен обязательно иметь ясную цель и конкретных пользователей, чтобы минимизировать накладные расходы на его создание.

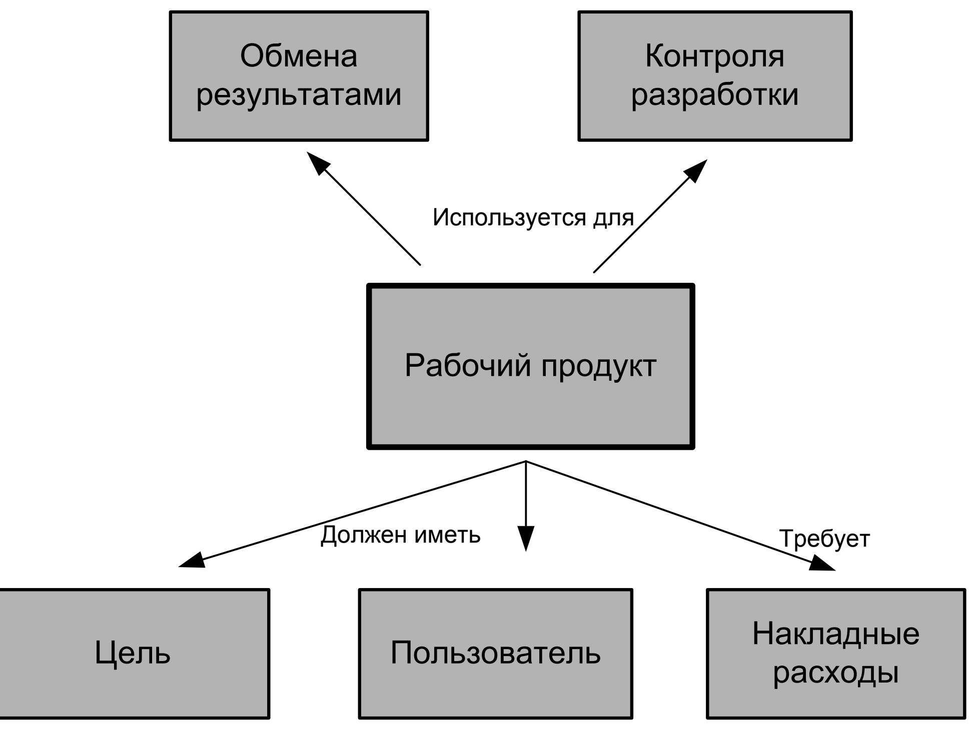

Рис. 3.2.

Дисциплина обязательств

В основе разделения обязанностей в бизнесе и промышленном производстве лежит, корпоративных правил и норм лежит определенная деловая этика, форма отношений – ***дисциплина обязательств.*** Она широко используется на практике и является одним из возможных форм социального взаимоотношения между людьми. Привнесение 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 deadline, 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

Классическое операционное разделение труда идет еще от Адама Смита и является сутью массового индустриального производства. То есть существует четко налаженный процесс работы и имеются области специализации – один цех точит, другой строгает, третий собирает, четвертый красит и т.д. Пропускная способность такого производства намного превосходит выполнение всей работы одним человеком или одной группой. Таким образом в XIX веке операционное разделение труда стало основой мануфактур, вытеснивших индивидуальное, ремесленное производство. В начале XX века эту структуру работ перенесли и на управление – то есть многочисленные менеджеры контролировали отдельные участки работ.

Однако высокий уровень сложности ряда задач в промышленности и бизнесе не позволяет (к счастью!) так работать везде. Существует много творческих, новых задач, где, быть может, в будущем и удастся создать конвейеры, но в данный момент для их решения требуется существенная концентрация сил и энергии людей, неожиданные решения, а также удача и легкая рука. Это и есть область проектов.

Прое́кт – это уникальная (в отличии от традиционной пооперационной промышленного производства) деятельность, имеющая начало и конец во времени, направленная на достижение определённого результата/цель, создание определённого, уникального продукта или услуги, при заданных ограничениях по ресурсами 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 в подобных проектов не только абсолютная – такого еще не было. Такое, может, уже и было, но только не снами, 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 этом 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 (project management) – 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.

- ***Stakeholedrs*** – 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
- ***Project scope*** – это границы проекта. Это очень важное понятие для программных проектов в виду 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. Одна их техник управления компромиссами будет рассказана в контексте изучения методологии MSF.

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

Область управления проектами	**Описание**
Планирование и мониторинг проекта, контроль за изменениями в проекте (Project planning / Tracking / Change Control)	Интеграция и синхронизация планов проекта; организация процедур и систем управления и мониторинга проектных изменений
Управление рамками проекта (Scope Management)	Определение и распределение объема работы (рамок проекта); управление компромиссными решениями в проекте
Управление календарным графиком проекта (Schedule Management)	Составление календарного графика исходя из оценок трудозатрат, упорядочивание задач, соотнесение доступных ресурсов с задачами,

	применение статистических методов, поддержка календарного графика
Управление стоимостью (Cost Management)	Оценки стоимости исходя из оценок временных затрат; отчетность о ходе проекта и его анализ; анализ затратных рисков; функционально-стоимостной анализ (value analysis)
Управление персоналом (Staff Resource Management)	Планирование ресурсов; формирование проектной команды; разрешение конфликтов; планирование и управление подготовкой
Управление коммуникацией (Communications Management)	Коммуникационное планирование (между проектной группой, заказчиком/спонсором, потребителями/пользователями, др. заинтересованными лицами); отчетность о ходе проекта
Управление рисками (Risk Management)	Организация процесса управления рисками в команде и содействие ему; обеспечение документооборота управления рисками
Управление снабжением (Procurement)	Анализ цен поставщиков услуг и/или аппаратного/программного обеспечения; подготовка документов об инициировании предложений (requests for proposals – RFPs), выбор поставщиков и субподрядчиков; составление контрактов и переговоры об их условиях; договора; заказы на поставку и платежные требования
Управление качеством (Quality Management)	Планирование качества, определение применяемых стандартов, документирование критериев качества и процессов его измерения

Литература

1. W.Humphrey. Managing the Software Process. Addison-Wesley, 1989. 494 p.
2. Д.Ахен, А.Клауз, Р.Тернер. CMMI: комплексный подход к совершенствованию процессов. Пер с англ. М.:МФК. 2005. 330 с.
3. Carnegie Mellon University Software Engineering Institute (2006). Retrieved on 22 August 2007. http://www.sei.cmu.edu/publications/documents/06.reports.

Лекция 4. Архитектура ПО

Обсуждение

Как-то раз один менеджер объяснял основные идеи одного достаточно крупного проекта, которым он руководил. Он начертил на доске три кубика: frontend, backend, tools. И сказал, что это и есть главное строение проекта. И в смысле внутреннего устройства продукта, и в смысле распределения работ в команде по трем дистанционно разнесенным центрам разработки. Задачи backend сложные, ресурсоемкие, выполняются пакетно. Они отделены от графического интерфейса продукта (frontend), который также непросто устроен. Frontend – это пользовательский интерфейс: сложный, параметризуемый, с рядом встроенных пользовательских сервисов (в частности, браузер информации), а также настраиваемый. Обе этих подсистемы взаимодействуют друг с другом через хорошо определенный и детально описанный программный интерфейс: алгоритмы backend разбиты на методы, которые frontend может вызывать по особым правилам, с параметрами, выстраивая в цепочку для достижения своих задач. «Сбоку» от всего этого находятся дополнительные tools. Они интегрируются во frontend, но не пользуются методами backend, а реализуют свои задачи самостоятельно. Эти задачи не требуют сложной пакетной обработки, а нацелены на интерактивное взаимодействие с пользователем. При их реализации особенно много внимания уделялось usability.

Каждая из трех подсистем требовала от разработчиков особых навыков. В случае backend это было умение и опыт по реализации такого рода пакетных алгоритмов, в случае с frontend – умение создавать сложный пользовательский интерфейс, в случае с tools требовалось искусство в проектировании и реализации «легковесных» инструментов, предоставляющих пользователям системы дополнительные сервисные возможности. В том, чтобы разделить работы таким образом, был еще и ряд политических аспектов. В частности, руководство проекта хотело иметь процесс разработки пользовательского интерфейса рядом с собой, в одном из трех центров разработки, который совпадал со штаб-квартирой. Считалось, что внешний вид продукта очень важен для его успешной продажи и требует особенного внимания.

В результате выполнения проекта (а он развивался более 15 лет, достигая в апогее до 150 человек, одновременно занятых в нем) такая четкая структура несколько сместилась – так географически интерфейс почти «переехал» в тот центр, где разрабатывался backend. Но в целом такое сквозное разделение проекта на части оставалось много лет и было основным скелетом всей разработки. Это и есть пример архитектуры программного проекта.

Определение

Будем понимать под ***архитектурой ПО*** внутреннюю структуру продукта (компоненты и их связи), основы пользовательского интерфейса продукта, а также квинтесенцию знаний и решений, являющихся инструментом разработки и управления проектом. То есть архитектура – это сквозная концепция или набор таковых для преодоления энтропии и хаоса, стремящихся «проглотить» разработку в виду сложности, нематериальности, согласовываемости и изменчивости ПО. При этом мы не разделяем продукт и проект, так как на практике это, как правило, одно целое, причем эта «сквозность», если она имеется, является «сильной» стороной данной разработки.

Часто под архитектурой понимают например, только внутреннее устройство ПО, выраженное в UML-диаграммах. Вот шутка на тему того, что архитектуру нельзя понимать односторонне. Одного известного трансляторщика спросили, почему в его знаменитом трансляторе ровно 21 просмотр. Ожидали услышать перечисление про алгоритмических проблем, которые таким способом удалось преодолеть, что-то про

особую эффективность алгоритмов, организованных таким образом, и т.д. Всех удивил ответ мэтра. Он сказал, что именно столько человек (то есть ровно 21), было у него в команде разработчиков…..

Итак, архитектура продукта оказывается инвариантом проекта, встречается и неожиданно возникает в его разных частях. Это и есть аналог «простым» естественно-научным постулатам и законам, отсутствие которых в разработке ПО, по мнению Брукса, является причиной сложности ПО (в смысле хаоса, то есть «плохой» сложности). Создавать такие структуры – непростое дело, требующее большого искусства. Но именно это путь к управлению хаосом, увеличивающейся энтропией в виде изменяющихся требований к системе, потере разработчиками ясного понимания, какую же именно систему они создают. И именно разработка таких структур доставляет истинное творческое наслаждение при разработке программных систем. Хорошо «работают» простые модели, которые не просто создавать. Они оказываются путеводной нитью проекта, ведущей его через пучины хаоса. Эти модели имеют такое свойство, что показывая или упоминая о них можно рассказывать о проекте очень долго, их можно красиво оформить и повесить на стенку, а можно этого и не делать.

В рамках многих проектов не создается оригинальной архитектуры, поскольку они являются типовыми и/или небольшими и основываются на готовых технологиях, архитектурных образцах, моделях команды и оргструктуры проекта.

Однако часто перед коллективами, которые хорошо себя зарекомендовали в таких проектах, возникает задача построить действительно оригинальную новую архитектуру, основывающуюся на прежних разработках. Или не основывающуюся – просто количество стремиться перейти в качество. Здесь прежде всего важно заметить этот переход, осознать, что старые методы работы не годятся и требуется принципиально новый опыт. Которого, очень часто, у коллектива и его лидеров нет…..

Множественность точек зрения

При разработке архитектуры ПО важным оказывается совмещение множества точек зрения. ПО оказывается настолько сложным, что его архитектуру не построить как единую модель – множество отдельных аспектов должны быть представлены в архитектуре, их связи сложны и плохо выразимы в явном виде. Полезнее оказывается создание множества моделей, созданных с разных точек зрения.

Причина множественности точек зрения при разработке ПО. Умение рассматривать предмет с разных точек зрения является важнейшей философией успешной практики при работе с большими объемами разнородной и сложной информации. Посмотрим на разработку ПО и то, почему там востребованы разные взгляды на процесс, систему и т.д.

Это происходит, прежде всего, из-за разных видов деятельности процесса разработки ПО (см. рис. 4.1). При составлении функциональных требований к ПО обращают внимание на то, какая именно функциональность должна быть реализована, но при этом опускаются принципы и детали реализации. При проектировании, наоборот, на первое место выходят принципы реализации ПО. А при тестировании детали реализации снова неважны — на ПО смотрят как на черный ящик, реализующий (не важно каким способом) некоторый набор пользовательской функциональности. При развертке у заказчика на ПО смотрят как на набор файлов, хранилищ данных и т. д.

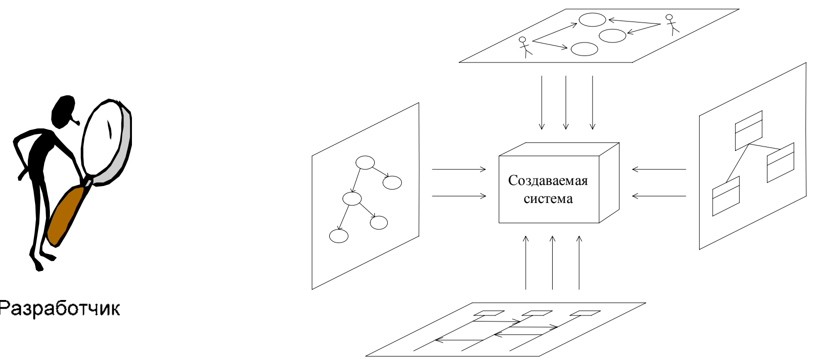

Рис. 4.1 Разные виды деятельности – разные взгляды на систему

Далее, в разработку/использование ПО вовлечено большое количество *очень разных* специалистов: программисты, инженеры, тестеры, технические писатели, менеджеры, заказчик, пользователи, продавцы-маркетологи и т. д. (см. рис. 4.2). Для всех эти специалистов нужна разная информация о программной системе. Представьте, что произойдет, если, например, продавцу или заказчику-непрограммисту в ответ на просьбу получше ознакомиться с ПО вы дадите почитать программные коды…

Рис. 4.2. Разные специалисты – разные взгляды на систему

Множественность точек зрения происходит также от того, что нет единых стандартов и норм разработки ПО. То есть разработка ПО во многом «state of art». Часто приходится изобретать новую точку моделирования зрения прямо по ситуации – чтобы именно этот эксперт тебя понял, чтобы именно эти особенности системы были отражены.

Часто здесь – как в лотереи: создается несколько описаний системы с разных точек зрения, какое-то оказывается удачным и его все используют в дальнейшем.

Итак, разные виды деятельности при разработке ПО, разные категории специалистов, задействованные в программном проекте, и уникальность каждой конкретной ситуации при разработке — все это приводит к созданию и использованию различных моделей, выполненных с разных точек зрения.

Точка зрения (viewpoint) — это определенный взгляд на систему, который осуществляется *для выполнения какой-то определенной задачи кем-либо из участников* проекта. Точку зрения нужно ясно осознавать при создании визуальных моделей, например, модели случаев использования. Важно понимать, что она может быть в каждом конкретном случае своя. Важнейшими характеристиками точки зрения моделирования является ***цель*** (зачем создается модель) и ***целевая аудитория*** (то есть, для кого она предназначается).

Важным вопросом, на который нужно честно себе ответить в самом начале моделирования — это *зачем* вы используйте диаграммы (в частности, UML). Это и есть определение цели моделирования. Потому, что так создавать модели правильно, нужно? И все проблемы (даже те, о которых ничего еще не известно) волшебным образом исчезнут, развеются? Очень часто, например, при создании модели случаев использования присутствует именно такая «цель» моделирования. А потом оказывается, что никакие проблемы не «вылечились», а наоборот, возникли новые (например, созданные нами диаграммы никто не понимает и не принимает). Да и сам аналитик чувствует, что диаграммы получились какие-то странные….

А может все происходить совсем не так. Например, аналитик действительно задался целью выявить требования к системе — не навязать свое собственное видение другим, а выяснить нужную информацию, смоделировать и изложить ее доступно. Для этого он и использует диаграммы случаев использования. Ему важно, чтобы будущие пользователи системы могли участвовать в этом процессе, диаграммы рисуются для них, они понятны и не избыточны. И эти же диаграммы структурируют и проясняют информацию для самого аналитика.

Подобных сюжетов на практике происходит множество. Тут важно понимать, что цель модели — это не какая-то гипотетическая задача типа «описания архитектуры, потому что так нужно, так правильно», а целевая аудитория — это не абстракция типа «люди, желающие познакомиться с ПО». И то и другое — что-то очень конкретное, реально существующее в проекте или рядом с ним. Ведь разработчики ПО не могут позволить себе за деньги заказчика создавать нечто на все века и для всех народов. И цель моделирования, и аудитория, которая будет работать с диаграммами, всегда существуют, важно лишь ясно понимать, какие они…

Вот полезный практический прием для ориентации на целевой аудиторию, для которой предназначена создаваемая вами модель. Можно выбрать одного представителя такой аудитории — конкретного и известного вам человека — и создавать диаграммы, понятные именно ему. При этом важно не обсуждать чрезмерно с ним ваши модели, поскольку это может создать дополнительный контекст, которого другие пользователи моделей будут лишены. Полезно представлять воображать себе этого человека при работе над моделями — его реакции, вопросы, недоумения и пр. И, исходя из этого, корректировать, исправлять созданное. И, конечно же, полезно проверить свои предположения, показав ему, что получилось.

Кроме того, важно, чтобы точка зрения была «живая», а не выдумывалась аналитиком или бездумно копировалась из книжек и тренингов, посвященных UML. Незаметно для себя аналитик может придумать свой собственный проект, своих собственных пользователей системы, заказчика и т.д. То есть аналитик исподволь,

навязывает самому себе определенное восприятие реально существующих людей, задач, сильно искажая реальное положение дел. И именно в контексте этой воображаемой ситуации он создает свои модели. Но ведь реальные люди, реальные ситуации обладают своеобразием, большим диапазоном вариативности. Соответственно, аналитик должен обладать гибкостью сознания, большим диапазоном техник, а также чуткостью и искренним стремлением к тому, чтобы сделать каждый конкретный проект, где он участвует, более гармоничным, более адекватным.

Язык UML

Часто понятие архитектуры сильно сужают, понимая под ним лишь описание основных, важных аспектов ПО, создаваемых, например, архитектором при разработке дизайна системы. Для этих целей используется язык моделирования UML (Unified Modeling Language).

Этот язык является итогом развития средств схематического описания программных систем, которые развивались с блок-схем, предложенных еще фон Нейманом в конце 40-х годов. Он предполагал, что эти схемы станут высокоуровневым языком ввода алгоритмов в вычислительные машины, но эволюция языков программирования пошла по пути текстовых языков. Тем не менее блок-схемы получили распространение при спецификации и документировании ПО, были стандартизованы, однако широкого практического применения не получили. В конце 60-х годов, в связи с поиском новых средств разработки ПО, рождением программной инженерии и общими следованиями в области проектирования и разработки искусственных систем появился термин структурный анализ (structured analysis) систем. Термин был введен ученым из MIT, Дугласом Россом, который также предложил диаграммный метод анализа и проектирования больших искусственных систем. Метод назывался SADT (Structured Analisys and Desing Technique), стал основой серии военных стандартов США серии IDEF и широко распространился в индустрии. Однако диаграммный язык в SADT был очень скромным – набор блоков и связей между ними, с поддержкой декомпозиции блоков. В 70-х годах, в связи с массовым выходом ПО на свободный рынок (то есть ghjuhfvvyst системы стали создаваться не только в военной области, для крупного бизнеса, но также для среднего и малого бизнеса) структурный анализ стал бурно эволюционизировать – набор диаграмм обогатился диаграммами состояний и переходов, сущность-связь, потоков данных и т.д. С развитием объектно-ориентированных средств разработки (конец 80-х – середина 90-х) структурный анализ превратился в объектно-ориентированный анализ и проектирование. Появилось большое количество методологий, и постепенно сложился единый язык моделирования, который и был закреплен в стандарте UML. Произошло это в 1997 году.

С тех пор вышло несколько версий стандарта UML. Текущая версия UML 2.1.

Виды диаграмм. «Скелетом» UML является диаграммная структура. Каждый вид диаграмм является типом моделей, реализующим определенную точку зрения на программную систему. Виды диаграмм не являются строго обязательными в UML – их можно перемешивать, создавать свои собственные виды диаграмм. Тем не менее стандартные виды диаграмм являются определенным достоянием программной инженерии, так как отражают опыт многих исследователей и практиков.

- Структурные диаграммы:
  - диаграммы классов (class diagrams) предназначены для моделирования структуры объектно-ориентированных приложений классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;
  - диаграммы компонент (component diagrams) используются при моделировании компонентной структуры распределенных приложений;

внутри каждая компонента может быть реализована с помощью множества классов;
  - диаграммы объектов (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;
  - диаграммы композитных структур (composite structure diagrams) используются для моделирования составных структурных элементов моделей – коопераций, композитных компонент и т.д.;
  - диаграммы развертывания (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);
  - диаграммы пакетов (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.
- Поведенческие диаграммы:
  - диаграммы активностей (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;
  - диаграммы случаев использования (use case diagrams) предназначены для «вытягивания» требований из пользователей, заказчика и экспертов предметной области;
  - диаграммы конечных автоматов (state machine diagram) применяются для задания поведения реактивных систем;
  - диаграммы взаимодействий (interaction diagram):
    - диаграммы последовательностей (sequence diagram) используются для моделирования временных аспектов внутренних и внешних протоколов ПО;
    - диаграммы схем взаимодействия (interaction overview diagram) служат для организации иерархии диаграмм последовательностей;
    - диаграммы коммуникаций (communication diagrams) являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);
  - временные диаграммы (timing diagrams) являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

Примеры. Центральным видом диаграмм являются диаграммы классов. Пример представлен на рис. 4.3.

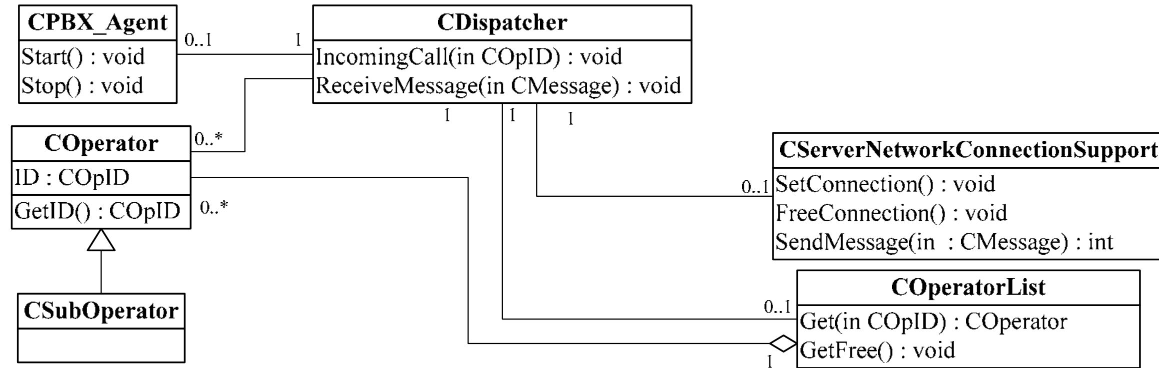

Рис. 4.3. Пример диаграмм классов.

Еще один вид структурных диаграмм – диаграммы размещений, пример представлен ниже.

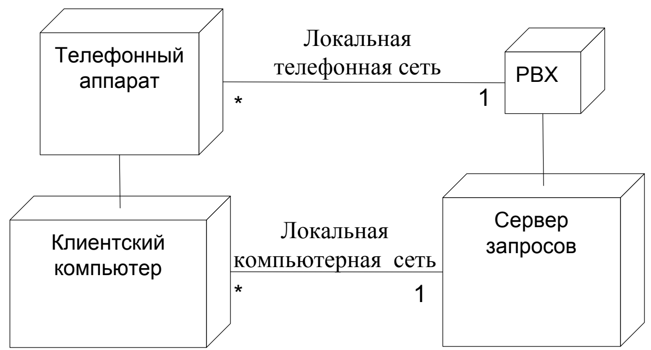

Рис. 4.4. Пример диаграмм размещений.

Отметим также еще один важный вид диаграмм UML – диаграммы компонент (пример представлен на рис. 4.5).

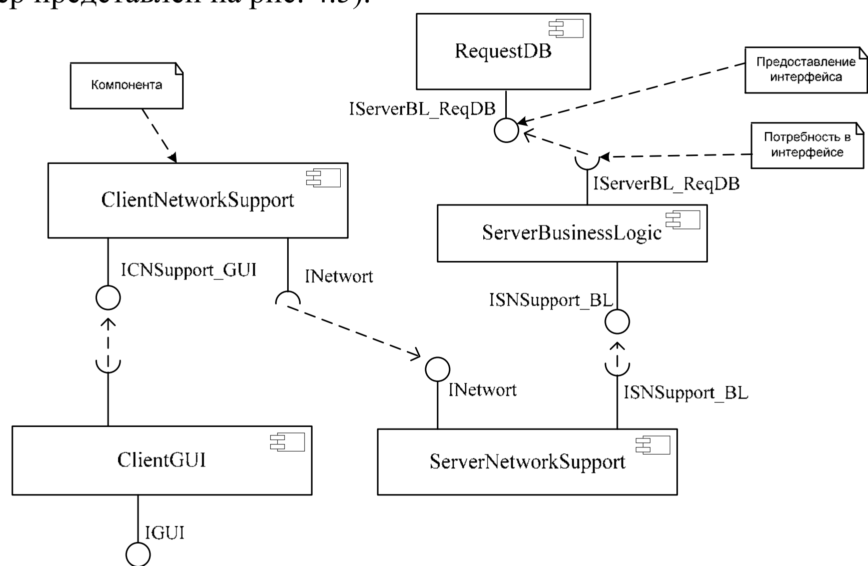

Рис. 4.5. Пример диаграмм компонент.

Интересен также вариант диаграмм композитных структур – сложные компоненты для систем реального времени и телекоммуникаций. Пример представлен ниже.

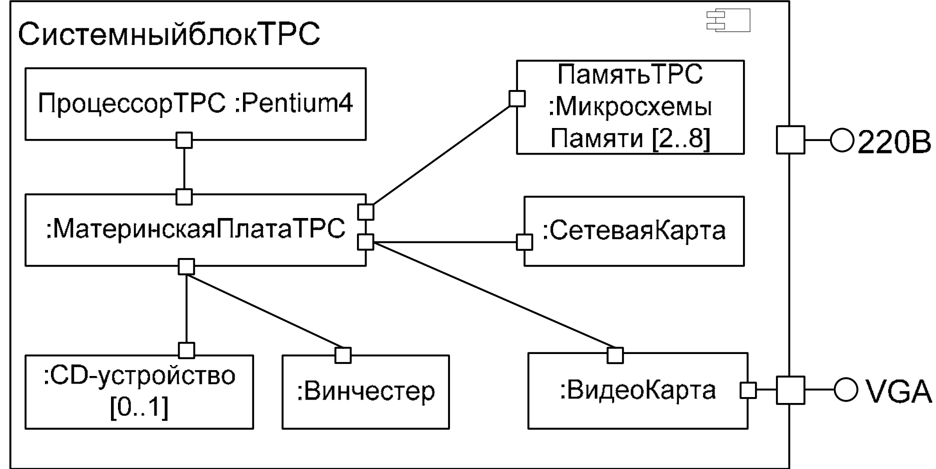

Рис. 4.6. Пример диаграмм композитных структур.

Ниже приводятся примеры на поведенческие диаграммы UML. Диаграммы конечных автоматов позволяют создавать полные спецификации поведения телекоммуникационных, событийно-управляемых алгоритмов и автоматически генерировать по этим описаниям программный код. Пример такой диаграммы для класса COperator представлена ниже.

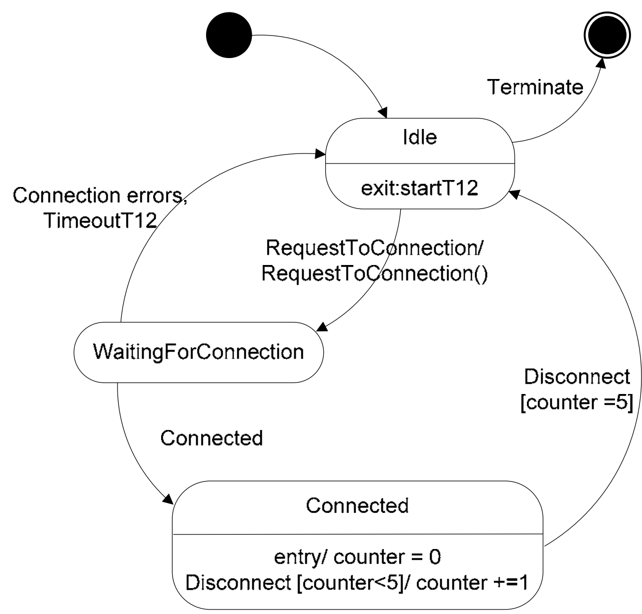

Рис. 4.7. Пример диаграмм конечных автоматов.

Еще один важный вид диаграмм – диаграммы последовательностей. Они позволяют задавать главные ветки сложных телекоммуникационных алгоритмов, а также рисовать цепочки вызовов для объектно-ориентированных приложениях, которые программируются в терминах объектов, но проектируются часто в терминах цепочек вызовов. Пример представлен ниже.

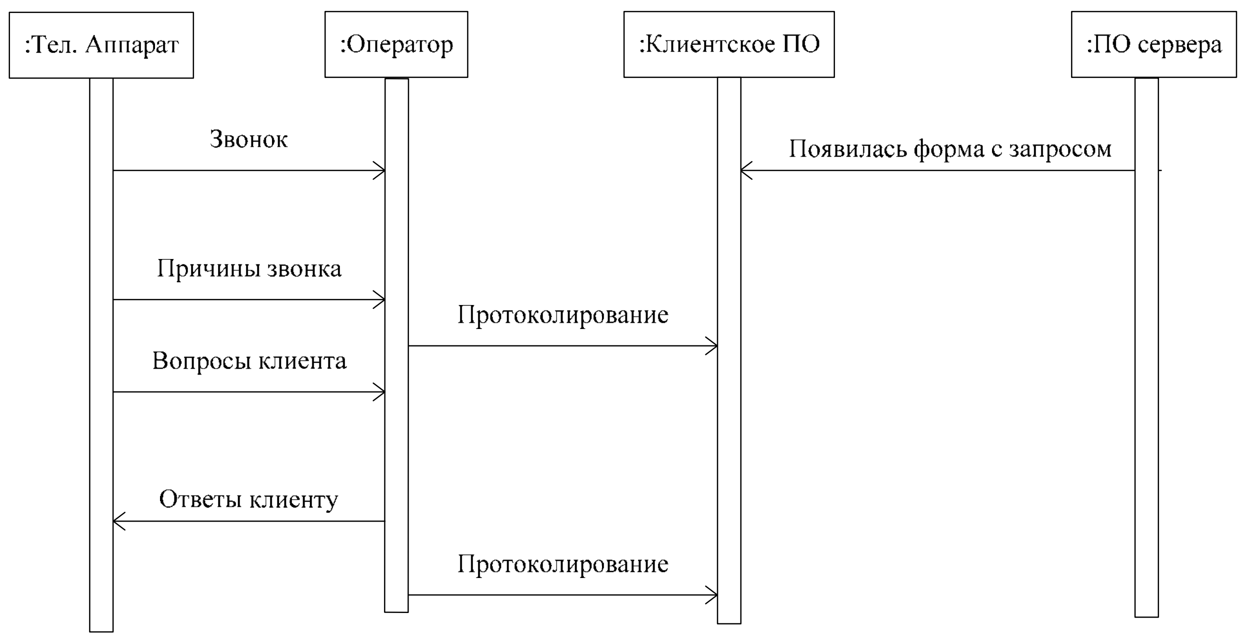

Рис. 4.8. Пример диаграмм последовательностей.

Литература

1. UML 2.1 Infrastructure Specification, September, 2006, http://www.omg.org/.
2. Kruchten P. The 4+1 View Model of Architecture. IEEE Software, 1995, 12(6), P. 42-50.
3. Буч Г., Якобсон А., Рамбо Дж. UML. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
4. Д. В.Кознов. Основы визуального моделирования / Д. В. Кознов. – М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. – 248 с.: ил. – (Серия «Основы информационных технологий»).

Лекция 5. Управление требованиями

Проблема

Например, строители строят дома, пусть разные: многоэтажные, отдельные коттеджи, офисные здания и пр. – однако, весь этот спектр вполне может охватить одна компания. Но все это дома. Строительной компании не приходится строить летающую тарелку, гиперболоид инженера Гарина, луноход, систему мгновенной телепартации и пр. А разработчики ПО, во многом, находятся именно в таком положении.

Велико разнообразие систем, которые создает одна компания, одна команда. Хотя сейчас и намечаются тенденции к специализации рынка разработки ПО, однако, причуды мировой экономики и многие другие причины приводят к тому, что строго специализированных компаний не так много, как хотелось бы. Многие области испытывают большой дефицит отдельных программистов и целых коллективов и компаний, хорошо разбирающихся в их специфики. Примером такой области может служить телевидение, где о данной проблеме открыто говорят на заседаниях различных международных сообществ.

Кроме того, ПО продолжает проникать во все новые и новые области человеческой деятельности, и сформулировать адекватные требования в этом случае вообще оказывается супертрудной задачей.

Но даже если речь идет об одной, определенной области, то процент новых, уникальных черт систем, принадлежащих этой области, высок: по сочетанию пользовательских характеристик, по особенностям среды исполнения и требованиям к интеграции, по распределенности информации о требованиях среди работников компании-заказчика. Все это несет на себе очень большой отпечаток индивидуальности заказчика – персональной или его компании, – сильно связано со спецификой его бизнеса, используемого в этой области оборудования.

Кроме того, существуют трудности в понимании между заказчиком и программистами, а еще – в изменчивости ПО (требования имеют тенденцию меняться в ходе разработки).

В итоге, далеко не очевидно, что та система, которую хочет заказчик, вообще можно сделать. Трудно найти черную кошку в темной комнате, особенно если ее там нет. Или то, как поняли и воплотили задачу разработчики, окажется удобным, востребованным на рынке.

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.

Несколько слов о трудности взаимопонимания заказчика и разработчиков. Здесь сказывается большой разрыв между программистами и другими людьми. Во-первых, потому, что чтобы хорошо разобраться, какой должна быть система автоматизации больницы и система поддержки химических экспериментов – надо поработать в соответствующей области достаточное время. Или как-то иным способом научиться видеть проблемы данной предметной области изнутри. Во-вторых, сказывается специфичность программирования как сферы деятельности. Для большинства пользователей и заказчиков крайне не просто сформулировать точное знание, которое необходимо программистам. На вопрос, сколько типов анализов существует в вашей лаборатории, доктор, подумав, отвечает - 43. И уже потом, случайно, программист уточнил, а нет ли других типов? Конечно, есть, ответил доктор, только они случаются редко и могут быть в некотором смысле, какими угодно. В первый же раз он назвал лишь типовые. Но, конечно же, информационная система должна хранить информацию обо всех анализах, проведенных в лаборатории….

Теперь чуть подробнее об изменчивости ПО и его причинах.

- Меняется ситуация на рынке, для которого предназначалась система или требования к системе ползут из-за быстро сменяющихся перспектив продажи еще неготовой системы.
- В ходе разработки возникают проблемы и трудности, в силу которых итоговая функциональность меняется (видоизменяется, урезается).
- Заказчик может менять свое собственное видение системы: толи он лучше понимает, что же ему на самом деле надо, толи выясняется, что он что-то упустил с самого начала, толи выясняется, что разработчики его не так поняли. В общем, всякое бывает, важно лишь, что теперь заказчик определенно хочет иного.

Нечего и говорить, что изменчивость требований по ходу разработки очень болезненно сказывается на продукте. Авторы сталкивались, например, с такой ситуацией, что еще не созданную систему отдел продаж начинает активно продавать, в силу чего поступает огромный поток дополнительных требований. Все их реализовать в полном объеме не удается, в итоге система оказывается набором демо-функциональности….

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.

Нефункциональные требования не являются описанием функций системы. Этот вид требований описывает такие характеристики системы, как надежность, особенности поставки (наличие инсталлятора, документации), определенный уровень качества (например, для новой Java-машины это будет означать, что она удовлетворяет набору тестов, поддерживаемому компанией Sun). 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.

Сформулируем ряд важных свойств требований.

- *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, тестеровщиков, 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 сильно не совпадать с таковыми для модели требований. 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 в виду 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 какого-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 очень 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 и нек. др.). 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.

- Описание возможных решений вместо требований.
- Нечеткие требования, которые не допускают однозначную проверку, оставляют недосказанности, имеют оттенок советов, обсуждений, рекомендаций: «Возможно, что имеет смысл реализовать также…..», «и т.д.».
- 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.
- Попуск 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 частью 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 определяется следующие виды деятельности при работе с требованиями.

- *Выделение требований (requirements elicitation)*, нацеленное на выявление всех возможных источников требований и ограничений на работу системы и извлечение требований из этих источников.
- *Анализ требований (requirements analysis)*, целью которого обнаружение и устранение противоречий и неоднозначностей в требованиях, их уточнение и систематизация.
- *Описание требований (requirements specification)*. 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.
- *Валидация требований (requirements validation)*, 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.

Литература

1. В. В. Кулямин, Н. В. Пакулин, О. Л. Петренко, А. А. Сортов, А. В. Хорошилов. Формализация требований на практике. Препринт ИСП РАН 2005. 50 с.
2. Guide to the Software Engineering Body of Knowledge: 2004 Edition – SWEBOK. IEEE, 2005.
3. Кулямин В.В. Технология программирования. Компонентный подход. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. 463 с.
4. I.Sommerville Software Engineering. 6th edition. Addison-Wesley, 2001. 693 p. / Русский перевод: И.Соммервилл. Инженерия программного обеспечения. Издательский дом «Вильямс», 2002. 623 с.

Лекция 6. Конфигурационное управление

Проблема

Всем известно, что на крупных промышленных предприятиях, в магазинах, книжных издательствах и пр. существуют склады. Основная задача склада – обеспечить хранение и доступ к материальным активам: товарам, изделиям, книгам и пр. То есть различных материальных активов становится так много, что необходима специальная служба по их учету. Оказывается, что не достаточно складывать, например, все, имеющиеся в книгоиздательстве книги в специальную комнату и выдавать их владельцам тиража, когда они за ним придут. Книг оказывается очень много, а процедура выдачи тиража – не совсем тривиальной. Нужно, чтобы владелец принес большое количество сопроводительных документов, и все они должны быть проверены перед выдачей книг. А на самом складе необходимо поддерживать порядок, чтобы было возможно быстро найти нужные книги (как показывает опыт, они могут там довольно долго находиться). Еще более сложная процедура работы с книгами в библиотеке – там добавляются еще каталоги, распределенные книжные хранилища, необходимость поддерживать хорошее состояние книг, а также контролировать возврат их в библиотеку после определенного срока. Аналогичным образом работает склад на любом заводе, фабрике и т.д.

Рассмотрим теперь проект по разработке программного обеспечения. Что в нем является аналогом материальных активов на обычном производстве? Определенно, не столы и стулья, которыми пользуются разработчики. И даже не компьютеры, запчасти к ним и прочее оборудование. Учета и контроля, сродни складскому, требуют ***файлы*** проекта. В программном проекте их очень много – сотни и тысячи даже для относительно небольших проектов. Ведь создать новый файл очень легко. Многие технологии программирования поддерживают стиль, когда, например, для каждого класса создается свой отдельный файл.

Файл – это виртуальная информационная единица. В чем главное отличие файла от материальных единиц учета? В том, что у файла может быть ***версия***, и не одна, и породить эти версии очень легко – достаточно скопировать данный файл в другое место на диске. В то время как материальные предметы существуют на складе сами по себе, и для них нет понятия версии. Да, может быть несколько однотипных предметов, разных заготовок изделия различной степени готовности. Но все это не то….. А версия файла – это очень не простой объект. Чем одна версия отличается от другой? Несколькими строчками текста или полностью обновленным содержанием? И какая из двух и более версий главнее, лучше? К этому добавляется еще и то, что многие рабочие продукты могут состоять из набора файлов, и каждый из них может иметь по несколько версий. Как собрать *корректную версию продукта*?

В итоге в программном проекте начинают происходить мистические и загадочные события.

- Тщательно оттестированная программа на показательных испытаниях не работает
- Функциональность, о которой долго просил заказчик и которая была, наконец, добавлена в продукт, и новая версия торжественно отослана заказчику, таинственным образом исчезла из продукта.
- На компьютере разработчика программа работает, а у заказчика – нет….

Разгадка проста – все дело в версиях файлов. Там, где все хорошо, присутствуют файлы одной версии, а там, где все плохо – другой. Но беда в том, что «версия всего продукта» – это абстрактное понятие. На деле есть версии отдельных файлов. Один или

несколько файлов в поставке продукта имеют не ту версию – все, дело плохо. Необходимо управлять версиями файлов, а то подобная мистика может стать огромной проблемой.

Она серьезно тормозит внутреннюю работу. То разработчики и тестеры работают с разными версиями системы, то итоговая сборка системы требует специальных напряжения усилий всего коллектива. Более того, возможны неприятности на уровне управления. Различные курьезные ситуации, когда заявленная функциональность отсутствует или не работает (опять не те файлы послали!), могут сильно портить отношения с заказчиком. Недовольный заказчик может потребовать даже денежной компенсации за то, что возникающие ошибки слишком по долгу исправляются. А будет тут не долго, когда разработчики не могут воспроизвести и исправить ошибку, так как не могут точно определить, из каких же исходных текстов была собрана данная версия!

Итак, становится понятно, что в программных проектах необходима специальная деятельность по поддержанию файловых активов проекта в порядке. Она и называется ***конфигурационным управлением***.

Выделим две основные задачи в конфигурационном управлении – ***управление версиями*** и ***управление сборками***. Первое 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, настройки компиляции, и интегрируемый с процессом автоматического тестирования. Эта процедура является мощным средством интеграции проекта, основой итеративной разработки.

Единицы конфигурационного управления

Так чем же мы управляем в рамках этой деятельности? Любыми ли файлами, которые имеются в проекте? Нет, не любыми, а только теми, которые изменяются. Например, файлы с используемым в проекте покупным ПО должны себе спокойненько покоиться на CD-дисках или в локальной сети. Книги, документы с внешними стандартами, используемыми в проекте (например, в телекоммуникациях очень много разных стандартов на сетевые интерфейсы) и пр. также должны просто храниться там, где каждый желающий их может взять. Как правило, такой информации в проекте немного, но, разумеется, она должна быть в порядке. Однако ради этого специальный вид деятельности в проекте не нужен.

Итак, конфигурационное управление имеет дело с меняющимися в процессе продуктами, состоящими из наборов файлов. Такие продукты принято называть ***единицами конфигурационного управления*** (configuration management items). Вот примеры:

1. пользовательская документация;
2. проектная документация;
3. исходные тексты ПО;
4. пакеты тестов;
5. инсталляционные пакеты ПО;
6. тестовые отчеты.

У каждой единицы конфигурационного управления должно быть следующее.

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 и пр. 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 там и пр. Более высокоуровневые правила, связанные, например, с правилами изменения тестов и тестовых пакетов при изменении кода. Однако, где-то здесь лежит водораздел между конфигурационным управлением и иными видами деятельности в проекте

4. Автоматическая процедура контроля целостности элемента – например, сборка для исходных текстов программ. Есть не у всех элементов, например, может не быть у документации, тестовых пакетов.

Элементы конфигурационного управления могут образовывать иерархию. Пример представлен на рис. 5.1.

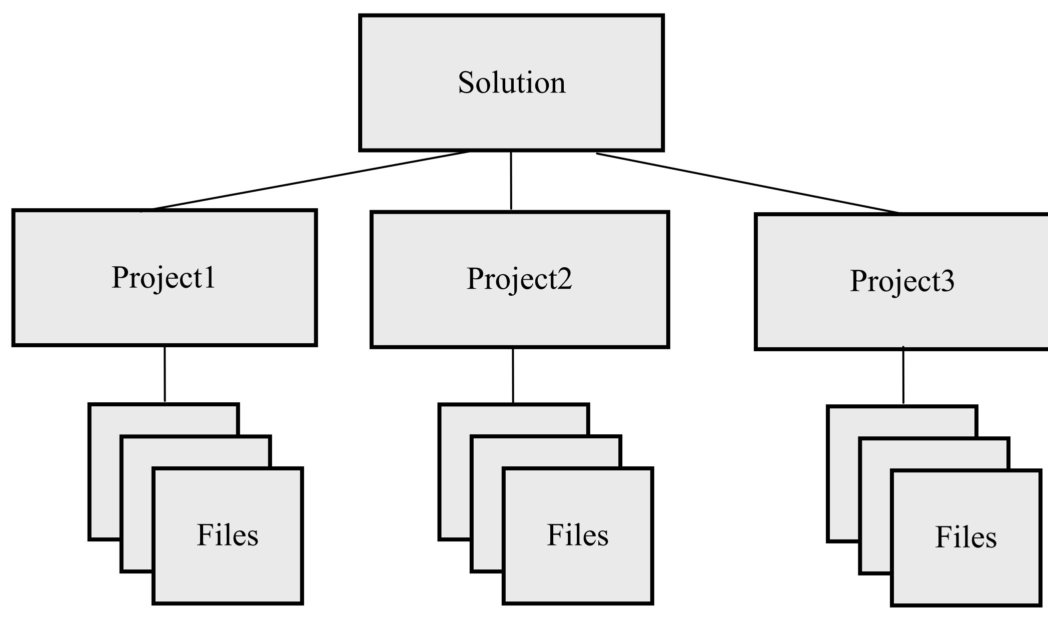

Рис. 5.1.

Управление версиями

Управление версиями файлов. Поскольку программисты имеют дело с огромным количеством файлов, многие файлы в один момент могут быть необходимы нескольким людям и важно, чтобы все они постоянно составляли единую, как минимум, компилирующуюся версию продукта, необходимо, чтобы была налажена работа с файлами с исходным кодом. Также может быть налажена работа и с другими типами файлов. В этой ситуации файлы оказываются самыми младшими (по иерархии включения) элементами конфигурационного управления.

Управление версиями составных конфигурационных объектов. Понятие «ветки» проекта. Одновременно может существовать несколько версий системы – и в смысле для разных заказчиков и пр. (так сказать, в большом, настоящем смысле), и в смысле одного проекта, одного заказчика, но как разный набор исходных текстов. И в том и в другом случае в средстве управления версиями образуются разные ***ветки***. Остановимся чуть подробнее на втором случае.

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.

- V1.0 – ветвь, соответствующая выпущенному релизу. Внесение изменений в такие ветви запрещены 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.
- Mainline – ветвь, соответствующая основному направлению развития проекта. По мере созревания именно от этой ветви отходят ветви готовящихся релизов.
- WCF Experiment – 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 убиваются по завершению работ после интеграции с основной веткой.

Управление сборками

Итак, почему же процедура компиляции и создания exe dll файлов по исходникам проекта – такая важная процедура? Потому что она многократно в день выполняется каждым разработчиком на его собственном компьютере, с его собственной версией проекта. При это отличается:

- 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 учете параметров компиляции (в частности, переход в Visual Studio с debug на release версию часто сопровождается появлением многочисленных проблем).

В связи с этим процедуру сборки проекта часто автоматизируют, то есть 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. 5.2.

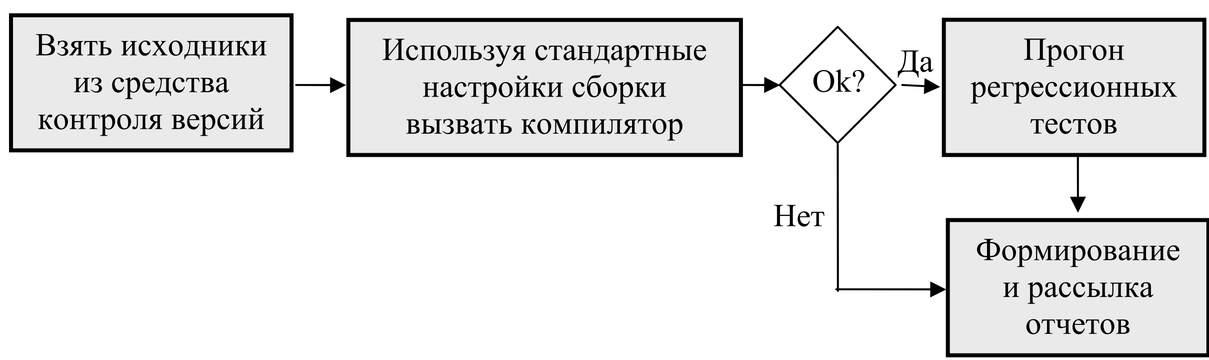

Рис. 5.2

Тестеровщики 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 тестеровщикам, 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. Но поддержка baseline может быть сложной формальной процедурой, как показано на рис. 5.3.

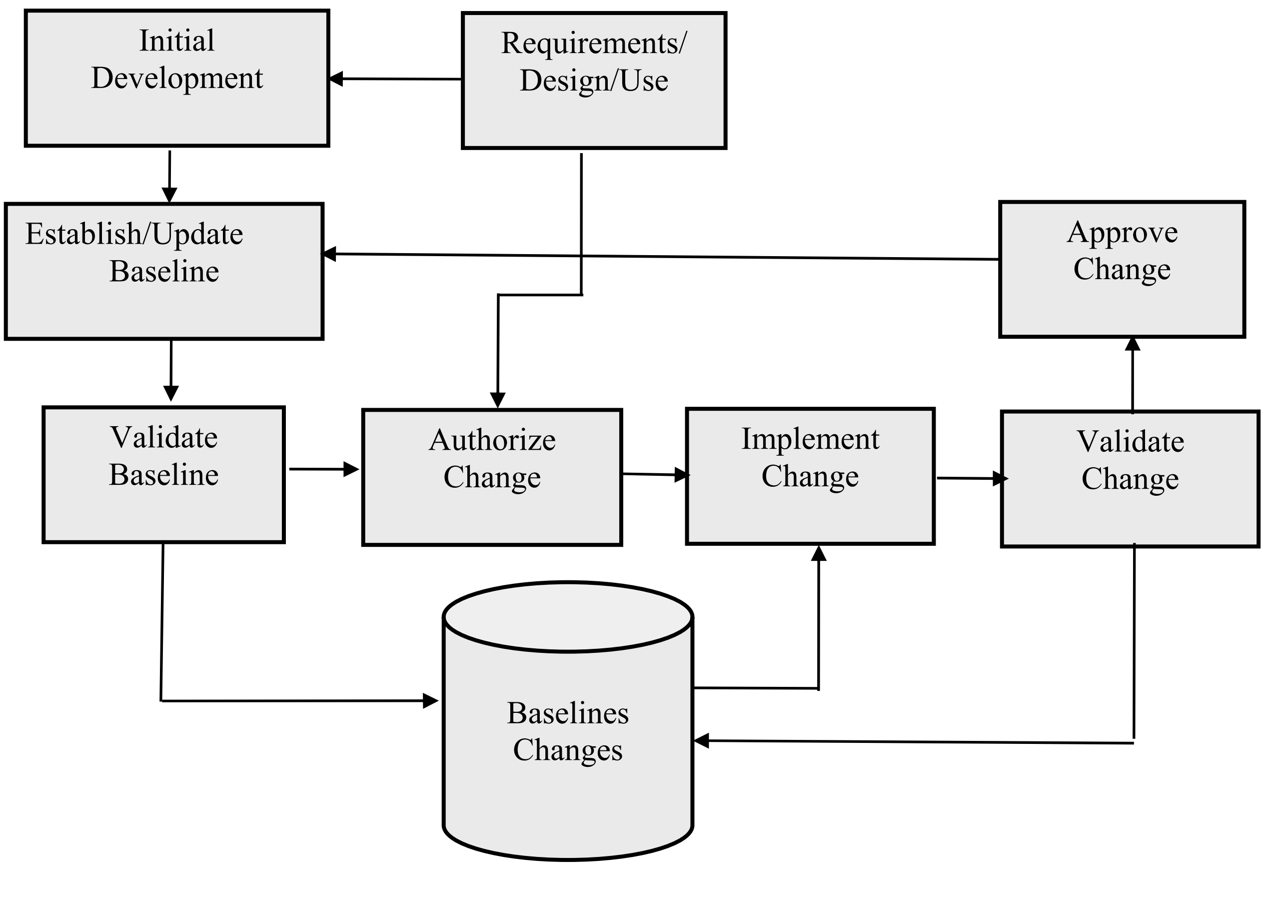

Рис. 5.3.

Baseline может также поддерживаться непрерывной интеграцией.

Важно, что Baseline (особенно в случае в 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 не востребованной. Но начиная с какого-то момента она просто необходима. Какой этот момент – решать членам команды. Наконец, существую проекты, где автоматическая сборка не нужна вовсе – это простые проекты, разрабатываемые небольшим количеством участников, где нет большого количество исходных текстов программ, проектов, сложных параметров компиляции.

Литература

1. W.Humphrey. Managing the Software Process. Addison-Wesley, 1989. 494 p.
2. R. W. Selby, M. A. Cusumano. Microsoft Secrets HarperCollins Business, 1997, 512 p.
3. Carnegie Mellon University Software Engineering Institute (2006). Retrieved on 22 August 2007. http://www.sei.cmu.edu/publications/documents/06.reports.

Лекция 7. Тестирование

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 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. Остановимся на самых известны международных комитетах.

1. 1865 год – образован комитет, который ныне называется ITU (International Telecommunication Union). Сейчас штаб-квартира в Женеве (Швейцария), а ITU является часть ООН. Его основная задача – стандартизация телекоммункационных 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. 1946 год – создана организация ISO (International Organization for Standardization). Цель – содействие развитию стандартизации, 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. 1988 год, образование организации ETSI (European Telecommunications Standards Institute), штаб-квартира в г. София 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. 1984 год – создание SEI (Software Engineering Institute) на базе университета Карнеги-Меллон в г.Питсбурге (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 стало международной деятельностью. Некоторые продукты SEI стандартизованы также ISO. На соответствие CMM/CMMI проводится сертификация.

2. 1963 год – создание IEEE (Institute of Electrical and Electronics Engineers). Ведет историю с конца XIX века, в контексте промышленной стандартизацией в США. Сейчас IEEE международная некоммерческая ассоциация 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. 1989 год – группа американских IT-компаний (в том числе Hewlett Packard, Sun Microsystems, Canon) организовали OMG (Object Management Group). Сейчас включает около 800-т компаний членов. Основное направление - разработка и продвижение объектно-ориентированных технологий и стандартов, в том числе для создания платформо-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 разработки. Здесь срабатывает идея о том, 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. Не претендуя на абсолютную полноту, перечислим различные способы контроля качества, используемые на практике при разработке ПО.

- **Наладка качественного процесса,** другими словами совершенствование процесса. Для комплексного улучшения процессов в компании (подход technology push) компаниями-разработчиками ПО используются стандарты CMM/CMMI, а также по стандартам серии ISO 9000 (с последующей официальной сертификацией). Применяются и локальные стратегии, менее дорогостоящие и более направленные на решению отдельных проблем (подход organization pull).

- **Формальные методы**[4] – использование математических формализмов для доказательства корректности, спецификации, проверки формального соответствия, автоматической генерации и т.д.:
  - доказательство правильности работы программ,
  - проверка на моделях определенных свойств (model cheking),
  - статический анализ кода по дереву разбора программы (например, проверка корректности кода по определенным критериям – аккуратная работа с памятью, поиск мертвого кода и пр.),
  - модельно-ориентированное тестирование (model-based testing): автоматическая генерация тестов и тестового окружения по формальным спецификациям требований к системе) и т.д.

  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 др. характеристик путем запуска и непосредственных наблюдений в виде графиков, отчетов и пр. В частности, этот подход используется при распараллеливании программ, при поиске «узких» мест. Еще пример – область, называемая «моделирование и анализ производительности» (performance modeling and analysis). Здесь моделируется

---

[4] Формальные методы понимаются в двух смыслах: в узком, как математизированные подходы к разработке ПО и в широком – как методы, основывающиеся на четких предписаниях, языкам и пр. Здесь мы будем рассматриваем формальные методы в узком смысле.

нагрузочное окружение системы (число одновременных пользователей системы, сетевой трафик и пр.) и наблюдается поведение системы.

- **Обеспечение качества кода.** Сюда относится целый комплекс различных мероприятий и методов. Вот некоторые, самые известные из них.

  - 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 и т.д.
  - Регулярный рефакторинг для предотвращения образования из кода «вермишели». Существует тенденция ухудшения структуры кода при внесении в него новой функциональности, исправления ошибок и пр. Появляется избыточность, образуются неиспользуемые или слабо фрагменты, структура становится запутанной и трудной для понимания. *Рефакторинг* – это регулярная деятельность по переписываю кода, но не с целью добавления ново функциональности, а для улучшения его структуры. Рефакторинг появился в контексте «гибких» методов, в данный момент активно поддерживается различными средами разработки ПО.
  - 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.
  - Есть еще такой подход, как «вычитка» кода, используемый, например, при разработке критических систем реального времени. Ею занимаются также разработчики, но их роль в данном проекте – вычитка, а не разработка.

- **Тестирование**. Самый распространенный способ контроля качества ПО, представленный, фактически, в каждом программном проекте.

Тестирование

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

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 тестеровщикам, а тестируется только ее интерфейс. Часто это закрепляется и организацией коллектива – тестеровщики оказываются отдельными сотрудниками и в некоторых компаниях они даже принципиально не общаются с разработчиками, чтобы минимально знать реализационных деталей и максимально полно выступить в роли проверяющей инстанции. Существует тестирование методом ***белого ящика***, когда код

программ доступен тестеровщикам и используется в качестве источника информации о системе[5]. Его схема представлена на рис. 6.1.

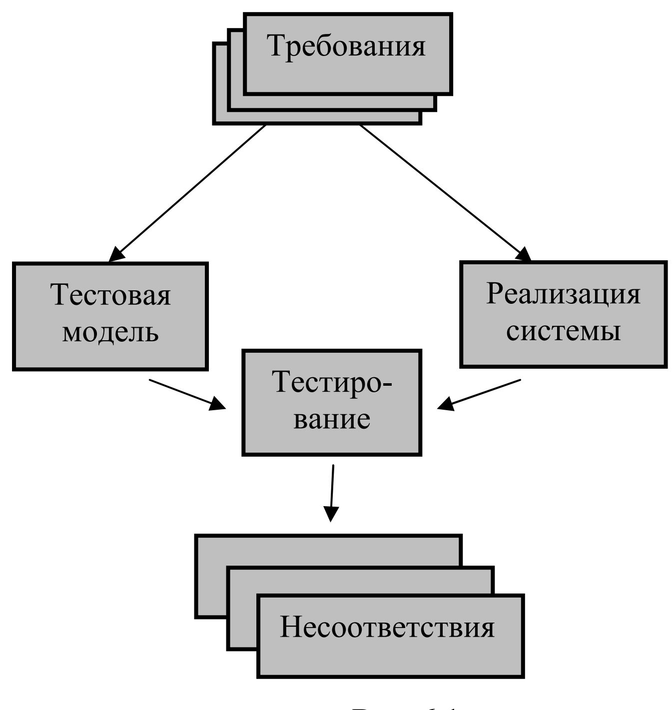

Рис. 6.1.

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 больше прока от их сопоставления. Иначе, если тестеровщики 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 тестеровщики могли неправильно понять требования, в тестах и средствах тестирования могли быть ошибки.

Данный подход закрепляется также и в организации коллективов программистов - тестеровщики, как правило, отделены от разработчиков. Это разные люди, несовместимые роли в MSF. Авторы слышали рассказ об одной американской компании где разработчики и тестеровщики сидели на разных этажах, ходили в разной одежде (тестеровщики в костюмах, разработчики – в свитерах) и начальство не поощряло нерабочие отношения между этими группами. Это, конечно же, крайность, но она еще раз подчеркивает, как важно, чтобы точка зрения на систему у тестеров отличалась от точки зрения разработчиков. Но, конечно, и та и другая должны исходить из общего видения системы – ее требований.

Специально заданные, искусственные условия, – те условия, где осуществляется тестирование. При этом ключевым аспектом здесь является наличие ***тестов*** – воспроизводимых шагов манипуляции с системой, приводящих к ее некорректной работе. Концепция теста очень важна, так как необходимо не просто обнаружить некорректное поведение системы, а создать и зафиксировать алгоритм воспроизведения ошибки – чтобы

---

[5] Необходимо отметить, что тестирование методом черного ящика является наиболее распространенным подходом, хотя, как это часто бывает, на практике часто реализуется смешанный вариант.

повторить его для разработчика или чтобы разработчик сам смог воспроизвести ошибку. Если ошибка не воспроизводится, то нет возможности ее исправить.

Тесты могут быть «ручными» и автоматизированными. «Ручной» тест – это последовательность действий тестеровщика, 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. Поговорим теперь о трудностях автоматического тестирования.

Во-первых, для того, чтобы тесты автоматически запускать, нужны соответствующие программные продукты, которые также являются неотъемлемой частью специально заданных, искусственных условий, которые мы сейчас обсуждаем. 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 оказывается 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 тестировании 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 тех аспектах ПО, которые наиболее важны -как для общей работоспособности системы, так и для заказчика. Например, при тестировании Web-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 (не на всех возможных!) 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 разным ее логическим веткам (порождаемым, например, условными операторами).

Виды тестирования. Не претендуя на полноту, выделим следующие виды тестирования.

- **Модульное тестирование** - тестируется отдельный модуль, в отрыве от остальной системы. Самый распространенный случай применения – тестирования 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** – две и более компонент 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 самим программистами, чтобы, как минимум, удостовериться, что все живет вместе в первом приближении. Далее тонкости интеграции могут исследовать тестеровщики. 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 тестеровщики, менеджеры и разработчики акцентируются на том, как ПО выглядит и работает в целом, удобно ли оно, удовлетворяет ли она 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 не регресс. 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 поведение серверного ПО при большом количестве

клиентских соединений, эксперименты с предельным трафиком для сетевых и телекоммуникационных систем, одновременное открытие большого числа файлов, проектов и т.д.

- **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.

Работа с ошибками

Между программистами и тестеровщиками необходим специальный интерфейс общения. Ведь ошибок находится много, их исправление требует времени, и их исправления разработчиками тестеровщики 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. 6.2. Что справиться с этим потоком информации и обеспечить необходимые в работе, удобные сервисы, существует специальный класс программных средств – ***средства контроля ошибок*** (bug tracking systems).

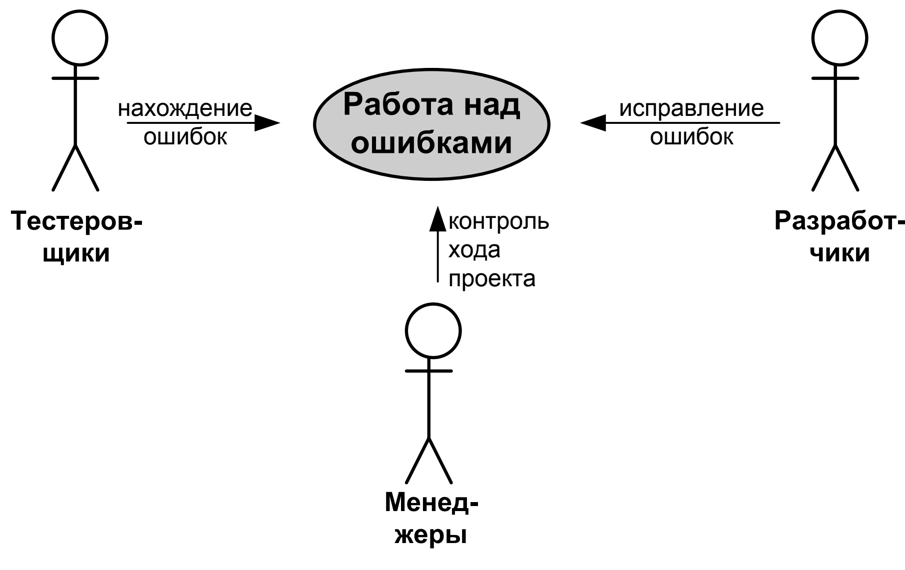

Рис. 6.2.

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

- ответственного за ее проверку – тестеровщика, 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, о чем будет подробно рассказано при обсуждении VSTS.

Литература

1. Кулямин В.В. Технология программирования. Компонентный подход. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. 463 с.
2. I.Sommerville Software Engineering. 6th edition. Addison-Wesley, 2001. 693 p. / Русский перевод: И.Соммервилл. Инженерия программного обеспечения. Издательский дом “Вильямс”, 2002. 623 с.

Лекция 8. Диаграммные техники в работе со знаниями

Метод случаи использования

Описание примера. В качестве примера рассмотрим «Телефонную службу приема заявок». Заказчиком данной системы является компания, владеющая сетью продуктовых магазинов. Эта компания, кроме обычной розничной торговли и оптовых поставок продуктов отдельным столовым и ресторанам, хочет предоставлять еще и сервис по обслуживанию клиентов по телефонным заявкам. Клиент регистрируется в компании, а потом по телефону, в удобное для себя время, делает заказ товаров, которые к нему привозят домой, и он расплачивается. Для этого компания хочет организовать у себя локальный телефонный центр, состоящий из офисной многоканальной АТС, штата операторов и соответствующего программного обеспечения. При этом в компании уже есть информационная система по обработке заявок от постоянных мелкооптовых клиентов, и заказываемая система должна быть с ней проинтегрирована.

Работа с требованиями. Случаи или варианты использования (use cases) были предложены в конце 90-х годов Айвером Якобсоном, одним из главных авторов языка UML, как диаграммный подход для извлечения и первичной формализации требований к системам. Выше уже говорилось о сложности по формированию единой и связной картины требований к ПО. Необходимо извлечь требования из всех возможных источников, формализовать в некотором виде и обсудить. Этот процесс – извлечение, формализация, обсуждение – итеративен, то есть все делается не за один присест. Более того, сам способ формализации должен быть удобен для обсуждения, и в первую очередь, с потенциальными пользователями системы, которые могут быть совершенно не IT не компетентны. Их комментарии, одобрения и несогласия часто являются основой итеративного извлечения требований к системе. Кроме того, этот способ работы с информацией должен вести к созданию моделей, удобных в дальнейшей реализации системы. Другими словами, ясно формулировать исходные задачи для разработки. То есть способ формализации должен быть прост, понятен и обладать достаточной строгостью. Этим требованиям удовлетворяют диаграммы случаев использования, являющиеся на сегодняшний день составной частью стандарта UML.

Пример диаграммы случаев использования представлен на рис. 7.1.

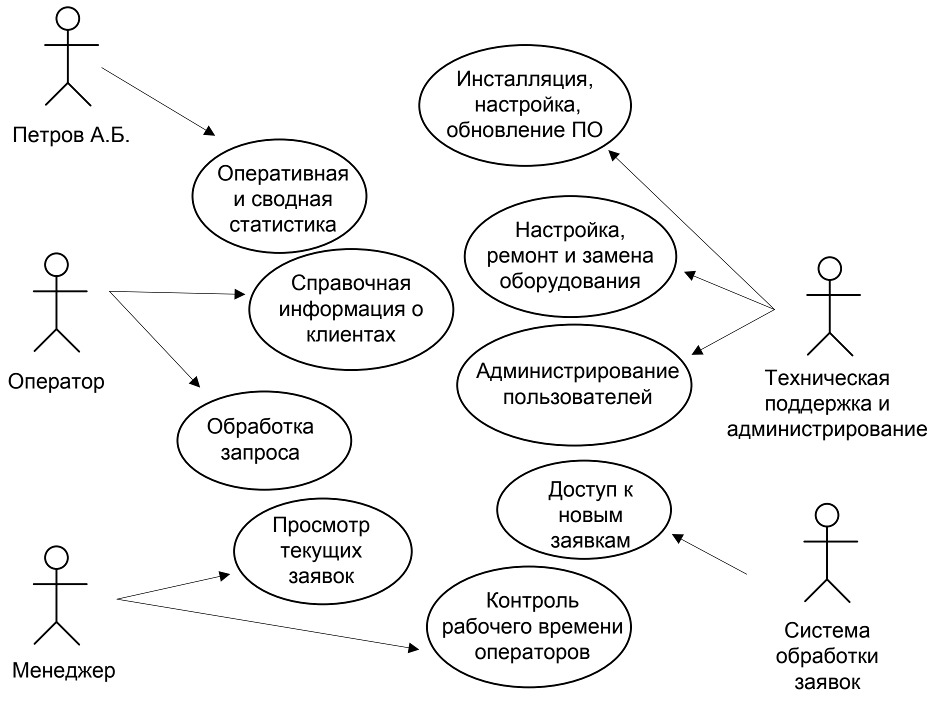

Рис. 7.1 Пример диаграммы случаев использования

Итак, все начинается с точной идентификации пользователей будущей системы. Это – основа хороших требований и хорошей системы, ведь основная задача системы – удовлетворять потребности будущих пользователей. Для этого нужно их знать в лицо….. В нашем случае пользователями системы являются оператор, менеджер и представители технической поддержки и администрирования. Система должна также поддерживать внешний интерфейс с системой обработки заявок. Это — четвертый пользователь. Еще одним пользователем системы является Петров А.Б. — директор департамента сбыта товаров, который хочет периодически отслеживать деятельность телефонной службы приема заявок. Для него создано специальное пользовательское место с экранными формами статистики.

Различные пользователи ПО, изображаемые на диаграммах случаев использования, называются ***актерами*** (actors). Актеры могут обозначать:

- типовых пользователей («Менеджер», «Оператор», «Техническая поддержка») — работников компании, сгруппированных по исполняемым обязанностям;
- другие системы, взаимодействующие с данной («Система обработки заявок»);
- выделенного пользователя («Петров А.Б.»).

Отметим, что выделенный пользователь существенно отличается от типового пользователя. Он, как правило, Важная Персона, и согласование функциональности для него согласуется лично с ним. Часто он влияет на оплату проекта, от его мнения о системе, во многом, зависит ее успешная сдача. Такие персоны, ради успеха проекта, нужно уметь идентифицировать и в рамках всей системы создавать некоторую функциональность специально для них и очень при этом стараться!

После идентификации пользователей происходит определение случаев использования ими системы. Прежде всего, определяется та функциональность системы, которая непосредственно помогают пользователям выполнять их работу, не связанную непосредственно с эксплуатацией системы. В нашем случае, для оператора важным плюсом от использования системы оказывается возможность получать быстрый доступ к справочной информации о клиентах, а также оперативно обрабатывать поступившие по телефону запросы на покупки (список товаров, цены, оформление заказа и пр.). Для менеджера важным является возможность оперативного просмотра текущих заявок (выполненных, в работе, отложенных, за определенный период времени и пр.), а также учет контроль рабочего времени операторов – кто и сколько времени потратил на разного вида работы (телефонные разговоры с клиентами, оформление заявки после окончания разговора и т.д.). При этом важно отметить, что функция учета рабочего времени может потребовать определенных действий со стороны операторов – например, нажимать соответствующую клавишу, уходя на обед или на перекур. Однако мы не обозначили соответствующую связь с этим случаем использования со стороны оператора, поскольку эта функциональность не помогает ему в непосредственной работе, а помогает его начальнику. Кроме того, мы не включили в случаи использования ряд сервисов, связанных с эксплуатацией системы, например, функцию логина в систему. Наличие четкой точки зрения при составлении диаграмм – залог их полезности.

Итак, ***случай использования*** (use case) — это независимая часть функциональности системы, обладающая результирующей ценностью для ее пользователей.

«*Независимость*» означает, что если случай использования всегда исполняется вместе с некоторым другим, то, по всей видимости, один из них нужно включить в другой (какой именно в какой, как назвать получившийся в итоге случай использования — зависит от обстоятельств).

«Результирующая ценность» случая использования для актера системы подразумевает, что он, данный случай использования, должен приносить актеру некоторый законченный и ценный с точки зрения его бизнеса результат. Будучи реализован системой, этот случай использования действительно делает бизнес актера эффективнее, производительнее. Тем самым разработка системы фокусируется на бизнес-целях, а незначительные случаи использования игнорируются, что важно для компактности модели. Ведь строится не абстрактная модель функций системы, а набор самых важных (для заказчика и пользователей) сервисов, чтобы каждый из них правильно понять и не один не упустить. И в дальнейшем контроль разработки системы будет осуществляться именно в терминах этого самого важного — того, что нужно заказчику и пользователям.

Случаи использования, соответствующие актерам «Техническая поддержка и администрирование» и «Служба обработки заявок» несколько не вписываются в представленное выше определение. Прежде всего, сами эти актеры не являются пользователями ПО, участвующими в основном бизнес-процессе обработки телефонных заявок. «Техническая поддержка и администрирование» занята поддержкой ПО и оборудования системы обслуживания телефонных заявок, а также ее администрированием (добавлением новых пользователей, назначением им соответствующих прав и пр.). «Служба обработки заявок» является уже существующей в компании информационной системой, имеющей базу данных и ряд сервисов по обработке заявок. Идентификация этих актеров и соответствующих им случаев использования важна с очки зрения определения требований к системе. Для представителей службы технической поддержки необходим специальный удобный интерфейс с набором соответствующих функций. А все поступившие по телефону заявки должны попасть в единую базу данных заявок и пройти единый цикл обработки. Упущение этих факторов может привести к серьезным недочетам и проблемам. Кроме того, они ни откуда не следуют напрямую и поэтому нуждаются в

особых начальных вершинах в дереве требований – то есть мы решили, что целесообразно поместить их на главную диаграмму случаев использования.

Отметим еще одну интересную деталь. Клиент магазина не является пользователем данного ПО. Он оказывается бизнес-пользователем всей системы в целом (включая соответствующий бизнес-процесс и оборудование). На рис. 7.2 представлена бизнес-диаграмма случаев использования.

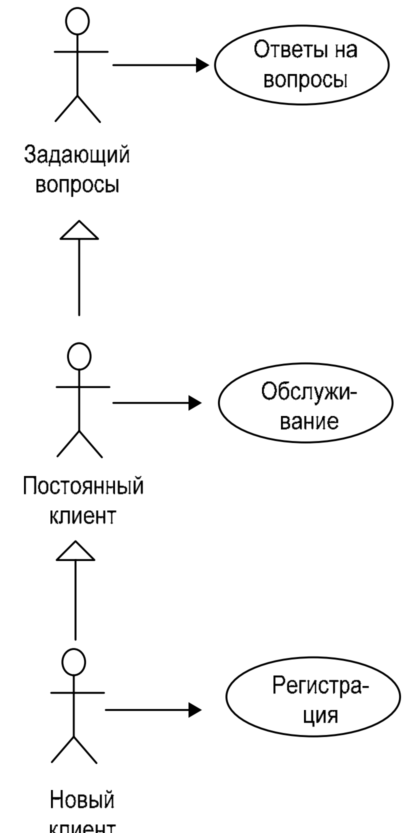

Рис. 7.2 Пример диаграммы бизнес-случаев использования

Ее можно рисовать отдельно (классики на этом настаивают), но можно пририсовывать клиента и на общую диаграмму, связав стрелкой и оператором. Часто бывает, что востребована не очень концептуальная, но компактная запись.

Каждый случай использования сопровождается небольшим текстовым описанием, а в дальнейшем может содержать целые главы в техническом задании. Диаграммы случаев использования могут служить структурой технического задания или его отдельных частей.

Другие версии. На практике диаграммы случаев использования создаются не только таким способом, как указано выше. Многие практики предпочитают строить очень детальные модели, прорисовывая на них все небольшие случаи использования, а также многочисленные связи между ними (использование, расширение и т.д.). Кто-то решительно протестует против включения в актеры системы, взаимодействующие с данной. Другие считают неприемлемым включать совмещать обычные и диаграммы и бизнес-диаграммы случаев использования и так далее. Какую именно вы изберете стратегию в конкретном случае, какую точку зрения поставите во главу угла – вам решать самим. Рецепта здесь нет.

Важно лишь отметить, что хорошо определенная точка зрения нужна. Она позволяет четко сфокусироваться, решать определенные, хорошо осознаваемые задачи. А

также такую точку зрения можно кое-где осознанно, в угоду практической полезности, нарушать.

Случаи использования в управлении разработкой. Итак, выше мы показали, как диаграммы случаев использования могут быть полезны при выявлении ыи первичной формализации требований. Но они могут оказаться полезными и после того, как это процесс завершен. Результирующие диаграммы случаев использования можно применять при управлении разработкой. Менеджер проекта может отслеживать прогресс проекта по тому, сколько реализовано функциональности, необходимой пользователю. Разработчики могут иметь диаграммы случаев использования где-то перед глазами, чтобы не забывать об основной цели разработки. Эти же диаграммы могут использоваться в рабочих встречах по проекту.

Казалось бы, что может быть проще — реализовать набор функций, необходимых пользователю. Однако на деле программный проект может незаметно потерять эту цель. Вместо этого можно, например, очень долго заниматься разработкой сложной и многофункциональной архитектуры, после реализации которой разработчики обещают, что все пользовательские функции получатся почти сразу же и очень легко. Однако, как правило, оказывается, что это «сразу же» было сильным преувеличением и проект весьма выбивается из расписания, а многие заказанные пользователем функции в этом окружении сделать тяжело или невозможно. Бывает, что чрезмерная ориентация на «внутреннее совершенство» ПО оканчивается для проекта либо крупными неприятностями, либо полным крахом. Однако бывают и другие случаи, когда только такая ориентация впоследствии и спасает проект. Последнее случается, когда система долго развивается и сопровождается, или когда требования к ней внезапно и сильно меняются, или когда на ее основе делаются другие системы. Необходим баланс между внутренним совершенством программного обеспечения и функциональностью, нужной для заказчика и доставленной ему в срок. Разработка ПО в терминах случаев использования — хороший способ контролировать, что процесс создания системы двигается в нужном направлении.

Итеративный цикл автор/рецензент

Опишем одну интересную и крайне полезную технику использования визуального моделирования при выявлении знаний о какой-либо предметной области через общение с экспертами (специалистами в этой предметной области). Эта техника называется ***цикл автор/рецензент*** (Reader/Author Cycle review process) и может применяться, например, при работе с диаграммами случаев использования. при работе как с UML, так и с любым другим языком визуального моделирования. Эта техника была определена в рамках методологии SADT.

Активный сотрудник — *автор* визуальных моделей (author), — изучает не вполне знакомую ему область знаний. При этом автору постоянно нужна обратная связь с экспертами в этой предметной области, чтобы он осознавал, насколько правильно он понял и адекватно формализовал тот или иной аспект изучаемых знаний.

В качестве такой области знаний может выступать предметная область, для которой создается информационная система. При разработке информационной системы ее авторы должны хорошо разобраться в данной предметной области. Если будущие пользователи или заказчик системы не имели возможности подробно ознакомиться с тем, как разработчики поняли и интерпретировали их предметную область, то это непременно приведет к созданию невостребованной системы: данные будут неверны или их не будет хватать, форматы отчетов окажутся неудобны и т. д.

Итак, для того, чтобы создать адекватное описание системы, необходимо своевременно получать оценку создаваемых моделей от тех людей, которые в конце концов будут ею пользоваться. Для этого вводятся следующие роли:

- *автор* (author) модели — тот, кто ее создает;
- *эксперт* (commenter) — это специалист в той предметной области, для которой строится данная модель; автор интервьюирует эксперта, получая необходимую для моделирования информацию; эксперт просматривает и комментирует созданные автором диаграммы; важно, что эксперт выражает свои комментарии в письменном виде и разделяет с автором ответственность за качество создаваемых моделей; эксперт может быть также архитектором системы, который активно участвует в процессе разработки модели анализа — но не как автор моделей (у него хватает других забот), а как активный критик (при разработке архитектуры системы он будет активно использовать эту модель);
- *читатель* (reader) — во всем похож на эксперта, но не обязан давать письменные комментарии к моделям и не несет ответственности за качество моделирования.

Получив диаграммы автора, эксперт их тщательно просматривает и пишет свои комментарии (прямо на диаграмме, в виде примечаний, красной ручкой). Автор, получив назад свои диаграммы с комментариями, обязан отреагировать на каждое замечание — пометить синей ручкой на той же копии, принимает ли он замечание или нет. Принятые замечания он учитывает в следующей версии диаграмм, неприятные отсылает обратно эксперту с мотивировкой. В случае возникновения непонимания организуется встреча автора и эксперта, на которой они улаживают все рассогласования.

Кроме автора, эксперта и читателя в цикле «читатель/автор» имеются также следующие роли:

- *библиотекарь* (librarian) — это главный координатор процесса моделирования; он следит за тем, чтобы все участники процесса вовремя получали свежие копии моделей, чтобы эти копии не терялись и вовремя попадали в архив, а последний был бы доступен; в его компетенцию входит также отслеживать, что все замечания экспертов и читателей обработаны автором, не оставлены без внимания; раньше, когда метод SADT только появился, роль библиотекаря была велика — модели строились на бумаге; теперь же для этого используют разные графические пакеты, а для хранения разных версий модели — программные средства управления версиями;
- *комитет технического контроля* (technical review committee) — это группа людей, которая следит за тем, насколько процесс моделирования отвечает целям проекта, будет ли возможность использовать в дальнейшей работе создаваемые диаграммы; этот комитет следит также за тем, когда моделирование нужно завершить; ведь время людей может стоить существенных денег, у проекта есть сроки, а процесс моделирования может продолжаться очень долго — например, автор может увлечься, изучая новую предметную область.

Следует заметить, что цикл «читатель/автор» может использоваться в различных ситуациях, когда необходимо эффективно извлекать информацию из экспертов некоторой предметной области. Например, такая ситуация может сложиться, когда технический писатель создает документацию о программном обеспечении, или тестеровщик изучает систему для того, чтобы эффективно ее тестировать, или новый менеджер проекта изучает систему, которая уже давно разрабатывается и созданием которой ему нужно будет руководить, и т. д.

Кроме того, цикл «автор/рецензент» может быть использован и вне контекста извлечения знаний, когда мы, зачем-либо создавая визуальные модели, хотим получать регулярную и упорядоченную обратную связь.

Разнообразие производственных контекстов, где может применяться данная техника, а также особенности человеческих и организационных отношений, приводят к тому, что цикл «автор/рецензент» на практике требует адаптации. Для его эффективного использования необходима «тонкая подстройка» под особенности конкретной ситуации.

В частности, могут варьироваться ответственности разных ролей. Например, эксперт может отвечать за процесс моделирования или совсем не отвечать (вся ответственность лежит на авторе). Само общение автора и эксперта также может быть организовано по-разному. Например, в отличие от приведенных выше рекомендаций, эксперт может высказываться только устно, при личных встречах с автором. На одной встрече эксперт выдает информацию, на другой проверяет то, как получилось у автора ее формализовать. Этот вариант предсьавлен в примерах ниже.

На рис. 7.3 показана начальная диаграмма, которую нарисовал автор для первой встречи с экспертом. На ней присутствуют только интерфейсы с диаграмм компонент UML и комментарии к ним. На рис. 7.4 представлена заключительная диаграмма, получившаяся в итоге многочисленных итераций. На ней присутвуют мнгочисленные компоненты системы, сгруппированные по уровням.

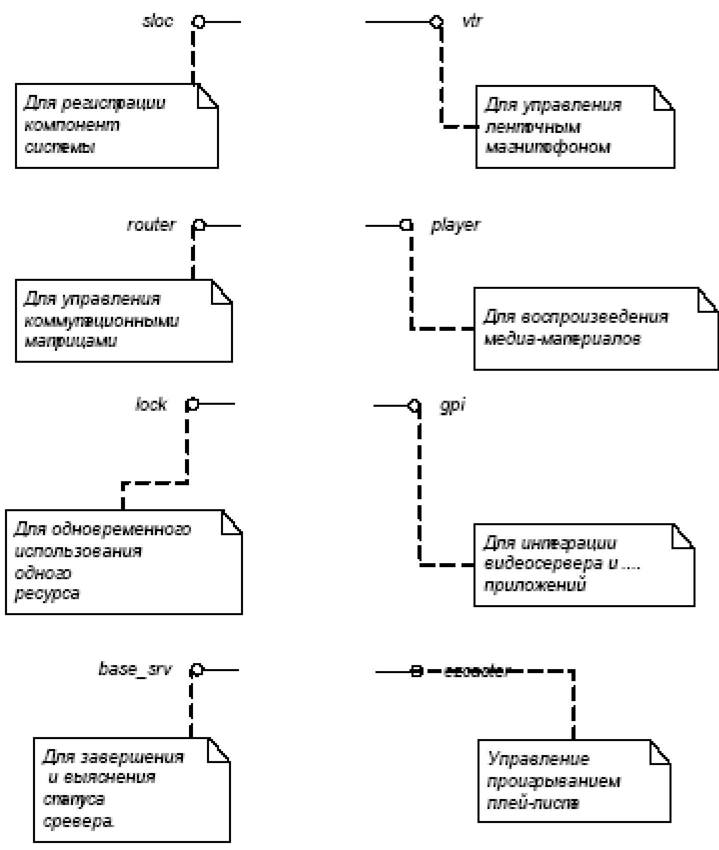

Рис. 7.3 Пример исходной, первой диаграммы.

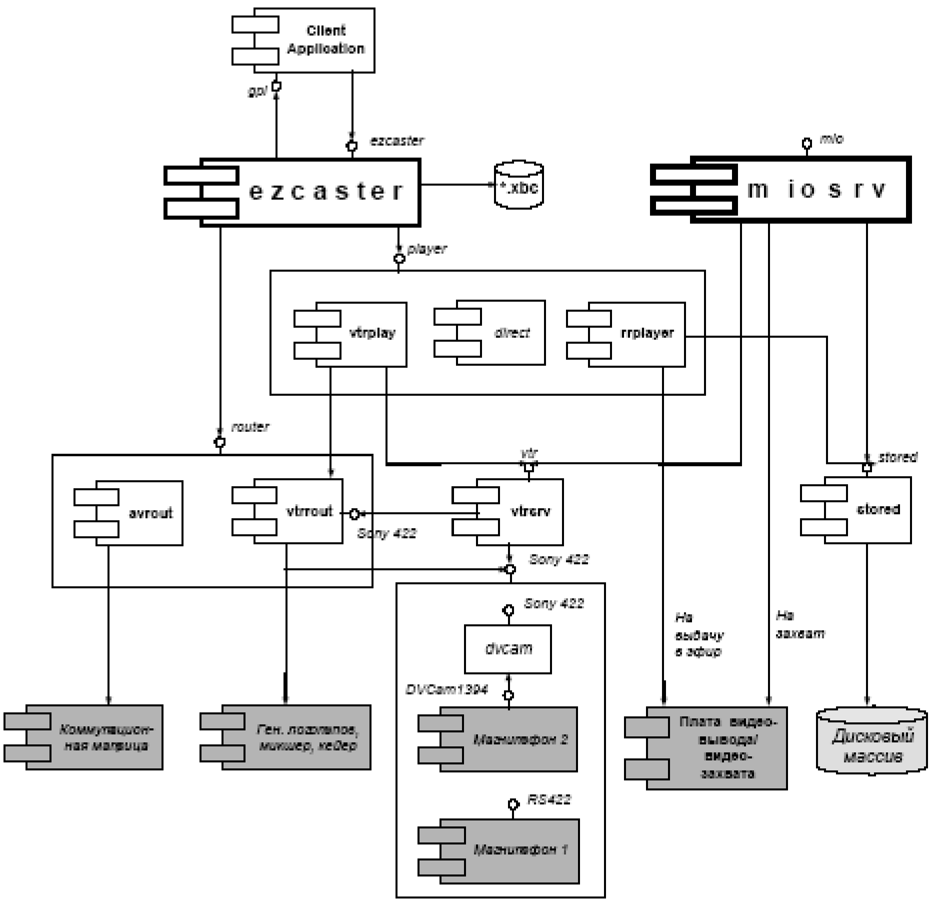

Рис. 7.4 Пример итоговой диаграммы.

Карты памяти

Карты памяти (Mind Maps) – техника работы с различными знаниями, предложенная и развитая английским психологом Тони Бьюзеном в конце 70-х годов прошлого века. Она очень простая и используется при работе с информацией любого вида, для ее структурирования, осмысления, лучшего усвоения и запоминания. На листе бумаги, в центре, рисуется объект, обозначающий ту тему или предмет, который мы рассматриваем. Далее рисуются вторичные объекты, которые поясняют и уточняют данный и соединяются с ним дугами. И так далее. Пример представлен на рис. 7.5.

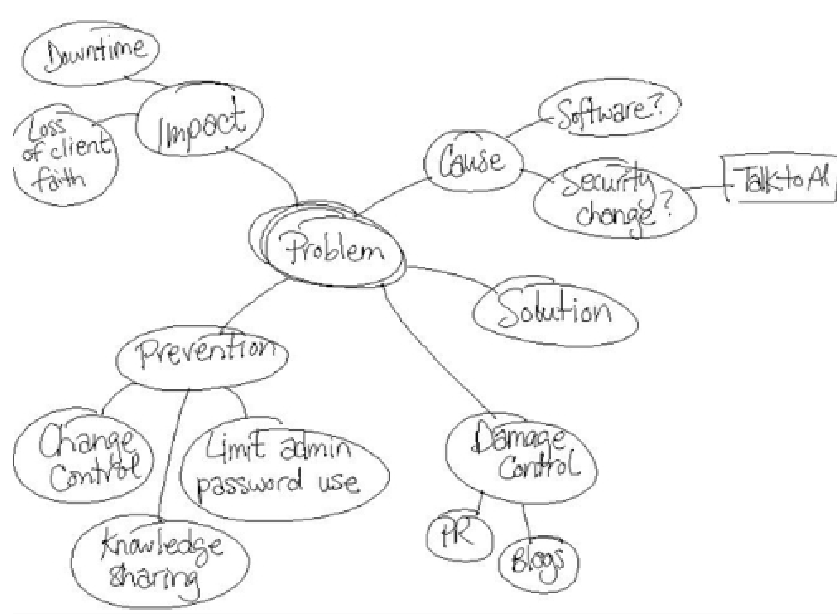

Рис. 7.5

Дизайн идей. Карты памяти позволяют выполнить «дизайн идей». Очень часто мы, как следует не подумав, начинаем что-то делать – писать большой текст, с кем-то встречаться, кем-то руководить и пр. И оказывается, что понимание по ходу дела возникает трудно и мучительно. Более того, мы вынуждены переделывать то, что уже сделали без этого понимания. Хорошая иллюстрация – работа над тестом (диплома, курсовой работы, статьи, книги и пр.). Кардинально переделывать текст очень тяжело. А если при соавтором несколько? Карты памяти здесь очень хорошо работают, так как позволяют в компактном виде делать пробы и ошибки, видя всю картину перед собой. Ее легко также обсуждать в таком виде с другими людьми. Но здесь не нужно фанатизма. Можно и написать текст, если он легко «выливается» из вас. И снова вернуться к схеме – многое на ней может проясниться.

Планирование детальной информации. Метод позволяет также выполнить детальное планирование большого объема информации, имеющей огромное количество важных деталей. Например, мы использовали карты памяти при проектировании анкеты - она содержала достаточное количество ветвлений, списки вопросов и пр. Все это в общем виде, сокращенно, было не представить, карты памяти, поддержанные программным инструментом Comapping[6] нам здесь очень помогли. Пример представлен на рис. 7.6.

---

[6] www.comapping.com

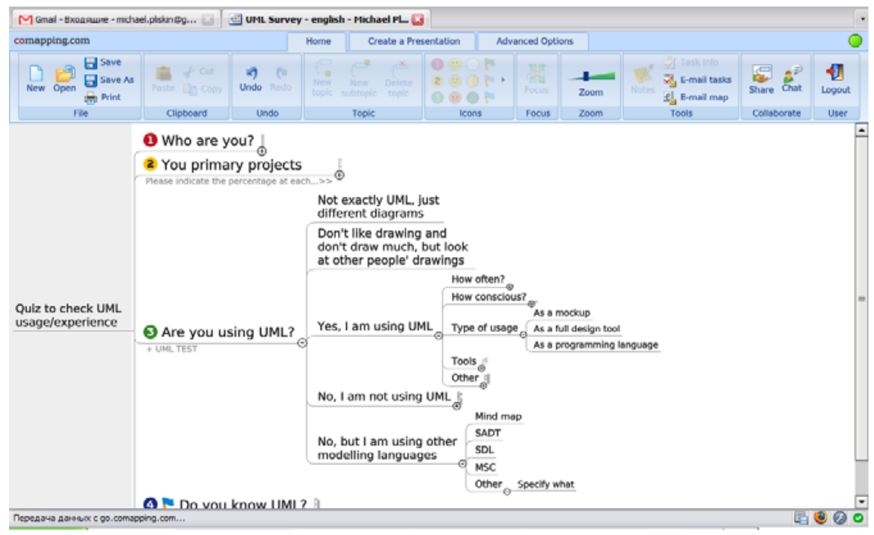

Рис. 7.6.

Реструктуризация. Карты памяти полезны при реструктуризации знаний. Например, при реструктуризации статьи. У нас был случай, когда результаты были получены, материал собран и изложен, но достаточно хаотично. Мы выполнили реструктуризацию статьи с помощью карт памяти (модель представлена на рис. 7.7) и по этой модели быстро переписали текст. Исправления прямо по тексту затянули бы весь процесс. Кроме того, карты памяти позволили разделить работу между соавторами – одни создал новый план, а второй его реализовал в новой версии текста.

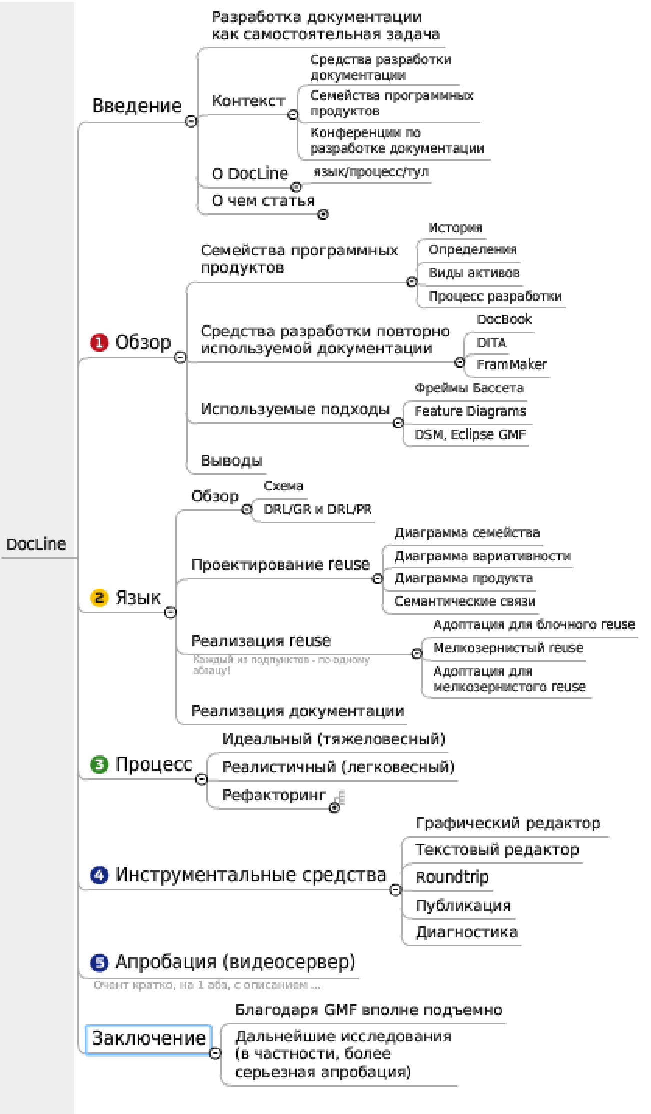

Рис. 7.7

Метод реструктуризации широко используется при работе со знаниями, например, при выявлении и анализе требований. Построить представление той же информации, но с другой точки зрения – верный способ найти незамеченные ранее противоречия, «темные углы», углубить свое понимание.

Работа с краткосрочной памятью. Часто бывает, что прослушав лекцию, мы каоке-то время помним ее содержание (обычно несколько дней), но по прошествии нескольких месяцев ее содержание начисто улетучивается из мозгов. Так вот, можно сразу, пока железо еще горячо, сделать себе набросок основных аспектов, и использованием карт памяти. Студенты говорят, что найдя потом такие конспекты-напоминалки, они быстро восстанавливают учебный материал в памяти. Это целесообразно делать после лекции, когда целостное впечатление от информации еще свежо.

Коллективная работа и продукт Comapping. Одно из главных достоинств диаграмм заключается в том, что их можно обсуждать с широким кругом людей. Тест, например, обсуждать труднее – его нужно сначала просчитать. А диаграмму можно тут же смотреть и обсуждать. И исправлять. Более того, с помощью диаграмм можно организовывать эффективные групповые территориально распределенные процессы

работы с информацией: планирование, создание текстов, обмен результатами бесед, общение преподавателя и студента и пр.

Расскажем про один программный продукт, реализующий карты памяти и поддерживающий широкие возможности групповой работы….

Литература

1. Marca D.A., McGowan C.L. SADT Structured Analysis and Design Technique. McGraw-Hill, 1988.
2. Integration Definition For Function Modeling (IDEF0). Draft Federal Information Processing Standards Publication 183, 1993, 79 p., http://www.idef.com/IDEF0.html.
3. Д.В. Кознов. Основы визуального моделирования / Д. В. Кознов. – М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. – 248 с.: ил. – (Серия «Основы информационных технологий»).
4. Д.В.Кознов. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Изд. СПбГУ, 2004. 170 с.
5. D.V.Koznov, M.Pliskin. Computer-Supported Collaborative Learning with Mind-Maps. T. Margaria and B. Steffen (Eds.): ISoLA 2008, CCIS 17, pp. 478–489, 2008. Springer-Verlag, Berlin Heidelberg, 2008.
6. Кознов Д.В. Методика обучения программной инженерии на основе карт памяти. С. 121-140.
7. Кознов Д.В.Кириленко Я.А. Опыт сочетания теории и практики в обучении программной инженерии. Труды III всероссийской конференции «Современные информационные технологии и ИТ-образование», М. 2008.

Лекция 9. MSF

История и текущий статус

В 90-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 году вышла версия MSF 3.1, состоявшая из пяти документов-руководств:

- модель процессов (process model),
- модель команды (team model),
- модель управления проектами (project management),
- дисциплина управления рисками (risk management),
- управление подготовкой (readiness management).

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 понималась создание ПО, обучение пользователей и полная передача продукта команде сопровождения.* Задача наладки полноценного сопровождения IT-решения - важная составляющая успешности проекта.

Основными новшествами MSF является следующее.

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 году MSF претерпело значительные изменения. Версия MSF 4.0. стала составной часть продукта Visual Studio Team System (VSTS) и разделилась на две ветки – MSF for Agile и MSF for CMMI. 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 ответственности, действия в рамках этих ответственностей, 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 предоставляет 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 не сильно 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. В этой лекции мы рассмотрим основные принципы MSF, то есть, фактически, MSF 3.1, а в лекциях, посвященных VSTS будут рассмотрены MSF for Agile и MSF for CMMI.

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 команды

Основные принципы. Главная особенность 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 обуславливают модель проектной группы и образуют ***ролевые кластеры*** (или просто ***роли***) в проекте. В каждом ролевом кластере может присутствовать по одному или несколько 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, team of peers), работающая над проектом, должна иметь четкую форму отчетности перед заинтересованными сторонами (stakeholders) при распределенной ответственности за достижение общего успеха. В MSF следующие ролевые кластеры (часто их называют ролями) – см. рис. 8.1.

- ***Управление продуктом*** (product management). 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 организации-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*** (program management) 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*** (development). Этот ролевой кластер занимается, собственно, программированием ПО.
- ***Тестирование*** (test) – отвечает за тестирование ПО.
- ***Удовлетворение потребителя*** (user experience). Дизайн удобного пользовательского интерфейса и обеспечение удобства эксплуатации ПО (эргономики), обучение пользователей работе с ПО, создание пользовательской документации.
- ***Управление выпуском*** (release management). 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*** (Architecture).[7] Организация и выполнение высокоуровневого проектирования решения, создание функциональной спецификации ПО и управление этой спецификацией в процессе разработки, определение рамок проекта и ключевых компромиссных решений.

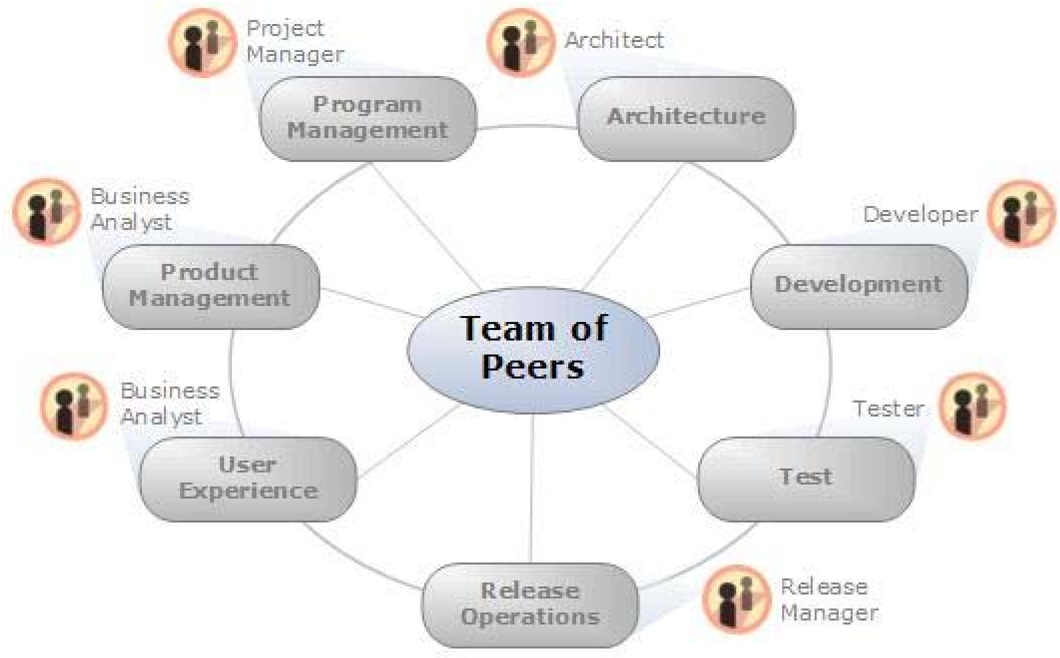

Рис. 8.1

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 ниже представлены рекомендации MSF относительно совмещения ролей в рамках одним членом команды. «+» означает, что совмещение возможно, «+-» - что совмещение возможно, но нежелательно, «-» означает, что совмещение не рекомендуется.

---

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

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

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

Модель проектной группы MSF предлагает разбиение больших команд (более 10 человек) на малые многопрофильные ***группы направлений*** (feature teams). Эти малые коллективы работают параллельно, регулярно синхронизируя свои усилия, каждая из которых устроена на основе модели кластеров. Это компактные мини-команды, образующие матричную организационную структуру. В них входят по одному или несколько 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*** (functional teams), 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 сосредотачивается 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.

Прочие особенности

Модель процесса

Водопадная модель – фазы работ и вехи

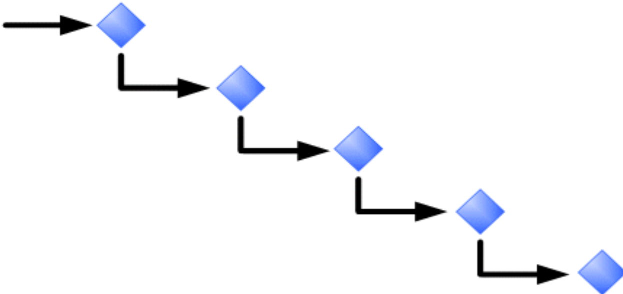

Рис. 8.2

Водопадная модель – постоянное уточнение требований, активное взаимодействие с заказчиком.

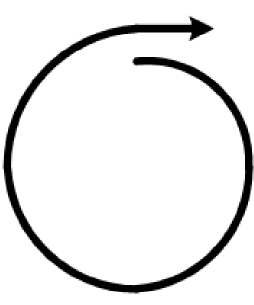

Рис. 8.3

В MSF объединяются водопадная и спиральная модели: сохраняются преимущества упорядоченности водопадной модели, не теряя при этом гибкости и творческой ориентации модели спиральной.

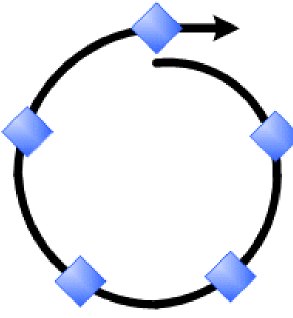

Рис. 8.4

Итак, процесс MSF ориентирован на “вехи” (milestones) – ключевые точки проекта, характеризующие достижение в его рамках какого-либо существенного (промежуточного либо конечного) результата. Этот результат может быть оценен и проанализирован, что подразумевает ответы на вопросы: “Пришла ли проектная группа к однозначному пониманию целей и рамок проекта?”, “В достаточной ли степени готов план действий?”,

“Соответствует ли продукт утвержденной спецификации?”, “Удовлетворяет ли решение нужды заказчика?” и т.д. А между вехами - итерации, итерации, итерации…..

Управление компромиссами. Хорошо известна взаимозависимость между ресурсами проекта (людскими и финансовыми), его календарным графиком (временем) и реализуемыми возможностями (рамками). Эти три переменные образуют треугольник, показанный на рис. 8.5.

Рис. 8.5

После достижения равновесия в этом треугольнике изменение на любой из его сторон для поддержания баланса требует модификаций на другой (двух других) сторонах и/или на изначально измененной стороне.

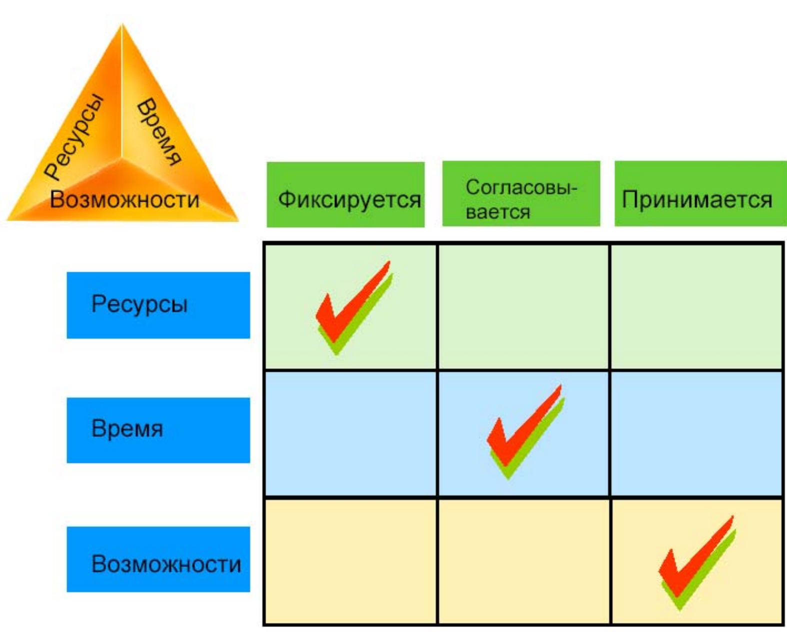

Рис. 8.6

Зафиксировав ___________, мы согласовываем ___________ и принимаем результирующий ___________.

Литература

1. Microsoft Solutions Framework, Process Model, Version 3.1. 2002. http://www.microsoft.com/msf (Русский перевод: Microsoft Solutions Framework. Модель процессов, версия 3.1. 41 с.)
2. Microsoft Solutions Framework, Team Model, Version 3.1. 2002. http://www.microsoft.com/msf (Русский перевод: Microsoft Solutions Framework. Модель команды, версия 3.1.)
3. Microsoft Solutions Framework, Project Management, Version 3.1. 2002. http://www.microsoft.com/msf (Русский перевод: Microsoft Solutions Framework. Упрвление проектами, версия 3.1.).

Лекция 10. CMMI

Что такое CMMI?

CMMI является некоторым описанием идеального процесса разработки ПО, предлагает некоторую модель процесса. То есть в процессе выделяются и тщательно описываются некоторые составные части, ключевые с точки зрения CMMI. Эта точка зрения CMMI – совершенствование процессов разработки. То есть эти значимые части процесса – ***области усовершенствования***. В CMMI различаются следующие группы областей усовершенствования: управление процессами, управление проектами, инженерные области, служебные области. При этом все области задаются в виде требований, определяющих не то, как они реализованы, а интерфейсные требования. Из этого имеется два следствия.

Следствие 1. CMMI допускает различные реализации и не является методологией разработки ПО, подобно MSF, Scrum, RUP и пр. Последние могут использоваться в его реализации. Так, существует, например, специальный шаблон процесса в VSTS для CMMI под названием MSF for CMMI.

Следствие 2. CMMI используется для сертификации компаний на зрелость их процессов. Изначально, в конце 80-х начале 90-х годов, CMM (тогда еще не CMMI) создавался именно как средство сертификации федеральных субподрядчиков. И только позднее, получив широкое распространение в мире, он начал использоваться а после и ориентироваться на совершенствование процессов.

Отметим еще одну важную характеристику CMMI. Он предназначен не только для разработки программных систем. Многие крупные компании выпускают не ПО, а целевые изделия, куда ПО входит как составная часть. Например, авиационная, аэрокосмическая индустрии. То есть разработка ПО происходит вместе с инженерными работами иных видов. И часто бывает, что в одном проекте участвует более двух различных видов инженерии. Задача CMMI – предоставить таким проектам и компаниям единую платформу организации процесса разработки.

Уровни зрелости процессов по CMMI

В отличии от классической модели CMM, которая была жестко иерархической и допускала только последовательное улучшение процессов по уровням, модель CMMI имеет два измерения – последовательное, такое же как и в CMM, и непрерывное, допускающее совершенствование процессов в организации до некоторой степени в произвольном порядке. Здесь мы остановимся на последовательной модели. Она имеет 5 уровней зрелости процессов, как показано на рис. 9.1.

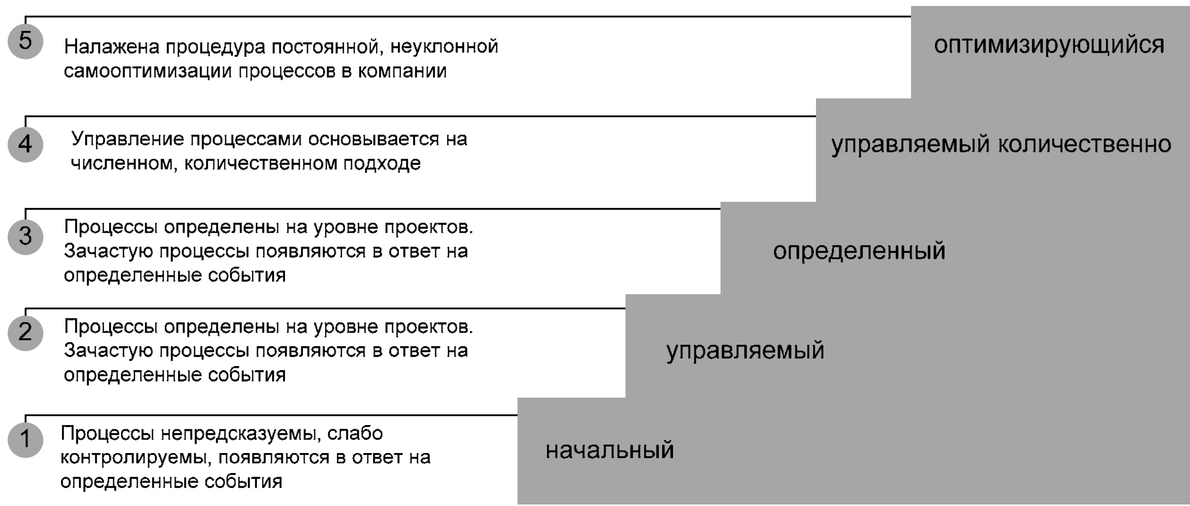

Рис. 9.1

Начальный уровень (уровень зрелости 1) – это уровень, на котором, по определению, находится любая компания. На этом уровне разработка ПО ведется более-менее хаотично.

Управляемый уровень (уровень зрелости 2) – здесь уже появляются политики и процедуры организации процессов, утвержденные на уровне компании. Но в полной мере процессы существуют лишь в рамках отдельных проектов.

Определенный уровень (уровень зрелости 3) – здесь появляется стандартный процесс на уровне всей компании в целом. Это большой и постоянно пополняющийся набор активов процесса – шаблонов документов, моделей жизненного цикла, программных средств, практик и пр. Любой конкретный процесс получается вырезкой, из этого стандартного.

Управляемый количественно уровень (уровень зрелости 4) подразумевает появление системы измерений в компании, которые происходят на базе стандартного процесса и позволяют количественно управлять разработкой.

Оптимизирующийся уровень (уровень зрелости 5) подразумевает постоянное улучшение процессов разработки, как постепенных, пошаговых, так и революционных. При этом данные изменения оказываются не вынужденными, а упреждающими проблемы и трудности. Процесс совершенствуется сам и постоянно – есть, реализованы соответствующие механизмы.

Области усовершенствования

Уровень зрелости	Области усовершенствования	
Уровень зрелости 2	Управление требованиями	
	Планирование проекта	
	Наблюдение за проектом и контроль	

	Управление договоренностями с поставщиком	
	Измерения и анализ	
	Проверка процессов и продуктов на соответствие стандартам	
	Конфигурационное управление	
Уровень зрелости 3	Разработка требований	
	Техническое решение	
	Сборка и поставка продукта	
	Проверка продукта на соответствие требованиям (верификация)	
	Проверка продукта на соответствие предназначению (валидация)	
	Фокусирование на процессах организации	
	Определение процессов организации	
	Организация обучения	
	Комплексное управление проектом	
	Управление рисками	
	Управление объединенной командой	
	Комплексное управление работой с поставщиком	
	Принятие решений: оценка альтернатив	
	Создание в организации условий для совместной работы	
Уровень зрелости 4	Установление показателей выполнения процессов организации	
	Управление проектами на основе количественных показателей	
Уровень зрелости 5	Отбор и внедрение улучшений в организацию	
	Анализ причин возникновения проблем и предотвращение их появления в будущем	

Литература

1. Д.Ахен, А.Клауз, Р.Тернер. CMMI: комплексный подход к совершенствованию процессов. Пер с англ. М.:МФК. 2005. 330 с.
2. W.Humphrey. Managing the Software Process. Addison-Wesley, 1989. 494 p.
3. CMMI for Development, Version 1.2. *CMMI-DEV* (Version 1.2, August 2006). Carnegie Mellon University Software Engineering Institute (2006). Retrieved on 22 August 2007. http://www.sei.cmu.edu/publications/documents/06.reports.

Лекция 11. «Гибкие» (agile) методы разработки

Общее

«Гибкие» (agile) методы разработки ПО появились как альтернатива формальным и «тяжеловесным» методологиям, наподобие CMM и RUP. Талантливые программисты не желают превращения разработки ПО в рутину, хотят иметь максимум свобод и обещают взамен высокую эффективность. С другой стороны, практика показывает, что «тяжеловесные» методологии в значительном количестве случаев неэффективны. Основными положениями гибких методов, закрепленных в Agile Manifesto в 2007 году являются следующее[8]:

- индивидуалы и взаимодействие вместо процессов и программных средств;
- работающее ПО вместо сложной документации;
- взаимодействие е с заказчиком вместо жестких контрактов;
- реакция на изменения вместо следования плану.

Фактически, гибкие методологии говорят о небольших, самоорганизующихся командах, состоящих из высококвалифицированных и энергичных людей, ориентированных на бизнес, то есть, например, разрабатывающих свой собственный продукт для выпуска его на рынок. У этого подхода есть, очевидно, свои плюсы и свои минусы.

Extreme Programming

Самым известным гибким методом является Extreme Programming (известное сокращенное название – XP). Он был создан талантливым специалистом в области программной инженерии Кентом Беком в результате его работы в 1996-1999 годах над системой контроля платежей компании "Крайслер".

Модель процесса по XP выглядит как частая последовательность выпусков (releases) продукта, столь частых, сколь это возможно. Но при этом обязательно, чтобы в выпуск входила новая целиковая функциональность. Ниже перечислены основные принципы организации процесса по XP.

1. Планирование (Planning Game), основанное на принципе, что разработка ПО является диалогом между возможностями и желаниями, при этом изменятся и то и другое.
2. Простой дизайн (Simple Design) – против избыточного проектирования.
3. Метафора (Metaphor) – суть проекта должна умещаться в 1-2 емких фразах или в некотором образе.
4. Рефакторинг (Refactoring) – процесс постоянного улучшения (упрощения) структуры ПО, необходимый в связи с добавлением новой функциональности.
5. Парное программирование (Pair Programming) – один программирует, другой думает над подходом в целом, о новых тестах, об упрощении структуры программы и т.д.
6. Коллективное владение кодом (Collective Ownership).

---

[8] Не надо понимать эти положения Agile Manifesto буквально так, что любая гибкая методология им в точности удовлетворяет. Agile Manifesto лишь оконтуривает некоторое пространство, обозначает определенное явление. Отдельные части этого явления – конкретные гибкие методологии могут иметь разную специфику, могут также являться пограничными объектами.

7. Участие заказчика в разработке (On-site Customer) – представитель заказчика входит в команду разработчика.
8. Создание и использование стандартов кодирования (Coding Standards) в проекте – при написании кода (создаются и) используются стандарты на имена идентификаторов, составление комментариев и т.д.
9. Тестирование – разработчики сами тестируют свое ПО, перемежая этот процесс с разработкой. При этом рекомендуется создавать тесты до того, как будет реализована соответствующая функциональность. Заказчик создает функциональные тесты.
10. Непрерывная интеграция. Сама разработка представляется как последовательность выпусков.
11. 40-часовая рабочая неделя.

Однако в полном объеме XP не была использована даже ее авторами и является, скорее, философией. Кроме того, известны и внедряются отдельные практики XP, как, например, парное программирование, коллективное владение кодом, и, конечно же, рефакторинг кода. Идея простого, неизбыточного дизайна проекта также оказала значительное влияние на мир разработчиков ПО.

Более практичным «гибким» методом разработки является Scum.

Scrum

История. В 1986 японские специалисты Hirotaka Takeuchi и Ikujiro Nonaka опубликовали сообщение о новом подходе к разработке новых сервисов и продуктов (не обязательно программных). Основу подхода составляла сплоченная работа небольшой универсальной команды, которая разрабатывает проект на всех фазах. Приводилась аналогия из регби, где вся команда двигается к воротам противника как единое целое, передавая (пасуя) мяч своим игрокам как вперед, так и назад. В начале 90-х годов данный подход стал применяться в программной индустрии и обрел название Scum (термин из регби, означающий - схватка), в 1995 году Jeff Sutherland и Ken Schwaber представили описание этого подхода на OOPSLA '95 – одной из самых авторитетных конференций в области программирования. С тех пор метод активно используется в индустрии и многократно описан в литературе. Scum активно используется также в России.

Общее описание. Метод Scrum позволяет гибко разрабатывать проекты небольшими командами (7 человек плюс/минус 2) в ситуации изменяющихся требований. При этом процесс разработки итеративен и предоставляет большую свободу команде. Кроме того, метод очень прост – легко изучается и применяется на практике. Его схема изображена на рис. 10.1.

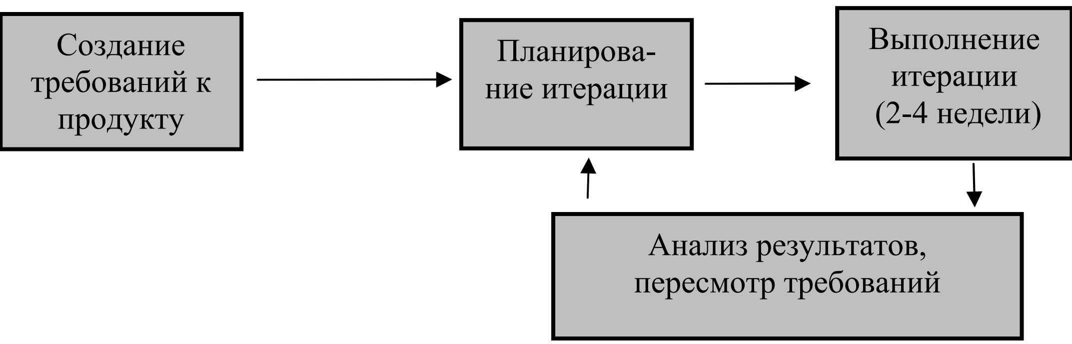

Рис. 10.1

Вначале создаются требования ко всему продукту. Потом из них выбираются самые актуальные и создается план на следующую итерацию. В течение итерации планы не меняются (этим достигается относительная стабильность разработки), а сама итерация длится 2-4 недели. Она заканчивается созданием работоспособной версии продукта (рабочий продукт), которую можно предъявить заказчику, запустить и подемонстрировать, пусть и с минимальными функциональными возможностями. После этого результаты обсуждаются и требования к продукту корректируются. Это удобно делать, имея после каждой итерации продукт, который уже можно как-то использовать, показывать и обсуждать. Далее происходит планирование новой итерации и все повторяется.

Внутри итерации проектом полностью занимается команда. Она является плоской, никаких ролей Scrum не определяет. Синхронизация с менеджментом и заказчиком происходит после окончания итерации. Итерация может быть прервана лишь в особых случаях.

Роли. В Scrum есть всего три вида ролей.

Владелец продукта (Product Owner) – это менеджер проекта, который представляет в проекте интересы заказчика. В его обязанности входит разработка начальных требований к продукту (Product Backlog), своевременное их изменение требований, назначение приоритетов, дат поставки и пр. Важно, что он совершенно не участвует в выполнении самой итерации.

Scrum-мастера (Scrum Master) обеспечивает максимальную работоспособность и продуктивную работу команды – как выполнение Scrum-процесса, так и решение хозяйственных и административных задач. В частности, его задачей является ограждение команды от всех воздействий извне во время итерации.

Scrum-команда (Scrum Team) – группа, состоящая из пяти–девяти самостоятельных, инициативных программистов. Первой задачей команды является постановка для итерации реально достижимых и приоритетных для проекта в целом задач (на основе Project Backlog и при активном участии владельца продукта и Scum-мастера). Второй задачей является выполнение этой задачи во что бы то ни стало, в отведенные сроки и с заявленным качеством. Важно, что команда сама участвует в постановке задачи и сама же ее выполняет. Здесь сочетается свобода и ответственность, подобно тому, как это организовано в MSF. Здесь же «просвечивает» дисциплина обязательств.

Практики. В Scrum определены следующие практики.

Sprint Planning Meeting. Проводится в начале каждого Sprint. Сначала Product Owner, Scurm-мастер, команда, а также представители заказчика и пр. заинтересованные лица определяют, какие требования из Project Backlog наиболее приоритетные и их следует реализовывать в рамках данного Sprint. Формируется Sprint Backlog. Далее Scurm-мастер и Scrum-команда определяют то, как именно будут достигнуты определенные выше цели из Sprint Backlog. Для каждого элемента Sprint Backlog определяется список задач и оценивается их трудоемкость.

Daily Scrum Meeting – пятнадцатиминутное каждодневное совещание, целью которого является достичь понимания того, что произошло со времени предыдущего совещания, скорректировать рабочий план сообразно реалиям сегодняшнего дня и обозначить пути решения существующих проблем. Каждый участник Scrum-команды отвечает на три вопроса: что я сделал со времени предыдущей встречи, мои проблемы, что я буду делать до следующей встречи? В этом совещании может принимать участие любое заинтересованное лицо, но только участники Scrum-команды имеют право принимать решения. Правило обосновано тем, что они давали обязательство реализовать цель

итерации, и только это дает уверенность в том, что она будет достигнута. На них лежит ответственность за их собственные слова, и, если кто-то со стороны вмешивается и принимает решения за них, тем самым он снимает ответственность за результат с участников команды. Такие встречи поддерживают дисциплину обязательств в Scrum-команде, способствуют удержанию фокуса на целях итерации, помогают решать проблемы «в зародыше». Обычно такие совещания проводятся стоя, в течение 15-20 минут.

Sprint Review Meeting. Проводится в конце каждого Sprint. Сначала Scrum-команда демонстрирует Product Owner сделанную в течение Sprint работу, а тот в свою очередь ведет эту часть митинга и может пригласить к участию всех заинтересованных представителей заказчика. Product Owner определяет, какие требования из Sprint Backlog были выполнены, и обсуждает с командой и заказчиками, как лучше расставить приоритеты в Sprint Backlog для следующей итерации. Во второй части митинга производится анализ прошедшего спринта, который ведет Scrum-мастер. Scrum-команда анализирует в последнем Sprint положительные и отрицательные моменты совместной работы, делает выводы и принимает важные для дальнейшей работы решения. Scrum-команда также ищет пути для увеличения эффективности дальнейшей работы. Затем цикл повторяется.

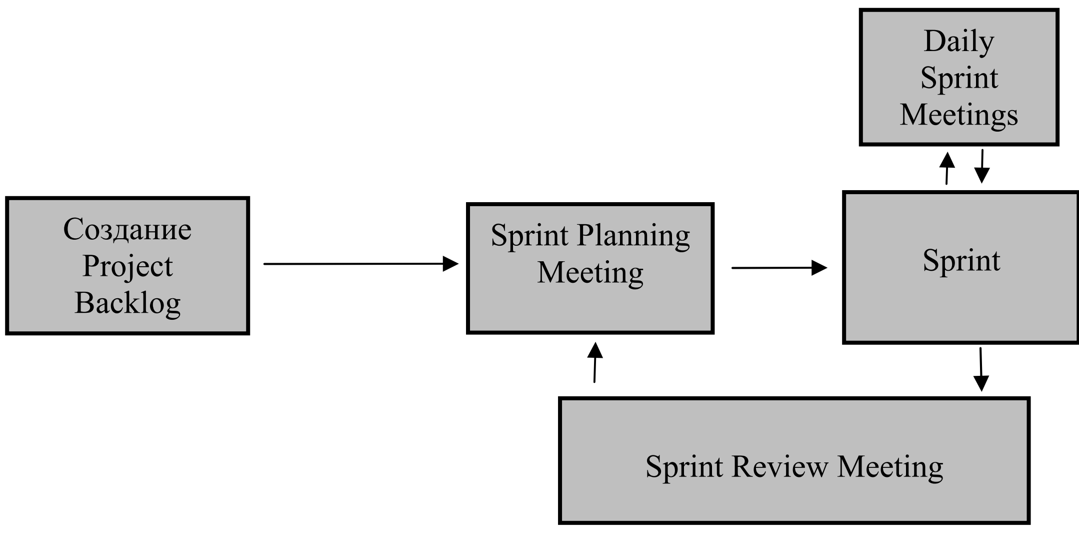

Рис.10.2

Литература

1. Manifesto for Agile Software Development. http://agilemanifesto.org/.
2. M.Fowler. The New Methodology. 2003. http://www.martinfowler.com/articles/newMethodology.html. (Русский перевод: М.Фаулер. Новые методологии программирования. 2001. http://www.maxkir.com/ sd/newmethRUS.html).
3. K.Beck, C. Andres. Extreme Programming Explained. 2004. Paperback. 118 p.
4. М.Борисов. Scrum: гибкое управление разработкой. 30/05/2007 №04. http://www.osp.ru/os/2007/04/4220063/.
5. Wikipedia http://en.wikipedia.org/wiki/Scrum_(development).

Лекция 12. Обзор технологии Microsoft Visual Studio Team System (VSTS)

Обзор

Анализируя собственный опыт разработки программного обеспечения, а также опыт других компаний, специалисты Microsoft пришли к выводу, что существенная часть проблем, возникающих при разработке программного обеспечения, вызвана «человеческим фактором» – взаимодействием различных специалистов в рамках одной команды. Это люди разного возраста, разного образования, разных жизненных принципов и интересов, решающие различные задачи и преследующие различные цели (хотя одна общая цель у них все же есть – сделать в конечном итоге качественное ПО), вынужденные работать вместе волею судьбы или начальства. Не удивительно, что при их взаимодействии часто возникают накладки и недопонимания, а истинно слаженные и эффективные команды встречаются не так часто, как хотелось бы. Для решения этой задачи корпорация Microsoft предлагается комплекс Visual Studio Team System (VSTS), который обеспечивает следующее.

- «Навязывание» процесса разработки. Инструменты VSTS позволяют задать процесс, который используется в проекте (то есть создать конкретный процесс, пользуясь нашей терминологией), и тем самым ограничить действия участников команды.
- Доступное описание процесса. VSTS предполагает доступное описание процесса разработки.
- Единая среда разработки – комплекс инструментов, поддерживающих все этапы процесса разработки ПО и применяемый всеми участниками команды, создавая не только единую интегрированную среду разработки, но и единую культурную среду, общий базис для всех участников команды.

Ядром VSTS является средства обеспечения жизненного цикла *элементов работы* (work items) – некоторых дискретных характеристик проекта, вокруг которых организуется вся работа команды (см. рис. 11.1). Вот примеры элементов работ:

- task – конкретная задача, которую необходимо выполнить в проекте;
- bag – ошибка, которая найдена, ждет своего исправления, исправляется, заново проверяется;
- risk – риск проекта, у которого тоже может быть разное состояние; как правило, за рисками их состояниями следят менеджеры проектов.

Каждый элемент работ имеет набор различных состояний, перечень событий, который могут изменять эти состояния, а также ответственное лицо. Элемент работы используется для оперативного управления проектом следующим образом. Каждый из участников команды видит связанные с ним элементы работ и после выполнения соответствующей работы меняет их состояния, а, возможно и ответственное лицо. Например, программист исправил ошибку и после этого для элемента работ, обозначающего эту ошибку, он меняет состояние (например, Fixed) и ответственного – соответствующего тестеровщика, чтобы последний протестировал изменения кода.

Кроме поддержки жизненного цикла элементов работы в VSTS входят дополнительные средства – контроля версий, поддержки сборки, средства интеграции с офисными приложениями (Project, Excel, Word), генераторы различных отчетов, средства

тестирования и нек. др. Кроме того, через открытый программный интерфейс VSTS можно надстраивать и другим сервисами, необходимыми в процессе разработки. На рис… эти возможные сервисы представлены пустыми кубиками, подобно алтарям неизвестным богам в одном древнем святилище.

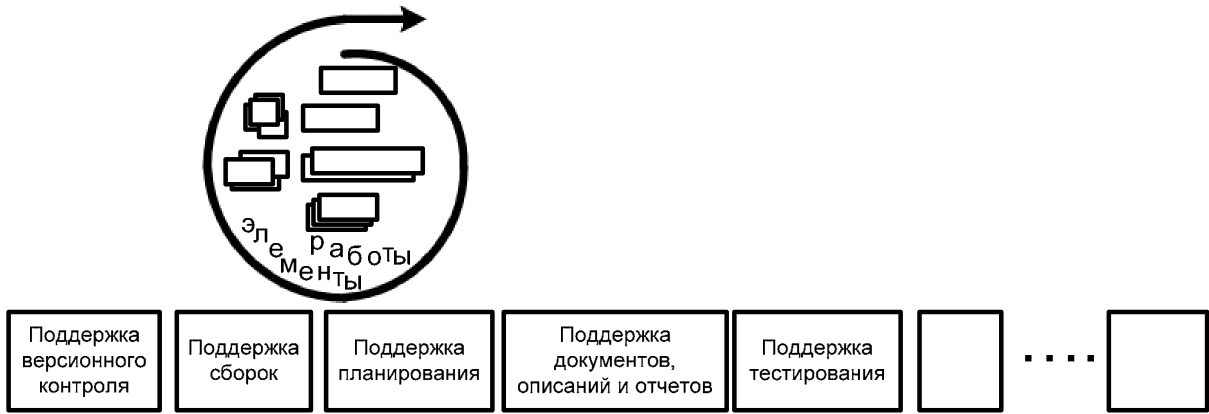

Рис. 11.1

Перечень доступных типов элементов работ, роли в проекте, правила перехода элементов работ из одного состояния в другое, всевозможные дополнительные автоматические действия, сервисы, а также ограничения, различные права ролей и членов команды на изменение элементов работы и перевод их в разные состояния и т.д. – все это являются предварительным описанием и настройкой процесса разработки. Эта настройка выполняется перед началом проекта через механизм настройки *шаблона процесса*.

Состав продукта

Обзор. Теперь посмотрим на VSTS как на программный продукт. Он является сложным, составным продуктом и разделяется на клиентское ПО и серверное ПО – см. рис. 11.2.

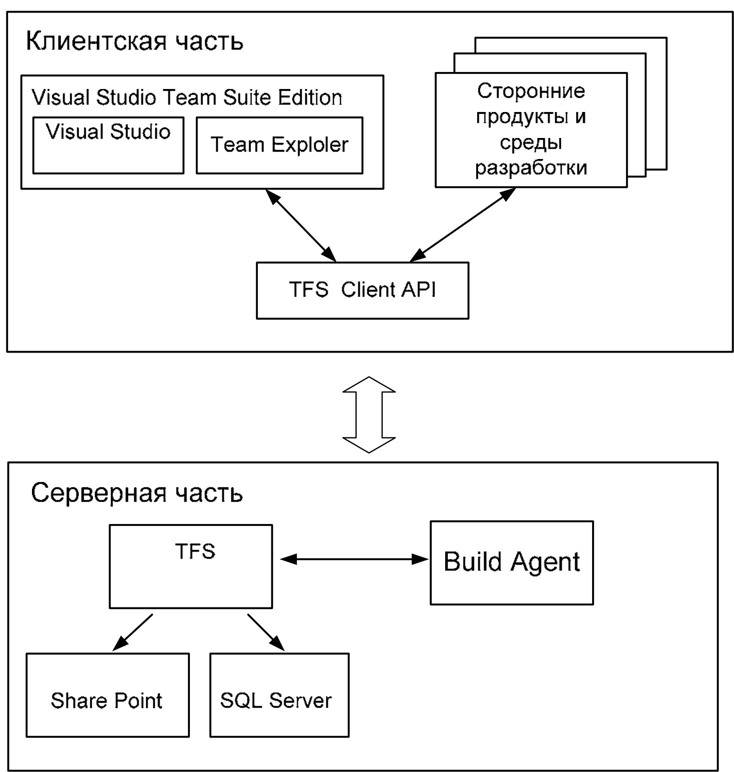

Рис. 11.2 Архитектура VSTS.

Рассмотрим подробнее клиентскую часть. Стандартным клиентом от компании Microsoft является продукт Visual Studio Team Suite Edition. Этот продукт является одной из редакций среды разработки Visual Studio с дополнительным продуктом – Team Explorer. Последний служит для доступа к сервисам серверной части VSTS и встраивается в Visual Studio. Кроме того, благодаря открытому программному интерфейсу к серверной части VSTS – библиотеки TFS Client API – она интегрируется с различными средами разработки, например, с Eclipse. Также существует значительное количество различных клиентских продуктов от сторонних производителей (наиболее успешные из которых Microsoft пытается ассимилировать)[9].

Серверная часть VSTS состоит из TFS (Team Foundation Server) – главной серверной компоненты, – а также компоненты Build Agent. TFS реализует главную функциональность серверной части и использует два других серверных продукта Microsoft – Share Point (для организации Web-портала с описанием используемого шаблона процесса разработки, других документов по процессу) и SQL Server (для хранения данных TFS). Build Agent – это серверная компонента, которая отвечает за выполнение сборок проектов. Вынесение сервера сборки в отдельное серверное приложение позволяет убрать процесс сборки с основной, серверной машины, где размещен TFS, на дополнительную машину, отвечающую именно за проведение сборок. Подобное разделение позволяет значительно снизить нагрузку на основной сервер, особенно в случае использования подхода непрерывной интеграции[10].

---

[9] Более подробную информацию о различных расширениях и дополнениях к TFS можно получить на следующих сайтах: http://blogs.msdn.com/mrod/archive/2008/04/28/external-team-foundation-server-tools.aspx, http://teamsystemrocks.com.

[10] Выделение Build Agent из TFS является run-time взглядом. Структурно, как продукт, TFS включает в себя серверные средства сборки.

Остановимся на клиентской и серверной частях VSTS более подробно.

Клиентская часть VSTS**.** Остановимся на стандартном клиентском ПО, основанном на среде разработки Visual Studio. Последняя выпускается в нескольких комплектациях (editions), ориентированных на разных пользователей. При этом издания, включающие инструменты комплекса VSTS имеют в своем названии слово «Team». Вот перечень этих изданий.

- *Microsoft Visual Studio Team System 2008 Architecture Edition* расширен средствами управления повторным использованием, средствами визуального моделирования с генераторами конечного кода и нек. др. возможностями.
- *Microsoft Visual Studio Team System 2008 Development Edition* предоставление средств анализа кода с целью повышения его качества, в частности, выявление сложного, трудного в обслуживании путем оценки отношений между классами, глубины наследования, цикломатической сложности, строк кода и индекса удобства обслуживания. Сюда же входят различные средства профиляции приложений.
- *Microsoft Visual Studio Team System 2008 Database Edition* включает в себя средства управление версиями всех основных объектов баз данных, модульного тестированиея баз данных, средства поддержки эволюции схем, поддержка синтаксиса SQL и многое другое.
- *Microsoft Visual Studio Team System 2008 Test Edition* предоставляет полный набор средств тестирования Web-приложений и Web-сервисов, интегрированный в среду Visual Studio. С помощью данных средств тестеровщики могут создавать, выполнять и управлять тестами и связанными с ними элементами работ VSTS непосредственно из среды Visual Studio. В это же издание входят средства нагрузочного тестирования, управления тестовыми пакетами и другие возможности.

Помимо четырех «ролевых» изданий, выпускается и издание, объединяющее функции всех четырех блоков – Microsoft Visual Studio Team System 2008 Team Suite. Условно взаимосвязь различных изданий отражена на рис. 11.3.

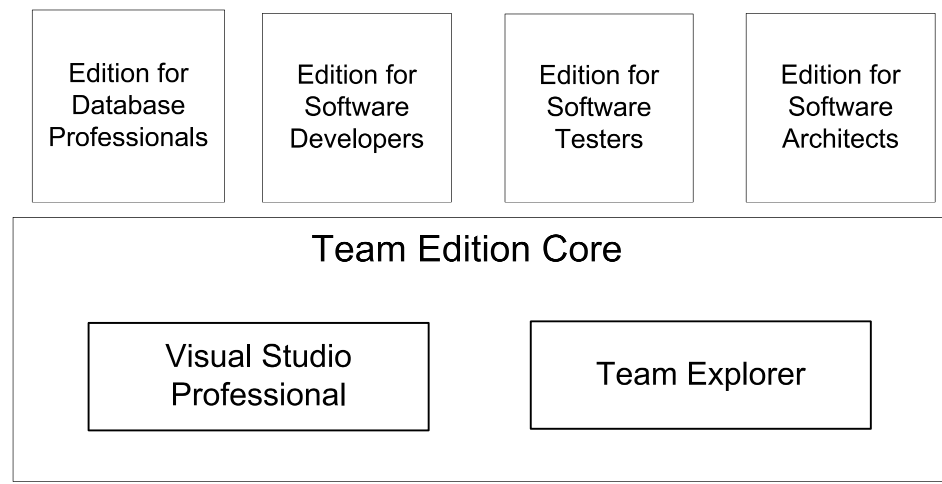

Рис. 11.3. Схема Microsoft Visual Studio Team System 2008 Team Suite**.**

Каждое их четырех «ролевых» изданий серии VSTS расширяет ядро (Team Edition Core) дополнительными инструментами, предназначенными для определенной роли

(разработчик, тестер, архитектор или специалист по базам данных), а издание Team Suite является объединением всех четырех «ролевых» изданий.

Ядро состоит из базовой конфигурацией Visual Studio – *Visual Studio Professional*,– которая является наиболее распространенным изданием среды Visual Studio и повсеместно используется для разработки программного обеспечения. Она дополняется *Team Explorer*, предназначенным для интеграции с TFS.

Серверная часть VSTS. Итак, ядром комплекса инструментов VSTS является TFS, который не является целостной системой, а представляет из себя набор стандартных продуктов (в частности, SQL Server и Share Point), соответствующим образом настроенных и объединенных в единое целое посредством прослойки Web-сервисов. Архитектура серверной части VSTS представлена на рис.11.4, где серыми прямоугольниками показаны компоненты VSTS, а белыми – компоненты других продуктов Microsoft. На этом же рисунке схематично обозначена и клиентская часть VSTS.

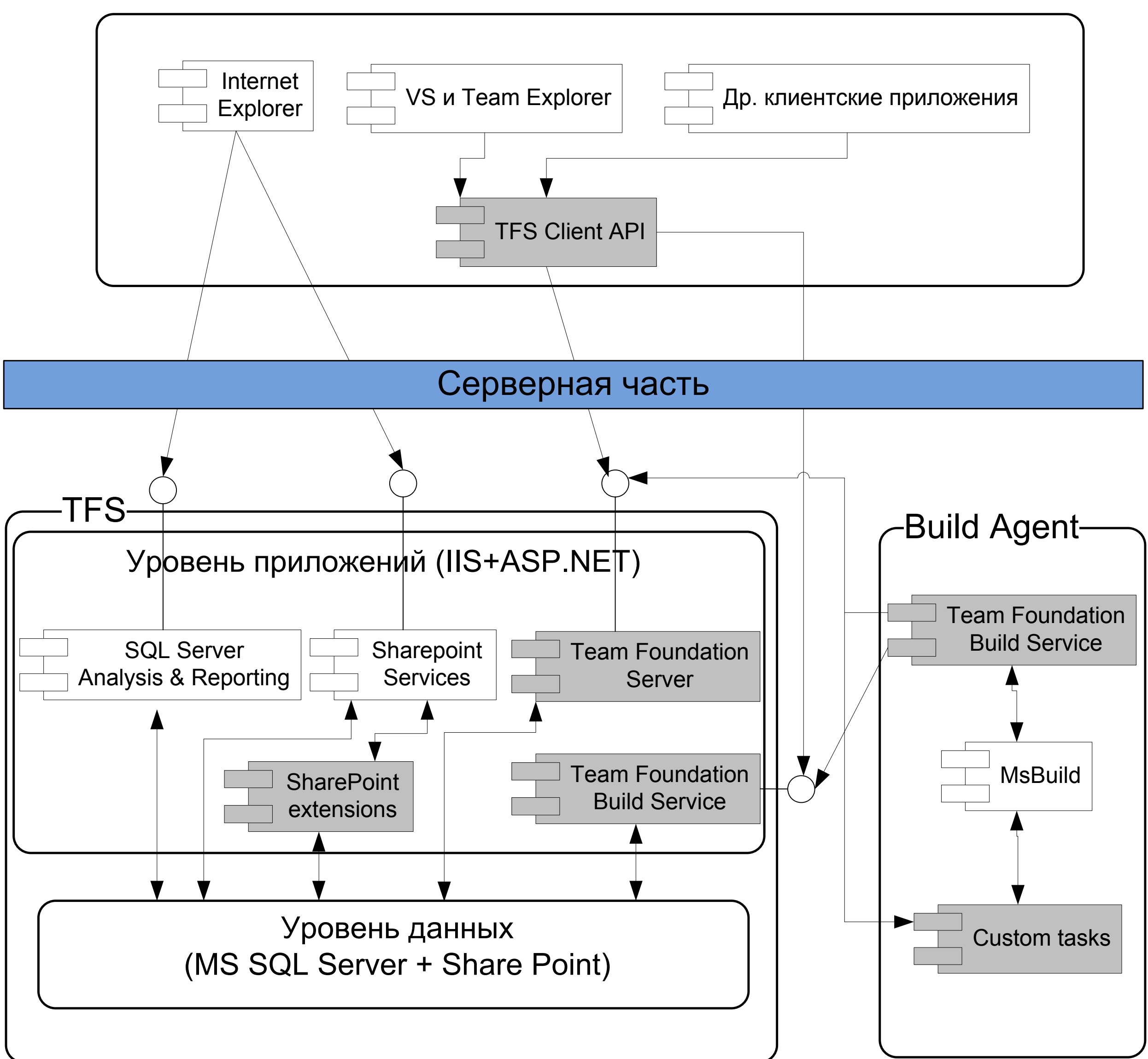

Рис. 11.4. Архитектура серверной части VSTS.

TFS, основная серверная подсистема VSTS, состоит из двух основных уровней: уровня приложений и уровня данных. Уровень приложений TFS включает в себя следующие компоненты.

- *SQL Server Analysis & Reporting* – компонента пакета SQL Server, используемая TFS для построения отчетов анализа статуса проектов. Доступ к этой компоненте с клиентской стороны осуществляется не через компоненту TFS Client API, а напрямую, средствами Web-браузера.

- *SharePoint Services* – компоненты из пакета Share Point, используется для хранения общедоступной информации и описания используемого процесса разработки. Доступ к этой компоненте с клиентской стороны осуществляется не через TFS Client API, а напрямую, средствами Web-браузера.

- *Share Point Extensions for TFS* – расширение Share Point для TFS, которое обеспечивает доступ к отчетам и некоторым функция TFS непосредственно с Web-портала.

- *Team Foundation Server* – главная компонента TFS, которая состоит из набора Web-сервисов, доступных через TFS Client API клиентскому ПО и реализующих основные сервисные функции TFS, в частности:
    - версионный контроль,
    - управление элементами работы,
    - работам с шаблонами процесса,
    - администрирование и т.д.

- *Team Foundation Build Service* в составе TFS – предназначена для инициации процесса сборки и передачи соответствующего задания компоненте Build Agent. Другой экземпляр этого приложения находится в Build Agent и выполняет там системные функции.

Уровень приложений реализован на технологии ASP.NET и работает под управлением IIS (Internet Infromation Service). IIS является Web-сервером, то есть средой для работы Web-сервисов TFS, обеспечивая доступ к функциональности сервера VSTS со стороны его клиентов.

Уровень данных состоит из набора баз данных, где TFS хранит свои данные. Он реализован на основе продуктов MS SQL Server и Share Point.

В зависимости от размера компании-разработчика ПО и предполагаемой нагрузки эти два уровня TFS могут быть установлены на одном сервере (single-server deployment) или на двух разных серверах (dual-server deployment). Для очень больших компаний возможно использование механизмов кластеризации, встроенных в Microsoft SQL Server и Internet Information Server.[11]

Build Agent – еще одна серверная подсистема VSTS. Как уже говорилось выше, она предназначается для выполнения сборки проектов. Выполнение сборки проекта происходит средствами пакета .NET Framework, с помощью стандартной утилиты этого пакета *MSBuild*, которая, получив задание на сборку, вызывает соответствующий компилятор из .NET Framework. Этот же механизм используется и для сборки проекта, запущенной из Visual Studio.

В случае Build Agent выполнение сборки происходит по следующему сценарию. Компонента *TFS Build Service* в составе TFS сообщает такой же компоненте на компьютере, где расположен Build Agent, что надо запустить выполнение сборки. А та, в свою очередь, являясь системным сервисом и будучи запущенной, оказывается тем процессом Windows, в рамках которого и будет происходит выполнение сборки под управлением компоненты MSBuild. При этом всю связь с TFS для выполнения сценария сборки осуществляет компонента *Custom tasks*. В сценарии сборки указывается, откуда нужно брать исходные тексты собираемого приложения, откуда брать регрессионные тесты и как их запускать, как создавать отчеты по результатам сборок и т.д.

---

[11] Подобная распределенная архитектура накладывает серьезное ограничение на схему развертывания TFS – нормальной его работы можно добиться только в сети с доменом Active Directory. Использование TFS в других условиях возможно, но сопряжено с дополнительной существенной нагрузкой по развертыванию и администрированию.

Правила инсталляции

Клиентская часть устанавливается легко, либо как расширению существующей установки Visual Studio, либо на чистую машину (в этом случае базовая инфраструктура Visual Studio будет установлена автоматически). Основная работа при установке VSTS – развертка серверной части, то есть TFS. В версии 2008-го года установка TFS значительно улучшена и упрощена по сравнению с версией 2005-го года, однако, требования на программное окружение по-прежнему достаточно жесткие:

- Microsoft Windows Server 2003 или 2008.
- Microsoft SQL Server 2005 или 2008[12].
- Internet Information Server 6 (для Windows Server 2003) или 7 (для Windows Server 2008).
- Active Directory Domain 5.0 или 2003 (TFS не работает с доменом 4.0). Важно отметить, что нормальное использование TFS вне домена наладить достаточно сложно – для этого приходится использовать технологию VPN или Web-клиента, что вдет к существенному увеличению затрат на администрирования и накладных расходов при работе.
- SharePoint Server 3.0 или Microsoft Office SharePoint Server 2007. При обновлении с TFS 2005 можно остаться на Windows SharePoint Server 2.0.

Пакет Team Explorer

Данный пакет является самым распространенным клиентским приложением VSTS. Он встраивается в среду Visual Studio в виде плавающего окна, а также ряда диалоговых окон и окон-документов. Его внешний вид представлен на рис. 11.5.

[12] Установку SQL сервера необходимо производить строго в соответствии с инструкцией для TFS.

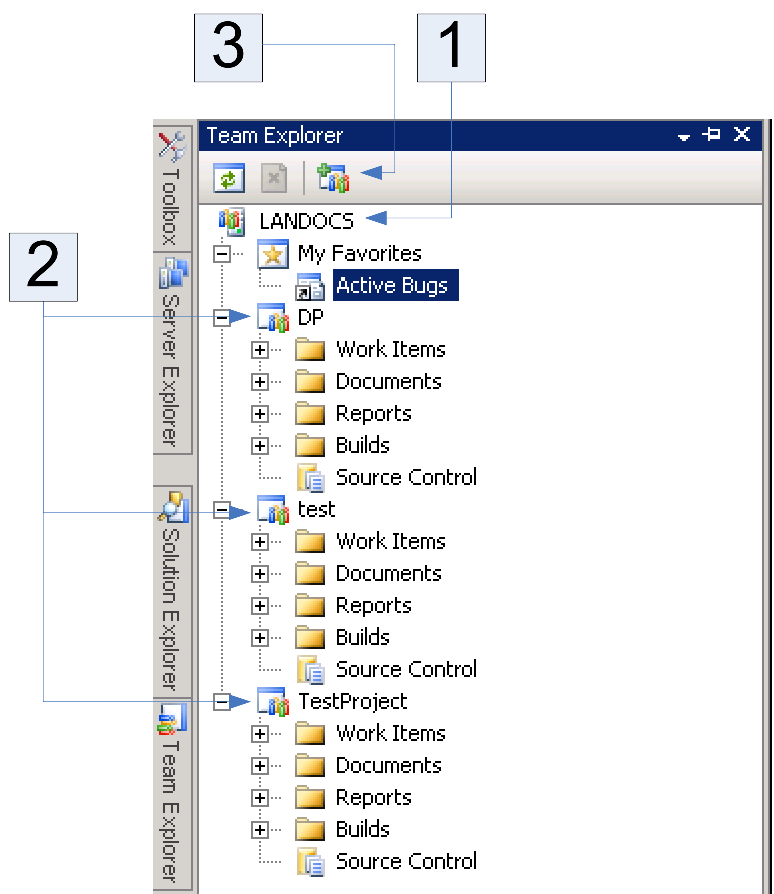

Рис. 11.5. Внешний вид Team Explorer.

Основное дерево Team Explorer содержит:

- список доступных TFS-серверов, (1); каждый такой сервер является экземпляром серверной части Team System и, как правило, располагается на отдельном компьютере[13];
- список доступных проектов для каждого из подключенных серверов (2);
- панели инструментов инструментального окна для того, чтобы подключить/добавить в TFS новый проект (3).

Для каждого из проектов в дереве Team Explorer отображается следующая информация.

- Список элементов работы (Work Items) проекта, то есть всех тех дискретных элементов работы в проекте, которые создают менеджеры и другие участники проекта для того, чтобы ни о чем не забыть, а также для коммуникации друг с другом.
- Список доступных документов (Documents). В этом списке отображаются документы, хранящиеся на портале проекта. Как правило, это нормативные или вспомогательные документы, не требующие хранения в системе контроля версий.
- Список доступных отчетов (Reports). В этом списке представлены доступные для проекта отчеты. Результат выполнения отчета открывается в отдельном окне документе.

[13] Исключение – это когда инсталляция TFS является кластерной и развертывается на нескольких компьютерах.

- Список сборок (Builds) проекта – описаний и результатов.
- Система контроля версий (Source Control). Позволяет получить доступ к версионному репозиторию с основными артефактами проекта (открывается в отдельном окне-документе).

Кроме того, через контекстные меню в дереве проектов можно выполнять следующие операции:

- создать новый проект или подключится к существующему;
- поменять настройки сервера или проекта;
- создать/удалить/изменить запрос на элементы работы;
- создать/удалить/изменить отчет;
- создать/удалить/изменить/запустить процесс сборки;
- создать/изменить/удалить документ;
- подписаться на определенные оповещения или отменить подписку.

Лекция 13. VSTS: управление элементами работ (Work Items)

Определение, свойства, жизненный цикл

Обзор. Вернемся к элементам работ VSTS – ключевым дискретным характеристикам проекта, таким как задача (task), ошибка (bug), риск (risk) и т.д. Эти характеристики выделены в VSTS с целью конкретизировать объекты управления в проекте, сделать это управление сквозным в следующих смыслах:

- обеспечить доступ к одной и той же информации для разных участников (и, главное, ролей!) в проекте; например, доступ к ошибкам для менеджеров, разработчиков и тестеров;
- прослеживать связи одних элементов с другими, например, изменений исходного кода и теми ошибками, для исправления которых эти изменения были сделаны.

Благодаря единой среде, включающей в себя средства поддержки различных видов элементов работы, в VSTS гораздо проще строить связи между элементами работы различного вида, отслеживать их изменения, чем при использовании отдельных продуктов поддержания процесса. Например, не нужно ждать момента, пока информация об ошибке или задаче будет перенесена из одной системы в другую. Ведь традиционно программные средства планирования (там, где определяются задачи), управления ошибками (там, где происходит учет ошибок), средства версионного контроля – это разные средства. Кроме того, в силу наличия единого информационного репозитория в VSTS возможны строгие ссылки на такие объекты, определенная сборка или тест, и получение, по соответствующему запросу, подробной информации по разным фильтрам, на разную глубину детализации. Возможно также настроить автоматическую генерацию элементов работы, например, ошибок при неудачной автоматической сборке или при автоматическом прогоне тестов.

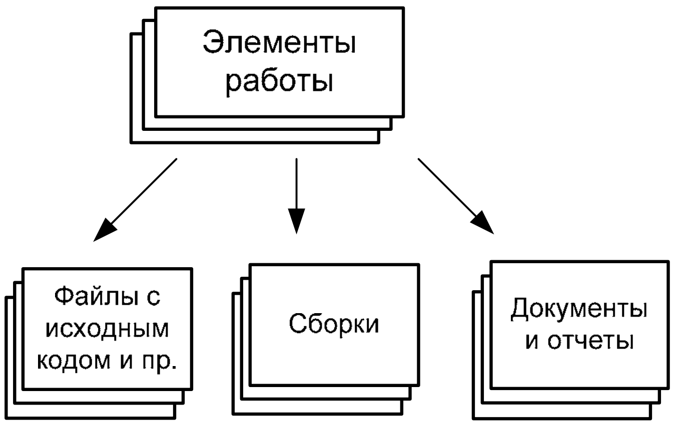

Рис. 12.1.

Элементы работы можно связывать с другими артефактами проекта - файлами с исходным кодом, сборками (как настройками, так и результатами), документами (по процессу, проектными, пользовательскими и др.), отчетами, которые могут также храниться в VSTS. Все это показано на рис. 12.1. Важно отличать элементы работы от этих артефактов – последние являются рабочими продуктами (точнее, из них рабочие продукты сформируются), а элементы работы являются управляющей информацией в

проекте. Правда, и по ним также можно создавать рабочие продукты – генерировать отчеты, документы и планы в продуктах Office и т.д. Но сами по себе элементы работы являются средством оперативной работы над проектом, а не результатами (в том или ином смысле).

Реквизиты. Каждый элемент работы принадлежит определенному типу. Тип элемента работы определяет набор реквизитов, для каждого из которых можно задать:

- имя, отображаемое в отчетах и пользовательском интерфейсе TFS;
- имя для ссылок (Reference Name), используемое для указания ссылок на данный реквизит из других мест шаблона процесса;
- тип – один из предопределенных типов для реквизитов: date, int, string, bool; типы могут *системными*, то есть являться ссылками в одно из типовых хранилищ TFS; например, тип build results указывает на информацию о результатах сборки;
- текстовое описание реквизита, отображаемое во всплывающих подсказках, отчетах и сообщениях;
- режим использования в отчетах – можно ли использовать этот реквизит в отчетах и как его следует использовать.

Бывают также системные реквизиты, имена, типы и обработка которых «прошита» в TFS. Это, например, такие реквизиты как состояние (state), причины (reasons), связи (links).

Отдельного внимания заслуживают имена для ссылок. Они позволяют идентифицировать данный реквизит не только в пределах одного типа элементов работы, но и в пределах всего TFS-проекта, а также при переносе элементов работы из проекта в проект. Для поддержания уникальности этих имен рекомендуется использовать концепцию пространств имен (namespaces). Кроме того, имена для ссылок служат для организации своего рода пула реквизитов – одинаковое имя для ссылок, использованное в разных типах элементов работы, подразумевает одинаковый смысл соответствующих реквизитов, а также одинаковую их роль с точки зрения формирования отчетов. Существует набор предопределенных имен для ссылок, соответствующих системным реквизитам. Реквизиты с соответствующими именами для ссылок могут подвергаться особой обработке со стороны TFS.

Каждый реквизит может либо не участвовать в отчетах вообще, либо участвовать в следующих режимах:

- как измерение (Dimension) – в качестве измерения при построении отчетов; этот режим допустим для чисел, дат и строковых полей с предопределенным набором значений;
- в деталях (Details) – то есть в качестве детальной информации отчета; этот режим допустим для чисел, дат и произвольных строк;
- как метрика (Measure) – то есть в отчетах используется некоторая статистическая функция, вычисленная для значений данного реквизита.

Жизненный цикл элемента работы определяется двумя системными реквизитами: состоянием и причиной.

Первый описывает текущее состояние элемента работы и определяет его текущую роль в процессе. Каждый тип элементов работы описывает допустимый набор состояний, например, «активный», «завершенный», «проверенный» и т.д.

Кроме того, каждый тип элемента работы имеет описание переходов между своими состояниями состояний, причины, вызывающие эти переходы и действия, выполняемые в них. Причинами могут являться, например, «выполнен», «устарел», «отложен» и т.д. Переходы может осуществлять сама система TFS, автоматически, но в большинстве

случае разработчик сам инициирует переход, внося в систему информацию о том, что выполнил задачу, исправил ошибку и так далее.

Таким образом, жизненный цикл элемента работы представляется ориентированным графом, нагруженным как по узлам, так и по дугам. Использование такого графа вместо нагруженного только по узлам, как например, часто можно встретить в системах управления ошибками, позволило резко сократить размер описания и повысить информативность.

Для настройки типов рабочих элементов используется продукт Team Foundation Power Tools[14]. Он является свободно распространяемым продуктом и содержит, в частности, визуальный редактор, позволяющий просматривать и редактировать жизненный цикл элемента работы в визуальном виде – см. рис. 12.2. На этом рисунке показан граф жизненного цикла для типа рабочего элемента «ошибка». Мы можем видеть три состояния – Active (обнаружена ошибка), Resolved (ошибка исправлена), Closed (ошибка закрыта), – и переходы между этими состояниями. В каждом переходе имеется прямоугольник Transition, в котором содержатся параметры перехода – причина, действия, которые нужно выполнить, список реквизитов, который должен быть изменен при переходе и некоторую другую информацию.

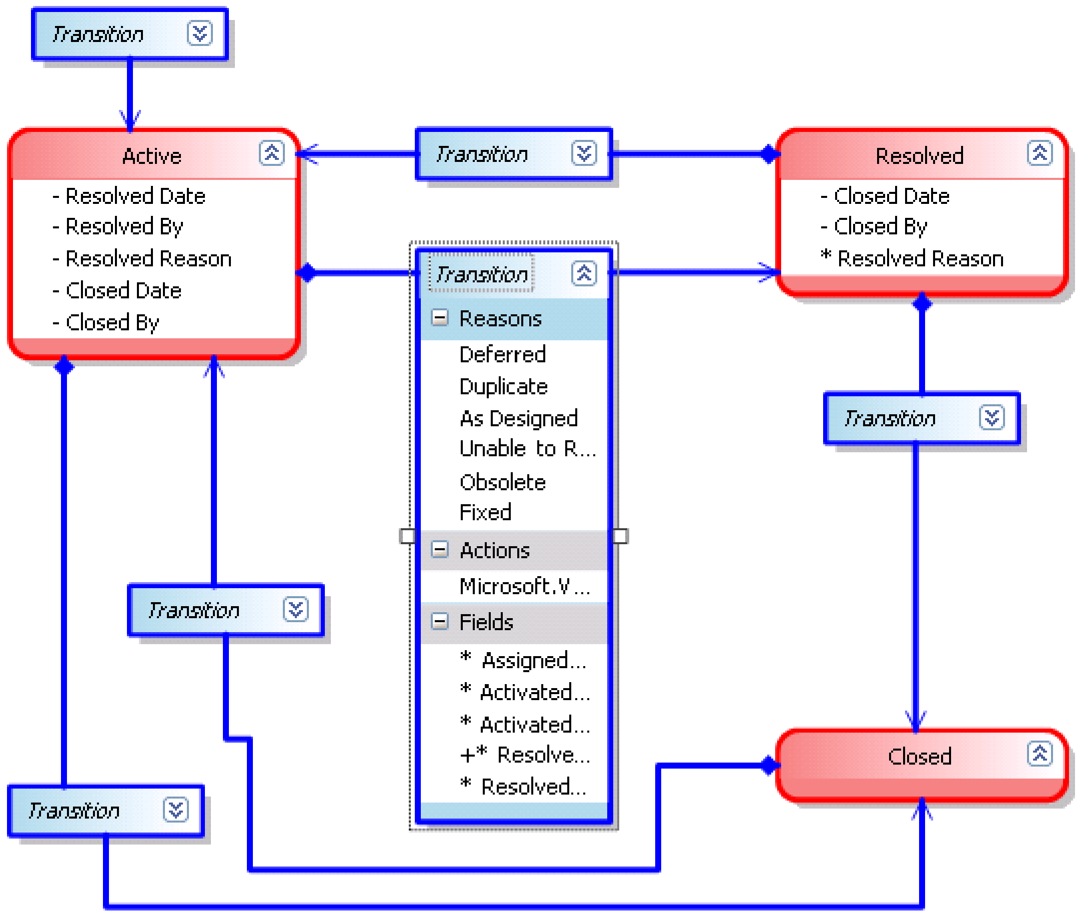

Рис. 12.2. Жизненный цикл элемента работы типа «Ошибка».

Еще одной важной составляющей описания жизненного цикла реквизита являются ***правила***, которые описывают различные ограничения на значения реквизитов элемента работы, в том числе:

[14] Его можно скачать отсюда http://msdn.microsoft.com/en-us/tfs2008/bb980963.aspx

- базовые ограничения: обязателен для заполнения или нет, доступен только на чтение, пусто, не может быть сброшено, может быть только сброшено;
- обязательное совпадение или несовпадение по значению с другим реквизитом;
- является датой в прошлом или в будущем;
- является именем пользователя, входящего в заданную группу, например то, что за этот элемент работы должен отвечать работник из только группы тестеровщиков.
- значение должно всегда удовлетворять шаблону некоторого регулярного выражения;
- значение является одним из предопределенных значений или наоборот, не является; для задания таких правил допустимо использование ссылок на внешние списки, например, на списки выполненных сборок; кроме того, можно определить список предполагаемых значений, которые будут предложены пользователю, но не обязательны к выбору.

При описании каждого правила можно указать имя пользователя или группы, для которых это правило будет применяться. Исключением является только правило «позволить сохранить текущее значение», применяемое всегда для всех пользователей.

Кроме того, можно описать правила, применяемые в разные моменты времени, в том числе:

- постоянно действующие ограничения или применяемые при создании правила;
- при изменении определенного реквизита, или наоборот, если реквизит остался неизменен;
- при совпадении или несовпадении значения реквизита с предопределенным значением;
- при переходе или во время нахождения элемента работы в определенном состоянии;
- при совершении определенного перехода;
- при совершении определенного перехода по определенной причине в жизненном цикле элемента работы.

Система правил предоставляет обширные возможности для спецификации различных тонкостей бизнес-процесса, однако платить за эту гибкость приходится сложностью настройки. Именно описание корректных правил потребовало от нас наибольших усилий при разработке собственного шаблона и его использовании в реальном промышленном проекте.

Еще один важный механизм настройки реквизитов в VSTS – это задания способа представления реквизита в экранных формах редактирования, просмотра в отчетах. Для этой цели в TFS существует достаточно гибкий диалект XML, включающий предопределенный набор элементов пользовательского интерфейса, а также позволяющий группировать их, разбивать по колонкам или размещать на закладках. Используя вложенность элементов и групп, можно описать строгий, удобный и красивый пользовательский интерфейс, но иногда для этого требуется много времени. Некоторые типы элементов пользовательского интерфейса можно связывать с реквизитами элемента работы, используя соответствующее имя для ссылок.

Средства использования

Пример: элемент работы task. Кратко опишем пример, который будет использован в дальнейшем для объяснения средств использования элементов работы.

Рассмотрим элемент работы типа task (задача). Как правило, в начале проекта некоторый эксперт (как правило, системный архитектор, ведущий разработчик и т.д.) проводит анализ всей необходимой работы по проекту и разбивает её на подзадачи,

устанавливая ответственных, сроки и т.д. Эти подзадачи с соответствующими атрибутами и являются элементами работы типа task.

Затем менеджер проекта, с учетом списка всех задач и их взаимосвязей, строит календарный план. На этом этапе менеджеру могут оказаться полезными средства Project и Microsoft Excel – он пользуется ими на основе соответствующих мостов, имеющихся в TFS.

Далее разработчики начинают реализовывать соответствующие задачи. После того, как было внесено последнее изменение и задача выполнена, разработчик переводит элемент работы в состояние Resolved и информация о нем войдет в следующий отчет по автоматической сборке. При обнаружении ошибок реализации тестер создаст новый элемент работы типа Bug и проставит ему связь с исходной задачей. Если же тестер обнаружит, что функциональность реализована не в полном объеме, то он может решить перевести задачу обратно в состояние Active. Если же реализованная функциональность достаточно стабильна, а имеющиеся ошибки не являются критичными, тестер переводит задачу в состояние Closed.

За всем этим процессом наблюдает менеджер проекта, используя как запросы на элементы работы, так и средства интеграции с офисными приложениями, а также средства построения отчетов. Таким образом, он получает возможность максимально оперативно реагировать на возникающие нештатные ситуации, отставания от плана и возникающие дополнительные незапланированные работы.

Создание элементов работы. Для создания нового элемента работы можно воспользоваться пунктом меню Team, добавляемым в Visual Studio вместе с Team Explorer, как показано на рис. 12.3.

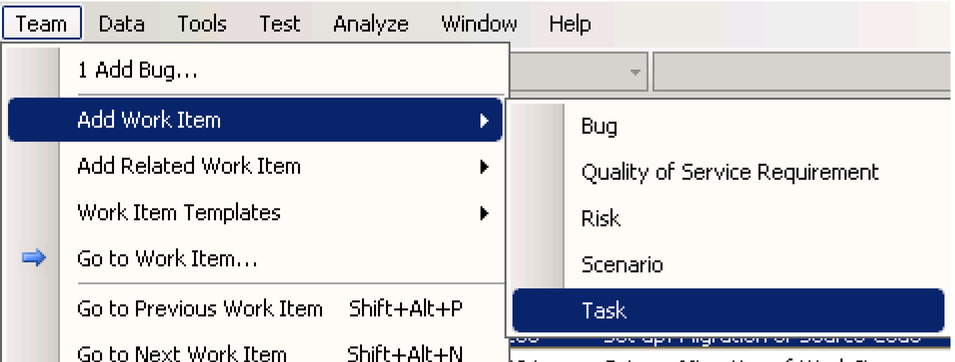

Рис. 12.3. Создание элемента работы.

После выбора пункта меню Add Work Item отобразился список из существующих в данном проекте типов элементов работы. Далее, после выбора соответствующего пункта меню будет открыто окно редактирования нового элемента работы, как показано на рис. 12.4:

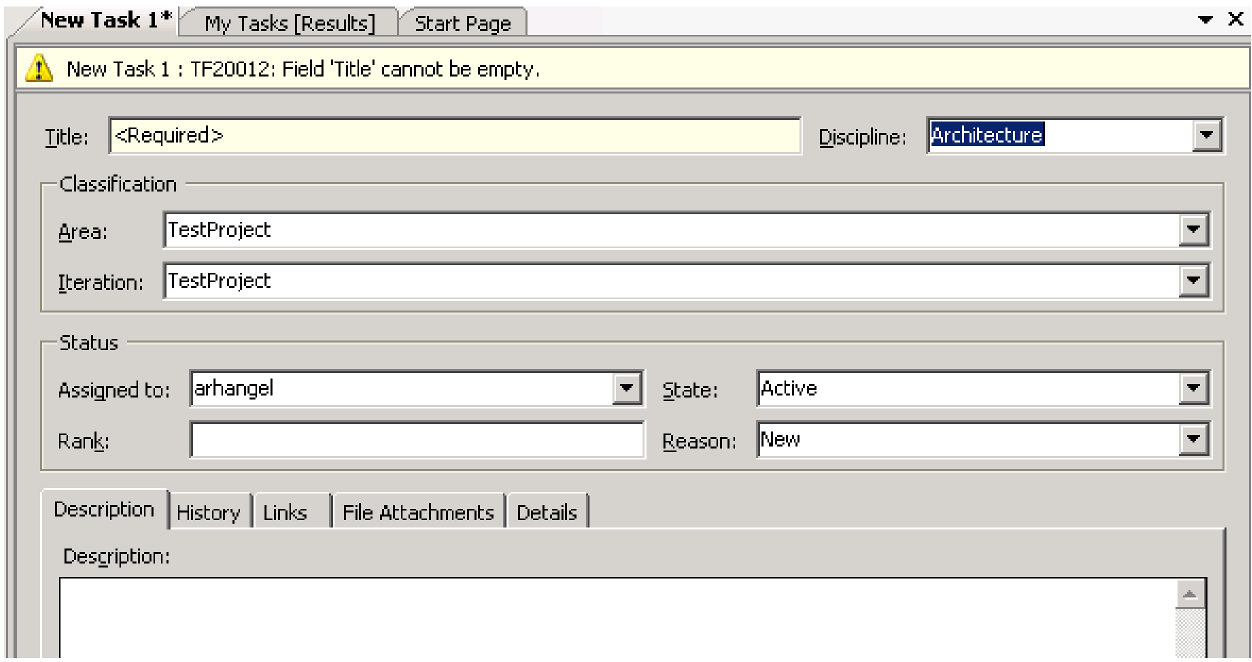

Рис. 12.4. Редактирование элемента работы.

Это окно позволяет заполнить все реквизиты элемента работы, а в случае ошибок выдаст соответствующее предупреждение в верхней части окна. После того, как все поля заполнены, сохранить элемент работы можно посредством кнопки панели инструментов. После сохранения элемент работы автоматически получит уникальный идентификатор и будет сохранен в системе управления элементами работы.

Для добавления связанных элементов работы можно воспользоваться командой контекстного меню Add Related Work Item, как показано на рис. 12.5:

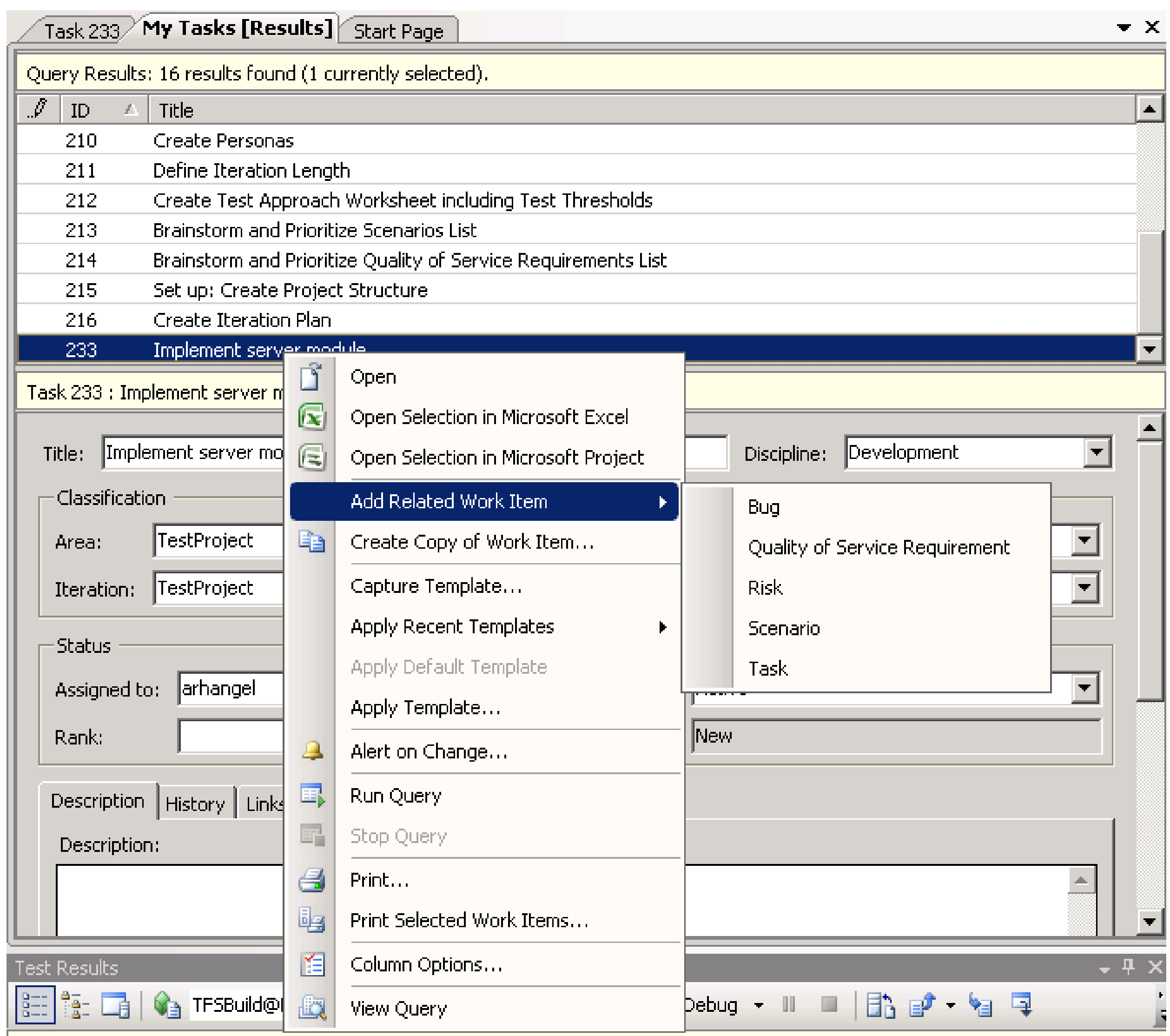

Рис. 12.5. Добавление связанного элемента работы.

После добавления связанного элемента работы откроется окно редактирования для вновь созданного элемента работы, при этом связь между двумя элементами работы будет добавлена автоматически, как показано на рис. 12.6.

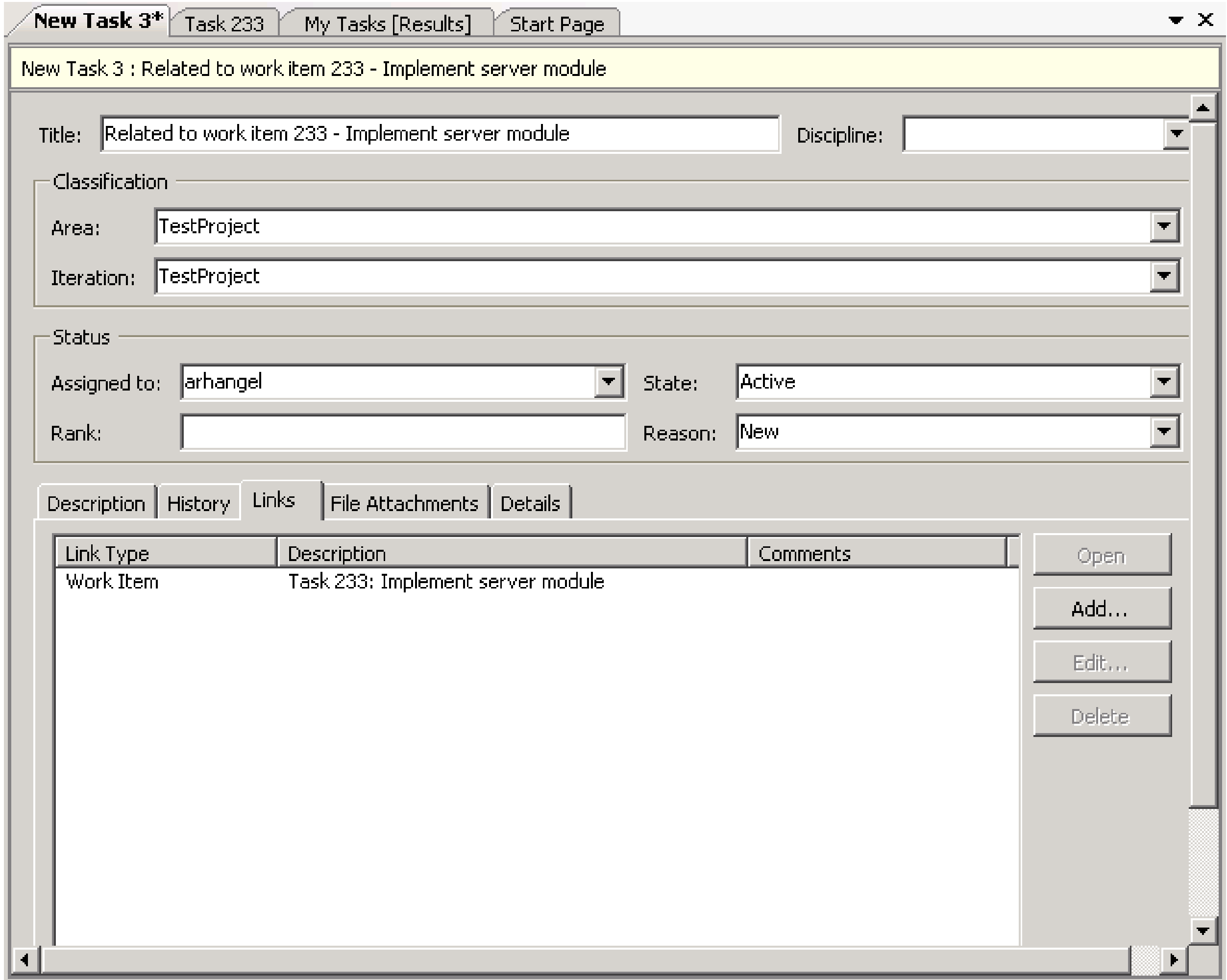

Рисунок 12.6. Список связей.

Созданной связи можно приписать соответствующий комментарий. К сожалению, он будет одинаковым для записей о связи в обоих созданных задачах, что затрудняет идентификацию концов связи.

Заметим, что контекстное меню на рис. 12.5 демонстрирует еще одну полезную при создании элементов работы возможность – шаблоны элементов работы. Шаблон определяется набором «предзаполненных» атрибутов элемента работы и может сильно облегчить жизнь участникам проекта, которым приходится создавать много однотипных элементов.

Доступ к элементам работы. Члены команды, работающие с VSTS через Team Explorer, имеют доступ к элементам работы, открыв в текущим проекте вкладку Work Items (см. рис 12.7). При этом элементы работ доступны не как огромная куча (легко понять, что их может быть очень много в каждом проекте – сотни и даже тысячи), а с помощью специальных фильтров – *запросов* (queries). Они распределены по папкам Team Queries и My Queries. В первой папке располагаются запросы, видимые и используемые всей командой. Во второй папке располагаются запросы, созданные конкретным пользователям для себя лично. Все это можно увидеть на рис. 12.7.

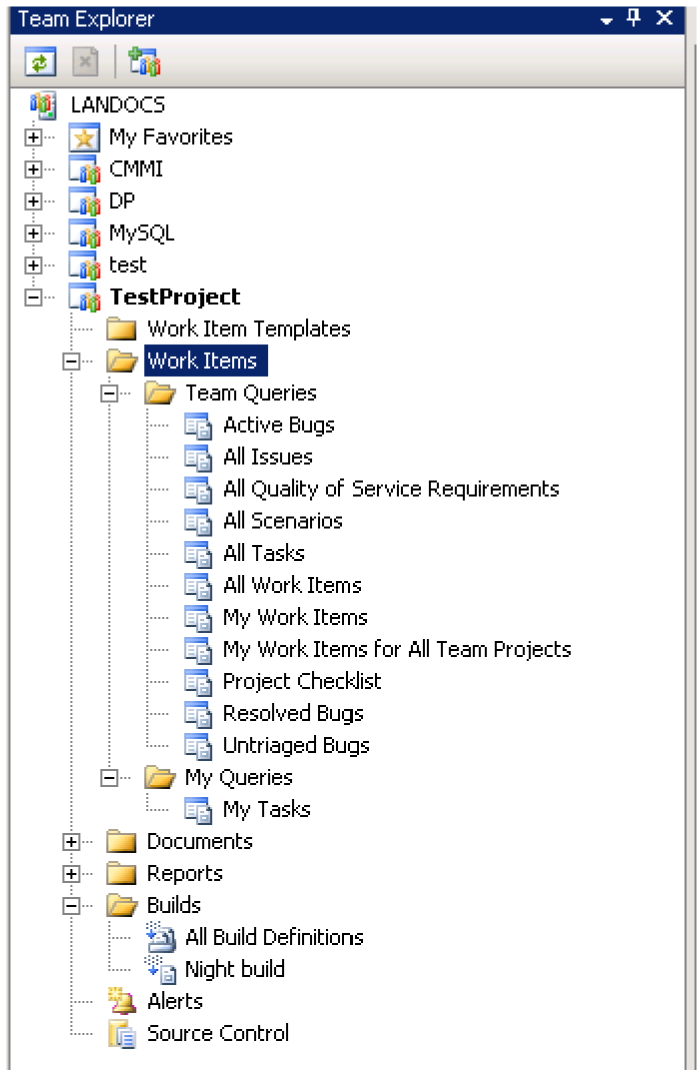

Рис. 12.7. Запросы на элементы работы.

Итак, когда разработчику понадобилось получить доступ к определенной группе элементов работ (например, к ошибкам, которые ему нужно исправить), он выбирает соответствующий запрос и выполняет его. В результате в специальном окне будет открыт список элементов работы, удовлетворяющих данному запросу (рис.12.8), а при выборе определенного элемента, в нижней части окна-списка отобразится детальная информация об этом элементе (а при двойном щелчке элемент работы будет открыт в отдельном окне).

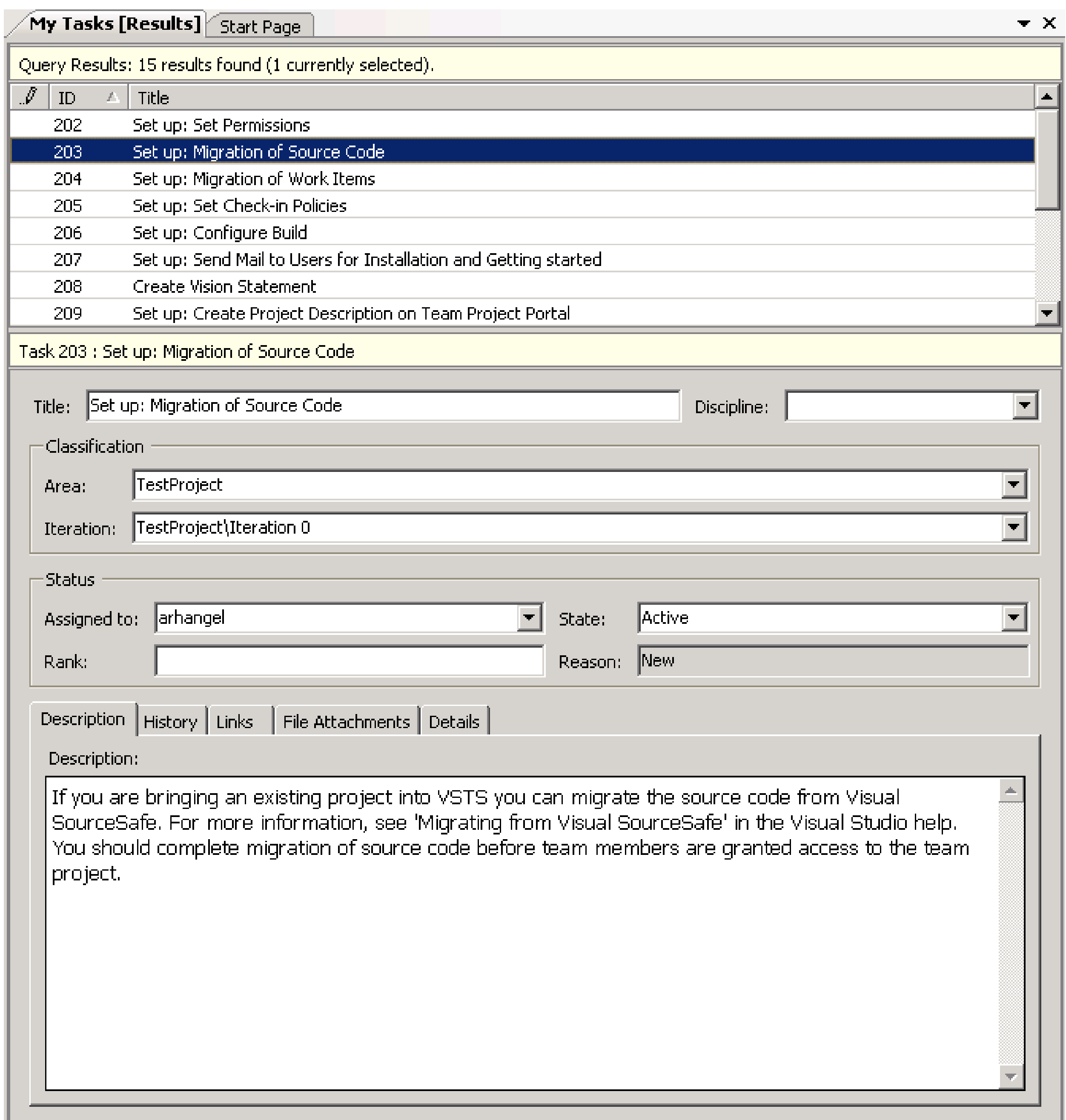

Рис. 12.8. Список элементов работы.

Способ отображения списка (набор колонок, сортировка, и т.д.) настраивается индивидуально для каждого пользователя, а форма детальной информации об элементе работы настраивается для проекта в целом для каждого типа элементов работы в отдельности.

При редактировании и создании элементов работы учитываются все те правила, заданные в шаблоне процесса, где определен данный тип элементов работы. Это выражается в том, что соответствующие поля формы свойств элемента работы допускают или запрещают редактирование, позволяют выбор значений только из определенного списка и т.д.

Выделенные (selected) элементы работы можно экспортировать в пакеты Microsoft Project, Word, Excel, используя соответствующие кнопки панели инструментов и соответственно. Изменения, произведенные с элементами работ, выполненными в этих

пакетах, можно затем загрузить обратно в TFS используя соответствующие библиотеки-расширения офисных приложений.

Элементы работы при планировании. Не сложно заметить, что такая важная роль как менеджер проекта, не получила собственного издания Visual Studio. Связано это с тем, что основная платформа Visual Studio плохо приспособлена для задач, которые приходится решать этой роли. Гораздо более удачно для этого подходят офисные приложения – Microsoft Excel и Microsoft Project. Поэтому для более полного вовлечения менеджера в информационное пространство проекта Team System предоставляет специальные мосты.

Рассмотрим пример с пакетом Project. В этом пакете, при наличии на том же компьютере Team Explorer, появляется пунгт меню Team. В нем нужно выбрать необходимый проект в VSTS, как показано на рис. 12.9.

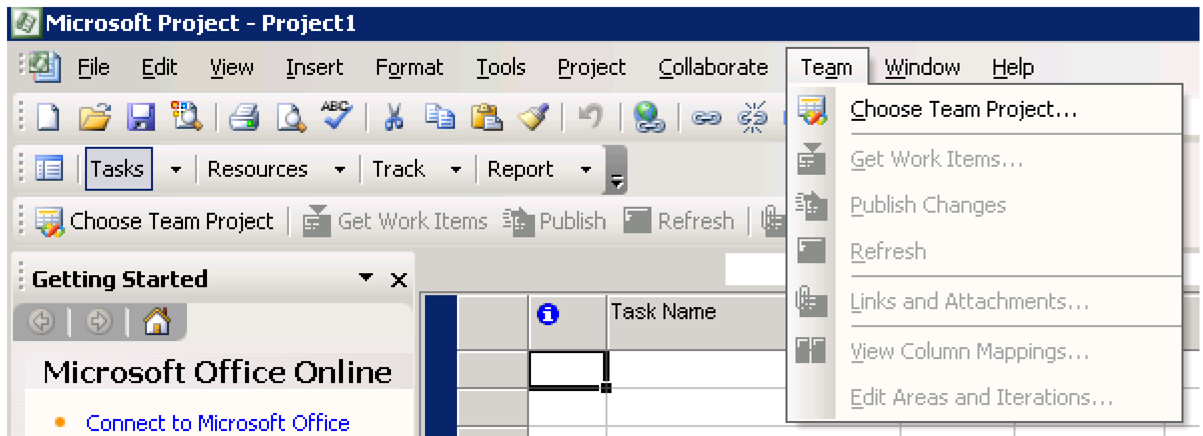

Рис. 12.9. Меню Team в Microsoft Project.

После этого появится возможность использовать другие пункты меню Team, в частности – пункт меню Get Work Items, позволяющий считать необходимые элементы работы с сервера. При выборе этого пункта меню появится диалог, показанный на рис. рис. 12.10. Поиск нужных элементов работы можно осуществлять в соответствии с существующим запросом, по заданным идентификаторам или по названию элемента работы.

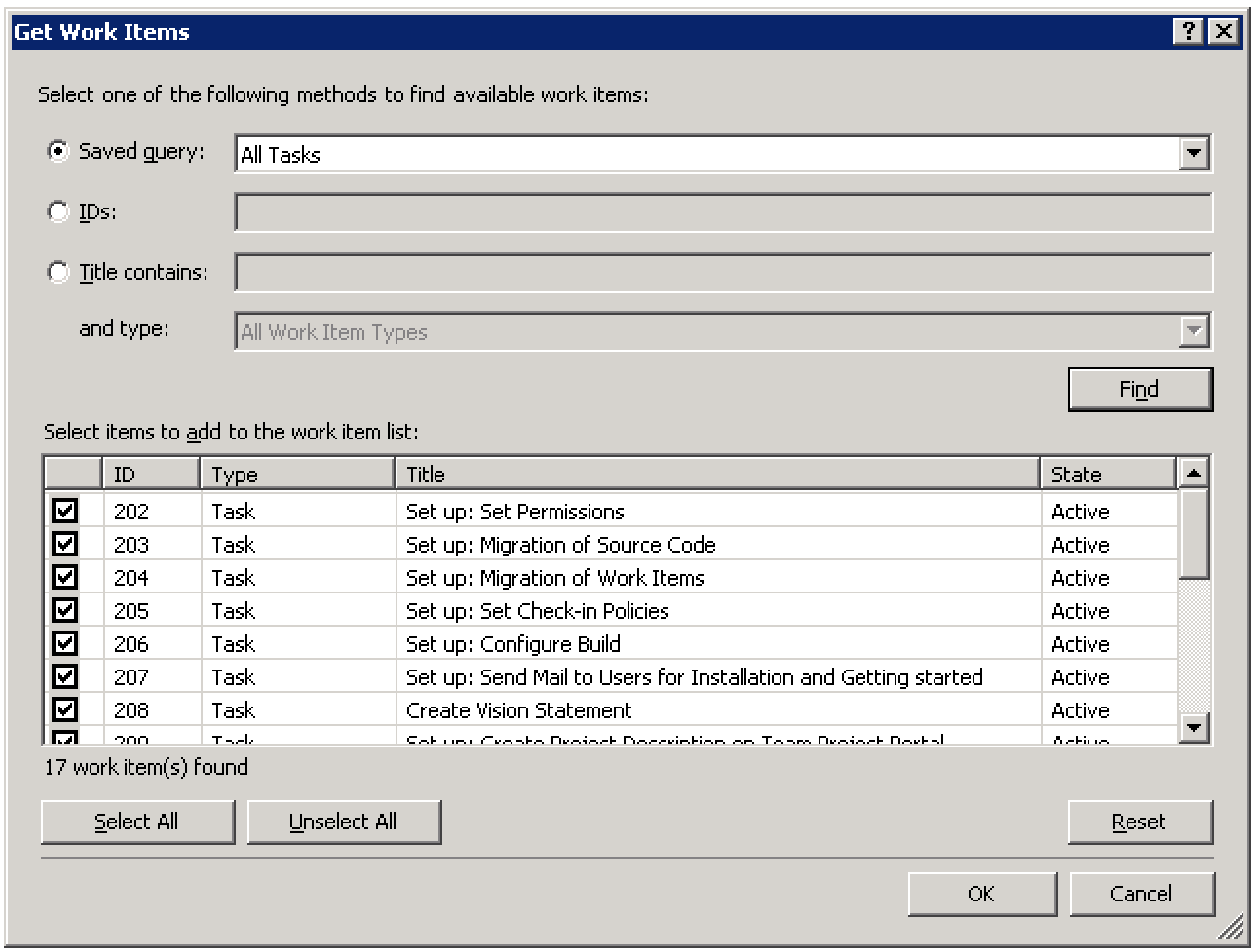

Рис. 12.10. Форма поиска элементов работы.

После того, как нужные элементы выбраны, они будут автоматически импортированы в Project – см. рис. 12.11:

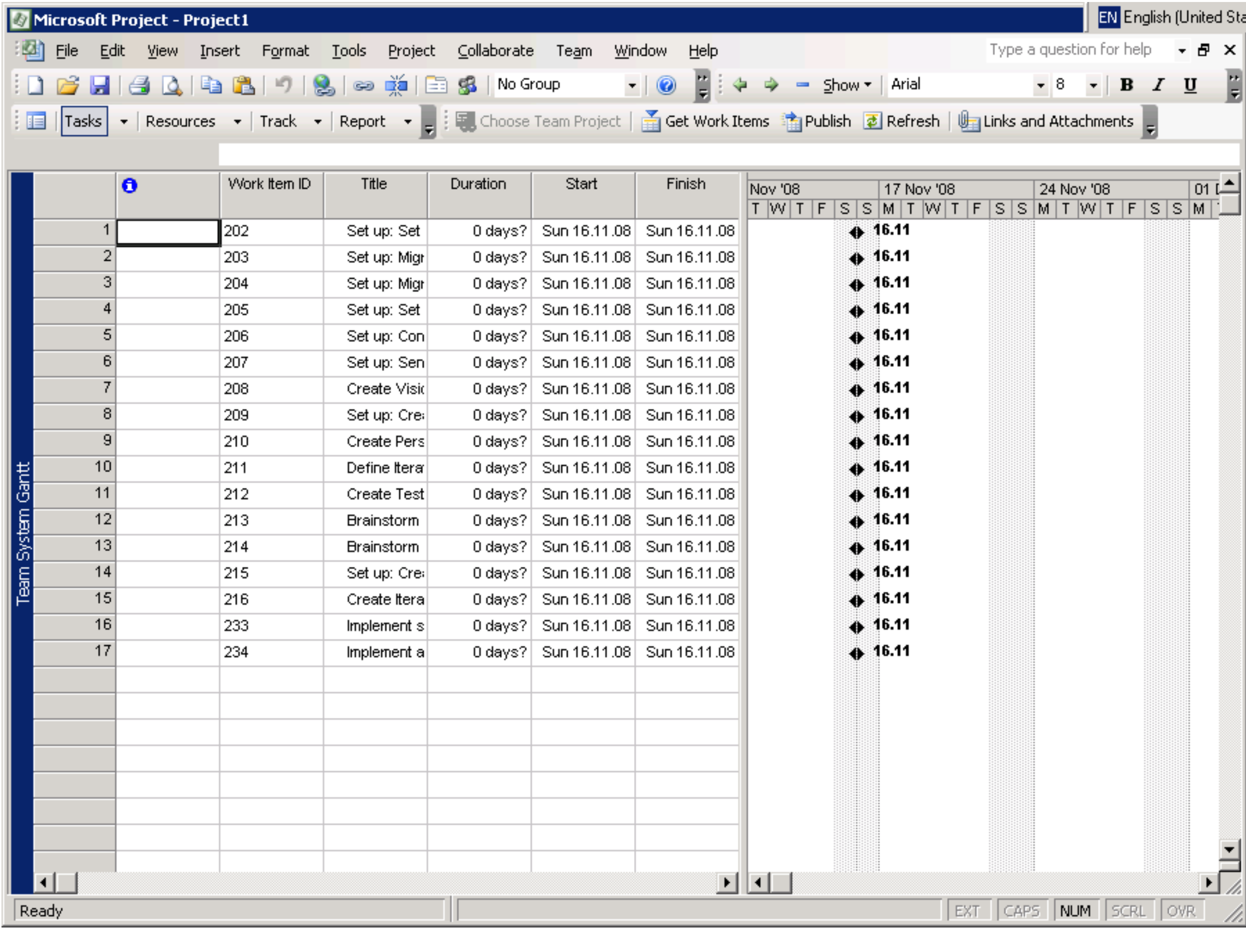

Рис. 12.11. Элементы работы в Microsoft Project.

После импорта элементов работы менеджер может проводить с ними все действия, которые он привык выполнять в Project. В данном случае он создает полноценный план работ, как показано на рис. 12.12.

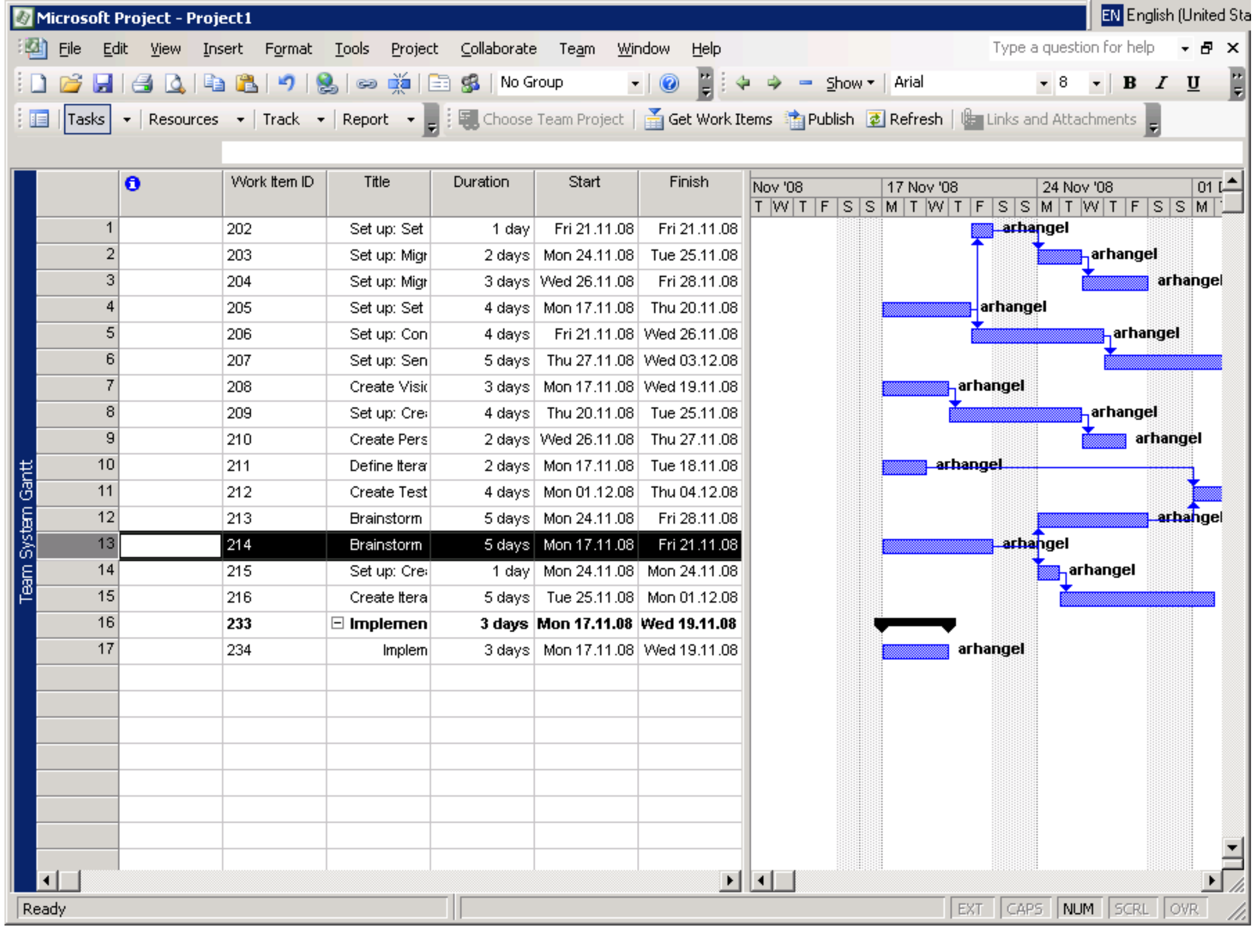

Рис. 12.12. Редактирование элементов работы в Microsoft Project.

Отображение реквизитов элементов работы task на атрибуты задач Project изначально задается в шаблоне процесса разработки. Кроме того, менеджер может получить доступ из Project ко всем остальным атрибутам задачи как к расширенным полям, как показано на рис. 12.13.

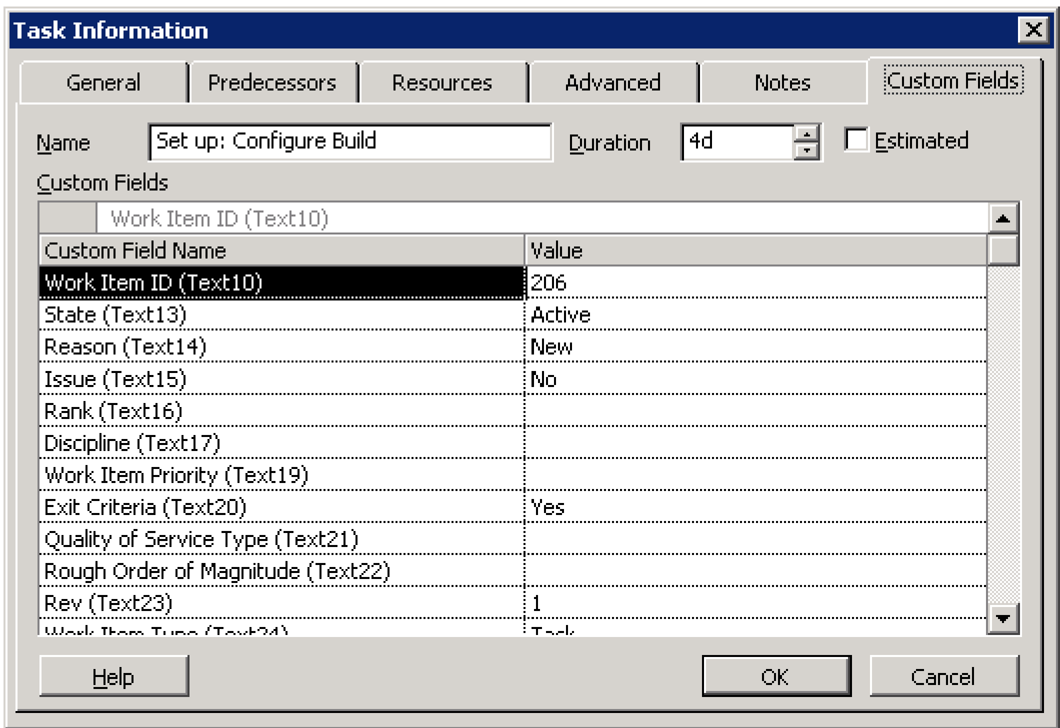

Рис. 12.13. Доступ к реквизитам элемента работы.

После того, как все действия в Project выполнены, для внесения их в VSTS воспользоваться командой Publish, а для получения обновлений – командой Refresh. Кроме того, сам файл с планом можно сохранить на диске или портале SharePoint. При этом информация о связи с сервером TFS так же сохранится.

Элементы работы в дальнейшей разработке. После того, как был построен план и назначены исполнители определенным задачам, ответственные исполнители увидят их в результатах соответствующих запросов (типа My Tasks). И начнут выполнять соответствующую работу. При этом придется вносить некоторые изменения в систему контроля версий, и в этот момент у них появляется возможность указать связанные с данными изменениями элементы работы – ошибки, которые исправляются и задачи, которые выполняются в данном изменении кода и т.д.

Элементы работы в отчетах. Одной из основных задач подсистемы работы с отчетами является отражения реального актуального статуса проекта и анализ его истории. Большинство отчетов в TFS базируются именно на элементах работы и отражают динамику их изменения. В частности, отчет Project Velocity, представленный на рис. 12.14, отражает количество закрытых задач в соответствии с днями (неделями или месяцами) и позволяет судить о том, насколько эффективно двигается проект.

По оси абсцисс на этом рисунке откладывается время, по оси ординат – количество элементов работы (в данном случае – дефектов). Далее мы можем видеть два графика – зеленый (сколько ошибок было закрыто), желтый – сколько было найдено. Серая пунктирная линия обозначает среднюю интенсивность работы в проекте, измеряемую как количество закрытых ошибок. Из рисунка видно, что в проекте были всплески производительности (в начале и в конце), а также спад в середине – в это время разработчик был в отпуске и тестер не тестировал его компоненту.

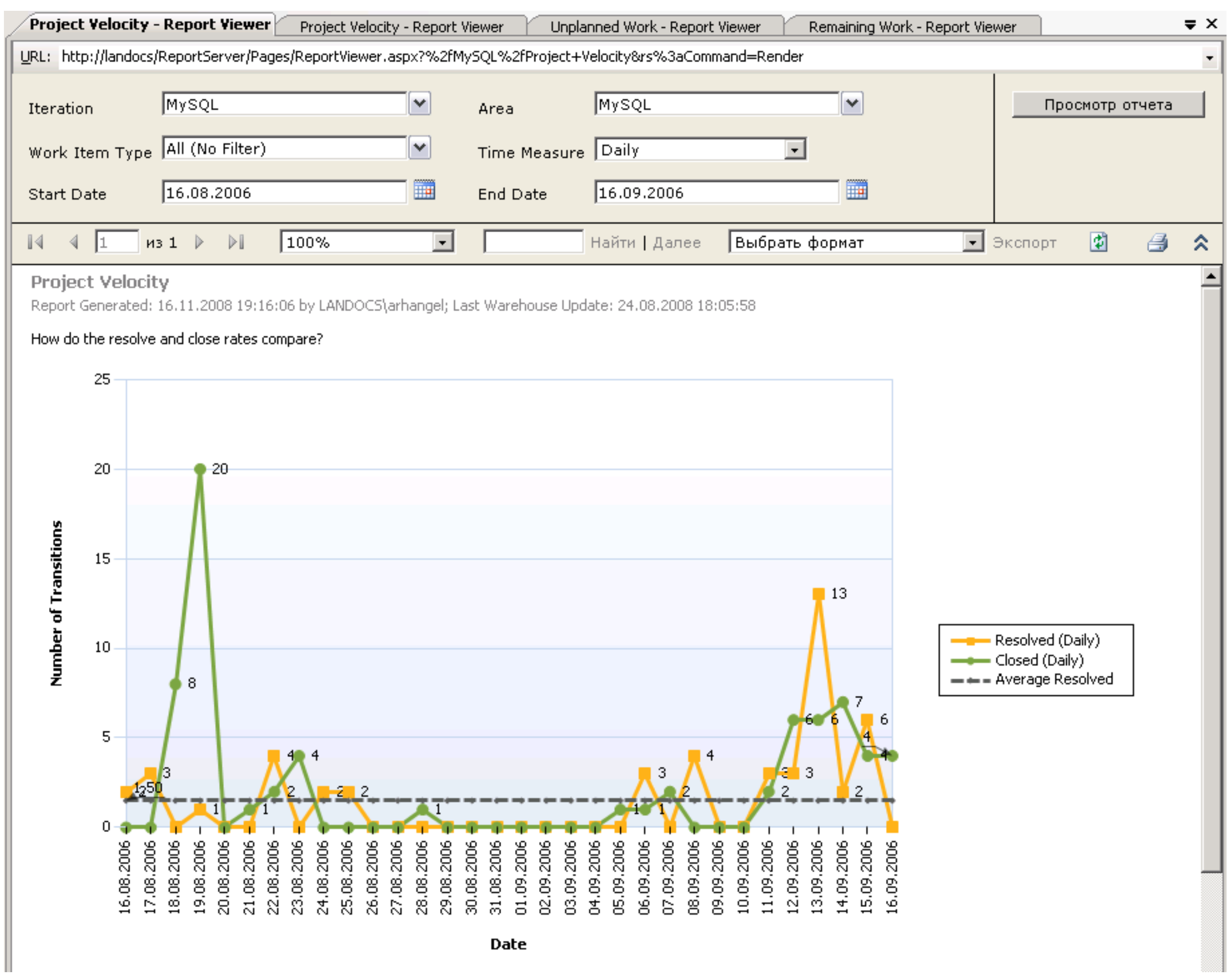

Рис. 12.14. Отчет Project Velocity.

Лекция 14. VSTS: конфигурационное управление

В VSTS есть два типа инструментов для поддержки конфигурационного управления – система контроля версий и система управления сборками. Первая используется для хранения всех основных артефактов, составляющих результат деятельности проектной команды (сюда входят исходные коды приложения, модульные тесты, тестовые пакеты и т.д.). Вторая система позволят автоматизировать получение образа конечного продукта в виде, готовом для тестирования и отправки заказчику. Кроме того, система управления сборок позволяет непрерывно контролировать качество конечного продукта благодаря автоматическому тестированию и статическому анализу кода. По сравнению с другими аналогичными системами в этом аспекте работы у VSTS есть несколько преимуществ:

- интеграция с системой управления элементами работы (а через нее и с системой отчетов) позволяют эффективнее отслеживать процесс разработки и управлять им;
- интеграция с интегрированной средой разработки является стандартом для любой системы контроля версий, однако для систем управления сборок это не так – благодаря удобному пользовательскому интерфейсу интегрированному в единую среду разработки управление сборками осуществляется значительно проще.

Система контроля версий

Функциональность ею предоставляемая в большинстве своем является стандартной, поэтому более подробно мы остановимся на следующих ее особенностях:

- отслеживание изменений отдельных файлов и их «провязка» с элементами работы;
- правила внесения изменений (check-in policies);
- средства управления «ветками»;
- сохранение изменений без внесения.

Отслеживание изменений отдельных файлов. Основным отличительным свойством системы контроля версий в TFS является интеграция его с другими подсистемами TFS, а также более тесная интеграция с Visual Studio, чем во многих других системах контроля версий. Большая часть этих возможностей наглядно демонстрируется самим внешнем вида check-in диалога, представленным на рис. 13.1.

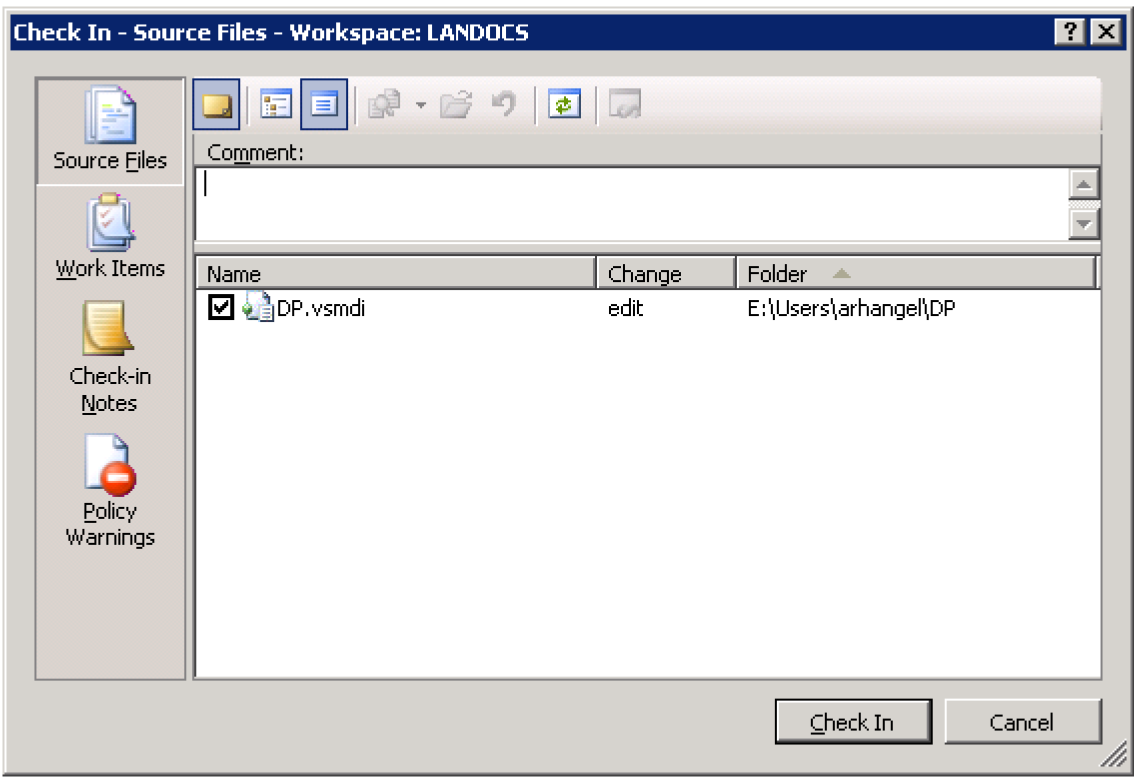

Рис. 13.1. Check-in диалог.

На первый взгляд диалог выгляди достаточно стандартно – список файлов и поля для внесения комментариев к вносимым файлам. Однако в глаза бросается панель с дополнительными закладками в левой части окна. Именно эта панель и позволяет получить доступ к специфической функциональности.

Наиболее востребованной является поддержка в TFS возможности связи вносимых изменений с элементами работы, которую можно выполнить на закладке Work Items, показанной на рис. 13.2.

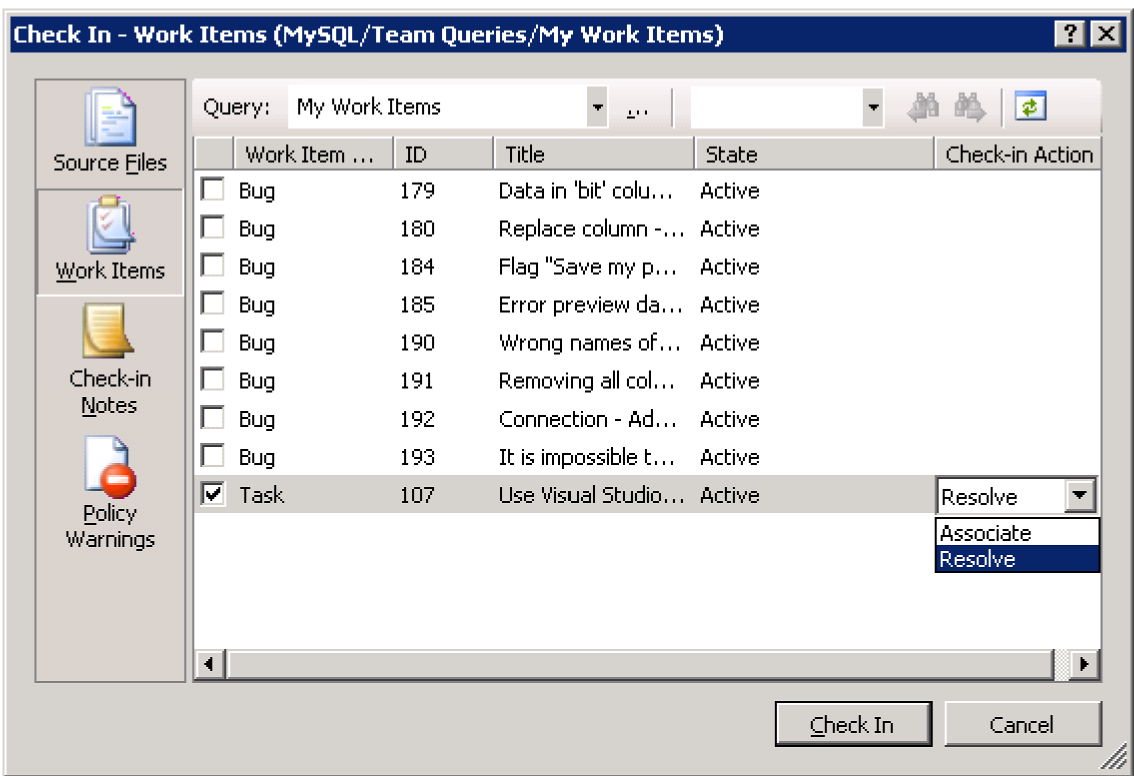

Рис. 13.2. Закладка Work Items.

Разработчик, который вносит изменения в файлы с исходными текстами, может найти соответствующие этим изменениям элементы работы (ведь он либо исправлял какую-нибудь ошибку, либо выполнял задачу и т.п.), используя любой из доступных запросов, а также текстовый поиск. Запрос задается в секции Query – см. рис. 13.2. Результат его выполнения отобразится в главном окне диалога на этом рисунке. Для того, чтобы связать конкретный элемент работы из этого запросы с данным изменением исходников, надо выбрать галочку в первом столбце. На рис. 13.2 она выбрана для последнего элемента в списке – элемента работы Task 107. Далее, бывает так, что это изменение исходников «закрывает» данный элемент работ. Тогда в столбце Check-in Action нужно выбрать действие Resolve – это действие, на равнее с другими, определяется в шаблоне процесса для всех элементов работы данного типа. Если же данное изменение не «закрывает» данный элемент работы, то в этом столбце нужно проставить действие Associate, и оно просто установит связь этого изменения с данным элементом работы.

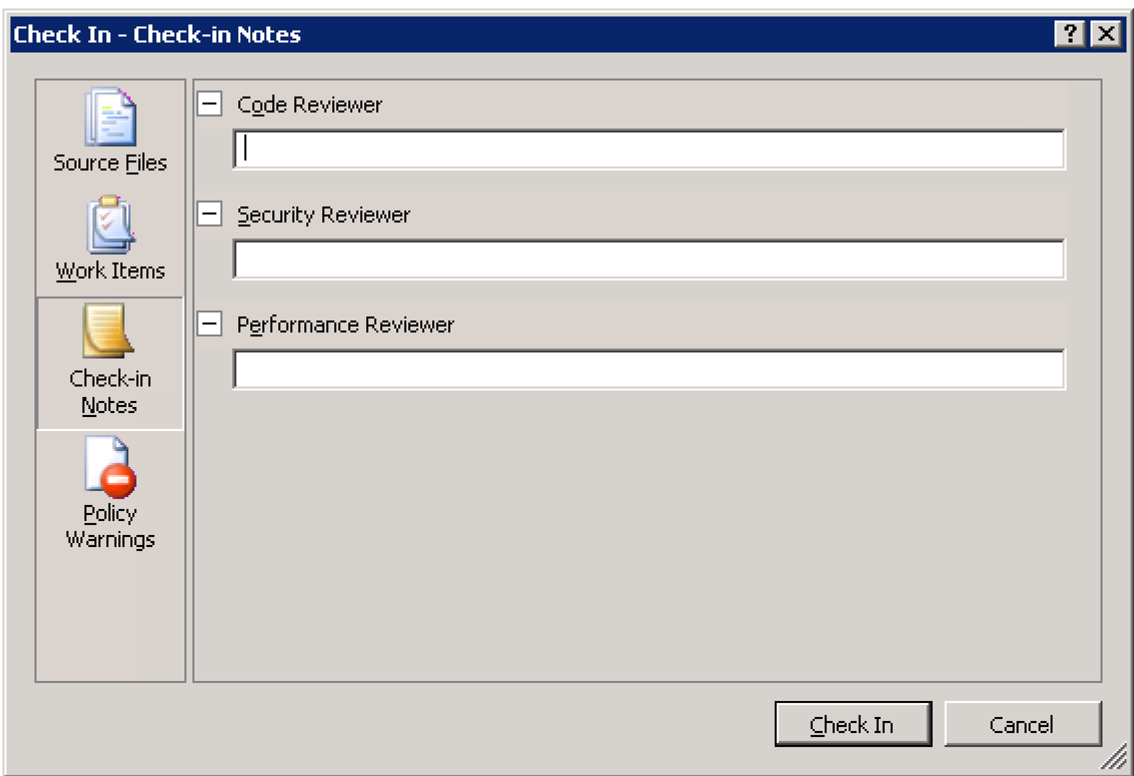

Рис. 13.3. Закладка Check-in Notes.

В момент внесения изменений или после к нему могут быть присоединены дополнительные комментарии людей, проинспектировавших данное изменение (рис. 13.3). По умолчанию TFS предполагает три вида инспекций – инспекцию кода, инспекцию безопасности и инспекцию производительности. Инспекции являются эффективным способом повышения качества кода и предполагают изучение написанного кода другим человеком. К сожалению, реализация поддержки инспекций в этом виде не дает возможность указать, кто именно производил инспекцию, что часто является важной информацией.

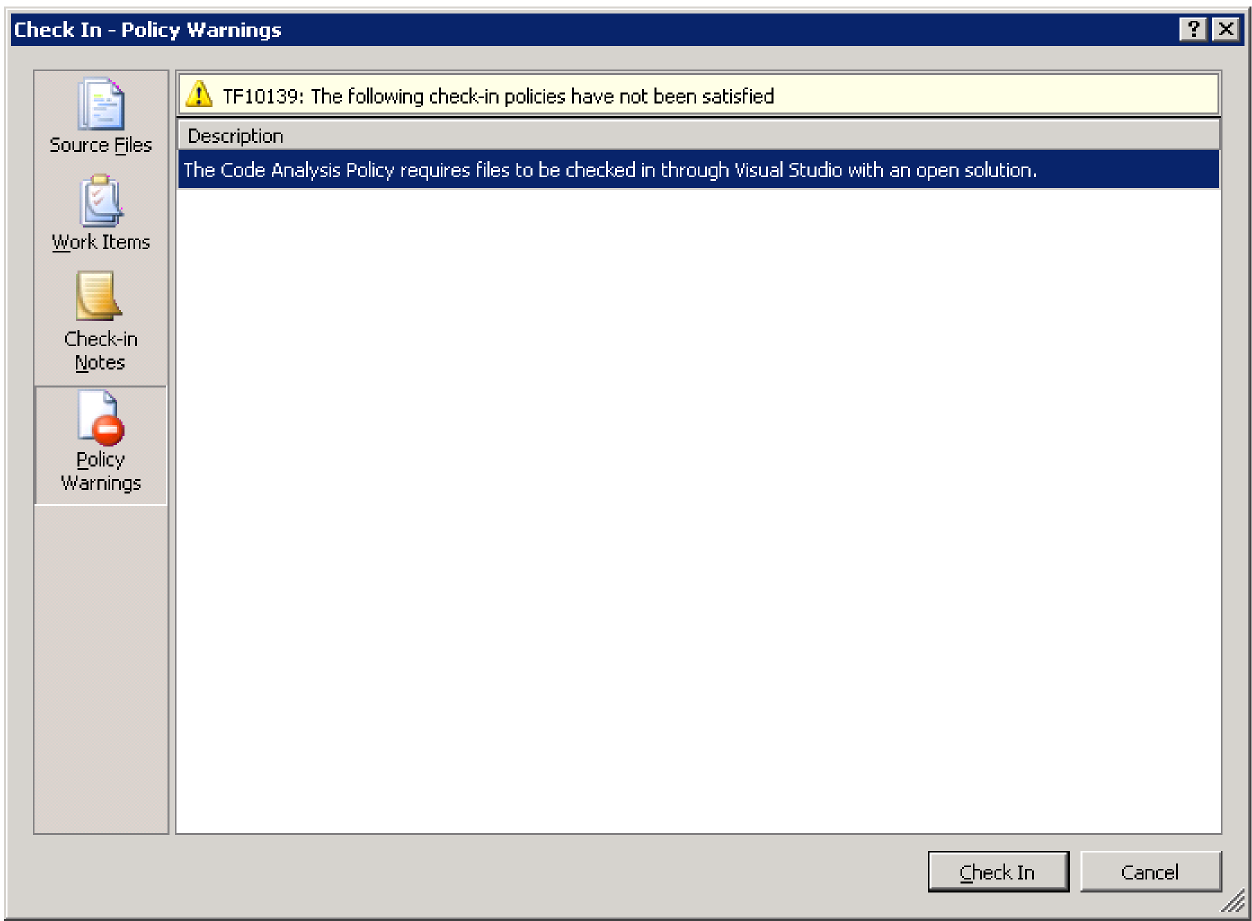

Рис. 13.4. Закладка Policy Warnings.

Интересным нововведение TFS как системы контроля версий является гибкая система задания правил внесения изменений (check-in policies), о которой будет подробнее рассказано позже. На закладке Policy Warnings (рис. 13.4) разработчику показывается список правил, с которыми вошло в конфликт его изменение (или процедура внесения изменений).

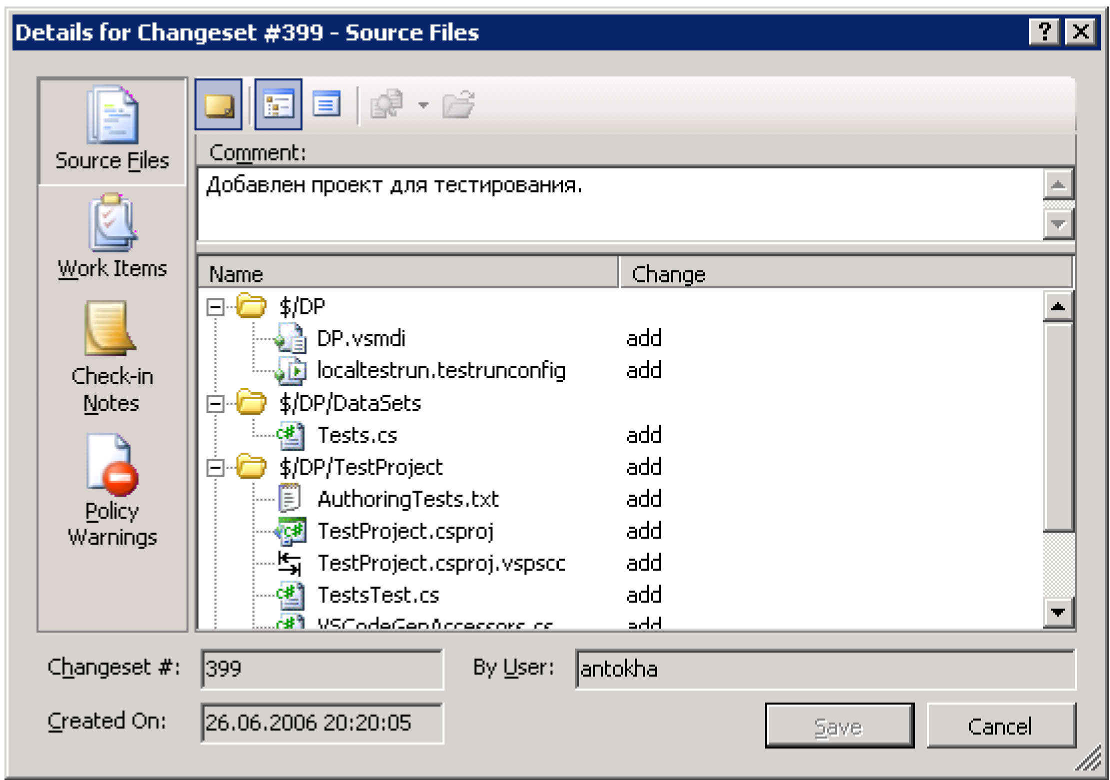

Рис.13.5. Свойства набора изменений.

Большинство свойств пакета изменений можно изменить в дальнейшем в окне редактирования пакета изменений (рис. 13.5). Единственное свойство, которое нельзя изменить из этого окна – ассоциации с элементами работы. Для установки ассоциаций элемента работы и пакета изменений необходимо обратится к редакторы элемента работы.

Правила внесения изменений. Одним из наиболее существенных преимуществ TFS как системы контроля версий является возможность задания ***правил внесения изменений***. Эти правила применяются непосредственно перед внесением изменений на компьютере разработчика и в том случае, если правила не выполняются, разработчику отказывается во внесении изменений.

Правила задаются с помощью специального вида .NET сборок, реализующих определенные интерфейсы. Несколько правил поставляется вместе с самим TFS, огромное количество правил реализовано сообществом разработчиков и находится в открытом доступе. Если же найти идеально подходящее правило так и не удалось, в Интернет можно найти огромное количество информации о написании собственных правил.

В стандартную поставку TFS входят следующие правила:

- Work Items – предполагает, что каждый пакет изменений кроме файлов должен иметь ассоциацию с элементом работы;
- Builds – проверяет, что перед внесением изменений разработчик убедился в собираемости проекта;
- Testing – проверяет, что перед внесением изменений разработчиком были исполнены тесты. К сожалению, правило не может определить какие именно тесты надо запускать для проверки данного изменения, поэтому данное правило не всегда эффективно;
- Code Analysis – выполняет статический анализ кода перед внесением изменений.

В пакете Team Foundation Power Tools имеются следующие дополнительные правила:

- Правило Forbidden Patterns позволяет запретить добавление файлов с определёнными шаблонами в именах, например, Form1.cs.
- Правило Custom Path позволяет увеличить гранулярность применения правил в проекте сконфигурировав правила только для определённых частей структуры папок системы контроля версий.
- Правило Changeset Comments проверяет, что изменения сопровождаются комментариями.
- Правило Work Item Query проверяет, что все элементы работы, возвращаемые в результате определённого запроса, ассоциированы с файлами (более продвинутый вариант правила Work Items).

Среди правил, доступных в открытом доступе можно выделить:

- Правило Code Comment Checking проверяет код на наличие комментариев перед внесением изменений (работает для кода на C#/VB.NET).
- Правило Code Review Workflow проверяет, что каждый пакет изменений ассоциирован с элементом работы, являющимся заданием на инспекцию кода в состоянии «Выполнено». Это правило позволяет проводить процесс инспекции вносимых изменений в более формальном виде.
- Правило Merge/Branch Only удостоверяет, что все изменения происходят в результате объединения с другой веткой либо ответвления. Представляет особый интерес с точки зрения процесса конфигурационного управления. Позволяет, например, запретить вносить изменения напрямую в одну из ветвей (например, ветвь релиза). Вносить изменения в ветвь, защищенную таким правилом, можно только через интеграцию изменений из других ветвей.

Выбрать список правил, применяемых для командного проекта можно с помощью окна настройки системы контроля версий (рис. 13.6).

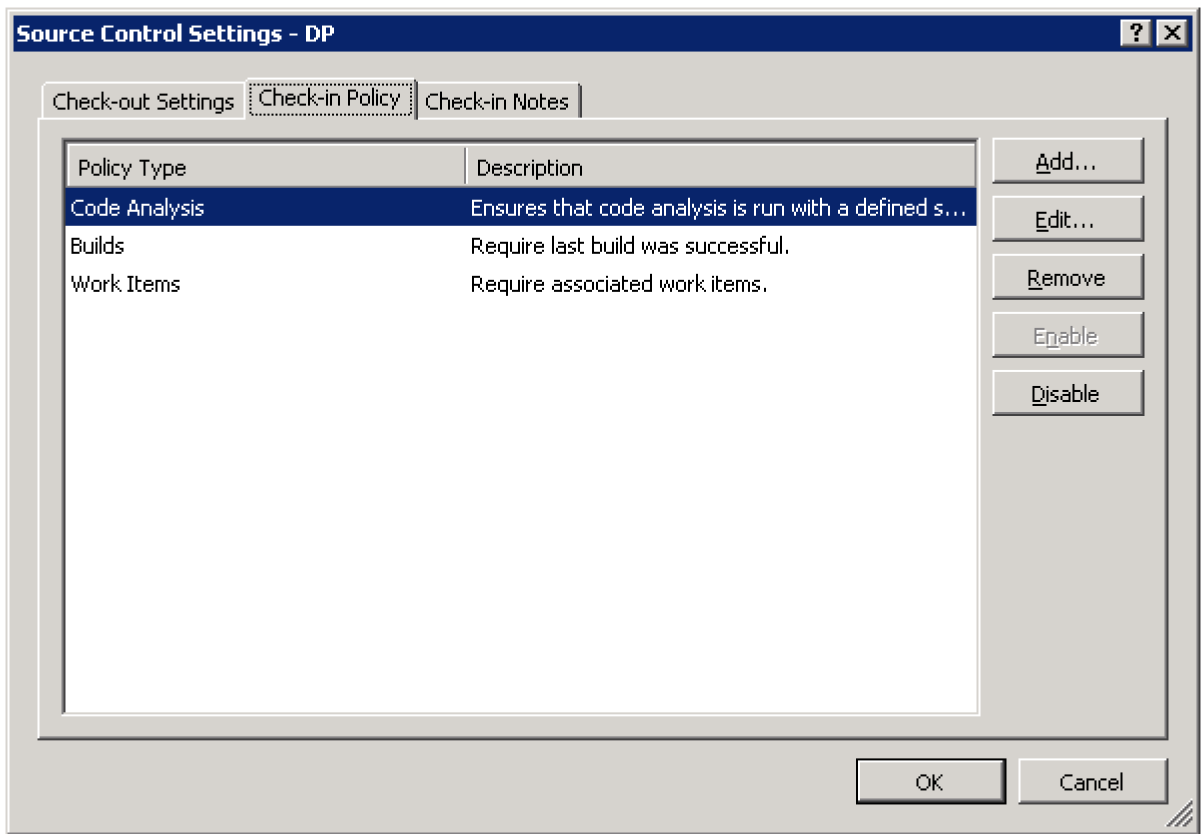

Рис. 13.6. Редактирование правил вноса изменений.

Следует заметить, что достаточно часто при разработке случаются ситуации, когда изменение необходимо срочно внести и на удовлетворение всех правил времени нету, либо конкретное правило не может быть выполнено по объективным причинам. В этом случае разработчик имеет право отменить правила для своего пакета изменений, написав при этом комментарий с объяснением причин отмены (рис. 13.7).

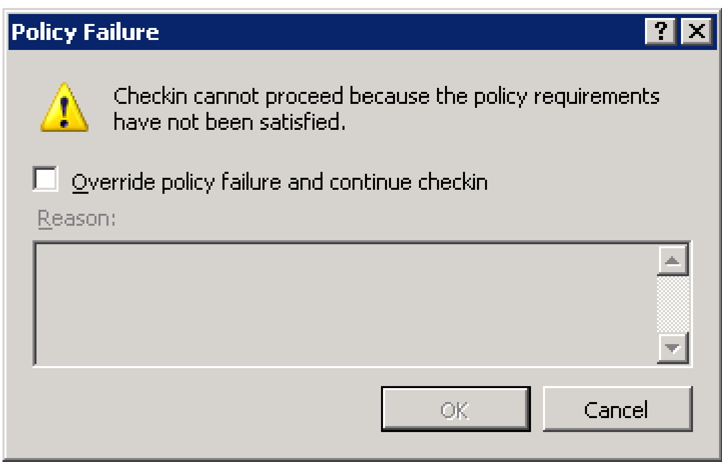

Рис. 13.7. Диалог Policy Failure.

Управление ветками. Для поддержки конфигурационного управления в системе контроля версий TFS реализовано две команды: создание ветви (Branch) и интеграция ветвей (Merge). Эти команды доступны на файлах и папках в системе контроля версий. При выборе команды создания ветви открывается диалог (рис. 13.8), позволяющий выбрать путь, куда следует скопировать (ответвить) выбранные файлы. После выполнение этой команды в системе контроля версий по указанному пути создастся полная копия выбранных файлов.

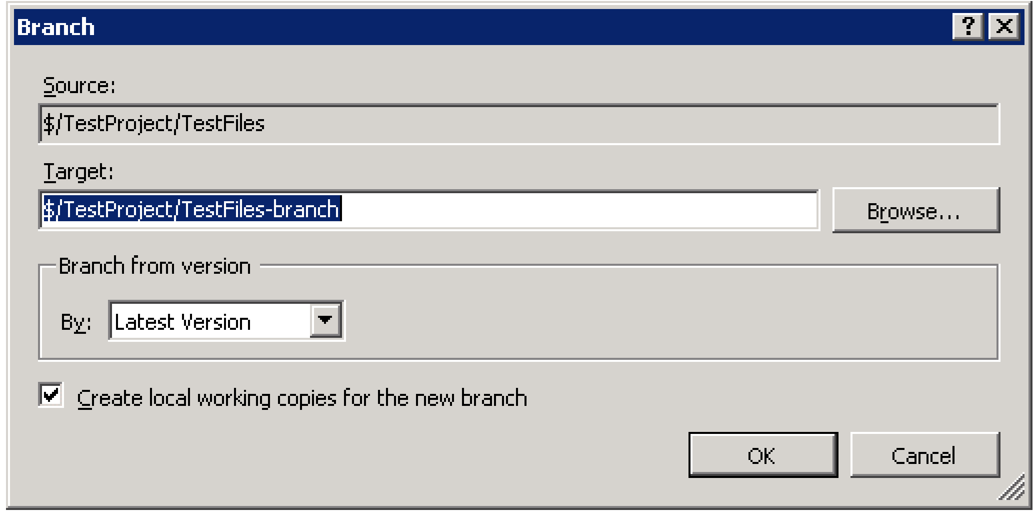

Рис. 13.8. Создание новой ветви.

Заметим, что после создания ветви она не попадает автоматически на сервер. Чтобы ветвь попала на сервер и стала доступна всем, необходимо выполнить операцию внесения изменений (рис. 13.9).

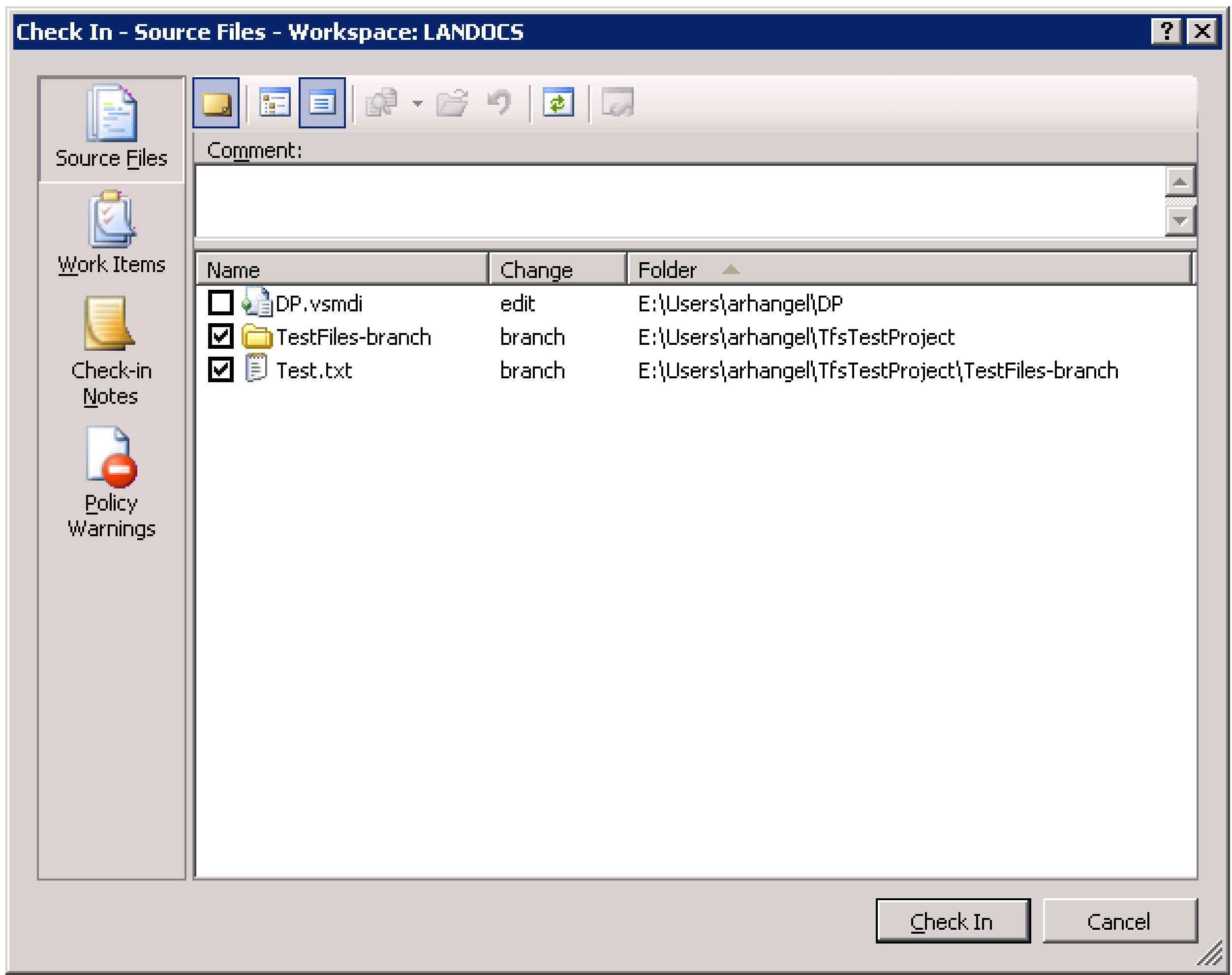

Рис. 13.9. Внесение изменений ответвления.

Гораздо более сложной, как правило, является операция переноса изменений из ветви в ветвь. Для выполнение этой операции (команда Merge) используется специальный мастер, позволяющий разработчику задать необходимые параметры слияния за несколько шагов.

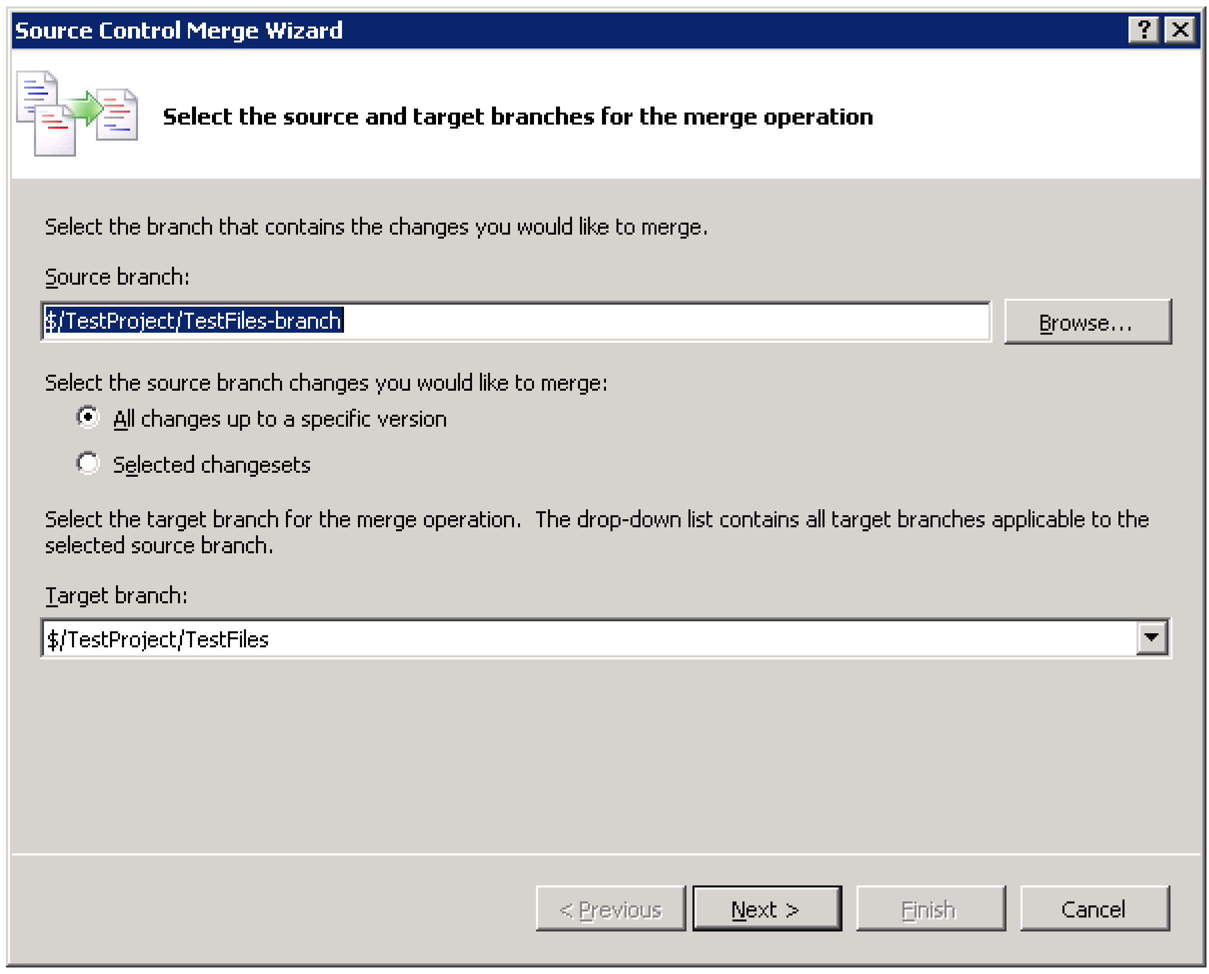

Рис. 13.10. Область интеграции.

На первом шаге (рис. 13.10) разработчик задает откуда (source branch) и куда (target branch), а также изменения из какой области он хочет перенести (все вплоть до определенной версии, или только выбранные пакеты изменений).

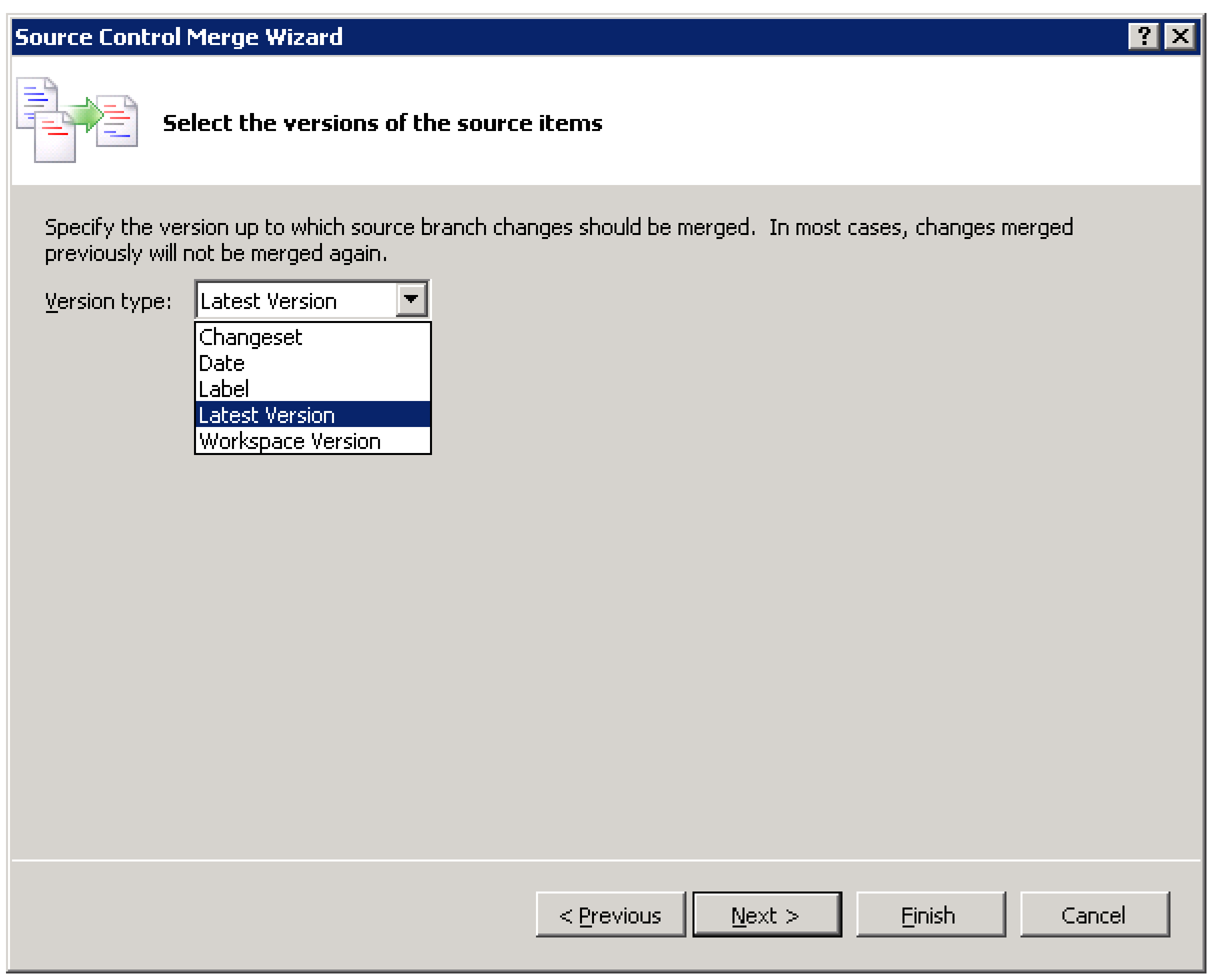

Рис. 13.11. Версия для интеграции.

В случае, если разработчик выбрал перенос всех изменений, ему предлагается выбрать версию исходной ветви вплоть до которой изменения нужно перенести (рис. 13.11). Разработчик может выбрать полный перенос, перенос до определенной даты, все предшествующие определенному пакету изменения, либо интеграцию до тех версий, которые находятся в текущем локальном рабочем пространстве.

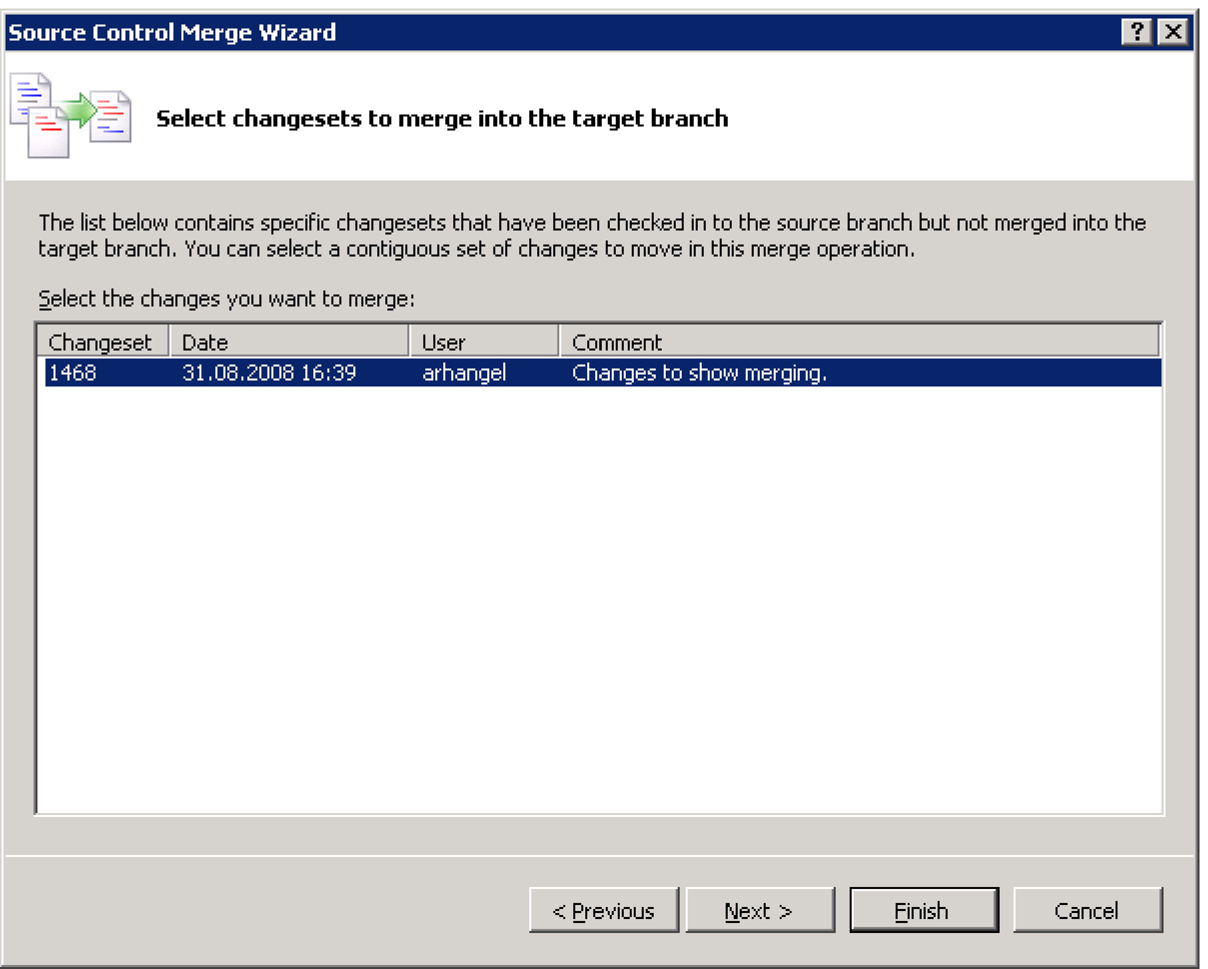

Рис. 13.12. Выбор пакетов для интеграции.

В случае, если разработчик выбрал интеграцию по пакетам изменений, ему предоставляется выбор среди не интегрированных пакетов (рис. 13.12), и он может выбрать один или несколько из них.

Так же как и при создании новых ветвей, при выполнении интеграции необходимо в явном виде внести пакет изменений, применив к нему все настроенные правила и ассоциировав с соответствующим элементом работы.

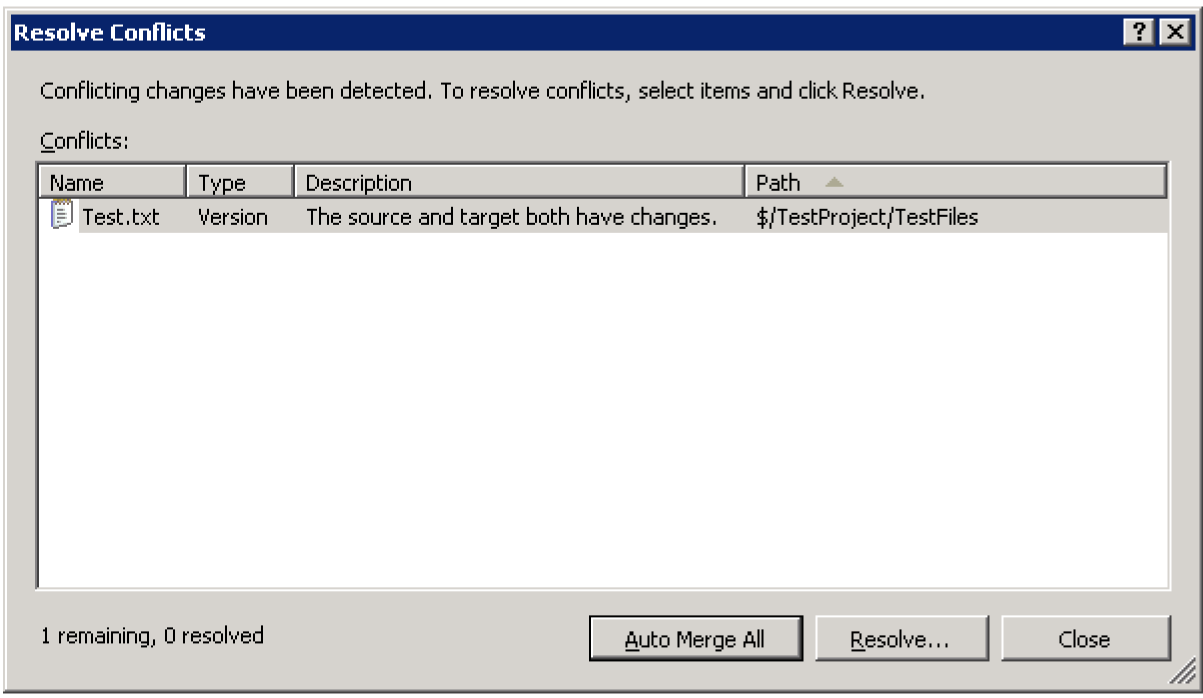

Рис. 13.13. Список конфликтов.

Достаточно часто при интеграции может возникнуть ситуация конфликта изменений, когда интегрируемый файл поменялся в обоих ветвях независимо друг от друга. В этом случае система контроля версий открывает окно со списком обнаруженных конфликтов (рис. 13.13) и предлагает выбрать способ разрешения.

Разработчик может выбрать автоматический способ разрешения, который сработает только для тех файлов, в которых были изменены разные части. В случае, если автоматическое разрешение невозможно, система откроет диалог ручного разрешения – см. рис. 13.14.

Resolve Version Conflict - Test.txt
E:\Users\arhangel\TfsTestProject\TestFiles\Test.txt could not be merged because the source and target both have changes.
Conflict details
Path: E:\Users\arhangel\TfsTestProject\TestFiles\Test.txt
Changes: 0 source, 0 target, 0 both, 1 conflicting
Resolution options
Merge changes for me
Merge changes in merge tool
Keep changes in target branch
Copy item from source branch
Compare...
OK
Cancel

Рис 13.14. Разрешение конфликта.

Для разрешения конкретного конфликта разработчик может выбрать несколько способов: автоматически объединить изменения (если возможно), принять изменения из исходной ветки, сохранить изменения целевой ветки или вручную разрешить все внутренние конфликты в специальном инструменте рис. 13.15.

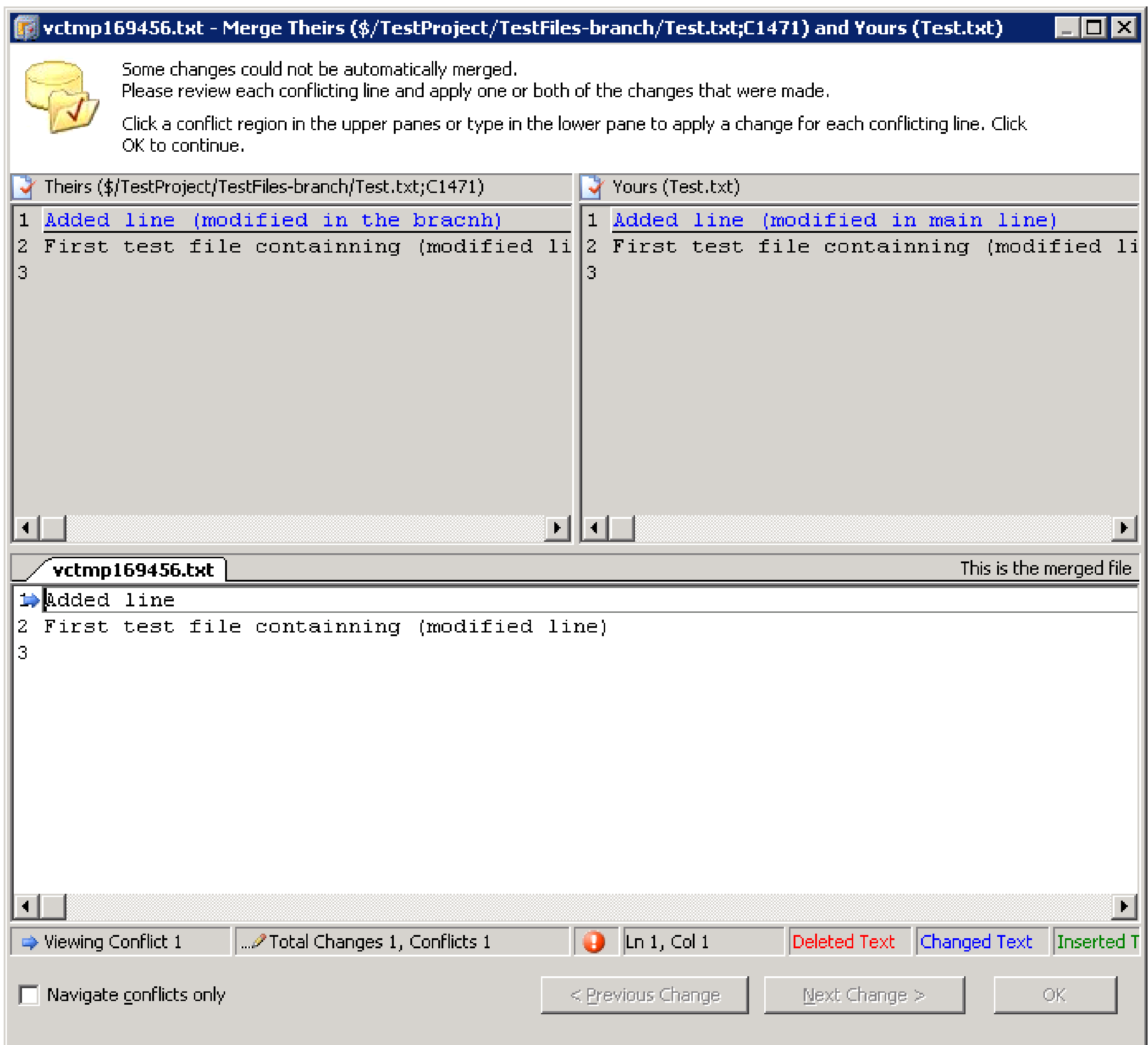

Рис. 13.15. Разрешение внутренних конфликтов.

Следует оговорится, что данный инструмент имеет достаточно ограниченную функциональность, поэтому некоторые разработчики предпочитаются использовать аналогичные инструменты других производителей.

Сохранение без внесения. Полезной возможностью системы контроля версий является возможность сохранить изменения в специальном хранилище на сервере, не внося их непосредственно в систему контроля версий. Для временно сохраненного кода не проверяются правила внесения изменений (если об этом не попросить явно) и он не доступен другим разработчиком (если они об этом явно не попросят).

Эта функциональность позволяет защитить важный пакет изменения от потери, если разработчик вынужден временно приостановить работу (например, чтобы поспать). Находясь на сервер эти изменения подпадают под политику создания резервных копий

базы данных и, следовательно, вероятность потери этих изменений становится минимальной.

Еще одним способом применение данной возможности является перенос изменений между разными машинами, в том случае, если разработчик использует несколько машин (например, рабочую или домашнюю). Сохраненные на сервере изменения разработчик сможет получить на другой машине в том же виде, чтобы продолжить работу, где бы он ни находился.

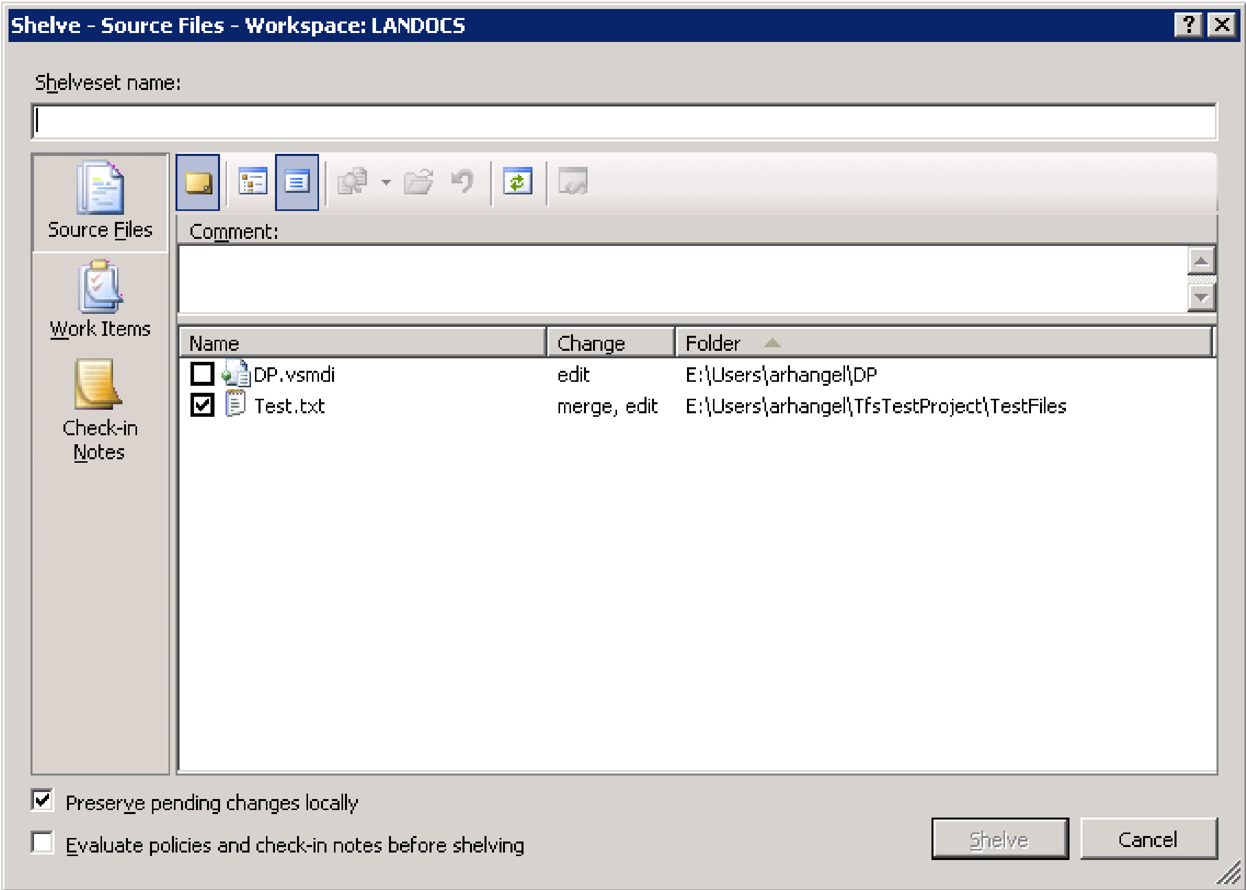

Рис. 13.16. Сохранение без внесения.

Для сохранения изменений служит команда Shelve, открывающая диалог, представленный на рис. 13.16, аналогичный диалогу внесения изменений за исключением поля для введения имени сохраненного пакета в верхней части и двух следующих дополнительных опций в нижней части.

- Сохранить локальные изменения. Если эта опция включена, то изменения останутся локально в рабочем пространстве пользователя (рекомендуется при временной остановки работы). Если же нет, то локальные изменения откатываются и остаются только в виде сохраненного на сервер пакета (рекомендуется при смене рабочей машины).
- Применить правила перед сохранением – позволяет проанализировать пакет, применив все те правила, которые действуют при внесении.

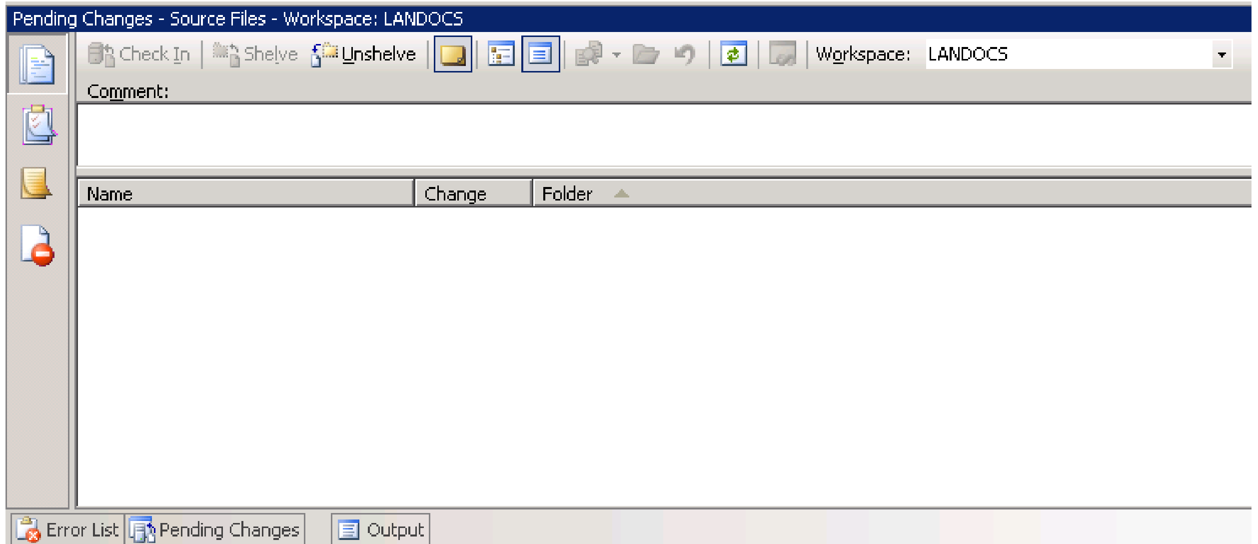

Рис. 13.17. Список текущих изменений.

Для того, чтобы восстановить сохраненные изменения необходимо воспользоваться командой Unshelve, доступной для файлов и папок, а также в глобальном контексте (из окна со списком невнесенных изменений – см. рис 13.17). Эта команда открывает диалог восстановления изменений (**Ошибка! Источник ссылки не найден.** 13.18), позволяющий выбрать один из сохраненных пакетов для восстановления. Из этого же диалога пакеты можно удалить, если они потеряют актуальность.

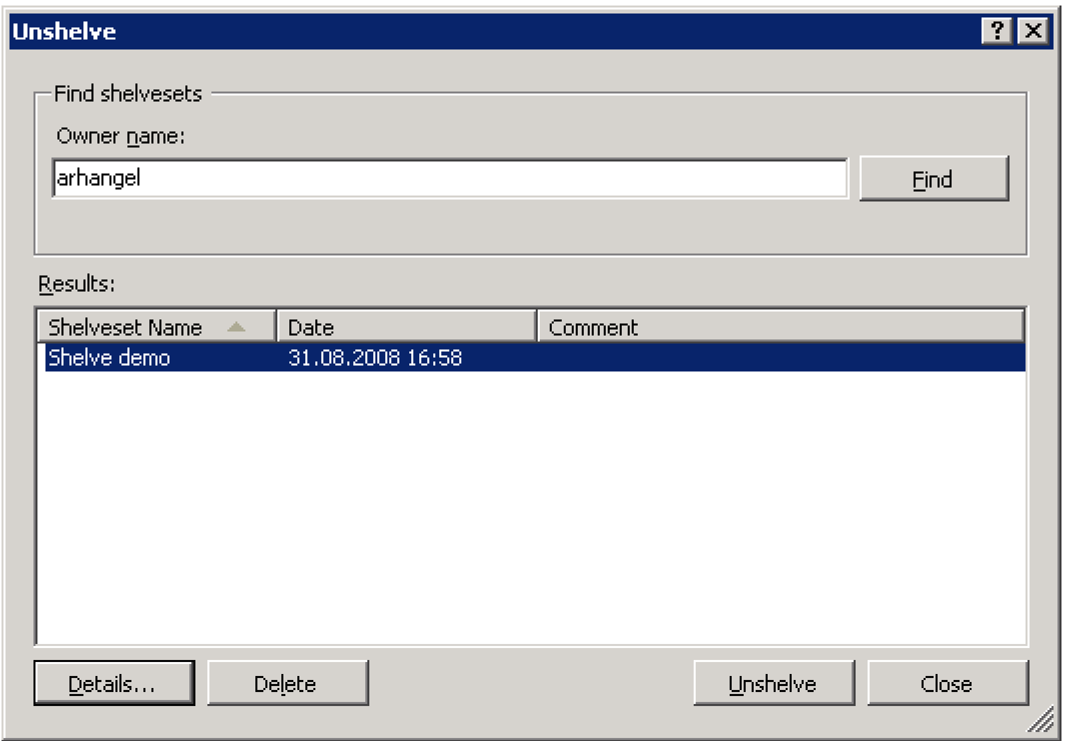

Рис. 13.18. Диалог восстановления изменений.

Автоматические сборки

Общее. Одним из существенных преимуществ TFS по сравнению с другими системами управления сборками является простота, с которой он позволяет создавать и настраивать процесс автоматической сборки. Не смотря на то, что в основе сборок TFS

лежит давно и широко известная технология MsBuild, именно TFS позволяет вывести её на принципиально новый уровень благодаря следующим улучшениям.

- TFS поставляется вместе с набором MsBuild-задач, позволяющих значительно упростить и ускорить настройку процесса сборки. Среди наиболее важных задач следует отметить следующие:
    - сборка проекта (при этом исходные тексты программ автоматически берутся из системы контроля версий),
    - автоматический запуск тестовых пакетов (как созданных в ручную, так и идентифицированных автоматически),
    - применение статического анализ кода,
    - размещение результатов сборки в сетевой папке,
    - автоматическое поддержание уникально идентификатора сборки и его регистрация,
    - выявление присоединенных элементов работы и т.д.
- TFS предоставляет серверную среду, позволяющую запускать процесс сборки в «чистом» окружении, в отличии от обычных MsBuild-сборок, где организация соответствующей инфраструктуры требует значительных усилий.
- Возможность автоматического запуска процесса сборки как в режиме непрерывной интеграции, так и по расписанию.
- Визуальное представление хода процесса сборки, результатов, а также истории более ранее отработавших сборок.

Очевидным преимуществом TFS здесь является то, что описание простого процесса сборки создается меньше чем за минуту, а описание более сложных создаются не сложнее, чем с помощью стандартного MsBuild.

Создание описания сборки. Для создания описания новой сборки проекта необходимо выбрать команду New Build Definition в соответствующем узле Team Explorer, и после этого откроется окно с несколькими закладками (рис. 13.19).

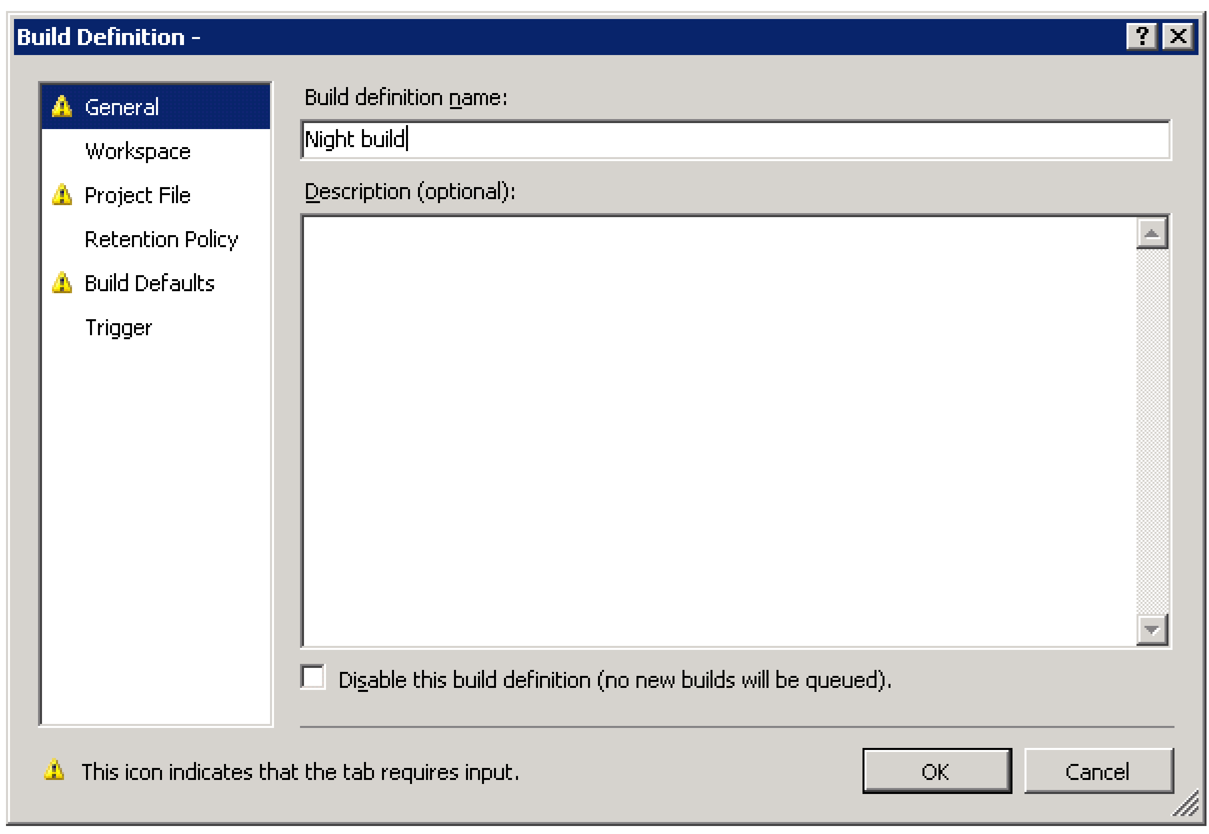

Рис. 13.19. Общие настройки описания сборки.

На первой закладке (General) находится общая информация – название и описание назначения этого сценария сборки.

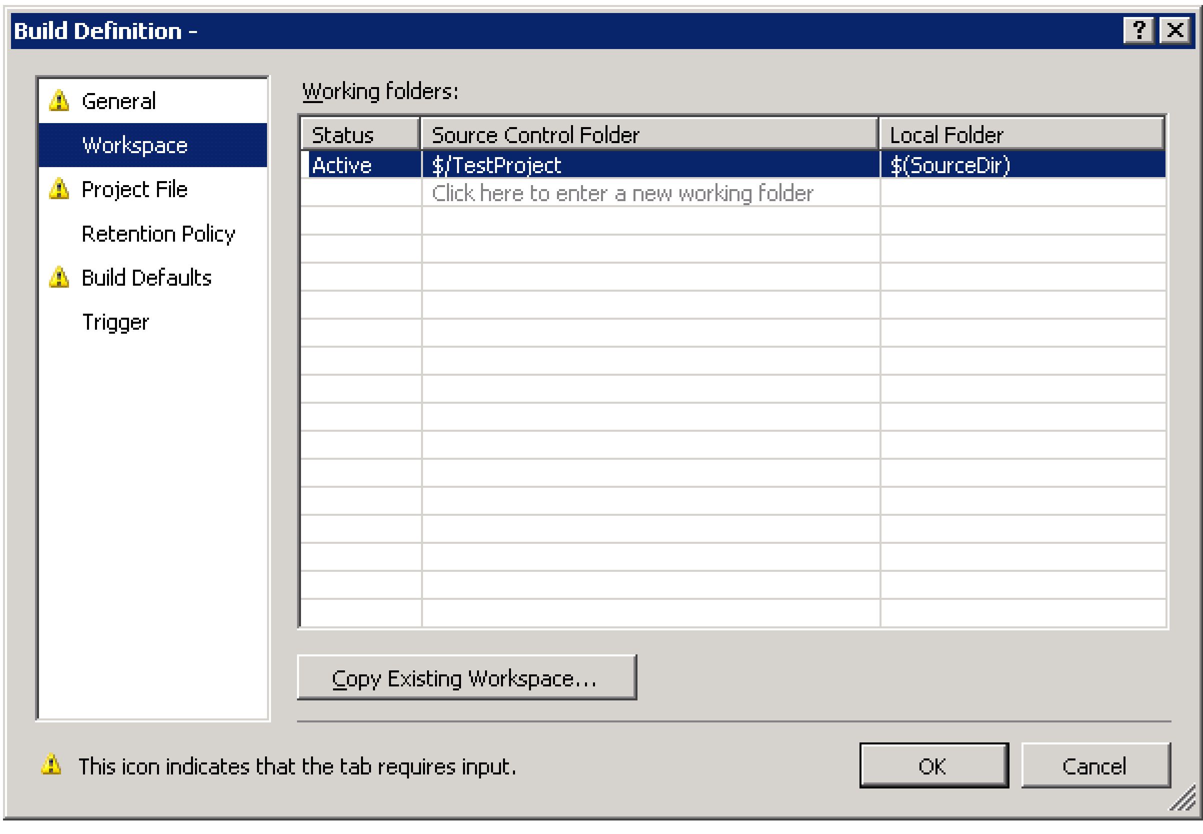

Рис. 13.20. Настройка рабочего пространства.

На второй закладке (Workspace) – см. рис 13.20 – описывается то, какие исходные тексты необходимо взять из системы контроля версий для сборки, а также то, как эти коды разместить на машине, где будет сборка происходить. Кроме того, эти настройки влияют на поведение при непрерывной интеграции – внесение изменений именно в выбранные области в средстве контроля версий будет являться сигналом для запуска данного процесса сборки.

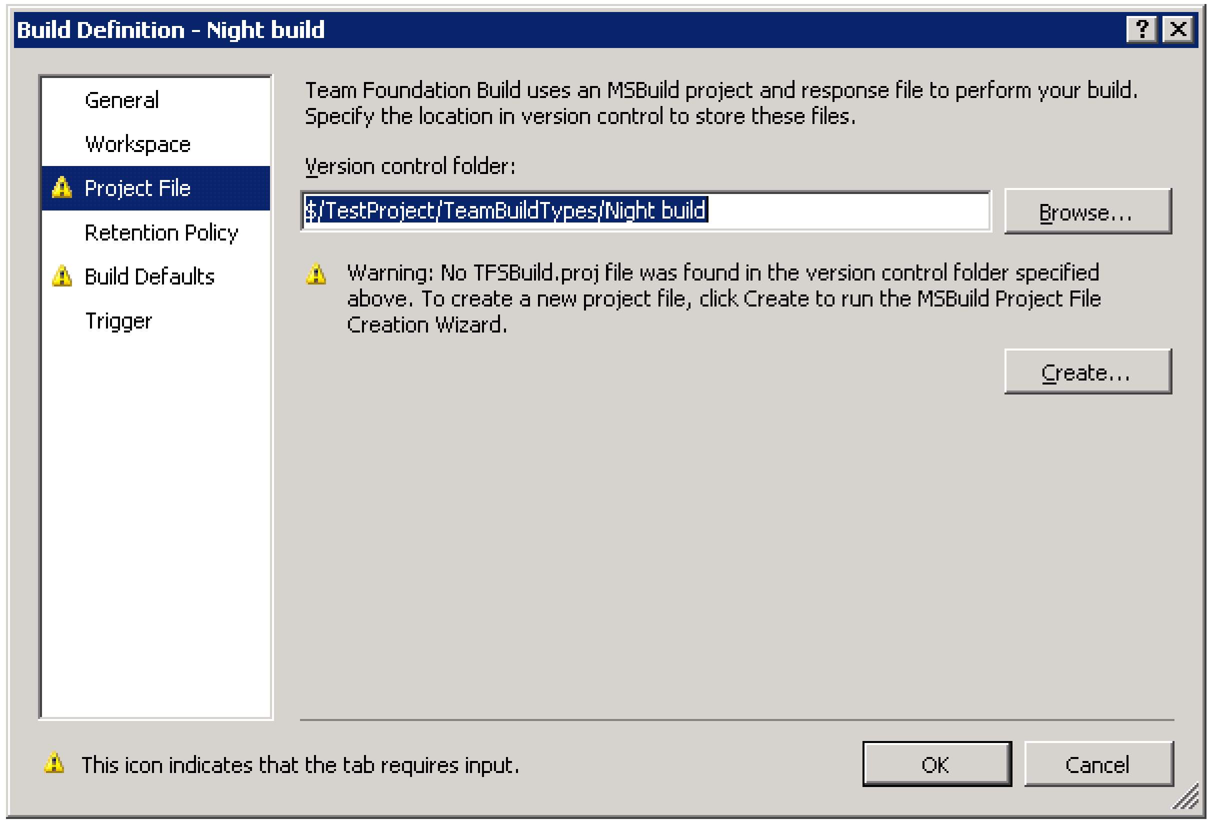

Рис. 13.21. Выбор или создание MsBuild-проекта.

Наиболее интересной является третья закладка Project File (рис. 13.21), где разработчик может выбрать один из существующих MsBuild-сценариев сборки или создать новый.

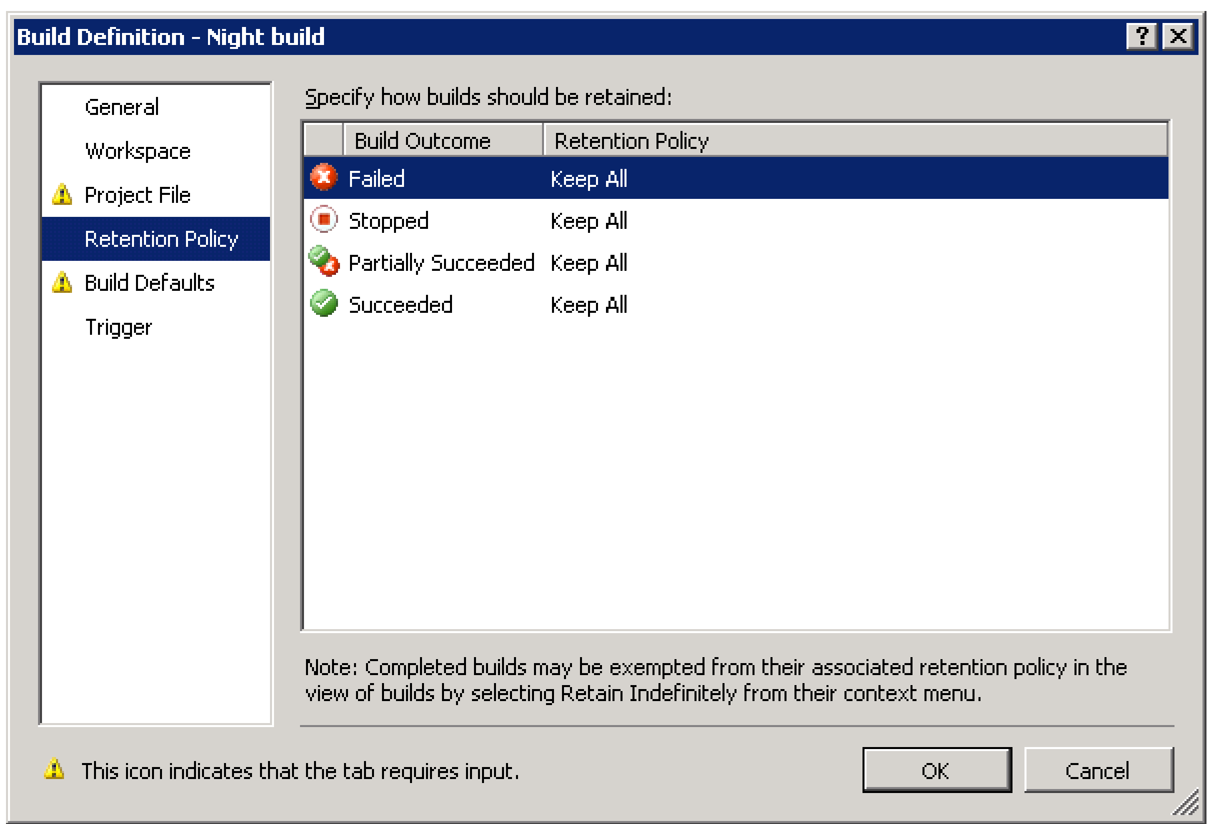

Рис. 13.22. Правила сохранения.

В TFS 2008, в связи с реализацией функциональности по непрерывной интеграции, возникла проблема большого числа описаний сборок и результатов, сохраненных в системе и на диске. Для устранения проблемы был реализован механизм автоматической очистки, получившей название Retention Policy (см. рис 13.22), позволяющий задать, сколько последних результатов нужно хранить. При этом для каждого из типов результат (неудачный, остановленный, частично успешный, успешный) можно задать свое число.

Полезной функцией является то, что политику уничтожения результатов можно отключить для конкретной сборки используя опцию Retain Indefinitely. Результаты, помеченные этой опцией, сохраняются в системе навсегда (или пока опция не будет снята).

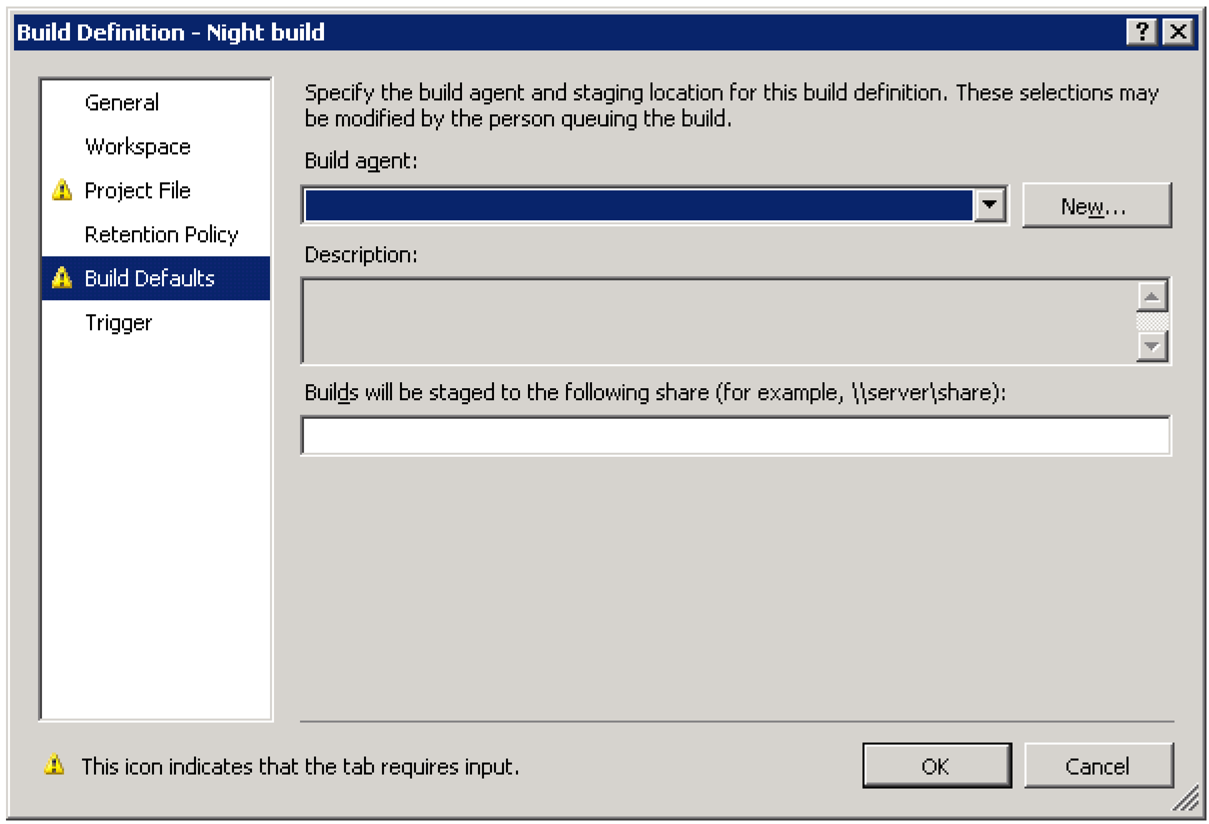

Рис. 13.23. Выбор агента.

На закладке Build Defaults (см. рис 13.23) мы задаем свойства окружения, которое будет использовано для автоматического запуска процесса сборки. Главным здесь является *сборочный агент* – процесс, в рамках которого будет выполняться процесс сборки.

Кроме того, именно на этой закладке можно задать сетевую разделяемую папку, в которой будут храниться результаты процесса сборки. При выборе папки нужно убедиться, что сборочный агент имеет к ней доступ на запись.

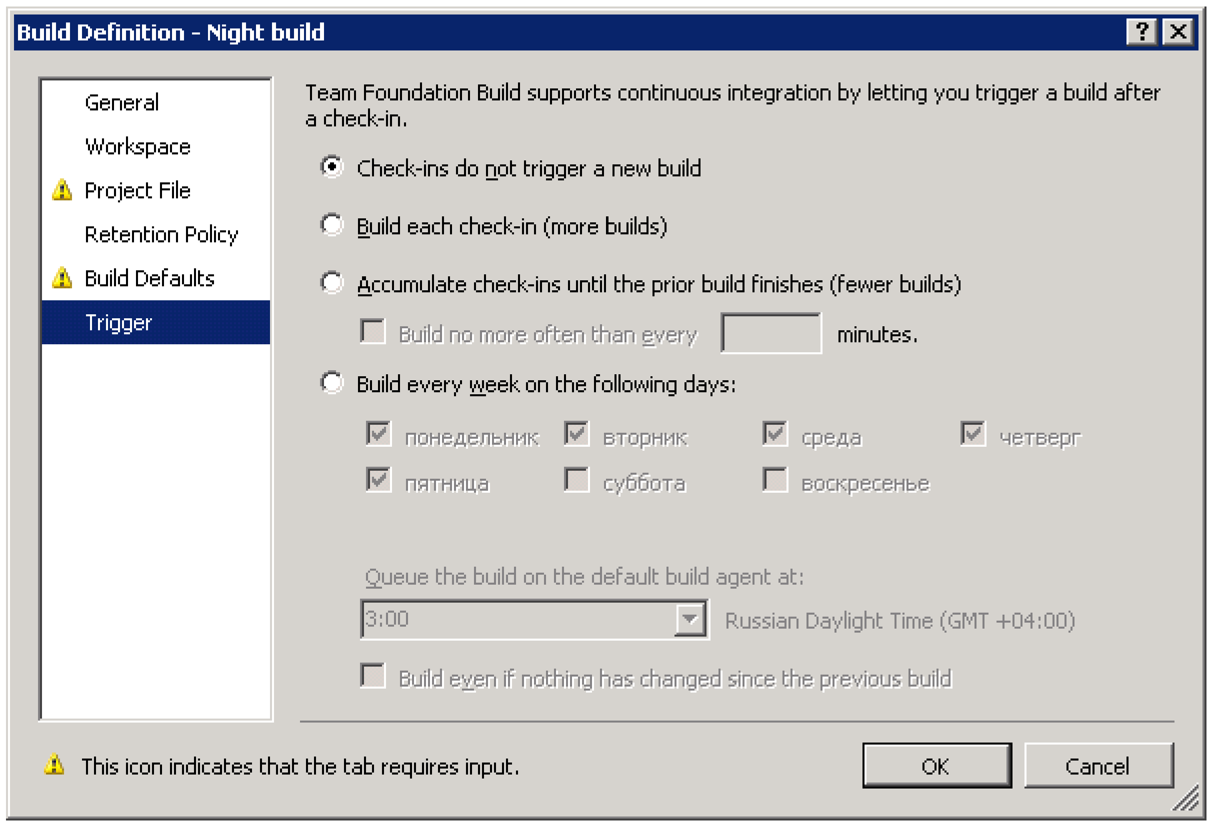

Рис. 13.24. Задание триггера.

На заключительном этапе настройки процесса сборки (см. рис. 13.24) мы задаем условие, при выполнении которого процесс сборки, определенный данным описанием, должен выполнятся. Это условие может быть одним из следующих:

- не запускать процесс сборки автоматически (то есть только «вручную»),
- запускать после каждого внесения изменений,
- аккумулировать изменения, пока не закончится предыдущая сборка и не истечет определенный интервал времени,
- запускать процесс сборки только по расписанию, даже в том случае, если ничего не менялось.

Создание проекта MsBuild. Проект MsBuild, не смотря ни на что, все-таки составляет основу системы автоматических сборок TFS. Однако специалистами Майкрософт потрачено немало усилий на то, чтобы мы могли забыть о необходимости поддерживать большие и сложные XML-файлы с описанием сборок. В TFS для создания проектов MsBuild реализован достаточно удобный мастер (рис. 13.25 – 13.27).

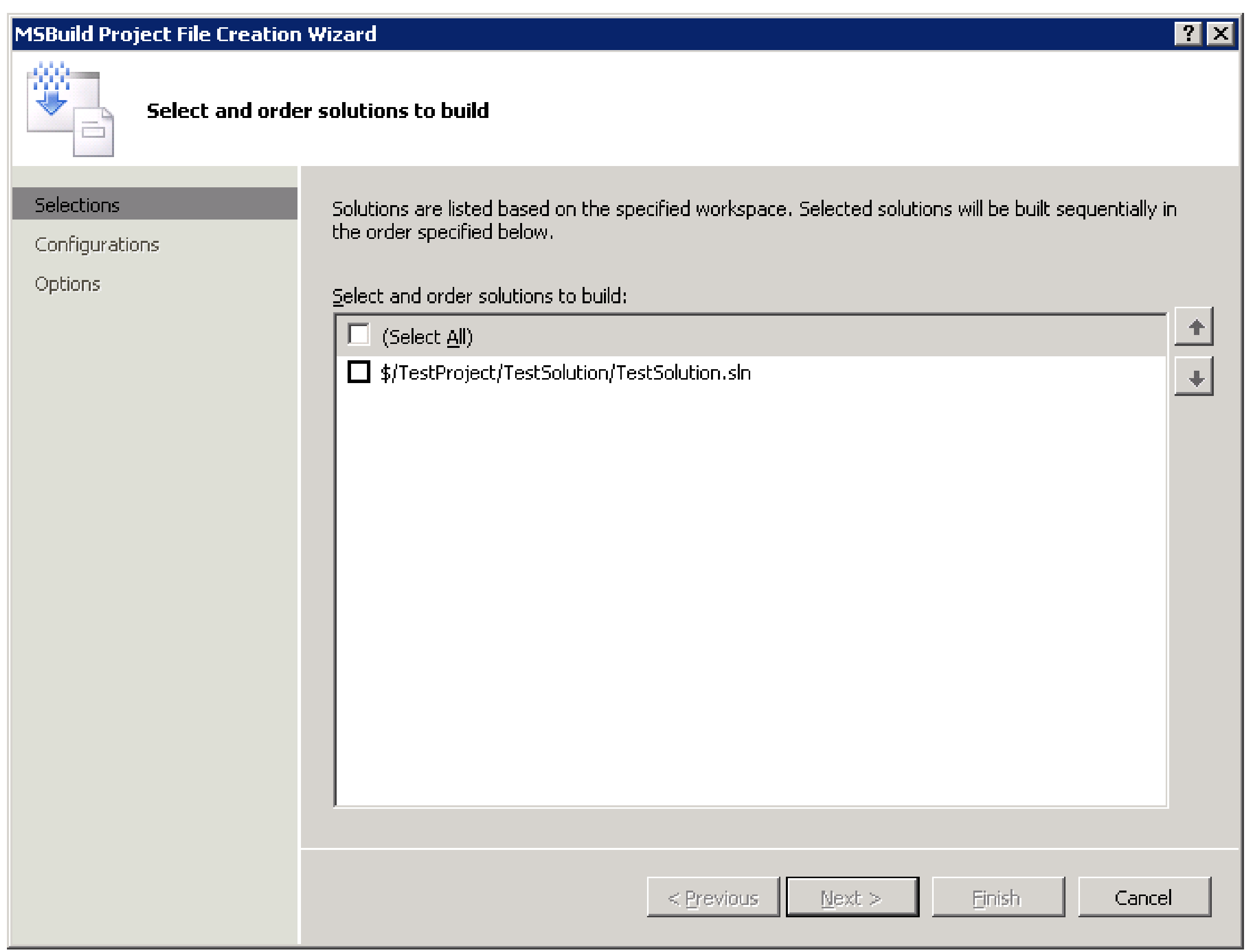

Рис. 13.25. Выбор решений.

На первом шаге (рис. 13.25) мы выбираем в системе контроля версий те решения, которые должны собираться в данной сборке.

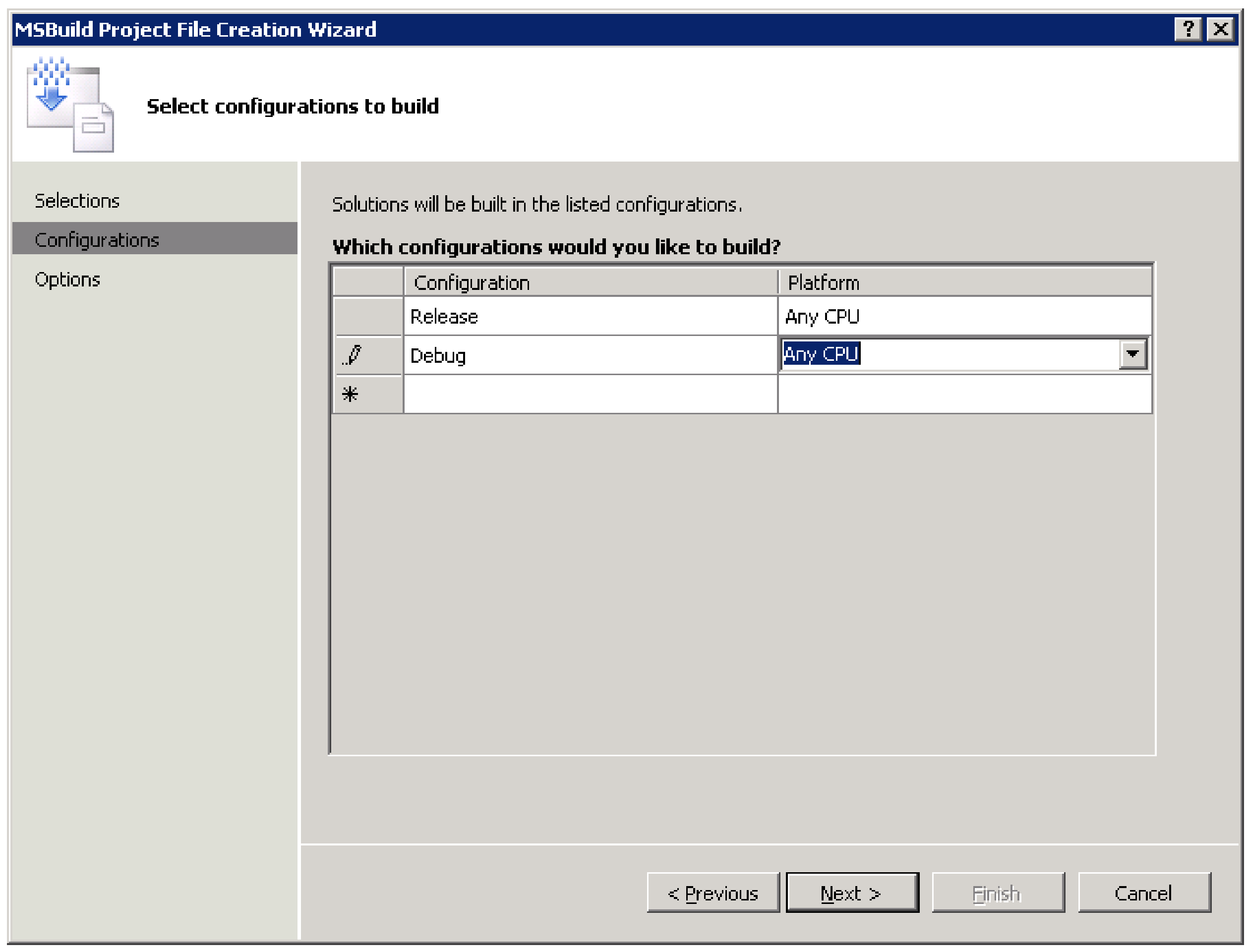

Рис. 13.26. Выбор конфигураций.

На втором шаге (см. рис 13.26) мы выбираем те конфигурации в этих решениях, которые нужно собирать.

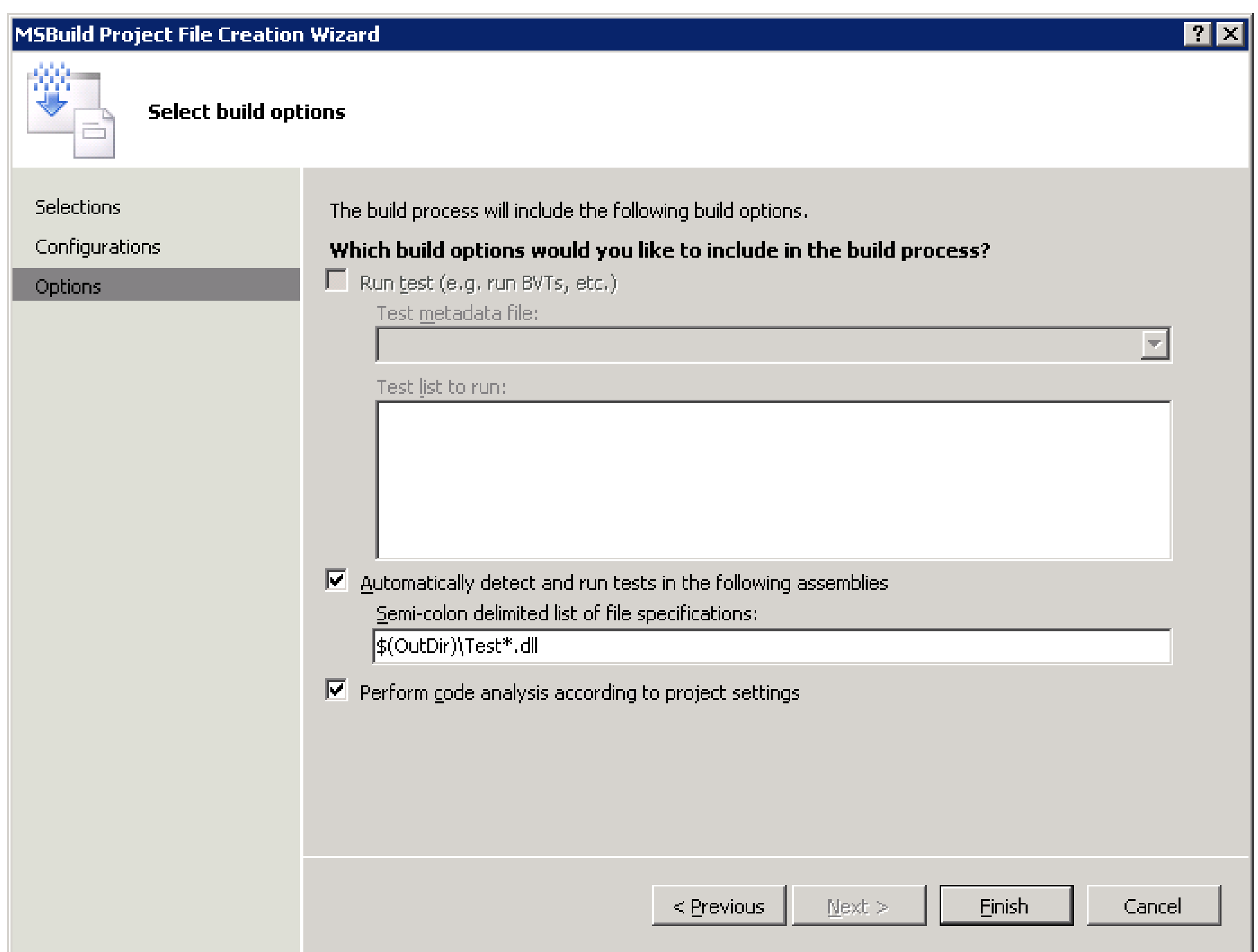

Рис. 13.27. Выбор тестов и правил анализа.

И, наконец, на третьем шаге (рис.13.27) мы выбираем те тесты, которые мы хотим запустить и отмечаем, хотим ли мы проводить статический анализ кода. В отличии от TFS 2005, где при выборе тестов можно было использовать только заранее подготовленные тестовые пакеты, в TFS 2008 появилась такая востребованная возможность как автоматическое подключение пакетов тестов по метке имени (обычно сборки начинающиеся или заканчивающиеся на Test). Эта возможность позволяет без дополнительных затрат включить выполнение модульных тестов в автоматическую сборку.

Создание сборочного агента. Появление сборочных агентов в TFS 2008 позволило значительно расширить возможности конфигурации автоматического выполнения процесса сборки. Сборочный агент – это процесс, запущенный на некоторой выделенной машине, в рамках которого и происходит автоматическая сборка. На самом деле, сборочные агенты выполняются процессом TFSBuild, запущенным в виде сервера или консольного приложения[15].

Одна машина может размещать у себя несколько процессов TFSBuild, каждый из которых доступен как Web-сервис на определенном порту. При этом каждый процесс TFSBuild может содержать несколько сборочных агентов, которые отличаются именами и рабочими папками, в которых они выполняют сборку.

---

[15] Последнее важно если в сборке используются автоматические тесты на пользовательский интерфейс – сервис не может его выполнить. Для сборок с такими тестами необходим интерактивный процесс.

Несмотря на сложность описанного процесса, настройка его достаточно проста и производится, в основном, с помощью окна свойств агента (см. 13.28).

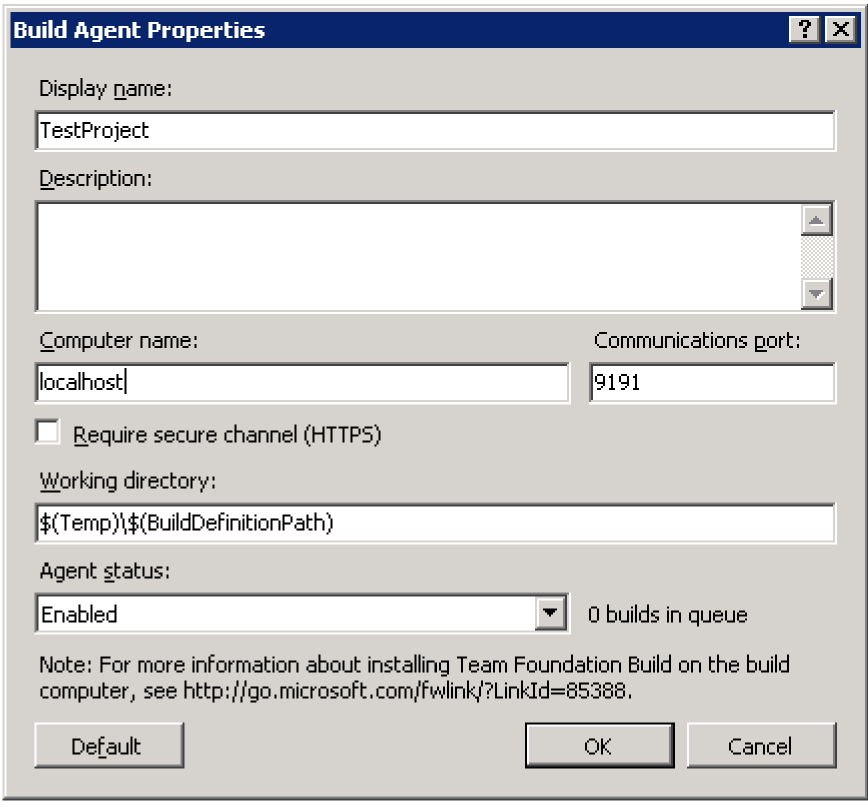

Рис. 13.28. Свойства сборочного агента.

Запуск процесса сборки и анализ результатов. Итак, после того как описание сборки создано и инфраструктура выполнения настроена, мы можем запустить процесс сборки с помощью команды Queue New Build, вызывающей окно, показанное на рис. 13.29.

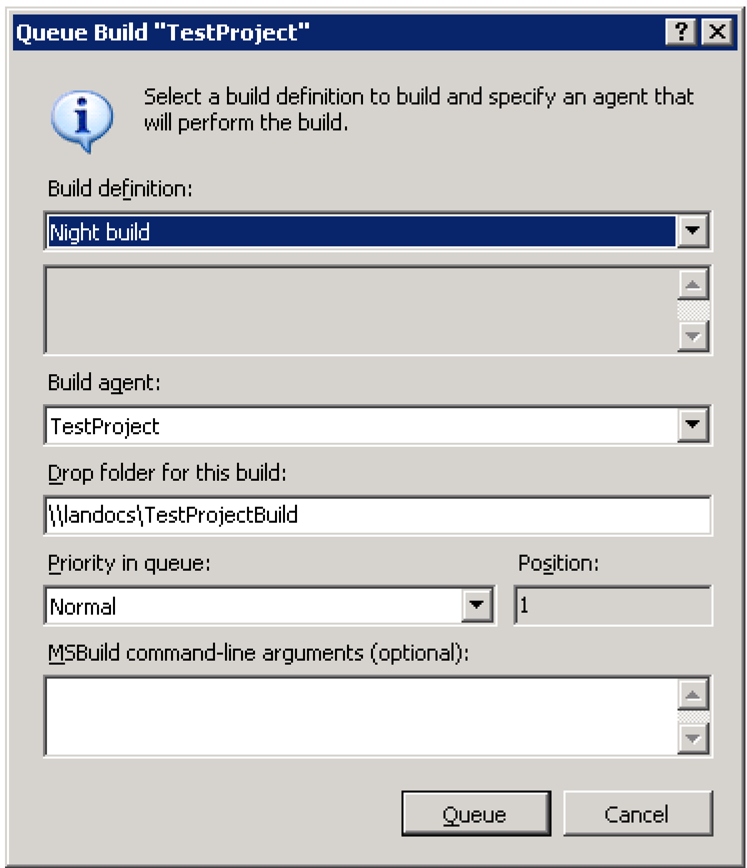

Рис. 13.29. Запуск новой сборки.

При запуске новой сборки пользователь может выбрать описание сборки, сборочного агента (по умолчанию используется агент, заданный в описании), папку для хранение результатов (так же берется по умолчанию из описания), приоритет и позицию в очереди (каждый сборочный агент может выполнять не более одной сборки за раз), а также дополнительные параметры командной строки для MsBuild. Как правило, в приведенном окне можно использовать все настройки по умолчанию, менять которые приходится только в особых случаях.

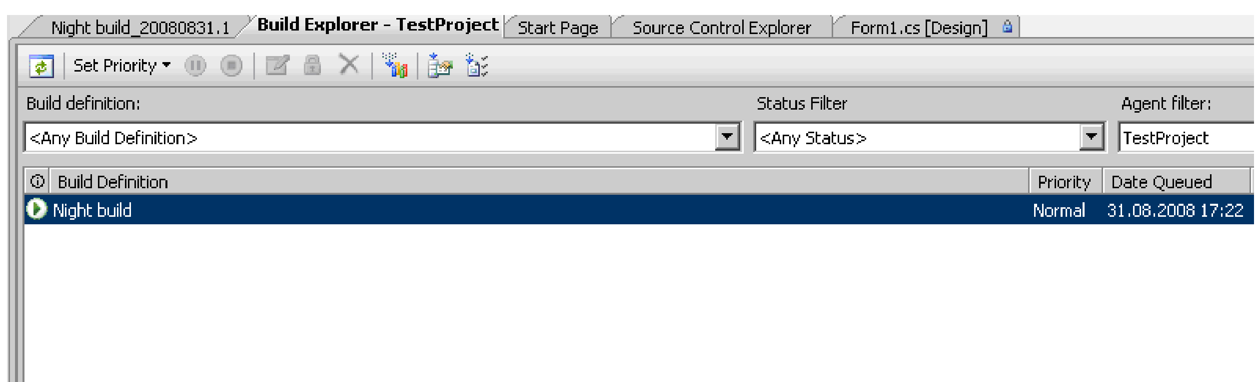

Рис. 13.30. Список описаний сборок.

После того, как сборка помещена в очередь, она отображается в списке сборок (рис. 13.30), откуда можно перейти к окну с детальной информацией о сборке (рис. 13.31). В этом окне отображается ход сборки, или её результаты, если она завершена.

Пример на рис. 13.31 наглядно демонстрирует средства Team Explorer по визуализации результатов. В открытом окне виден список не прошедших тестов, а также имеются ссылки на все файлы с логами, которые можно открыть одним щелчком в окружении студии (при условии, что у вас есть доступ к сетевой папке, куда они были скопированы). Кроме того, в информации о прошедшем процессе сборке можно увидеть, какие изменения исходных текстов программ в нее попали, а также то, какими элементами работы они были обусловлены.

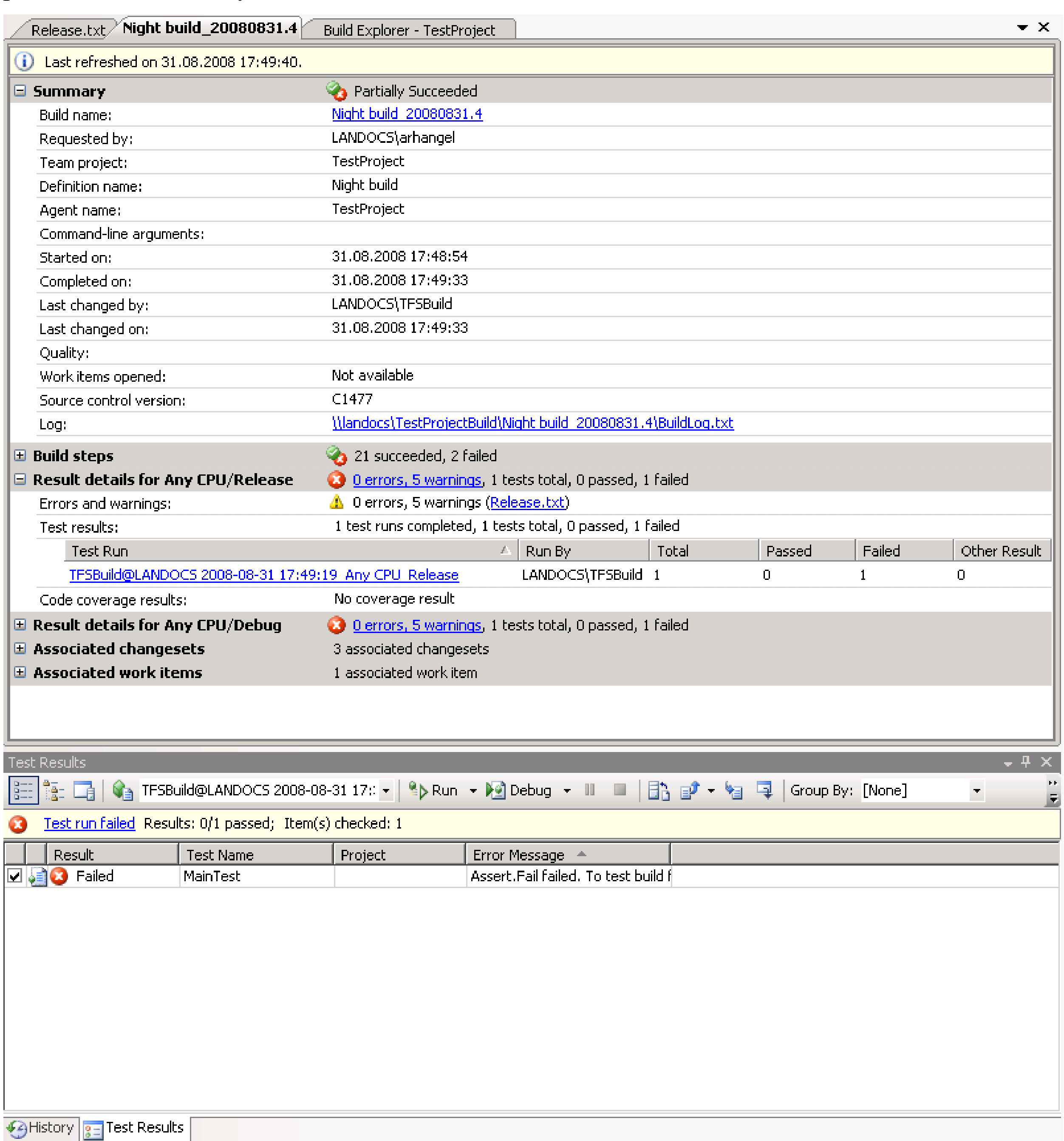

Рис. 13.31. Результаты сборки.

Управление процессом сборки. Все продемонстрированные нами средства визуальной описания сборок доступны не только при создании такого описания, но и для любого из существующих описаний сборок (команда Edit Build Definition). Единственным

исключением является файл MsBuild, который, будучи однажды сгенерирован с помощью мастера, далее поддерживается вручную.

Этот файл можно найти в системе контроля версий (рис. 13.32) в той папке, которая была выбрана при его создании. Эта папка содержит два файла:

- PROJ-файл – основной файл, описывающий задачи MsBuild. Этот файл содержит достаточное количество сгенерированных комментариев, позволяющих производить простую настройку (добавить или удалить решение, включить или отключить статический анализ и т.д.) достаточно легко, однако для более сложных изменений необходимо знакомство как с принципами работы MsBuild, так и со спецификой MsBuild-задач, используемых в TFS.
- RSP-файл, который содержит параметры командной строки для передачи при запуске MsBuild.

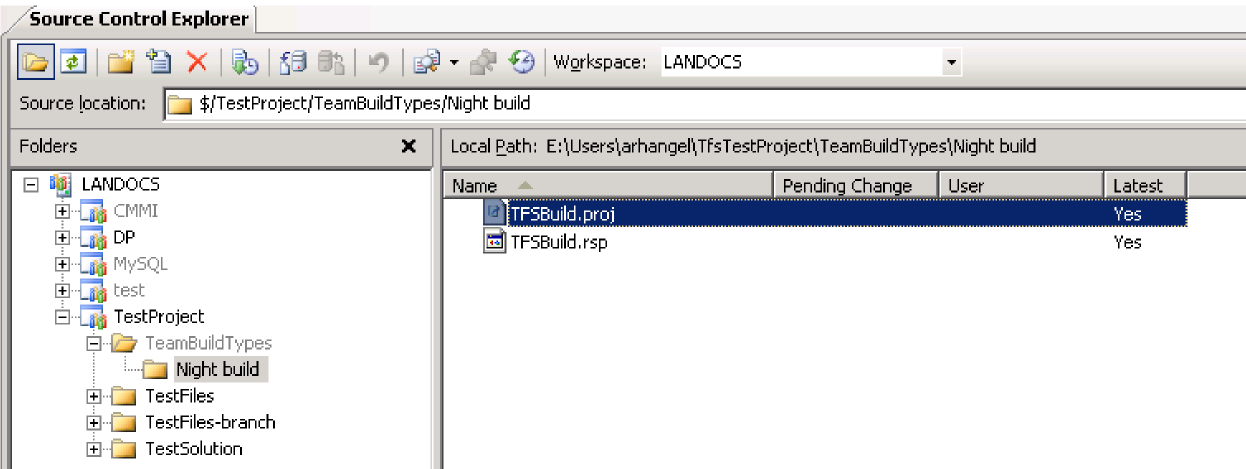

Рис. 13.32. Файл определения проекта.

Заметим, что для большинства простых проектов обычно хватает настроек, доступных через визуальные редакторы, и изменять эти файлы приходится относительно редко.

Лекция 15. VSTS: тестирование

Выделим следующие возможности VSTS по тестированию:

- интегрированная система отслеживания ошибок;
- средства разработки модульных тестов;
- средства организации тестовых пакетов;
- автоматическое тестирование Web-приложений (в том числе и нагрузочное).

Помимо перечисленных выше возможностей в процесс тестирования в VSTS до некоторой степени вовлечены практически все остальные системы – система контроля версий используется для хранения описаний тестов, система управления сборок позволяет автоматически выполнять тестовые пакеты, система отчетов позволяет следить за изменением качества продукта, а интеграция с офисными приложениями позволяет строить планы по тестированию и исправлению дефектов. Однако, каждая из выше перечисленных систем используется только в рамках своих стандартных возможностей, поэтому мы не будем подробно рассматривать их в этом разделе.

Система отслеживания ошибок

Общее. Система отслеживания ошибок в VSTS реализована на базе системы управления элементами работ. Ведь ошибки (bugs) – особый тип элементов работ. По сравнению с другими системами отслеживания ошибок, интегрированная система на основе системы управления элементами работы обладает рядом следующих серьезных преимуществ.

- Возможность задания связи изменений программного кода с ошибками, которые они предназначены исправить, позволяет легче поддерживать и развивать систему, избегая при этом значительной регрессии.
- Интеграция с системой автоматической сборки позволяет легко отслеживать то, в какую сборку вошло исправление той или иной ошибки, не требуя от разработчиков дополнительных действий.
- Возможность легко строить сводные отчеты позволяет легко отслеживать текущее качество продукта.
- Возможность интеграции с офисными продуктами и, в частности, с продуктом Microsoft Project, позволят проще планировать и управлять процессом исправления ошибок, в тоже время система автоматических оповещений позволят сделать этот процесс более оперативным.

На рис. 14.1 представлено описание жизненного цикла элемента работы «ошибка» (Bug) из шаблона процесса VSTS под названием MSF for Agile 4.2. У этого типа элемента работы определено три состояния:

- Active – ошибка нуждается в исправлении,
- Resolve – ошибка исправлена,
- Close – ошибка проверена и исправление принято.

На стрелках-переходах указана причины, в силу которых ошибка перешла в данное состояние. Опишем некоторые, самые часто встречаемые переходы.

В состояние Active ошибка попадает, во-первых, после своего создания – то есть тестеровщик нашел новую ошибку и создал соответствующий элемент работы (это начальное действие на картинке не показано). Далее, в это состояние ошибка может попасть из состояние Resolved, после того, как тестеровщик проверил исправление программиста и обнаружил, что тесты все равно «падают» (причина Test failed). Если исправление ошибки было проведено некорректно (поведение системы не соответствует желаемому), то ошибка переходит в состояние Active по причине Wrong Fix. Если же

способ закрытия ошибки является неприемлемы (например, тестер не согласен, что данная ошибка является дубликатом другой), то используется причина Resolution Denied. Наконец, ошибка может перейти в стояние Active, если она вновь стала появляться – причины Reactivation и Regression. При этом для тестеровщика важно не создавать новую ошибку, а понять, что это старя, закрытая, вновь проявилась. Эта информация поможет разработчикам быстрее разобраться с исправлениями – проглядеть те изменения исходных текстов, которые закрывали эту ошибку и исправить их. При этом может очень эффективно работать связь, которую обеспечивает VSTS для элементов работ и изменениями в средстве контроля версий – что подробно обсуждалось в лекции о поддержке в TFS конфигурационного управления.

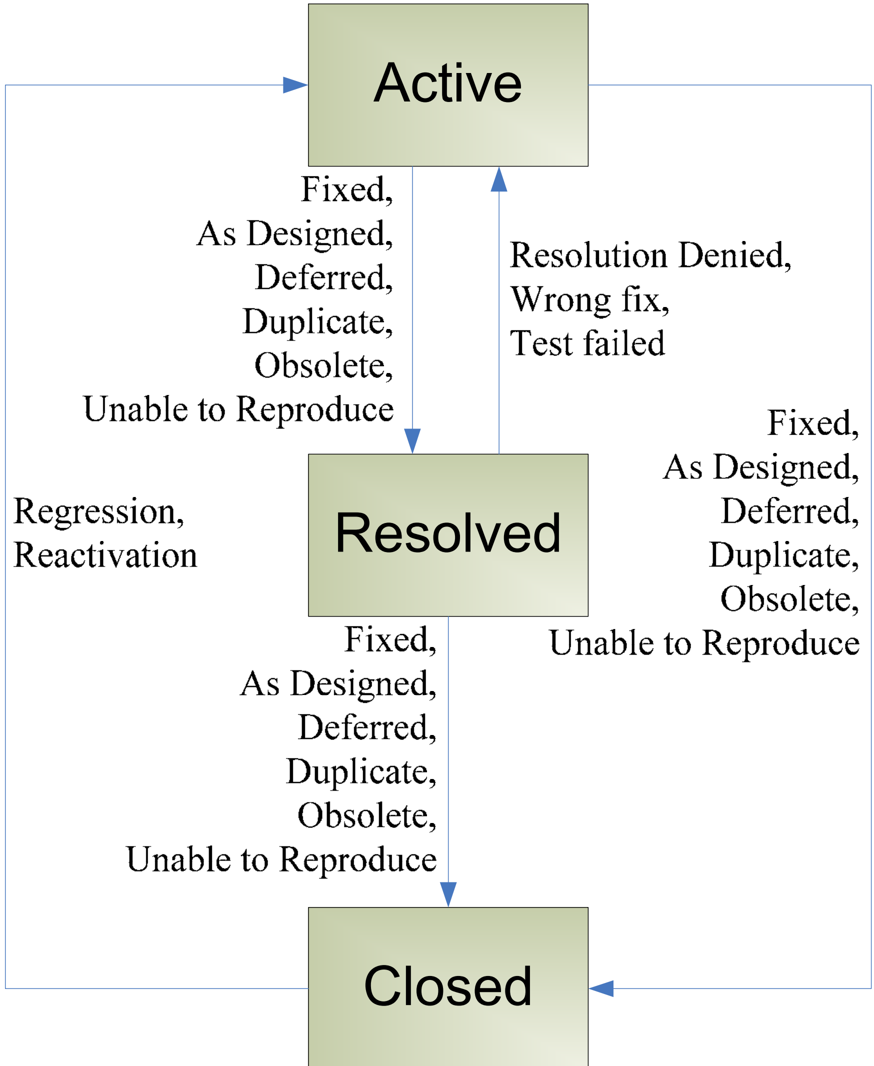

Рис. 14.1. Жизненный цикл.

В состояние Resolve ошибка переходит, во-первых, после того, как разработчик ее исправил. В это состояние разработчик может перевести ошибку еще и по тому, что это не ошибка, а свойство (тестеровщик неправильно понял требования к системе или проектную спецификацию) – причина As Designed. А также потому, что ошибка повторяет другую, найденную ранее ошибку (Duplicate), ошибка не воспроизводится у разработчика (Unable to reproduce) и т.д.

В состояние Close ошибка переходит, во-первых, когда тестеровщиек принял ее исправление (причина Fixed). Во-вторых, когда он согласился с мнением разработчика, что она повторная (Duplicated), не воспроизводится (Unable to reproduce) и пр. По этим же

причинам сам разработчик может перевести ошибку в состояние Close прямо из состояния Active. Правда, это может быть не любой разработчик, а, например, технический руководитель проекта или архитектор. Все остальные разработчики могут не иметь прав переводит ошибки самостоятельно в состояние Close, а обязаны действовать через тестеровщиков.

Как создается описание ошибки. Создание новой ошибки может происходить либо с помощью пункта меню в Team Explorer “Team/Add Bug…”, либо посредством добавления связанных элементов работы для задач, при реализации которых ошибки были обнаружены. Кроме того, провал автоматической сборки или прогона тестов может служить триггером для автоматического создания ошибки. Окно для описания ошибки показано на рис. 14.2.

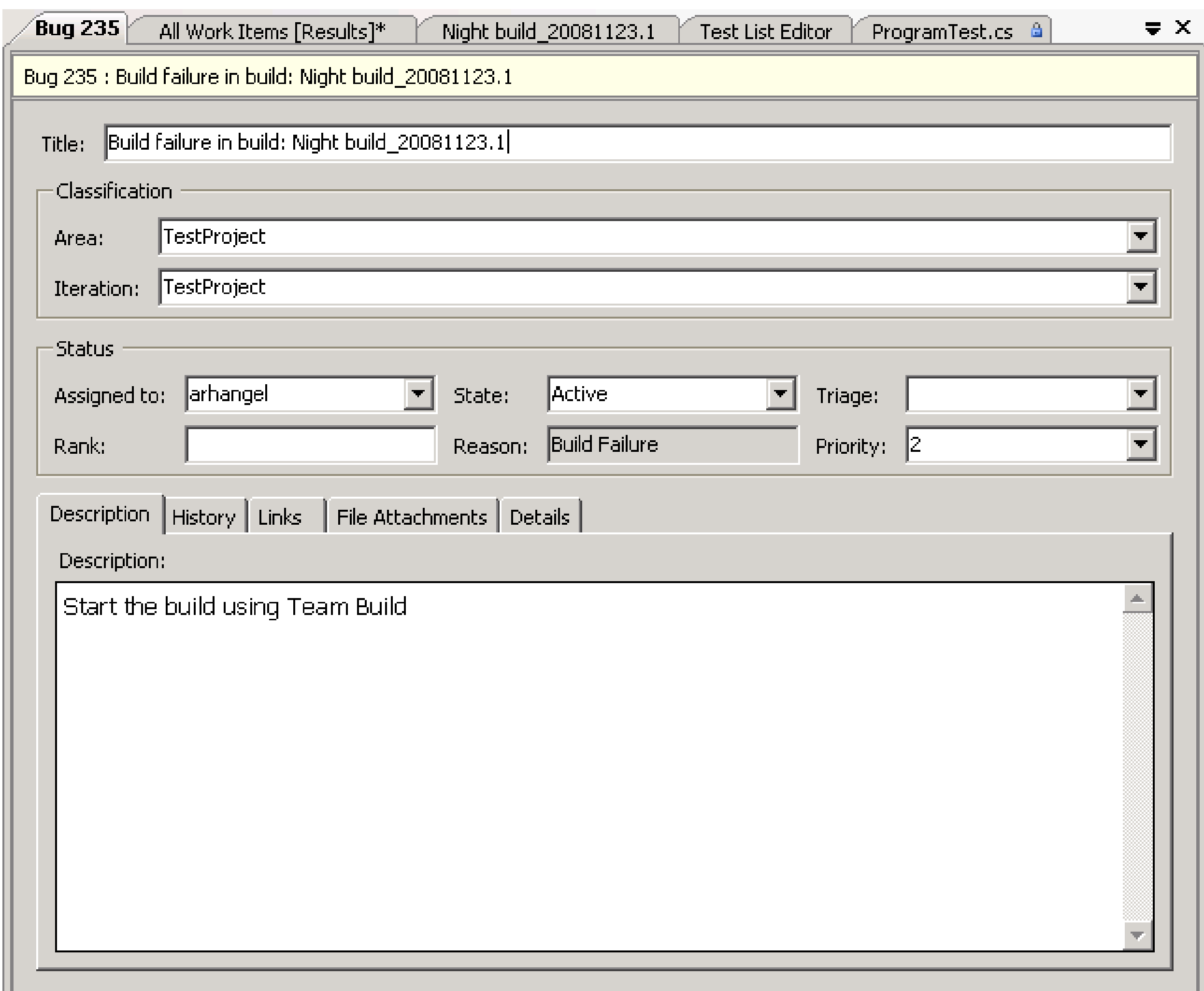

Рис. 14.2. Автоматически созданная ошибка.

Связь изменений исходных текстов ПО и ошибок. В лекции про конфигурационное управление мы подробно рассмотрели связь изменения исходников кода с элементами работы. Теперь посмотрим на то, как ошибка связан с этими изменениями (то есть мы смотрим на ту же задачу, но с другой стороны – со стороны элементов работы, и выбираем один специфический тип элемента работы – ошибку). Все изменения в коде, связанные с исправлением определенной ошибки, легко можно отследить используя закладку «Links» в диалоге описания ошибки. Так, на рис.14.3 показано, что ошибка связана с пакетом изменений, внесенным в систему контроля версий.

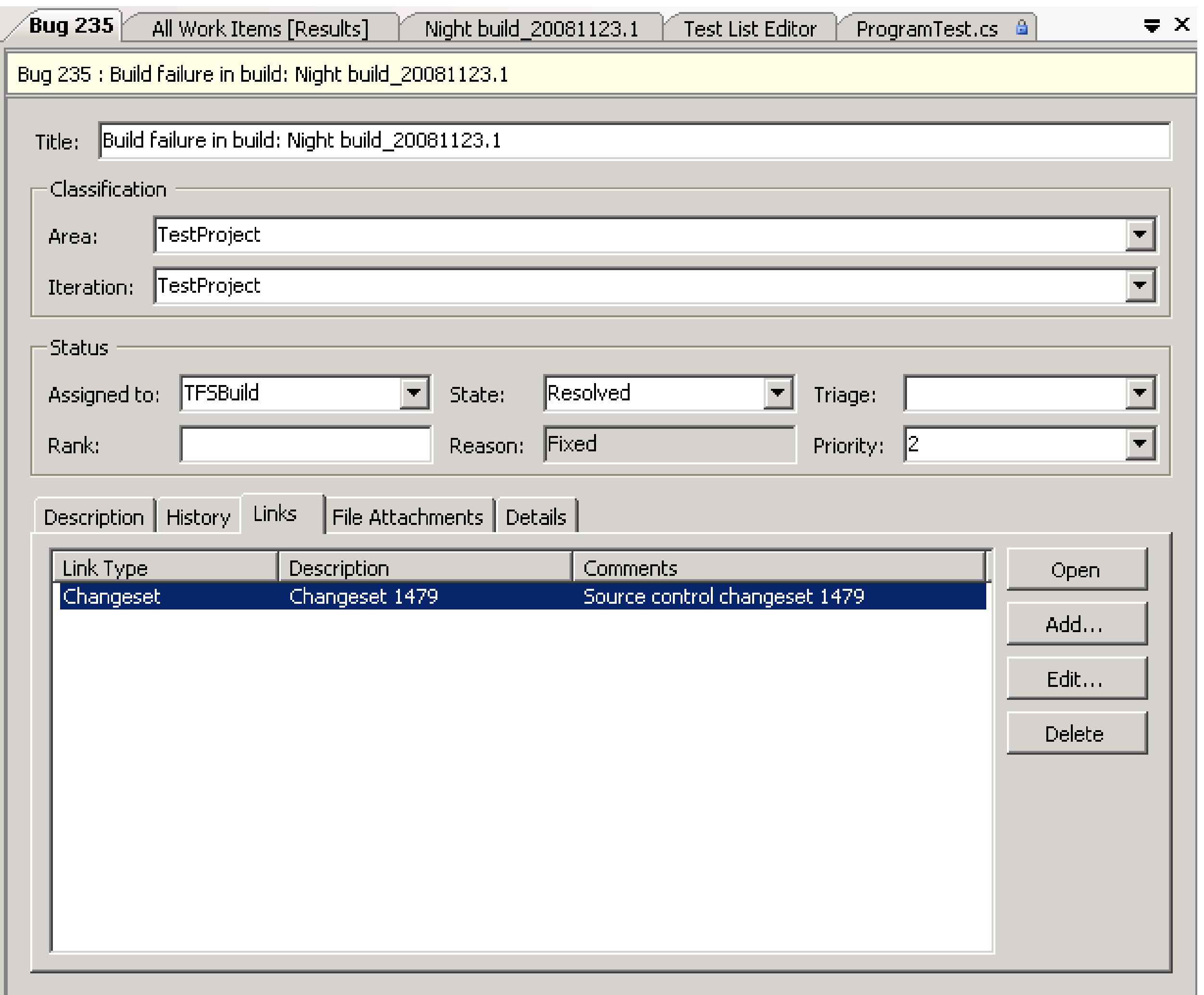

Рис. 14.3. Отслеживание изменений по ошибке.

Система оповещений о событиях в проекте является отдельной подсистемой TFS, оповещающей членов команды по средством электронной почты о различных событиях, например, о завершении процесса сборки проекта или об изменении элемента работы. Эта система используется для оперативного мониторинга состояния проекта, что особенно важно при тестировании. Рассылка оповещений осуществляется посредство электронной почты.

Настроить условия, при которых следует отправлять оповещения, а также список получателей, можно выполнить, используя специальную команду из меню проекта, как показано на рис 14.4. Все это конфигурируются на уровне проекта в целом, но каждый участник разработки может определить свои правила отправки оповещений.

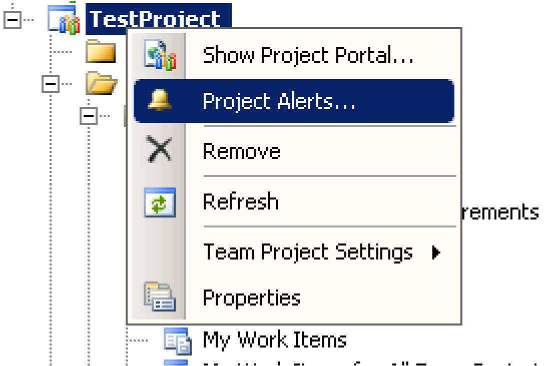

Рис. 14.4. Управление подписками.

В TFS, как показано на рис. 14.5, поддержаны следующие типы автоматических оповещений.

- При изменении элемента работы (оповещение отправляется при изменении любого реквизита). Этот вид оповещений позволяет оперативно узнавать о появлении новых элементах работы и об изменении существующих. Например, разработчик может оперативно получать сообщения о переведенных на него ошибках.
- Внесение изменений в систему контроля версий. Как правило, этот тип оповещения используется архитекторами или техническими лидерами команды для контроля качества вносимого кода посредством регулярной проверки вносимых изменений.
- При автоматическом или ручном изменении атрибута «качество» в описании результатов автоматической сборки[16]. Этот тип оповещений позволяет руководителям проекта узнавать об изменении состояния проекта.
- При завершении процесса автоматической сборки, не зависимо от результатов. Данный тип оповещений полезен для всех участников проекта.

[16] «Качество» является атрибутом сборки, устанавливаемым вручную тестером после проведения тестирования. На основании значения этого атрибута может быть принято решение о готовности или нет определенной сборки к выпуску или переводу на дальнейшие этапы тестирования.

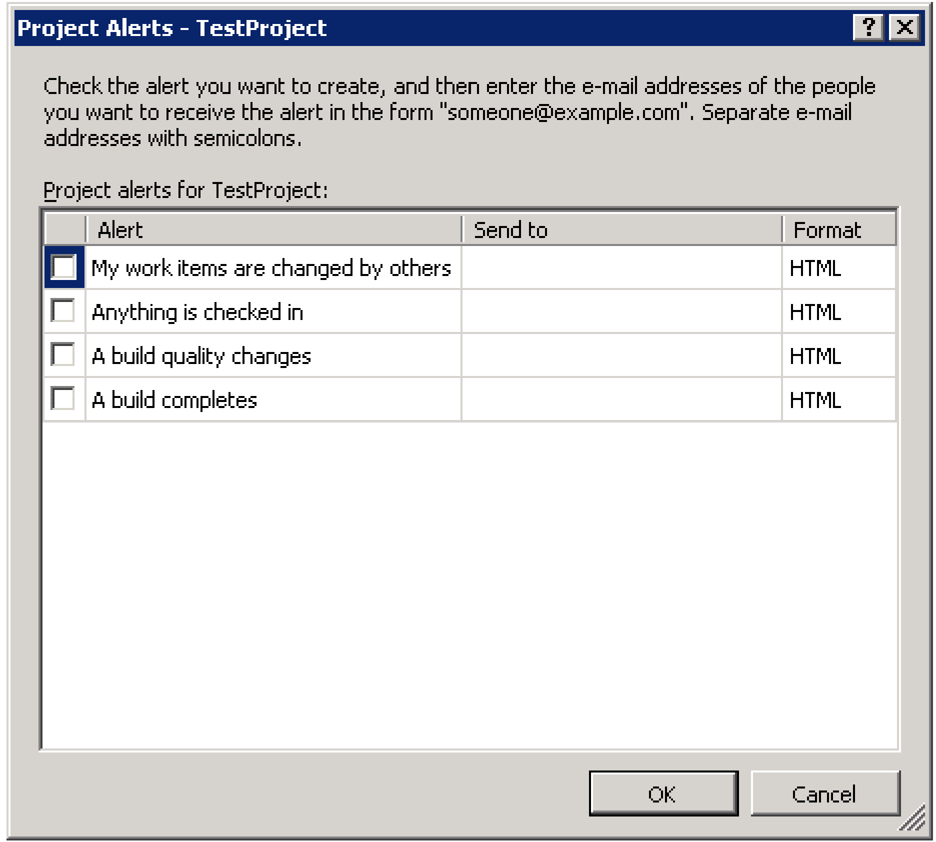

Рис. 14.5. Настройка получателей оповещений.

Оповещения, рассылаемые TFS, содержат только базовую информацию о произошедшем событии и ссылку, позволяющую просмотреть детали о события через Internet Explorer, на Share Point портале проекта. В частности, информация о результатах автоматической сборки и о тех ошибках, исправления которых вошли в соответствующую ей версию исходных кодов проекта, представлена на рис. 14.6.

Build Night build_20081123.2

Summary	Partially Succeeded
Build name:	Night build_20081123.2
Requested by:	LANDOCS\arhangel
Team project:	TestProject
Definition name:	Night build
Agent name:	TestProject
Command-line arguments:	
Started on:	23.11.2008 17:36:56
Completed on:	23.11.2008 17:38:24
Last changed by:	LANDOCS\TFSBuild
Last changed on:	23.11.2008 17:38:24
Quality:	
Work items opened:	Not available
Source control version:	C1479
Log:	\\landocs\TestProjectBuild\Night build_20081123.2\BuildLog.txt

…

Associated work items

ID	Title	State	Assigned To
107	Use Visual Studio core editor for SQL editing	Closed	arhangel
235	Build failure in build: Night build_20081123.1	Resolved	TFSBuild

Note: All dates and times are shown in Russian Standard Time (GMT +03:00:00).
Provided by: Microsoft Visual Studio® Team System 2008.

Рис. 14.6. Результаты автоматической сборки.

Модульные тесты

Модульные тесты как средство повышения качества программного обеспечения появились достаточно давно, однако они долго оставались не поддержанными продуктами Microsoft. Впервые поддержка модульных тестов появилась в Visual Sutiod 2005 и доступна в изданиях Professional и выше (в том числе, и во всех изданиях группы Team).

Основная идея модульных тестов заключается в том, что работоспособность кода можно проверить автоматически с помощью написания дополнительного кода, вызывающего тестируемы и анализирующего результаты. При этом, если для системы в целом такой подход достаточно затруднителен в виду сложности системы, то для отдельных частей системы (модулей), этот метод применим и дает хорошие результаты. Как правило, основной единицей для модульного тестирования являются классы и методы классов.

Исторически одним из первых популярных инструментов, ориентированных на организацию модульного тестирования, был jUnit для Java-приложений, клонированный затем под многие другие платформы. Для платформы .NET пионером здесь являлся nUnit, который до сих пор занимает лидирующую позицию в этой нише. Однако, лидерство nUnit серьезно пошатнулось с появлением поддержки модульных тестов в Visual Studio. По сравнению с классическими системами Visual Studio обладает рядом следующих преимуществ.

- Поддержана полная интеграция в пользовательский интерфейс, включая запуск и анализ результатов (для других систем интеграция доступна за отдельные деньги и не так обширна).

- Реализованы возможности для легкой интеграции в средства автоматической сборки (только для TFS Team Build).
- Предложены дополнительные средства для описания процедуры развертывания теста (больное место большинства остальных систем) и конфигурации других аспектов выполнения.
- Имеются средства автоматической генерации сигнатур тестов и средств доступа к приватным частям тестируемых классов.
- Поддержано управление тесповыми данными, а также тестами, использующими данные на уровне платформы.

В версии Visual Stuido 2008 поддержка модульного тестирования была перенесена из изданий семейства Team в издание Professional, что было вполне логичным шагом, так как модульное тестирование является общей практикой, применяемой как в личной, так и в командной разработке. Так как модульное тестирования более не является чем-то специфичным для Visual Studio Team System, а его реализация соответствует большинству стандартных пакетов в этой области, мы не будем останавливаться на этой возможности более подробно.

Пакеты тестов

Как правило, модульные тесты и тестовые конфигурации разрабатываются самими разработчиками, основная задача тестера в этом случае – организовать все тесты в упорядоченную структуру пакетов и указать, какие пакеты должны исполняться в каких условиях. Вся иерархия тестов хранится в так называемом файле метаданных, имеющем расширение vsmdi. Этот файл находится в системе контроля версий и может быть включен в решение как отдельный элемент – см. рис. 14.7.

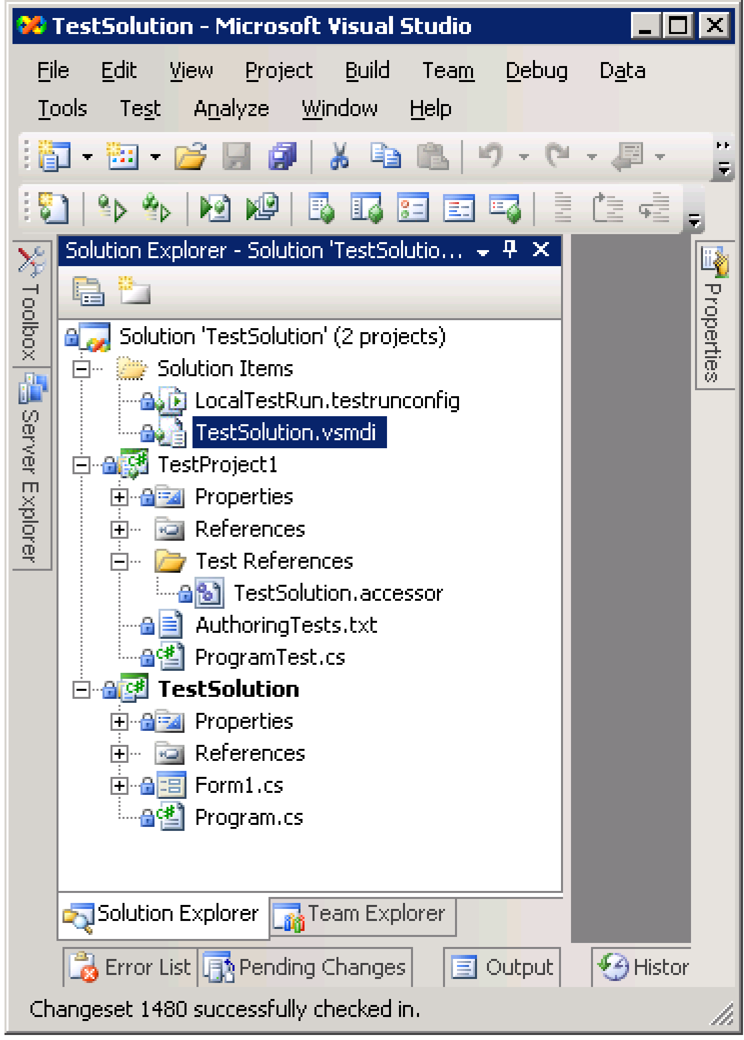

Рис. 14.7. Пакет с иерархией тестов.

Для создания тестового пакета можно воспользоваться меню «Create New Test List», как показано на рис. 14.8, и после этого откроется окно для задания имени нового пакета и определения его места в иерархии тестовых пакетов (см. рис. 14.9). В том случае, если для решения уже создан файл метаданных, пакет тестов будет добавлен к нему, если же файла метаданных еще создано не было, он будет создан автоматически.

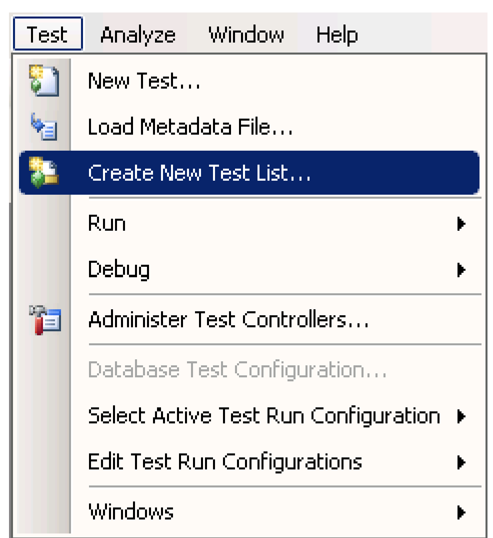

Рис. 14.8. Создание списка тестов.

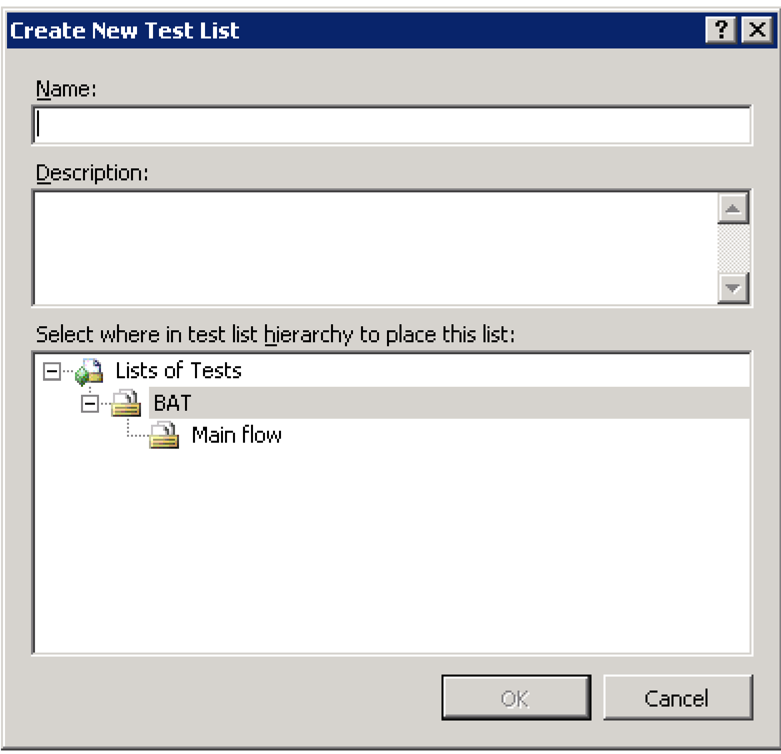

Рис. 14.9. Свойства нового списка тестов.

Содержимое пакетов тестов редактируется с помощью специального редактора, показанного на рис. 14.10. В пакеты могут быть включены тесты, находящиеся в одном из проектов текущего решения.

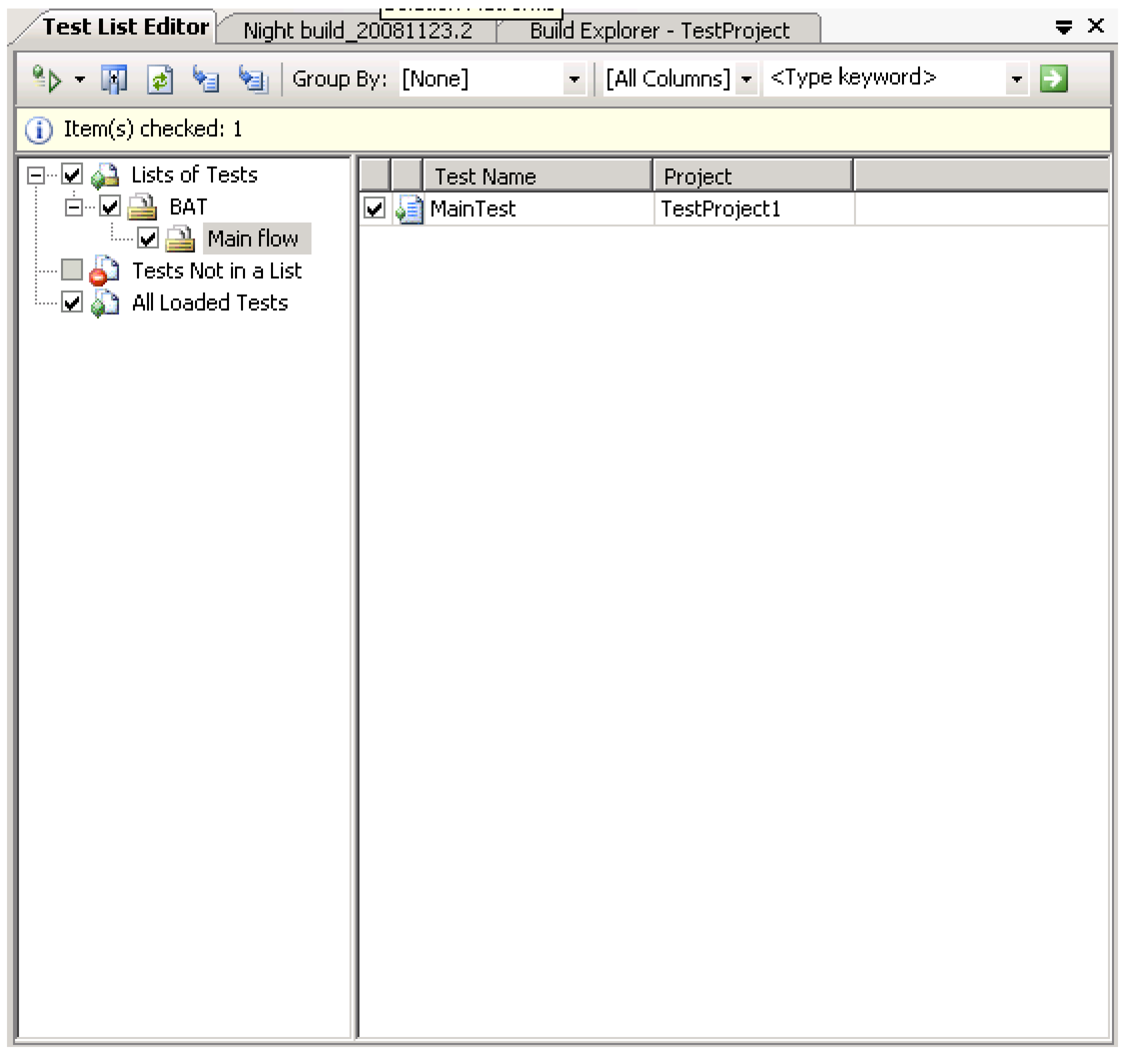

Рис. 14.10. Редактор список тестов.

Тестовые пакеты могут использоваться как для ручного прогона тестов определенной тематики (команда «Run checked», рис. 14.11), так и для автоматического прогона в рамках автоматической сборки.

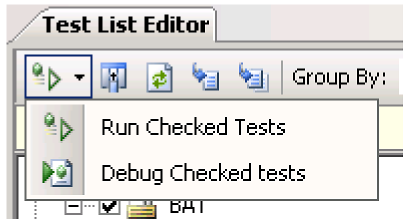

Рис. 14.11. «Ручной» запуск пакета тестов.

Указать тесты, которые будут запускаться при определенной сборке можно при создании файла с описание сборки MsBuild, или в последствии через модификацию проекта MsBuild. В первом случае достаточно на соответствующем шаге мастера выбрать файл метаданных и отметить галочками интересующие пакеты тестов (рис. 14.12). Во втором случае необходимо открыть проект MsBuild в редакторе XML, найти элемент MetaDataFile, или в писать необходимые пакеты вручную:

```
<MetaDataFile
   Include="$(BuildProjectFolderPath)/../../TestSolution/TestSolution.vsmdi">
   <TestList>BAT/Main flow</TestList>
</MetaDataFile>
```

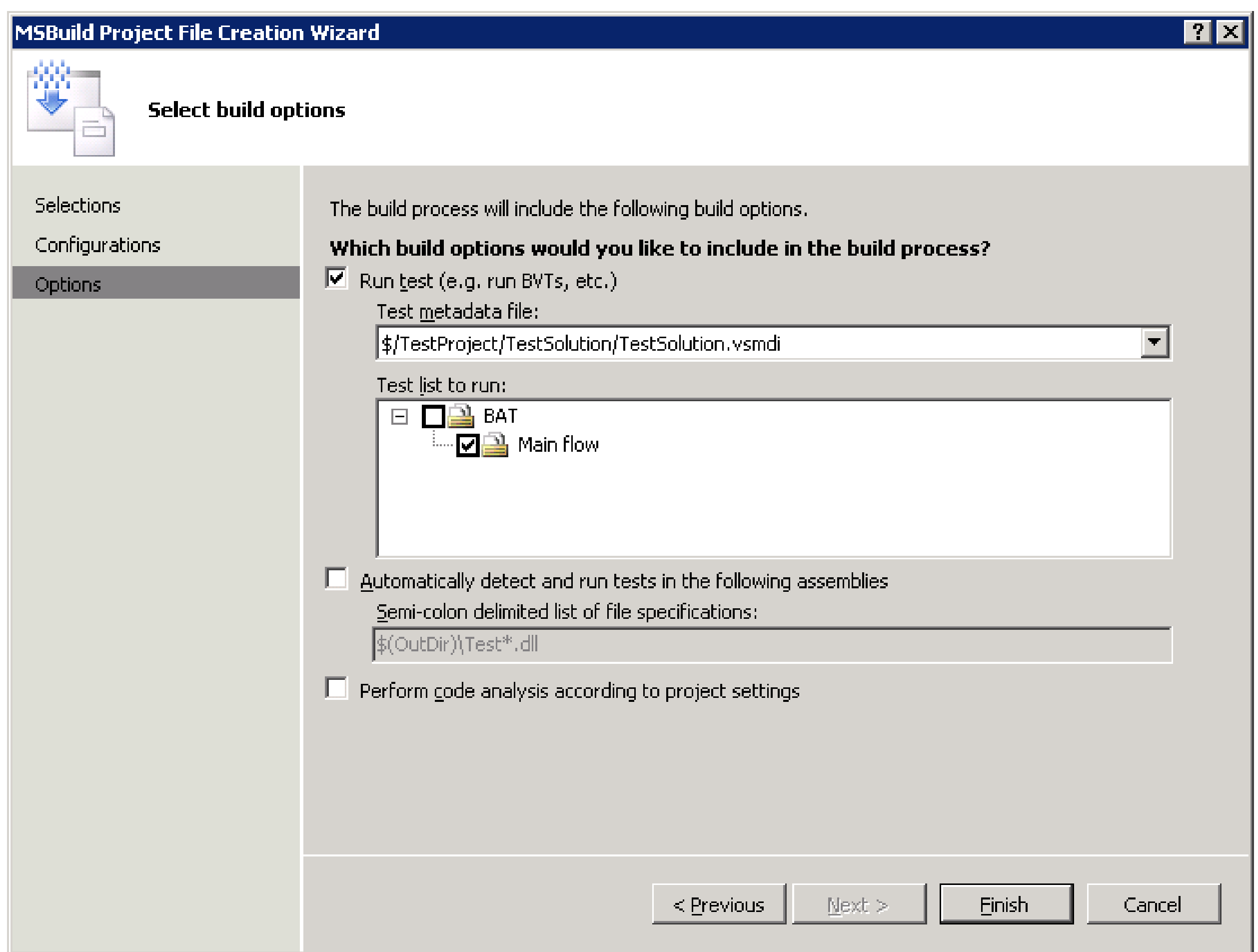

Рис. 14.12. Выбор пакета тестов при автоматической сборке.

Автоматическое тестирование Web-приложений

Capture & Playback подход. Этот подход к тестированию пользовательских интерфейсов выглядит очень эффектно и основан на следующей идее. Тестеровщик проходится мышкой по окнам, пунктам меню и другим элементам интерфейса. Специальная программа записывает его шаги и потом их воспроизводит в пакетном режиме. То есть очень просто получаются повторяемые тесты. Технически это устраивается так.

Специальное тестовое окружение с той или иной точностью распознает, куда именно было нажато мышкой на экране и создает соответствующий код в специальном скрипте. Потом этот скрипт «прогоняется» в пакетном режиме, воспроизводя действия тестеровщиков. Весь вопрос в том, каким образом распознается клик тестеровщика мышкой. Идеально, когда этот клик связывается с соответствующим элементом управления интерфейса. То есть если тестеровщик нажал кнопку в каком-то диалоге, то в скрипте эта информация сохраняется в полном объеме. Другая, более грубая ситуация имеет место тогда, когда тестовое окружение не может распознать, какой элемент

пользовательского интерфейса активировал тестеровщик. Тогда в скрипт заносится информация о тех координатах на экране, куда бал клик мышкой.

Чем плоха последняя ситуация? Дело в том, что при малейшем изменении пользовательского интерфейса (а это типичная ситуация, ведь ПО развивается, дорабатывается и тестируется одновременно) автоматический тест-скрипт, созданный таким образом, выходит из строя. Там, куда раньше клик мышкой попадал, например, на нужную кнопку, теперь находится совсем другой элемент управления.

Если же тестовое окружение распознало элемент интерфейса, то такой тест-скрипт оказывается более «живучим». Это происходит на уровне перехвата сообщений на уровне операционной системы (в частности, Windows). Но для того, чтобы это было возможно, код приложения должен быть написан «правильным» образом. Далеко не все интерфейсные приложения написаны «правильно».

То, насколько успешно для конкретного приложения можно применить данный подход, определяется несколькими факторами. Основным является то, какая используется платформа (например, Java Swing, AWT, Windows Forms, WPF, etc.) и дополнительные библиотеки с элементами пользовательского интерфейса. Наиболее зрелой платформой с этой точки зрения на данный момент является MS Windows Forms.

Capture & Playback при тестировании Web-интерфейсов. В случае с тестированием Web-интерфейса ситуация с точностью распознавания элементов управления (interface controls) значительно проще, чем при тестировании произвольного пользовательского интерфейса. Взаимодействие с сервером происходит по строго описанному протоколу HTTP, что позволяет при записи перехватить отправляемые и получаемые сообщения. Кроме того, визуальное представление страницы задано в структурированном формате HTML, что позволяет легко опознать отдельные элементы на странице.

В издание Visual Studio Team Edition for Software Testers включен дополнительный пакет, облегчающий автоматизацию тестирования Web-приложений методом Capture & Playback. Он позволяет, как автоматически генерировать просты тестовые сценарии на основе записи действия пользователя, так и писать более точные тесты на любом языке платформы .NET.

Добавить новый Web-тест к решению можно с помощью команды «Test/New Test», выбрав в возникшем диалоге (рис 14.13) тип теста Web Test.

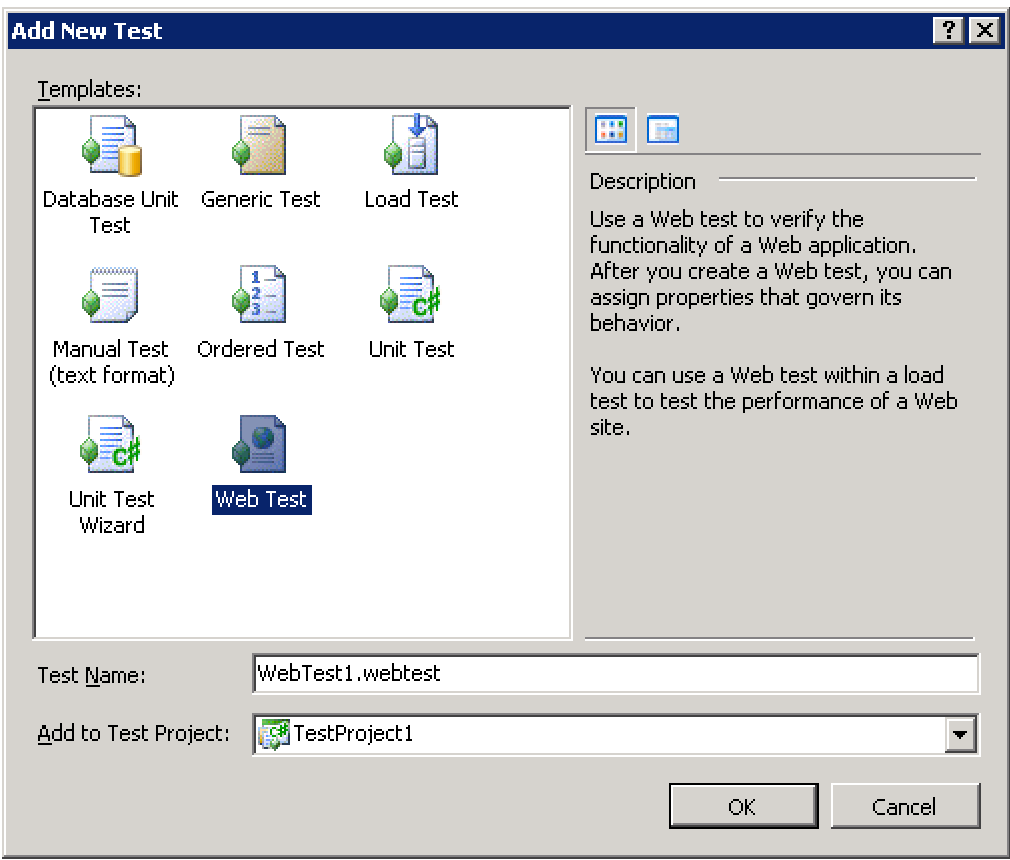

Рис. 14.13. Создание Web-теста.

После создания нового теста автоматически будет запущена процедура записи сценария теста в браузере (см. рис. 14.14). На этом этапе достаточно ввести www-адрес приложения, которое следует протестировать, после чего выполнить тестовый «проход» по Web-интерфейсу непосредственно в браузере.

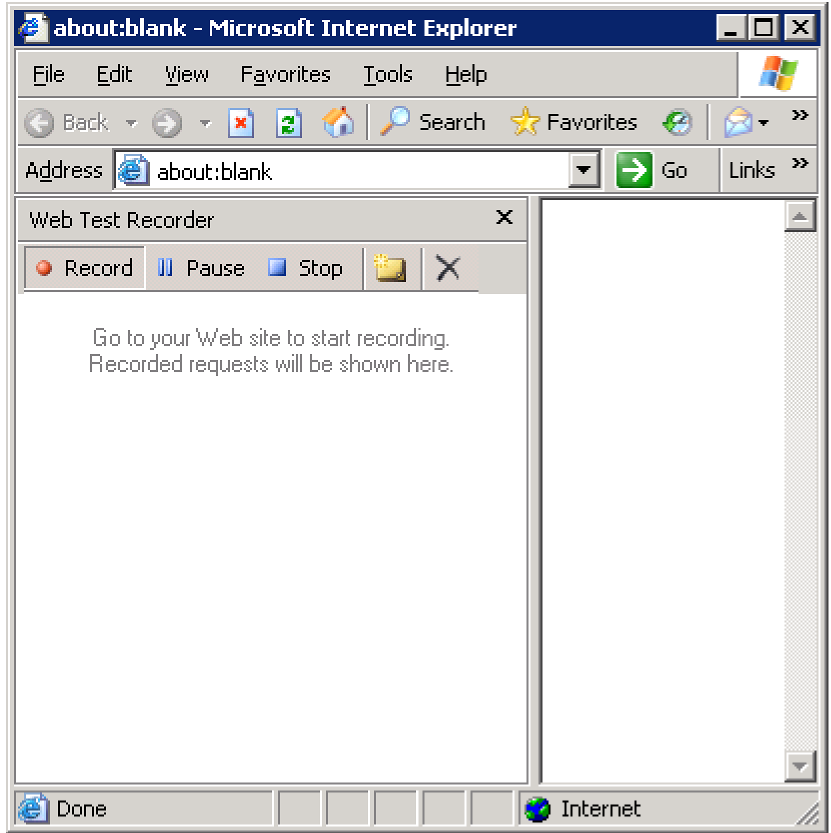

Рис. 14.14. Запись шагов тестреровщика Web-приложения в Internet Explorer.

После окончания записи будет автоматически сгенерирован тест, включающий все отправленные на сервер http-запросы и все полученные ответы (см. рис. 14.15). При этом генератор автоматически добавит некоторые правила, по которым будет проверятся корректность работы теста.

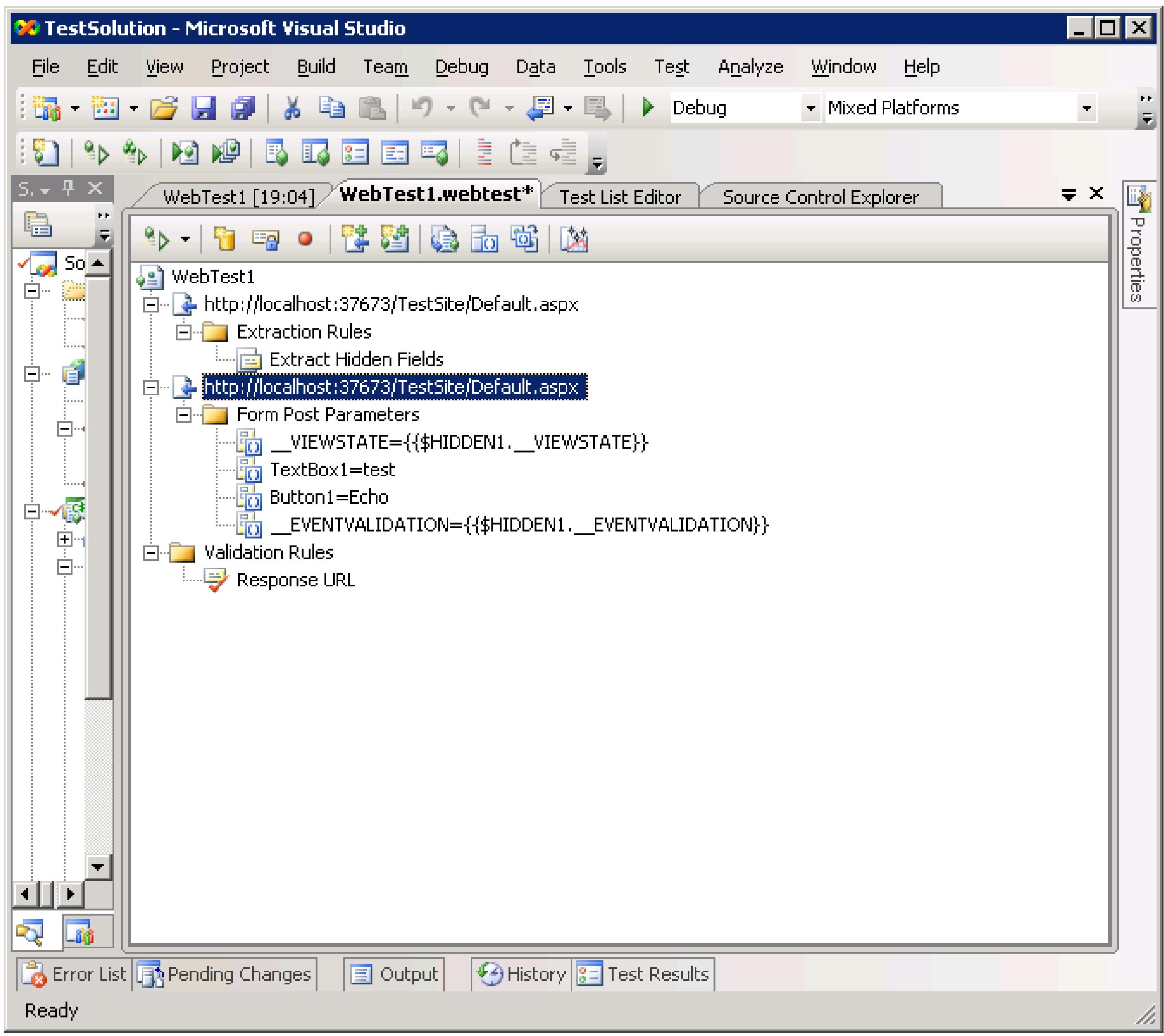

Рис. 14.15. Редактор Web-теста.

Для каждого из шагов теста можно, с помощью визуального редактора, добавить дополнительные проверки или опции, управляющие ходом выполнения: поиск подстроки в ответе сервера (тексте полученного HTML), валидацию HTML через задание регулярного выражения, проверка наличия или отсутствия определенных тегов или атрибутов на странице и т.д. Допускается также возможность разработки собственных правил на любом .NET языке. В тех же случаях, когда гибкости редактора недостаточно, можно сгенерировать C# код для данного теста и реализовать необходимую логику «вручную».

При написании правил валидации очень важно правильно выбрать необходимый уровень детализации. Чем более детально сформулировано правило и чем более специфично оно для данного HTML, тем больше вероятность того, что тест придется изменять при изменении кода тестируемого приложения, даже если эти изменения не касались напрямую этой части. Хорошее правило валидации должно проверять только то, что является важной частью бизнес логики приложения или то, что является ключевым свойством данной HTML-страницы. При этом правило не должно проверять детали верстки и дополнительные визуальные эффекты. К сожалению, автоматически сгенерированные правила далеко не всегда оказываются наиболее эффективными.

Созданный в редакторе или в ручную тест является полноправным тестом и может быть включен в тестовый пакет, а следовательно, и в процедуры автоматической сборки

(см. рис. 14.16). Однако, для того, чтобы автоматическое тестирование было возможным, необходимо соответствующим образом настроить сервер автоматических сборок – на нем должен быть развернут сервер IIS с тестовым сайтом, а одним из этапов сборки должно быть обновление кода этого сайта.

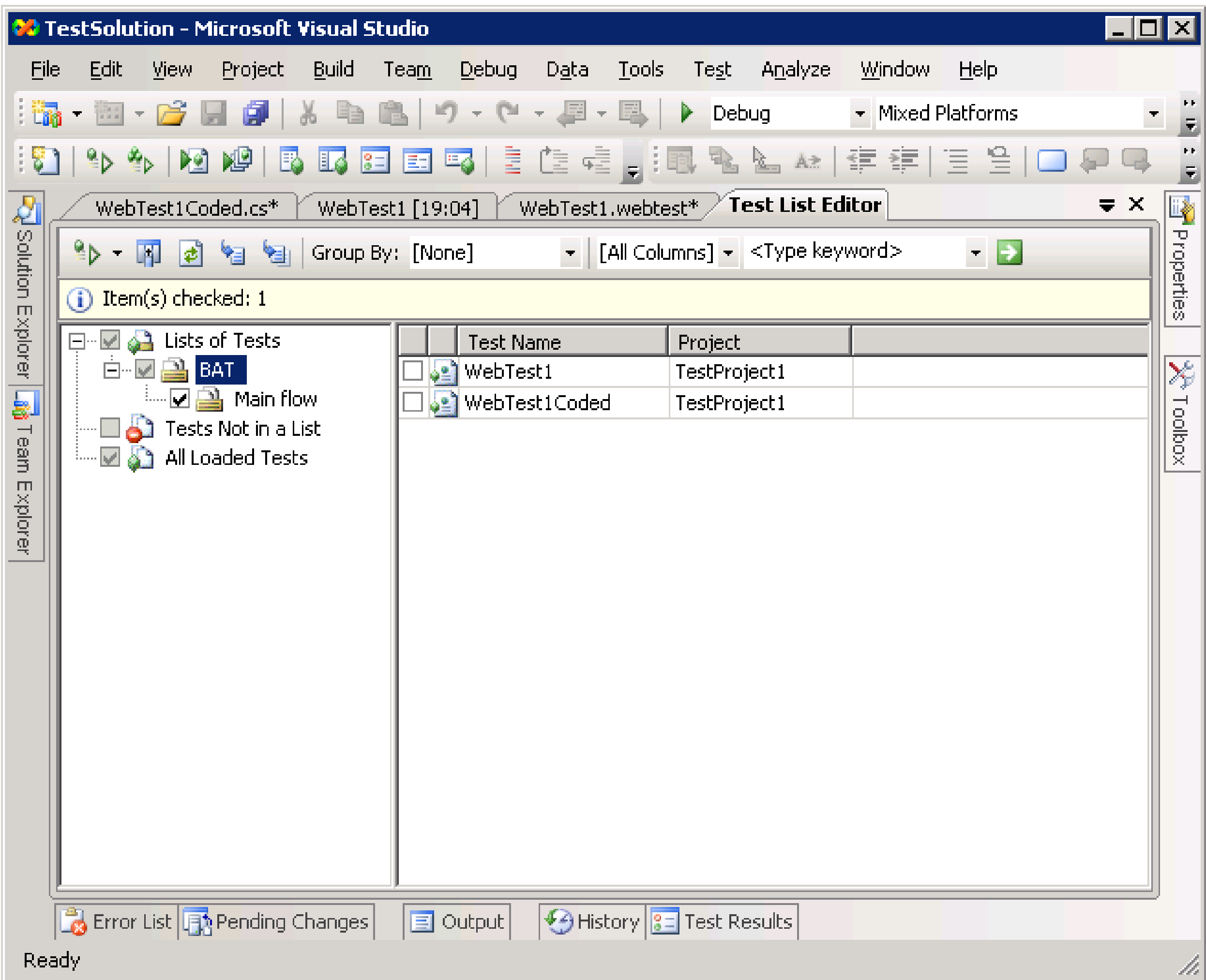

Рис. 14.16. Web-тесты в пакетах тестов.

Лекция 16. VSTS: поддержка различных моделей процесса

Поддержка шаблонов процесса

Общее. Чем активнее инструментарий интегрируется в процесс разработки, чем больше функциональности по контролю и поддержке бизнес-процессов разработки ПО он предоставляет, тем более востребованными становятся механизмы настройки этого инструментария. Естественно, для такой системы как VSTS вопрос настройки стоит особенно остро.

При создании в VSTS каждого нового проекта, сразу после выбора имени проекта, пользователь выбирает шаблон процесса разработки для этого проекта. Весь последующий процесс создания проекта определяется этим шаблоном. Шаблон может быть отдельно разработан или подправлен существующий.

Шаблон процесса содержит шесть основных разделов, в рамках которых можно осуществлять настройку работы TFS.

1. Классификация (Classification) – описание областей работы, итераций и настройка интеграции с Microsoft Project. *Области работы* – это способ для категоризации работ в проекте. Примером области работы может быть как направление деятельности (разработка, тестирование, документирование и т.д.), так и работа над определенной частью проекта (серверная часть, клиент, инфраструктура и т.д.).
2. Отслеживание элементов работы (Work Item Tracing) – описание типов элементов работы, включая задание их жизненного цикла, определение набора элементов работы и запросов, создаваемых по умолчанию для нового проекта.
3. Отчеты (Reports) – описание отчетов проекта на специальном XML-языке RDL (Report Definition Language). Имеющиеся в шаблоне отчеты по умолчанию можно использовать «as is», а также исправлять и дополнять. Создание полностью нового вида отчетов является довольно трудоемкой работой.
4. Портал (Portal) – настройка шаблона портала в SharePoint с тестовым описанием процесса разработки, а также набором рабочих документов проекта (планов, дизайн-спецификаций и пр.). На основании этого шаблона будет автоматически формироваться портал для каждого нового проекта.
5. Группы и права доступа (Groups & Permissions) настраиваются для использования системы управления версиями, а также для различных правил жизненного цикла элементов работы.
6. Контроль версий (Source Control) – набор настроек для системы контроля версий, включая набор политик внесения изменения, позволения или запрещения множественного взятия файлов на редактирование и т.д.

Шаблон процесса разработки действует в двух следующих основных направлениях: ограничивает действия участников процесса таким образом, чтобы они максимально соответствовали шаблону, и предоставляет некоторую инфраструктуру, позволяющую легче решать основные задачи, возникающие в рамках данного процесса. К ограничениям можно отнести настройки жизненного цикла элементов работы, а также права и политики работы с системой контроля версий, а к предоставлению инфраструктуры: списки отчетов, запросы на элементы работы и основная часть – портал SharePoint, содержащий информацию об использовании процесса, а также необходимые административные документы. Этот портал является важной частью с «человеческой» точки зрения, однако, с точки зрения автоматизации его роль минимальна.

Естественно, этих средств недостаточно для того, чтобы гарантировать, что все участники процесса будут четко его придерживаться. Основным элементом в процессе по-прежнему остается человеческий фактор. Но, с другой стороны, внедряя лишь

относительно небольшое число ограничений, TFS позволяет сохранить общую гибкость, что может принести огромную пользу. Таким образом, в отношении TFS, как и в отношении большинства успешных систем этого класса, верно утверждение – «не столько важен сам инструмент, сколько то, как им пользуются».

Инструменты настройки. Для управления шаблонами процесса разработки используется утилита Process Template Manager (рис. 15.1), доступная из меню TFS Settings (эта утилита устанавливается вместе с Team Explorer). Этот менеджер позволяет загружать и выгружать шаблоны, а также определять шаблон, используемый для новых проектов по умолчанию.

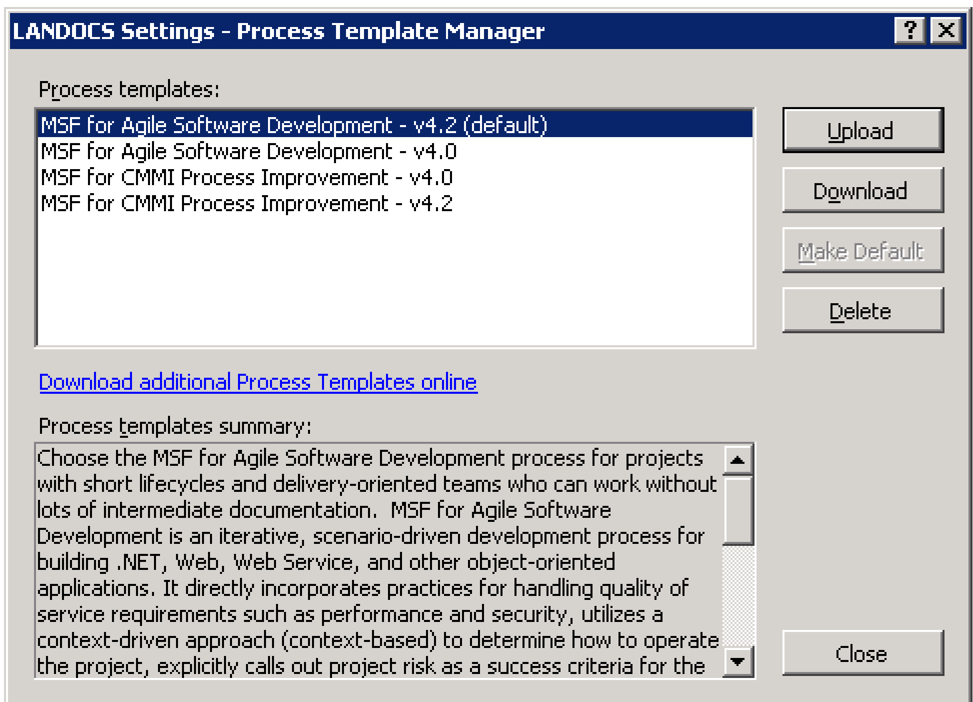

Рис. 15.1. Менеджер шаблонов.

С точки зрения реализации шаблон процесса разработки является набором XML-файлов, которые, пожалуй, никому не захочется редактировать «вручную». К счастью, существует инструменты, позволяющие значительно упростить этот процесс[17]. Эти инструменты входят в уже упоминавшийся нами выше пакет Power Tools. Эти инструменты доступны из меню Tools/Process Editor (см. рис. 15.2). Кратко перечислим и охарактеризуем их.

- Редактор типов элементов работы, который позволяет редактировать определения типов элементов работы, включая наборы реквизитов и жизненный цикл. Может быть использован как для редактирования файлов с описанием типов элементов работы, экспортированных с помощью Process Template Manager, так и для редактирования типов, находящихся внутри одного шаблона процесса. Кроме того, этот редактор может быть использован для импорта/экспорта типов элементов работы в/из шаблон процесса, что особенно полезно при распространении сделанных изменений по разным проектам. Более подробно этот редактор уже рассматривался выше.
- Редактор шаблона процесса разработки позволяет редактировать остальные аспекты шаблона процесса разработки. Может быть использован только для редактирования шаблона, выгруженного из TFS в файловую систему.

[17] http://msdn.microsoft.com/en-us/vsts2008/aa718802.aspx.

- Редактор глобальных списков позволяет определить списковые типы для реквизитов элементов работы, а также управлять множеством значений в них. По умолчанию TFS автоматически поддерживает глобальный список сборок, однако пользователь может определить и другие списки[18].
- Просмотрщик реквизитов элементов работы – это небольшая утилита, позволяющая просмотреть все используемые реквизиты всех типов элементов работы.

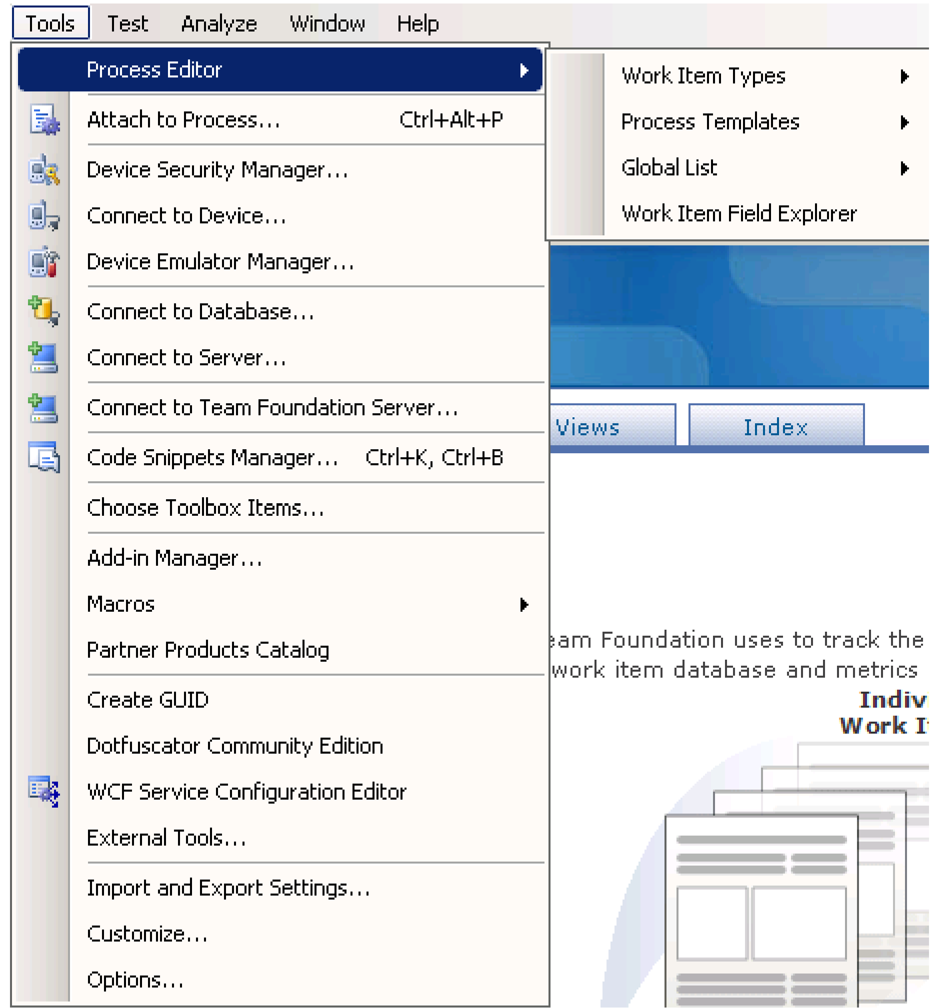

Рис. 15.2. Инструменты редактирования шаблона процесса.

Заметим, что разработка шаблона процесса является очень трудоемкой задачей, требующей от исполнителя как знания принципов работы TFS, так и хорошего знакомства с бизнес-процессами своей компании. Расходы на разработку собственного шаблона будут оправданы только для достаточно больших компаний и в долговременной перспективе. Для небольших компаний и проектов можно рекомендовать другой подход – использования одного из стандартных, существующих шаблонов, наиболее близко подходящих под бизнес процесс, и постепенная настройка необходимых параметров.

[18] Заметим, что помимо настраиваемых глобальных списков в TFS существуют и встроенные системные списки, связанные с различными аспектами деятельности: группы и пользователи, состояния элементов работы и причины перехода, списки итераций и областей работы и т.д.

Обзор существующих шаблонов

На данный момент существует достаточно широкий выбор свободно распространяемых шаблонов процесса разработки, которые можно было бы использовать в качестве основы[19]. Наиболее известными являются следующие шаблоны.

- MSF for Agile Software Development – один из двух шаблонов, входящих в поставку TFS. Описывает достаточно простой вариант методологии MSF, используемый для разработки небольших проектов. Входит в стандартную комплектацию TFS.
- MSF for CMMI – шаблон, используемый для более компаний, подразумевающий большее число типов элементов работы, а также больше формальных процедур разработки. Входит в стандартную комплектацию TFS.
- Conchango SCRUM – шаблон, описывающий широко известную гибкую методологию SCRUM. Не входит в стандартную комплектацию TFS.

MSF for Agile Software Development

В этом шаблоне используются стандартные роли MSF, которые подробно обсуждались выше. В рамках поддержки процесса MSF for Agile соответствующий шаблон объявляет следующие основные типы элементов работы.

- Сценарий (scenario) – функциональное требование к системе в виде некоторого сценария взаимодействия пользователя и системы, описанное на естественном языке как последовательность действий. Сценарий описывает только линейный путь развития событий, а для описания различных ветвлений используются дополнительные сценарии.
- Требование к качеству сервиса (Quality of Service Requirement, QoS) – описание нефункционального требования к системе (то есть требования, которое, которое не может быть выражено в терминах сценария взаимодействия), например, быстродействие и эффективность использования памяти.
- Задача (Task) – задание на выполнение некоторой ограниченной по объему работы в проекте. Для каждой роли задачи могут иметь свою специфику: для разработчика это написание кода, реализующего часть сценария или направленного на достижение определенного качества, а для тестера – написание тестовых сценариев.
- Ошибка (Bug) – элемент работы, использующийся для того, чтобы отслеживать и устранять проблемы и ошибки, обнаруженные в системе.
- Риск (Risk) – некоторый аспект управления проектом, который может оказать влияние на ход проекта (как правило, негативное).

По умолчанию вместе с активацией этого шаблона на портале Share Point разворачиваются следующие документы.

- План разработки в Microsoft Project, настроенный на импорт элементов работы, относящихся к области работы «Разработка». Используется для планирования работы разработчиков.
- План разработки тестов – то же самое, только направленно на планирования работы тестеров.
- Список проблем – список выявленных проблем, которые необходимо отслеживать (элементов работы с проставленным флагом «Is Issue»).
- Список (Check List) основных требований к проекту и их текущий статус.
- Список неупорядоченных элементов работы, которые нужно приоретизировать и запланировать.

---

[19] http://msdn.microsoft.com/ru-ru/teamsystem/aa718801(en-us).aspx

Кроме того, для поддержки процесса используются следующие отчеты.

- Ошибки по приоритету – показывает процентное соотношение в проекте ошибок разной степени серьезности и, таким образом, позволяет оценить степень эффективности тестирования.
- Рейтинг ошибок – показывает соотношение вновь открытых ошибок к закрытым и оставшихся открытых. Позволяет судить о «здоровье» продукта.
- Сборки – позволяет оценивать изменение качества сборок.
- Скорость проекта – отчет, демонстрирующий насколько активно закрываются элементы работы, т.е. насколько быстро команда решает поставленные перед ней задачи.
- Индикаторы качества – объединяет несколько индикаторов, включая количество дефектов, уровень тестового покрытия и т.д.
- Отчет о нагрузочном тестировании – показывает результаты нагрузочного тестирования.
- Регрессия – показывает тесты, которые проходили раньше, но теперь стали падать.
- Реактивация – показывает то, сколько элементов работы заново переходит в активное состояния после исправления.
- Связанные элементы работы – еще один способ просмотра связей элементов работы друг с другом.
- Оставшаяся работа – показывает количество элементов работы, которые закрываются, а которые остаются и появляются с течением времени.
- Незапланированная работа – показывает соотношение сделанного к запланированному и к незапланированному.

Помимо выше перечисленных есть еще несколько отчетов, возвращающих списки элементов работы, однако он редко используются для анализа.

Scrum

Шаблон для работы в методологии Scrum был разработан сообществом разработчиков, в настоящий момент поддерживается компанией Conchagp и доступен здесь http://scrumforteamsystem.com. В этом шаблоне используются стандартные роли Scrum, которые подробно обсуждались выше. Определяются следующие элементы работы.

- Product Backlog Item – высокоуровневое описание определенной функционального или нефункционального требования. Содержит описание, приоритет, установленный Product Owner, а также предварительную оценку трудоемкости. Во многом аналогичен сценарию MSF for Agile.
- Sprint – содержит информацию о текущем или запланированном Sprint, включая текущее состояние и объем доступных для спринта ресурсов.
- Sprint Backlog Item – более низкоуровневое описание задачи, сформированное самое командой. Аналогична задаче MSF.
- Bug – ошибка, обнаруженная при тестировании. Ошибки становятся частью Product Backlog и получают приоритеты от Product Owner.
- Sprint Retrospective – описание некоторого элемента (например, проблемы, удачного нововведения), идентифицированного в рамках Sprint Review Meeting и требующего дальнейшего изучения или выполнения некоторых действий.
- Impediment – нечто, реально или потенциально мешающее эффективной работе команды. Во многом аналогично риску из MSF.

Следует отметить, что описание элементов работы в шаблоне Scrum выгодно отличается от MSF отсутствием большого количества дополнительных, в большинстве случаев ненужных реквизитов, что позволяет сконцентрироваться на наиболее важной информации. Но, с другой стороны, в шаблоне для Scrum фактически не используются ни причины переходов (для каждого перехода определена ровно одна причина), ни правила, ограничивающие переходы. Таким образом, этот шаблон использует лишь часть возможностей, предоставленных TFS.

Среди поставляемых с шаблоном отчетов следует выделить следующие.

- Bugs Count – показывает количество ошибок, отсортированных по состоянию и степени влияния на процесс тестирования.
- Bugs Fixed and Found – соотношение найденных ошибок к закрытым.
- Bug History – показывает количество открытых ошибок в соответствии с их влиянием на процесс тестирования, а также изменение этого количества со временем.
- Bug Priority – показывает распределение ошибок по отношению их влияния на процесс тестирования.
- Bug Resolution Time – показывает насколько быстро исправляются найденные ошибки.
- Development To Test cycle time – демонстрирует, сколько проходит времени между выполнением работы и началом тестирования по этой работе.
- Product Burndown – отчет, демонстрирующий, как быстро снижается количество работы, которую нужно сделать в контексте Product Backlog. Может показывать прогресс по дням или спринтам, а также позволяет увидеть общую тенденцию и предсказать дату завершения работ.
- Product Cumulative Flow – показывает общее состояние Product Backlog в виде соотношения открытых и закрытых элементов, а также их изменения со временем.
- Sprint Burndown и Cumulative Flow – тоже самое, только для Sprint Backlog.

Отдельного упоминания заслуживает система Task Board for Team System, которая позволяет визуализировать представления основного средства управления и мониторинга в Scrum – доски с задачами. Эта система получает информацию о всех элементах Sprint Backlog с сервера и визуализирует их в виде стикеров на доске, позволяя изменять их состояния путем перетаскивания, смю рис. 15.3. Этот продукт можно бесплатно скачать по ссылке http://www.scrumforteamsystem.com/en/TaskBoard/default.aspx.

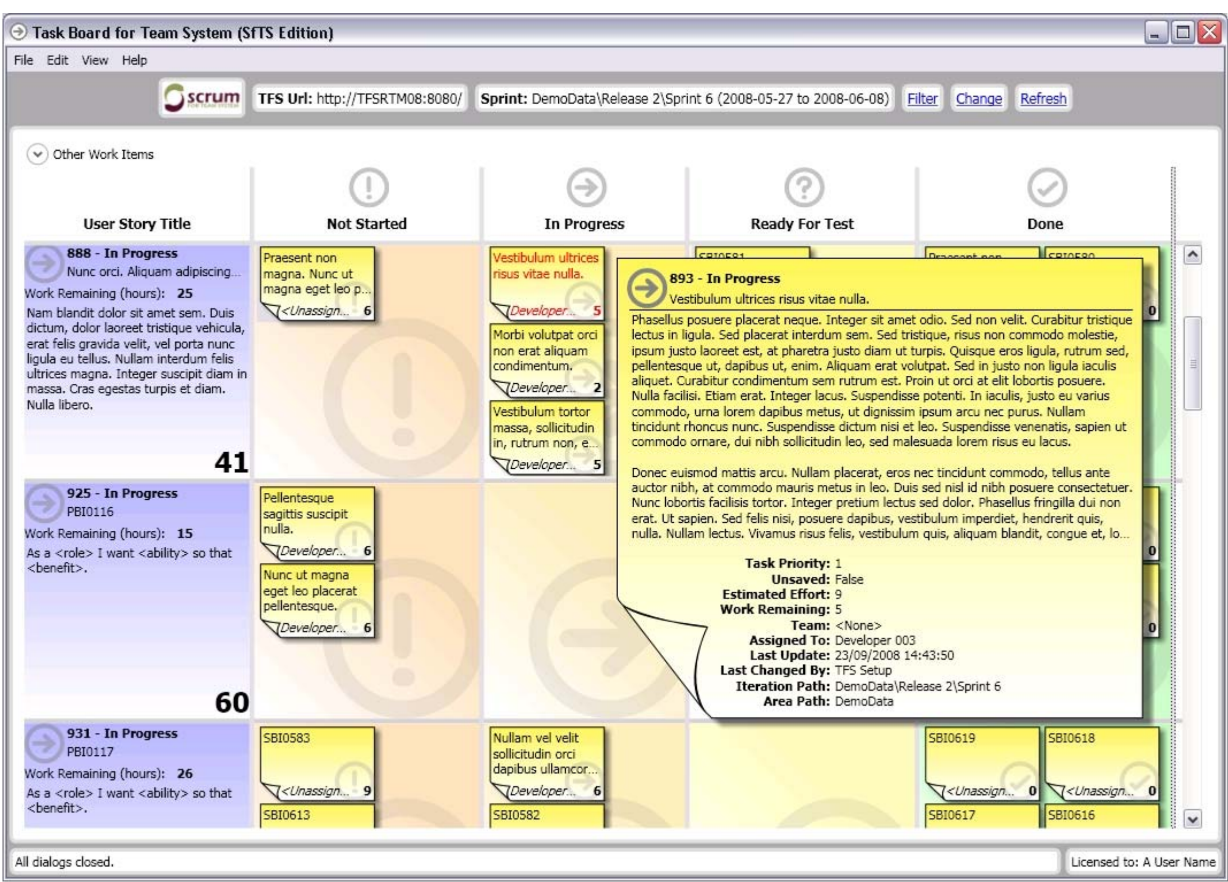

Рис. 15.3. Доска задач.