МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ ТА ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

І. Л. Бородкіна, Г. О. Бородкін

ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Посібник
для студентів вищих навчальних закладів

КИЇВ 2018

УДК 004.41
Б 833

Рекомендовано до друку Головною Вченою Радою
Національного університету біоресурсів та природокористування України
(Протокол № ___ від __ _______ 2018 р.)

Рецензенти:

Б 833 Інженерія програмного забезпечення: Посібник для студентів вищих навчальних закладів / І. Л. Бородкіна, Г. О. Бородкін ; М-во освіти і науки України, Національний університет біоресурсів та природокористування України. – Київ: , 2018. – с.

УДК 004.41

© Бородкіна І. Л., Бородкін Г. О.
© Національний університет біоресурсів та природокористування України

ЗМІСТ

ВСТУП 6

ЧАСТИНА 1. ОСНОВИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ 8

ГЛАВА 1. ПРО ПРЕДМЕТ ВИВЧЕННЯ 8

1.1. Інженерія програмного забезпечення (програмна інженерія) 8
1.2. Програмне забезпечення 11
Контрольні запитання 13

ГЛАВА 2. ПРОЦЕС РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 15

2.1. Процес 15
2.2. Вдосконалення процесу 17
2.3. Класичні моделі процесу 19
Контрольні запитання 26

ГЛАВА 3. РОБОЧИЙ ПРОДУКТ, ДИСЦИПЛІНА ЗОБОВ'ЯЗАНЬ, ПРОЕКТ ТА УПРАВЛІННЯ НИМ 29

3.1. Робочий продукт 29
3.2. Дисципліна зобов'язань 32
3.3. Проект 34
Контрольні запитання 40

ГЛАВА 4. АРХІТЕКТУРА ПЗ 41

4.1. Обговорення 41
4.2. Визначення 42
4.3. Множинність точок зору 44
4.4. Мова UML 48
4.5. Види діаграм 50
4.6. Приклади. 52
Контрольні запитання 56

ГЛАВА 5. УПРАВЛІННЯ ВИМОГАМИ 57

5.1. Проблема 57
5.2. Види і властивості вимог 59
5.3. Варіанти формалізації вимог 63
5.4. Деякі помилки підчас документуванні вимог. 65
5.5. Цикл роботи з вимогами 66
Контрольні запитання 66

ГЛАВА 6. КОНФІГУРАЦІЙНЕ УПРАВЛІННЯ 68

6.1. Проблематика 68
6.2. Одиниці конфігураційного управління 70
6.3. Управління версіями 72
6.4. Управління збірками 74
6.5. Поняття baseline 76
Контрольні запитання 78

ГЛАВА 7. УПРАВЛІННЯ ЯКІСТЮ ТА ТЕСТУВАННЯ 79

7.1. Стандартизація в сучасному бізнесі і промисловості. 79
7.2. Стандартизація якості. 82
7.3. Методи забезпечення якості ПЗ. 83
7.4. Тестування 85
7.5. Види тестування. 90
7.6. Робота з помилками 92
Контрольні запитання 94

ГЛАВА 8. ДІАГРАМНА ТЕХНІКА В РОБОТІ ІЗ ЗНАННЯМИ........... 96

8.1. Метод «випадки використання»..96
8.2. Ітеративний цикл автор/рецензент..103
Контрольні запитання..111

ЧАСТИНА 2. ПРОГРАМНІ ЗАСОБИ ПІДТРИМКИ ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 113

ГЛАВА 9. MICROSOFT SOLUTIONS FRAMEWORK(MSF)............. 113

1.3. Історія і поточний статус..113
1.4. Основні принципи MSF..115
1.5. Модель команди...116
1.6. Інші особливості...121
Контрольні запитання...124

ГЛАВА 10. CAPABILITY MATURITY MODEL INTEGRATION (CMMI) (ІНТЕГРАЦІЯ ЗАВЕРШЕНИХ ПРАЦЕЗДАТНИХ МОДЕЛЕЙ).. 126

10.1. Що таке CMMI?...126
10.2. Рівні зрілості процесів за CMMI...127
10.3. Галузі вдосконалення..129
Контрольні запитання...130

ГЛАВА 11. "ГНУЧКІ" (AGILE) МЕТОДИ РОЗРОБКИ..................... 131

11.1. Загальні положення...131
11.2. Extreme Programming...131
11.3. Scrum...133
Контрольні запитання...137

ГЛАВА 12. ОГЛЯД ТЕХНОЛОГІЇ MICROSOFT VISUAL STUDIO TEAM SYSTEM (VSTS)... 138

12.1. Інженерія програмного забезпечення (програмна інженерія)..........138
12.2. Склад продукту...140
12.3. Правила інсталяції..147
12.4. Пакет Team Explorer...148
Контрольні запитання...150

ГЛАВА 13. VSTS: УПРАВЛІННЯ ЕЛЕМЕНТАМИ РОБІТ (WORK ITEMS).. 152

13.1. Визначення, властивості, життєвий цикл...........................152
13.2. Засоби використання елементів роботи...............................159
Контрольні запитання...172

ГЛАВА 14. VSTS: КОНФІГУРАЦІЙНЕ УПРАВЛІННЯ 174

14.1. Загальні положення...174
14.2. Система контролю версій..174
14.3. Автоматичні збірки..191
Контрольні запитання...205

ГЛАВА 15. VSTS: ТЕСТУВАННЯ.. 207

15.1. Загальні положення...207
15.2. Система відслідковування помилок.....................................207
15.3. Модульні тести..215
15.4. Пакети тестів...216
15.5. Автоматичне тестування Web-додатків...............................220
Контрольні запитання...232

ГЛАВА 16. VSTS: ПІДТРИМКА РІЗНИХ МОДЕЛЕЙ ПРОЦЕСУ 234
16.1. Підтримка шаблонів процесу 234
16.2. Огляд існуючих шаблонів 238
16.3. MSF for Agile Software Development 239
16.4. Scrum 242
Контрольні запитання 244
ЛИТЕРАТУРА 246

ВСТУП

Метою викладання дисципліни “Інженерія програмного забезпечення” є - подати програмну інженерію у вигляді цілісного викладу, концентруючись на концепції процесу, на різних типах методологій розробки ПЗ (CMMI, MSF, Scrum), окремих видах діяльності процесу – розробці архітектури, конфігураційному управлінні, роботі з вимогами, тестуванні. Осторонь залишені питання програмування, оскільки у межах загального курсу їх неможливо ефективно розглянути. Як програмні засоби, що підтримують цілісний процес розробки ПЗ, розглядається технологія компанії Microsoft - Visual Studio Team System (VSTS) з акцентом на Team Foundation Server (TFS). Показується, як викладений вище теоретичний матеріал можна реалізувати на практиці, з підтримкою програмних засобів розробки.

Теоретичний матеріал, наданий в першому розділі, умовно може бути поділений на три частини. Перша частина присвячена питанням процесу розробки програмного забезпечення. Друга частина розглядає питання управління процесом розробки програмного забезпечення. Третя частина присвячена питанням використання діаграмної техніки в роботі із знаннями.

Мета другого розділу даного курсу - ознайомити студентів з програмними засобами, що підтримують цілісний процес розробки ПЗ. Детально розглядається технологія компанії Microsoft - Visual Studio Team System (VSTS) з акцентом на Team Foundation Server (TFS). Показується, як викладений вище теоретичний матеріал можна реалізувати на практиці, з підтримкою програмних засобів розробки.

Матеріал умовно може бути поділений на дві частини. Перша частина присвячена IT рішенням для управління розробкою програмного забезпечення (ПЗ) з використанням різних методів розробки. Друга частина

присвячена огляду технології Microsoft Visual Studio Team System (VSTS) для розробки ПЗ.

Теоретичний матеріал, що подається в посібнику, розрахований на студентів, які вперше вивчають цю дисципліну.

В кінці кожної глави посібника наведені контрольні запитання, які дозволять студентам перевірити ступінь засвоєння викладеного матеріалу.

Розділ 1. Основи програмної інженерії

Глава 1. Про предмет вивчення

Поняття інженерії програмного забезпечення. Основні визначення: інформатика, системотехніка, бізнес-реінжиніринг. Програмне забезпечення: визначення, властивості.

1.1. Інженерія програмного забезпечення (програмна інженерія)

Чим *програмування* відрізняється від програмної інженерії? Тим, що перше є деякою абстрактною діяльністю і може відбуватися в багатьох різних контекстах. Можна програмувати для задоволення, для того, щоб навчитися (наприклад, на практичних заняттях, на семінарах в університеті), можна програмувати в межах наукових розробок. А можна займатися промисловим програмуванням. Як правило, це відбувається в команді, і абсолютно точно – для замовника, який платить гроші за виконану роботу. При цьому необхідно точно розуміти, що потрібне замовникові, виконати роботу в певні терміни і результат повинен бути потрібної якості – такий, який задовольнить замовника і за який він заплатить. Щоб задовольнити цим додатковим вимогам, програмування набуває різних додаткових видів діяльності: розробкою вимог, плануванням, тестуванням, конфігураційним управлінням, проектним менеджментом, створенням різної документації (проектною, призначеною для користувача і ін.).

Розробці програмного коду передує аналіз і проектування (перше означає створення функціональної моделі майбутньої системи без урахування реалізації, для усвідомлення програмістами вимог і очікувань замовника; друге означає попередній *макет*, ескіз, план системи на папері). Трудовитрати на *аналіз* і проектування, а також форма представлення їх

результатів сильно варіюються від видів проектів і переваг розробників і замовників.

Потрібні також спеціальні зусилля *для* організації процесу розробки. У загальному вигляді це ітеративно-інкрементальна модель, за якою необхідна функціональність створюється порціями, яку менеджери і замовник можуть оцінити, і тим самим, є можливість управління перебігом розробки. Проте, ця загальна модель має безліч модифікацій і варіантів.

Розробку системи також необхідно виконувати з урахуванням зручностей її подальшого супроводу, повторного використання і інтеграції з іншими системами. Це означає, що система розбивається на компоненти, які є зручними в розробці, придатними для повторного використання і інтеграції та мають необхідні характеристики, наприклад, *за* швидкодією. Для цих *компонент ретельно* опрацьовуються інтерфейси. Сама ж система документується на багатьох рівнях, створюються правила оформлення програмного коду – тобто залишаються численні семантичні сліди, що допомагають створити, зберегти і підтримувати єдину, струнку архітектуру, одноманітний стиль, порядок.

Всі ці та інші додаткові види діяльності, що виконуються в процесі промислового програмування і є необхідними для успішного виконання замовлень, і називатимемо **програмною інженерією** (software engineering). Так ми позначаємо, по-перше, деяку практичну *діяльність*, а подруге, спеціальну **галузь знання**, або іншими словами, наукову дисципліну. Адже для полегшення виконання кожного окремого проекту, для можливості використовувати різноманітний позитивний *досвід*, що був досягнутий іншими командами і розробниками, цей самий *досвід* піддається осмисленню, узагальненню і належному оформленню. Так з'являються різні методи і практики (*best practices*), саме: тестування, проектування, робота над вимогами, архітектурні шаблони та ін. Також з'являються стандарти і методології, що стосуються всього процесу в цілому (наприклад, *MSF*, *RUP*,

CMMI, Scrum). Ось ці-то узагальнення і входять в програмну інженерію, як у галузь знання.

Необхідність в програмній інженерії, як у спеціальній галузі знань, була усвідомлена світовою спільнотою в кінці 60-х років минулого століття, більш ніж на 20 років пізніше за народження самого програмування, якщо вважати таким фактом появу відомого звіту фон Неймана "*First Draft*of a *Report*onthe EDVAC" («Перший проект звіту на EDVAC»), що був надрукований ним в 1945 році. Народженням програмної інженерії є 1968 рік – конференція *NATO Software Engineering*, м. Гарміш (ФРН), яка цілком була присвячена розгляду цих питань. У сферу програмної інженерії потрапляють всі питання і теми, які пов'язані з організацією і поліпшенням процесу розробки ***ПЗ,*** управлінням колективом розробників, розробкою і впровадженням програмних засобів підтримки життєвого *циклу розробки ПЗ.Програмна інженерія* використовує досягнення інформатики, тісно пов'язана з системотехнікою, часто передує бізнес-реінжинірингом. Трохи докладніше про цей контекст програмної інженерії.

Інформатика (*computer science*) – це комплекс теоретичних наук, які засновані на математиці і присвячені формальним основам обчислювальності. Сюди відносять математичну логіку, теорію граматик, методи побудови компіляторів, математичні формальні методи, що використовуються у верифікації і модельному тестуванні і так далі. Важко точно відокремити програмну інженерію від інформатики, але в цілому спрямованість цих дисциплін різна. *Програмна інженерія* націлена на вирішення проблем виробництва, *інформатика* – на розробку формальних, математизованих підходів до програмування.

Системотехніка (*system engineering*) об'єднує різні інженерні дисципліни *із* розробки всіляких штучних систем – енергоустановок, телекомунікаційних систем, вбудованих систем реального часу і так далі. Дуже часто ПЗ виявляється складовою подібних систем, що виконують завдання управління відповідним устаткуванням. Такі системи називаються *програ-*

мно-апаратними, і беручи участь в їх створенні, програмісти вимушені глибоко розбиратися в особливостях відповідної апаратури.

Бізнес-реінжиніринг (business reengineering) – в широкому сенсі позначає модернізацію бізнесу в певній компанії, впровадження нових практик, що підтримуються відповідними новими інформаційними системами. При цьому акцент може бути як на внутрішній реорганізації компанії, так і на розробці нового клієнтського сервісу (як правило, ці питання взаємозв'язані). Бізнес-реінжиніринг часто передує розробці і впровадженню інформаційних систем на підприємстві, оскільки потрібно спочатку навести певний порядок в діловодстві, а лише потім закріпити його інформаційною системою.

Зв'язок програмної інженерії (як галузі практичної діяльності) з інформатикою, системотехнікою і бізнес-реінжинірингом показаний на **Рис. 1.1.**

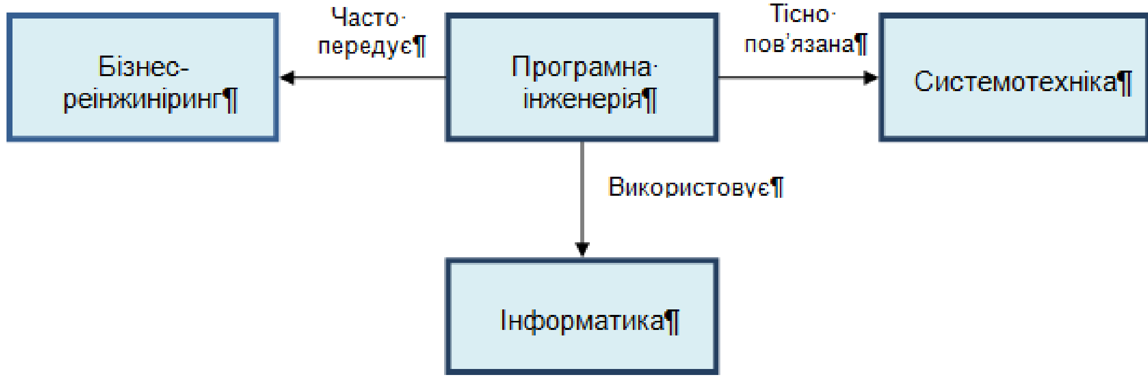

Рис. 1.1. Зв'язок програмної інженерії (як галузі практичної діяльності) з іншими галузями знань

1.2. Програмне забезпечення

Визначення. **Програмне забезпечення** (*ПЗ*) безліч логічних розпоряджень, що розвиваються в часі, за допомогою яких деякий колектив лю-

дей управляє і використовує багатопроцесорну і розподілену систему обчислювальних пристроїв.

Це *визначення*, яке дане Харальдом Мілсом, відомим фахівцем в галузі програмної інженерії з компанії *IBM*, містить в собі таке.

1. Логічні розпорядження – це не лише самі програми, але і різна документація (наприклад, з експлуатації програм) і ширше – певна система відносин між людьми, що використовують ці програми в межах деякого процесу діяльності.

2. Сучасне ПЗ призначене, як правило, для одночасної роботи з багатьма користувачами, які можуть бути значно віддалені один від одного у фізичному просторі. Таким чином, обчислювальне середовище (персональні комп'ютери, сервера і так далі), в якому ПЗ функціонує, виявляється розподіленим.

3. Завдання, які вирішуються сучасним ПЗ, часто вимагають різних обчислювальних ресурсів через різну спеціалізацію цих завдань, із-за великого об'єму виконуваної роботи, а також з міркувань безпеки. Наприклад, з'являється сервер бази даних, сервер додатків і ін. Таким чином, обчислювальне середовище, в якому ПЗ функціонує, виявляється багатопроцесорним.

4. ПЗ розвивається в часі – виправляються помилки, додаються нові функції, випускаються нові версії, міняється його апаратна база.

Властивості. ПЗ є складною динамічною системою, що включає технічні, психологічні і соціальні аспекти. ПЗ помітно відрізняється від інших видів систем, що створюються (створених) людиною, – механічних, соціальних, наукових і ін. і має наступні особливості, виділені Фредеріком Бруксом в його статті "Срібної кулі немає".

1. ***Складність*** програмних об'єктів, яка істотно залежить від їх розмірів. Як правило, більше ПЗ (більша кількість користувачів, більший об'єм оброблюваних даних, жорсткіші вимоги по швидкодії і ін.) з аналогічною функціональністю – це інше ПЗ. Класична наука будувала прості мо-

делі складних явищ, і це вдавалося, оскільки складність не була характеристичною межею даних явищ. (Порівняння програмування саме з наукою, а не з театром, кінематографом, спортом і іншими галузями людської діяльності, виправдано, оскільки воно походить, головним чином, з математики, а перші його плоди – програми – призначалися для використання при наукових розрахунках. Крім того, більшість програмістів мають природничо-наукову, математичну або технічну освіту. Таким чином, парадигми наукового мислення широко використовуються при програмуванні – явно або неявно.)

2. ***Узгодженість*** – ПЗ ґрунтується не на об'єктивних посилках (подібно до того, як різні системи в класичній науці ґрунтуються на постулатах і аксіомах), а повинно бути узгоджено з великою кількістю інтерфейсів, з якими згодом воно повинне взаємодіяти. Ці інтерфейси погано піддаються стандартизації, оскільки ґрунтуються на численних людських угодах, що погано формалізуються.

3. ***Змінність*** – ПЗ легко змінити і, як наслідок, вимоги до нього постійно міняються в процесі розробки. Це створює багато додаткових труднощів при його розробці і еволюції.

4. ***Нематеріальність*** – ПЗ неможливо побачити, воно віртуальне. Тому, наприклад, важко скористатися технологіями, якізасновані на попередньому створенні креслень, що успішно використовуються в інших промислових галузях (наприклад, в будівництві, машинобудуванні). Там на кресленнях в схемному вигляді відтворюються геометричні форми створюваних об'єктів. Коли об'єкт створений, ці форми можна побачити.

Контрольні запитання

1. Що таке програмна інженерія?
2. Назвіть дату зародження програмної інженерії як окремої науки.
3. Охарактеризувати відмінність програмної інженерії від інформатики?
4. Охарактеризувати відмінність програмної інженерії від системотехніки?

5. Наведіть приклади дисциплін інформатики і програмної інженерії (дисципліни не плутати з учбовими предметами).
6. Що таке ПЗ?
7. Перерахувати характеристики ПЗ за Бруксом. Коротко пояснити суть кожної.
8. З якими іншими видами людської діяльності співвідноситься створення ПЗ в даному розділі?

Глава 2. Процес розробки програмного забезпечення

Поняття процесу розробки ПЗ. Універсальний процес. Поточний процес. Конкретний процес. Стандартний процес. Вдосконалення процесу. Pull/push стратегії. Класичні моделі процесу: модель водоспаду, спіральна модель. Фази і види діяльності.

2.1. Процес

Як ми працюємо, яка послідовність наших кроків, які *норми* і правила в поведінці і роботі, який регламент відносин між членами команди, як проект взаємодіє із зовнішнім світом і т.і.? Все це разом ми схильні називати процесом. Його усвідомлення, вибудовування і *поліпшення* – основа будь-якої ефективної групової діяльності. Тому не випадково, що процес виявився одним з основних понять *програмної інженерії*.

Центральним об'єктом вивчення *програмної інженерії* є **процес** створення ПЗ – безліч різних видів діяльності, методів, методик і кроків, що використовуються для розробки і *еволюції ПЗ* і пов'язаних з ним продуктів (проектних планів, документації, програмного коду, тестів, призначеної для користувача документації та ін.).

Проте, на сьогоднішній день не існує **універсального процесу** розробки ПЗ – набору методик, правил і розпоряджень, які б відповідали б ПЗ будь-якого виду, для будь-яких компаній, для команд будь-якої національності. Кожен **поточний процес** розробки, який здійснюється деякою командою в межах *певного проекту*, має велику кількість особливостей і індивідуальностей. Проте доцільно перед початком проекту спланувати процес роботи, визначивши ролі і обов'язки в команді, *робочі продукти* (проміжні і фінальні), порядок участі в їх розробці членів команди і так далі. Називатимемо цей попередній опис **конкретним процесом**, відрізняючи його від плану *робіт*, проектних специфікацій і ін. Наприклад, в системі Microsoft *Visual Team System* опиняється *шаблон* процесу, що створюється або адаптується (у разі використання стандартного) перед початком розро-

бки. У VSTS існують заготовки для конкретних процесів на базі *CMMI*, Scrum і ін.

В межах окремої компанії можлива і корисна стандартизація всіх поточних процесів, яку називатимемо **стандартним процесом**. Останній, таким чином, виявляється деякою базою даних, що містить такі дані:

- інформацію, правила використання, документацію і інсталяційні пакети *засобів розробки*, які використовуються в проектах компанії (систем версійного контролю, засобів контролю помилок, засобів програмування – різних *IDE*, *СУБД* і так далі);
- опис практик розробки – проектного *менеджменту*, правил роботи із замовником і т.д.;
- шаблони проектних документів – *технічних завдань*, проектних специфікацій, планів тестування і так далі і ін.

Також можлива стандартизація процедури розробки конкретного процесу як частки стандартного. Основна ідея стандартного процесу – курсування усередині компанії передового досвіду, а також *уніфікація засобів розробки*. Дуже вже часто в компаніях різні департаменти і проекти сильно відрізняються *за зрілістю процесу розробки*, що ускладнює повторне використання передового досвіду. Крім того, трапляється, що компанія використовує декілька засобів паралельних інструментів розробки, наприклад, *СУБД* засобу версійного контролю. Іноді це буває виправдано (наприклад, такі вимоги *замовника*), часто це необхідно, наприклад, *Java .NET* (компетентність офшорної компанії дозволяє їй обирати ширший спектр замовлень). Але дуже часто це довільний вибір самих розробників. У будь-якому випадку, така множинність істотно ускладнює міграцію фахівців з проекту в проект, використання результатів одного проекту в іншому і так далі. Проте, у разі організації стандартного процесу необхідно стежити, щоб стандартний процес не виявився лише формальним, бюрократичним апаратом. Поняття стандартного процесу введене і детально описане в підході *CMMI*.

Необхідно відзначити, що наявність стандартного процесу свідчить про наявність "єдиної волі" в організації, яка існує саме на рівні процесу. На рівні продажу, бухгалтерії та інших звичних для всіх компаній процесів і *активів* єдність здійснити не важко. А ось на рівні процесів розробки дуже часто кожен проект виявляється сам *по* собі (особливо в офшорних проектах) –буденність захоплює і ізолює проекти один від одного дуже міцно.

2.2. Вдосконалення процесу

Визначення. Вдосконалення процесу (*software process improvement*) – це *діяльність щодо* змін існуючого процесу (як поточного, в межах одного проекту, так і стандартного, для всієї компанії) з метою поліпшення якості створюваних продуктів і/або зниження ціни і часу їх розробки. Причини актуальності цієї діяльності для компаній-виробників полягає в наступному.

1. Відбувається швидка зміна технологій розробки ПЗ, потрібні вивчення і впровадження нових засобів розробки.
2. Спостерігається швидке зростання компаній і їх вихід на нові ринки, що вимагає нової організації робіт.
3. Має місце висока конкуренція, яка вимагає пошуку ефективніших, більш економічних способів розробки.

Що і яким чином можна покращувати.

1. Перехід на нові засоби розробки, *мови програмування* і так далі
2. *Поліпшення* окремих управлінських і інженерних практик – тестування, *управління вимогами* і ін.
3. Повна, комплексна перебудова всіх процесів в проекті, департаменті, компанії (у відповідності, наприклад, до*CMMI*).
4. Сертифікація компанії (*CMM/CMMI*, *ISO 9000* і ін.).

Ми відокремили п. 3 від п. 4 тому, що на практиці 4 далеко не завжди означає дійсну творчу роботу *із* поліпшення процесів розробки *ПЗ,* а часто зводиться до підтримки відповідного документообігу, який є необ-

хідним для отримання сертифікації. Сертифікат потім використовується як засіб, козир в боротьбі за замовлення.

Головна важкість реального вдосконалення процесів в компанії полягає в тому, що вона при цьому повинна працювати і створювати *ПЗ,* її не можна "засинити на переоблік".

Звідси витікає ідея безперервного поліпшення процесу, так би мовити, малими порціями, щоб процедура була не такою хворобливою. Це є тим більш логічним, що нові технології розробки, які з'являються на ринку, а також розвиток тих, що вже існують потрібно постійно відслідковувати. Ця стратегія, зокрема, отримала віддзеркалення в стандарті вдосконалення процесів розробки *CMMI*.

Pull/push стратегії. У контексті впровадження *інновацій* у виробничі процеси бізнес-компаній (не обов'язково компанії для створення *ПЗ*) існують дві *парадигми*.

1. *Organization pull – інновації*, які націлені на вирішення конкретних проблем компанії.
2. *Technology push* – широкомасштабне впровадження *інновацій* із стратегічних міркувань. Замість конкретних проблем, які будуть вирішені після впровадження *інновації*, в цьому випадку розглядаються показники компанії (ефективність, *продуктивність*, річний оборот засобів, збільшення *вартості акцій* публічної компанії), які будуть збільшені, покращені після впровадження *інновації*. При цьому передбачається, що будуть автоматично вирішені численні окремі проблеми організації, у тому числі і ті, про які в даний момент нічого не відомо.

Приклад використання стратегії *organization pull* – впровадження нових засобів тестування за ситуації, коли є високі вимоги *до якості* в проекті, або коли якість програмної системи не задовольняє замовника.

Приклад використання стратегії *technology push* – перехід компанії із засобів структурної розробки на об’єктно-орієнтоване програмування. Ще один приклад використання тієї ж стратегії – впровадження стандартів

якості ISO *9000* або CMMI. У обох цих випадках компанія не вирішує якусь одну проблему або ряд проблем – вона хоче радикально змінити ситуацію, вийти на нові позиції і так далі.

Проблеми застосування стратегії *technology push* полягають в тому, що потрібна глобальна перебудова процесу. Але компанію не можна "закрити на реконструкцію" – за цей час місце компанії на ринку може бути зайняте конкурентами, акції компанії можуть впасти і так далі. Таким чином, впровадження *інновацій*, як правило, відбувається паралельно із звичайною діяльністю компанії, поетапно, що у випадку з *technology push* зв'язано з великими труднощами і ризиками.

Використання стратегії *organization pull* менш ризиковане, зміни, що вносяться нею в процес, менш глобальні, локальні. Але і переваг такі *інновації* приносять менше, *в порівнянні з* вдалими впровадженням відповідно до стратегії *technologypush*.

Необхідно також відзначити, що існують проблеми, які неможливо усунути точковими переробками процесу, тобто необхідно застосовувати стратегію *technology push*. Наведемо процес супроводу і розвитку сімейства *програмних продуктів*, який, наприклад, зайшов у *безвихідь,* – компанія стає збитковою, якщо і далі буде супроводжувати встановлені раніш продукти, інструментальні засоби проекту безнадійно застаріли і знаходяться в жалюгідному стані, *менеджмент* засмучений, всі спроби керівництва змінити процес натрапляють на нерозуміння колективу, сварки і *конфлікти*. Можливо, що у такому разі без "революції" не обійтися.

Ще одна відмінність обох стратегій: у випадку з *organization pull*, як правило, повернення інвестицій від впровадження відбувається швидше, ніж у випадку з *technology push*.

2.3. Класичні моделі процесу

Визначення моделі процесу. Процес створення *програмного забезпечення* не є однорідним. Той або інший метод розробки *ПЗ,* як правило,

визначає деяку динаміку *розгортання* тих або інших видів діяльності, тобто, визначає модель *процесу* (*process model*).

Модель є доброю *абстракцією* різних методів розробки *ПЗ,* яка дозволяє лаконічно, стисло і інформативно їх унаочнити. Проте, сама ідея *моделі процесу* є однією з найбільш ранніх в *програмній інженерії*, коли вважалося, що вдала модель – це найголовніше, що сприяє успіху розробки. Пізніше прийшло усвідомлення, що існує безліч інших аспектів (принципи управління і розробки, структура команди і так далі), які повинні бути узгоджені один з одним. І тоді почали розвиватися інтегральні *методології розробки*. Проте, існує декілька класичних моделей процесу, які корисні на практиці і на яких слід зупинитись.

Фази і види діяльності. Кажучи про моделі процесів, необхідно розрізняти фази і види діяльності.

Фаза (*phase*) – це певний етап процесу, який має початок, кінець і вихідний результат. Наприклад, фаза перевірки здійсненності проекту, здача проекту і так далі. Фази слідують одна за одною в лінійному порядку, характеризуються наданням звітності замовникові і, часто, виплатою грошей за виконану частину роботи.

Поодинокий замовник погодиться перший раз побачити результати лише після завершення проекту. З іншого боку, підрядчики вважають за краще отримувати гроші поступово, *у міру того, як* виконуються окремі частини роботи. Таким чином, з'являються фази, що дозволяють створювати і пред'являти проміжні результати проекту. Фази корисні також безвідносно взаємодії із замовником – з їх допомогою можна синхронізувати *діяльність* різних *робочих груп*, а також відслідковувати перебіг проекту. Прикладами фаз може служити узгодження із замовником технічного завдання, реалізація певної функціональності *ПЗ,* етап розробки, що закінчується здачею системи на тестування або випуском альфа-версії.

Вид діяльності (*activity*) – це певний тип роботи, що виконується в процесі розробки *ПЗ.* Різні види діяльності часто вимагають різних профе-

сійних навичок і виконуються різними фахівцями. Наприклад, *управління проектом виконується менеджером проекту*, *кодування – програмістом, тестування –тестувальником*. Є види діяльності, які можуть виконуватися одними і тими самими фахівцями, – наприклад, *кодування* і *проектування* (особливо в невеликому проекті) часто виконують одні і ті ж самі люди.

В межах однієї фази може виконуватися багато різних видів діяльності. Крім того, один вид діяльності може виконуватися на різних фазах – наприклад, тестування: на фазі аналізу і проектування можна писати тести і налагоджувати тестове оточення, а саме тестування проводити під час розробки і перед здачею. На поточний момент для складного *програмного забезпечення* використовуються *багатовимірні моделі* процесу, в яких відокремлення фаз від видів діяльності істотно полегшує управління розробкою *ПЗ*.

Види діяльності, фактично, присутні, під різними назвами, в кожному методі розробки *ПЗ*. У *RUP* вони мають назву робочі *процеси* (*work flow*), в *CMM* – ключовими галузями процесу (key *process area*). Слід зберігати традиційні назви, які прийняті в тому або іншому методі, щоб не створювати плутанини.

Модель водоспаду була запропонована в 1970 році Вінстоном Ройсом. Фактично, вперше в процесі розробки ПЗ були виділені різні етапи розробки і змінені примітивні уявлення про розробку ПЗ у вигляді аналізу системи і її кодування.

Були визначені такі етапи: розробка *системних вимог*, розробка вимог до *ПЗ, аналіз*, проектування, *кодування*, тестування, експлуатація і супровід (**Рис. 2.1)**.

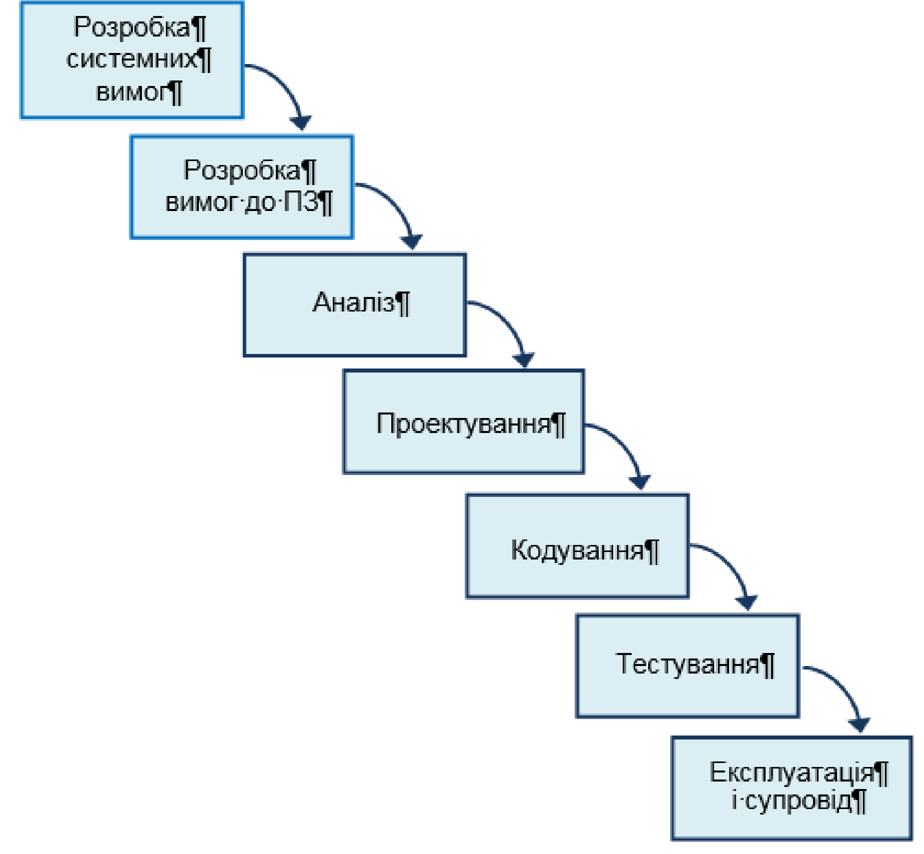

Рис. 2.1. Модель водоспаду

Іноді каскадну модель зображають з іншою послідовністю етапів розробки ПЗ (**Рис. 2.2**).

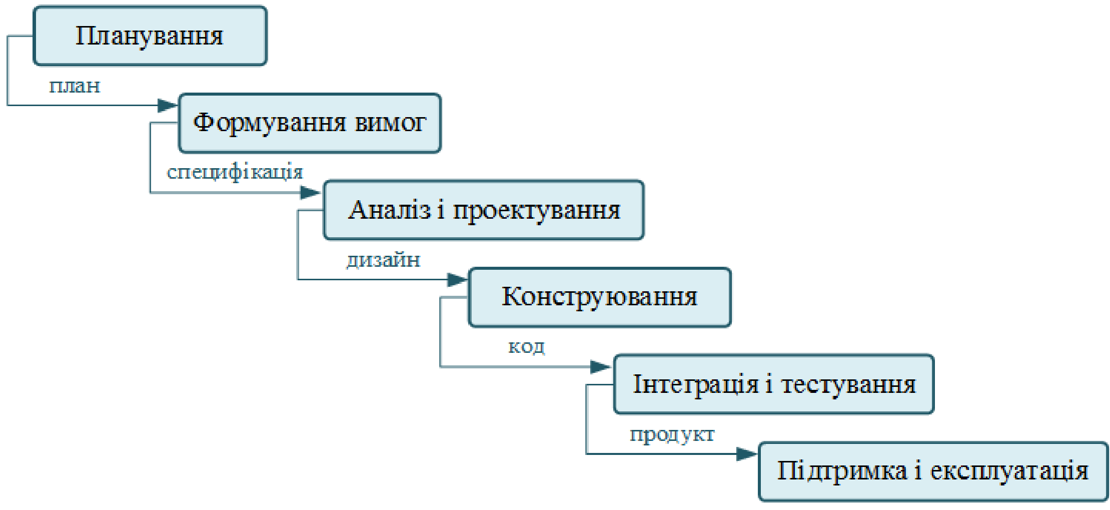

Рис. 2.2. Альтернативна модель водоспаду

Перевагою цієї моделі є обмеження можливості повернення на довільну кількість кроків назад, наприклад, від тестування – до аналізу, від розробки – до роботи над вимогами і так далі. Наголошувалося, що такі повернення можуть катастрофічно збільшити вартість проекту і терміни його виконання. Наприклад, якщо при тестуванні виявляються помилки проектування або аналізу, то їх виправлення часто призводить до повної переробки системи. Цією моделлю допускалися повернення лише на попередній крок, наприклад, від *тестування* до *кодування*, від *кодування* до *проектування* і так далі.

Нарешті, в межах цієї моделі було введено прототипіювання, тобто пропонувалося розробляти систему двічі, аби зменшити ризики помилок підчас розробки. Перша версія – прототип – дозволяє побачити основні ризики і обґрунтовано ухвалити головні архітектурні рішення. На створення прототипу відводилося до однієї третини часу всієї розробки.

У 70-80 роках минулого століття ця модель через свою простоту і схожість з моделями розробки інших, не програмних, систем широко використовувалася в процесі розробки ПЗ. Надалі, у зв'язку з розвитком *програмної інженерії* і усвідомленням ітеративного характеру процесу розробки ПЗ, ця модель активно критикувалася, практично, кожним автором відповідних статей і підручників. Стала загальноприйнятою думка, що вона не відображає особливостей розробки *ПЗ.* Недоліками моделі водоспаду є:

- ототожнення фаз і видів діяльності, що призводить до втрати гнучкості розробки, зокрема, до труднощів підтримки ітеративного процесу розробки;
- вимога повного закінчення фази-діяльності, закріплення результатів у вигляді докладного відповідного документа (технічного завдання, проектної специфікації); проте досвід розробки ПЗ показує, що неможливо цілком завершити розробку вимог, дизайн системи і так далі – все це вимагає змін; і причини тут не лише в тому, що оточення проекту є змінним, але і в тому, що заздалегідь важко точно визначити і

сформулювати багато рішень, вони з'ясовуються і деталізуються лише згодом;

- інтеграція всіх результатів розробки відбувається в кінці проекту, унаслідок чого інтеграційні проблеми дають про себе знати занадто пізно;
- користувачі і замовник не можуть ознайомитися з варіантами системи під час розробки, а бачать результат лише в самому кінці; тим самим, вони не можуть вплинути на процес створення системи, і тому збільшуються ризики непорозуміння між розробниками і користувачами або замовником;
- модель є нестійкою до збоїв у фінансуванні проекту або перерозподілу грошових коштів, розробка яку розпочали, фактично, не має альтернатив "за перебігом справи".

Проте дана модель продовжує використовуватися на практиці – для невеликих проектів або при розробці типових систем, де ітеративність не є важливою. З її допомогою зручно відстежувати розробку і здійснювати поетапний *контроль* за проектом. Ця модель також часто використовується в офшорних проектах з почасовою оплатою праці. Модель водоспаду увійшла як складова частина в інші моделі і методології, наприклад, в *MSF*.

Спіральна модель була запропонована Бері Боемом в 1988 році для подолання недоліків моделі водоспаду, перш за все, для кращого управління ризиками. Згідно цієї моделі розробка продукту здійснюється у вигляді спіралі, кожен виток якої є певною фазою розробки. На відміну від моделі водоспаду в спіральній моделі немає зумовленого і обов'язкового набору витків, кожен виток може стати останнім підчас розробки системи, а після його завершення складаються плани наступного витка. Нарешті, виток є саме фазою, а не видом діяльності, як в моделі водоспаду, в його межах може здійснюватися багато різних видів діяльності, тобто модель є двовимірною (Рис. 2.3).

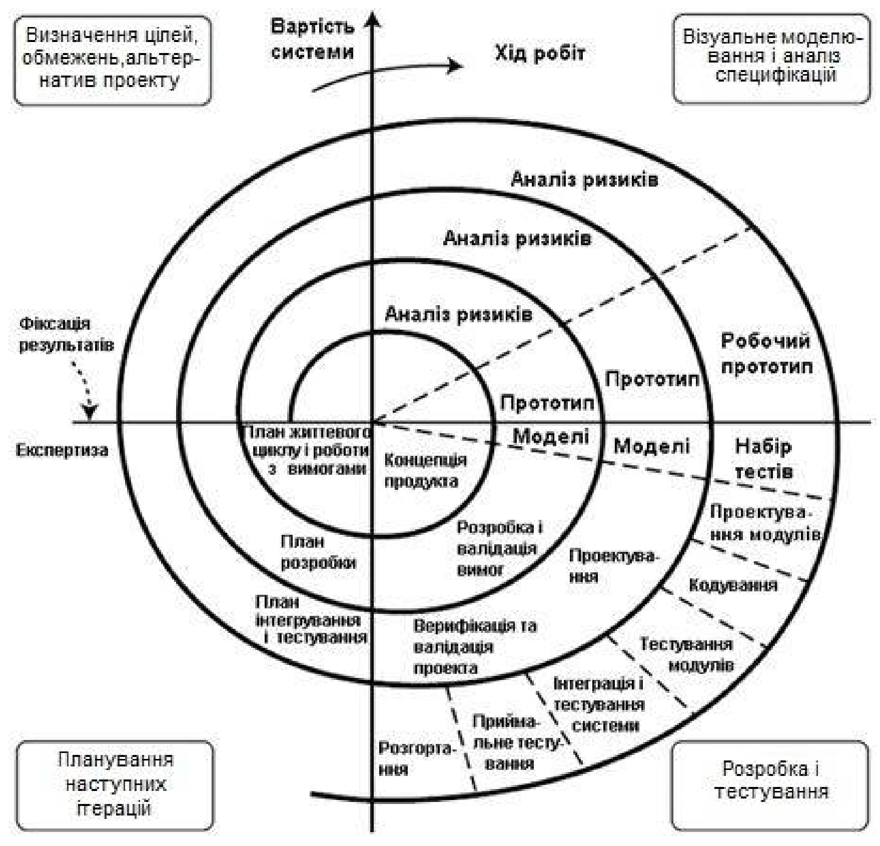

Рис. 2.3. Модель спірального процесу розробки ПЗ

Послідовність витків може бути такою: на першому витку ухвалюється рішення про доцільність створення *ПЗ,* на наступному визначаються *системні вимоги*, потім здійснюється проектування системи і так далі. Витки можуть мати і інші значення.

Кожен виток має таку структуру (сектори):

- визначення цілей, обмежень і альтернатив проекту;
- оцінка альтернатив, оцінка і дозвіл ризиків; можливе використання прототипіювання (зокрема створення серії прототипів), симуляція системи, візуальне моделювання і аналіз специфікацій; фокусування на самих ризикових частинах проекту;

- розробка і тестування – тут можлива модель водоспаду або використання інших моделей і методів розробки ПЗ;
- планування наступних ітерацій – аналізуються результати, плани і ресурси для подальшої розробки, ухвалюється (або не приймається) рішення про новий виток; аналізується, чи має сенс продовжувати розробляти систему чи ні; розробку можна і припинити, наприклад, через збій у фінансуванні; спіральна модель дозволяє зробити це коректним чином.

Окрема спіраль може відповідати розробці деякою програмною компоненти або внесенню чергових змін до продукту. Таким чином, у моделі може з'явитися третій вимір.

Спіральну модель недоцільно застосовувати в проектах з невеликим ступенем ризику, з обмеженим бюджетом, для невеликих проектів. Крім того, відсутність хороших засобів прототипіювання може також зробити незручним використання спіральної моделі.

Спіральна модель не знайшла широкого застосування в індустрії і важлива, швидше, лише в історико-методологічному плані: вона є першою ітеративною моделлю, має красиву метафору – спіраль, – і, подібно до моделі водоспаду, застосовувалася надалі у випадку створення інших моделей процесу і методології розробки *ПЗ.*

Контрольні запитання до Глави 2

1. Що таке процес створення ПЗ?
2. Охарактеризувати причини відсутності *універсального* процесу розробки ПЗ.
3. Чому можливо і доцільно стандартизувати процес на рівні компанії?
4. Що таке стандартний процес?
5. Що таке конкретний процес?
6. Як стандартний і конкретний процеси співвідносяться між собою?
7. Чим відрізняються між собою поточний і конкретний процеси?

8. Які *методології розробки ПЗ* підтримують поняття конкретного процесу і якими засобами?
9. Дати визначення діяльності із вдосконалення процесу.
10. У чому головна важкість вдосконалення процесів в компаніях?
11. Перерахувати основні напрями *поліпшення* процесу.
12. Охарактеризувати стратегію *organization pull* до впровадження *інновацій*. Навести приклади.
13. Охарактеризувати стратегію *technology push* до впровадження *інновацій*. Навести приклади.
14. Охарактеризувати переваги *organization pull.*
15. Охарактеризувати недоліки *organization pull.*
16. Охарактеризувати можливі ризики *organization pull*.
17. Охарактеризувати переваги *technology push.*
18. Охарактеризувати недоліки *technology push.*
19. Охарактеризувати можливі ризики *technology push*.
20. Що таке *модель процесу*?
21. Що таке фаза процесу?
22. Що таке вид діяльності?
23. Чому не можна ототожнювати фази і види діяльності?
24. Коли і за яких причин на практиці відбувається ототожнення фаз і виду діяльності?
25. Охарактеризувати переваги моделі водоспаду?
26. Охарактеризувати історичну роль моделі водоспаду?
27. Охарактеризувати недоліки моделі водоспаду?
28. Як в рамках моделі водоспаду передбачається працювати з ризиками?
29. Чому модель водоспаду використовується дотепер?
30. Чому модель водоспаду зручно використовувати в офшорних проектах з почасовою *оплатою*?
31. Чим виток *спіральної моделі* відрізняється від фази в моделі водоспаду?
32. Навести приклад послідовності витків спіральної *моделі*.

33. За яких умов спіраль спіральної *моделі* завершується.
34. Охарактеризувати друге і третє вимірювання *спіральної моделі*. Опишіть різні сектори витка спіралі.
35. Охарактеризувати переваги *спіральної моделі*?
36. Охарактеризувати недоліки *спіральної моделі*?
37. Які обмеження *спіральної* моделі?
38. Як в рамках *спіральної* моделі передбачається працювати з ризиками?

Глава 3. Робочий продукт, дисципліна зобов'язань, проект та управління ним

Робочий продукт. Дисципліна зобов'язань. Проект. Управління проектами.

Через творчий характер програмування, молодий вік учасників розробки ПЗ, виявляються актуальними деякі питання звичайного промислового виробництва, що стали давно загальним надбанням. Перш за все, це дисципліна зобов'язань і робочий продукт. Дані знання, коли залучені у практику, є надзвичайно корисними в командній роботі. Крім того, широко вживані зараз на практиці методології розробки ПЗ, що підтримані відповідним програмним інструментарієм, активно використовують ці поняття, уточнюючи і конкретизуючи їх.

3.1. Робочий продукт

Однією з істотних умов для керованості промисловим процесом є наявність окремо оформлених результатів роботи – як в остаточному вигляді, так і проміжних. Ці окремі результати у складі загальних результатів робіт допомагають ідентифікувати, планувати і оцінюватирізні частини результату. Проміжні результати допомагають менеджерам різних рівнів відслідковувати процес втілення проекту, замовник отримує можливість ознайомитися з результатами задовго до закінчення проекту. Більш того, самі учасники проекту в своїй щоденній роботі отримують простий і ефективний спосіб обміну робочою інформацією, а саме – обмін результатами.

Таким результатом є робочий продукт (work product) – будь-який артефакт, який був створений в процесі розробки ПЗ, наприклад, файл або набір файлів, документи, складові частини продукту, сервіси, процеси, специфікації, рахунки і так далі (Рис. 3.1).

Ключова різниця між робочим продуктом і компонентом ПЗ полягає в тому, що перший необов'язково матеріальний і відчутний (not to be engineered), хоча і може бути таким. Нематеріальний робочий продукт – це, як правило, деякий налагоджений процес – промисловий процес виробництва якої-небудь продукції, учбовий процес в університеті (на факультеті, на кафедрі) і так далі.

Важливо відзначити, що робочий продукт зовсім не обов'язково є складовою частиною підсумкового постачання. Наприклад, налагоджений процес тестування системи не поставляється замовникові разом з самою системою. Уміння управляти проектами (не лише в галузі програмування) багато в чому пов'язане з мистецтвом визначати потрібні робочі продукти, наполягати на їх створенні і в їх термінах вести приймання проміжних етапів роботи, організовувати синхронізацію різних робочих груп і окремих фахівців.

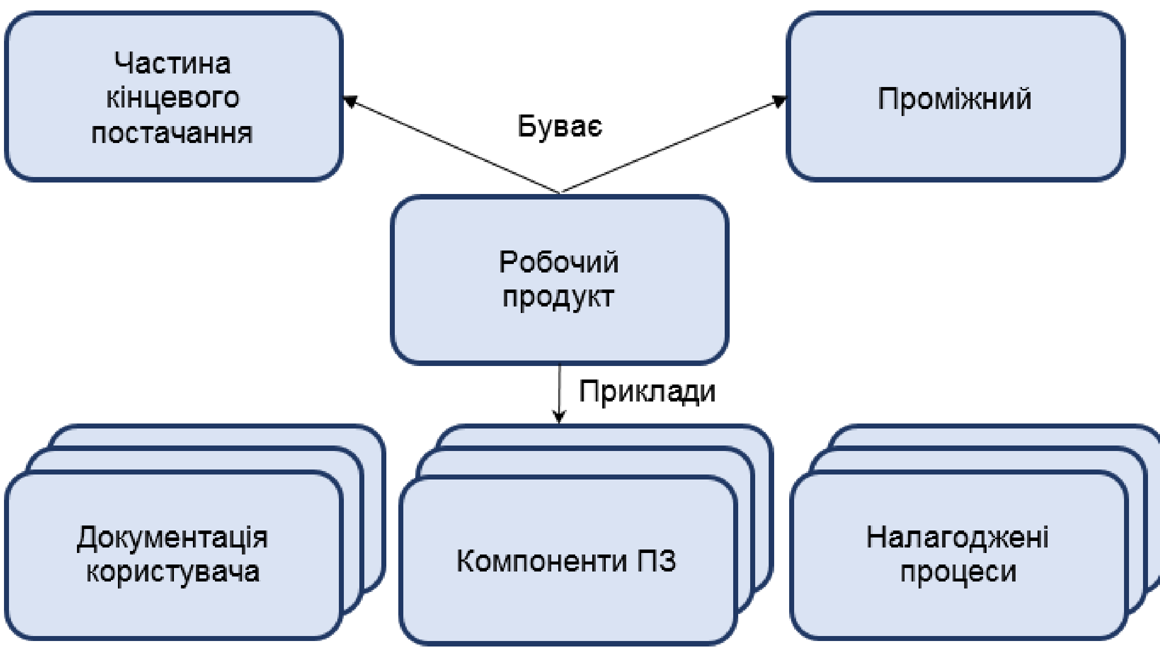

Рис. 3.1. Види типового робочого продукту

Багато методології включають опис специфічних робочих продуктів, які використовуються в процесі, – *CMMI*, *MSF*, *RUP* і ін. Наприклад, в *MSF* це програмний код, *діаграми* додатків і класів (*application diagrams* і *class diagrams*), план *ітерацій* (*iteration plan*), *модульний* тест (*unit test*) і ін. Для кожного з них обов'язково визначено зміст, відповідальні за розробку, *місце в* процесі та інші аспекти.

Зупинимося трохи детальніше на проміжних робочих продуктах. Компонента ПЗ, яка створена в проекті одним розробником і надана для використання іншому розробникові, виявляється робочим продуктом. Її треба мінімально протестувати, виправити назви класів інтерфейсів і методів, прибрати зайве, (таке, що не має відношення до функціональності даної компоненти) розділити на public і private, і так далі. Тобто, виконати деяку додаткову роботу, яку, мабуть, розробник і не почав би робити, якби продовжував використовувати компоненту лише сам. Об'єм цих додаткових робіт суттєво зростає, якщо компонента повинна бути надана для використання в розробці, наприклад, в інший центр розробки (іноземним партнерам, що є звичною ситуацією для офшорної розробки). Виготовлення гарних проміжних робочих продуктів є дуже важливим для успішності проекту, але вимагає додаткової роботи від їх авторів. Працювати одному, і не надавати робочих продуктів – легше, а для багатьох і зручно. Але робота в команді вимагає накладних витрат, у тому числі і у вигляді витрат на створення проміжних робочих продуктів. Звичайно, якість цих продуктів і трудовитрати на їх виготовлення дуже варіюються залежно від ситуації, але тут важливо розуміти сам принцип.

Отже, підсумуємо, що проміжний робочий продукт повинен обов'язково мати ясну мету і конкретних користувачів, щоб мінімізувати накладні витрати на його створення.

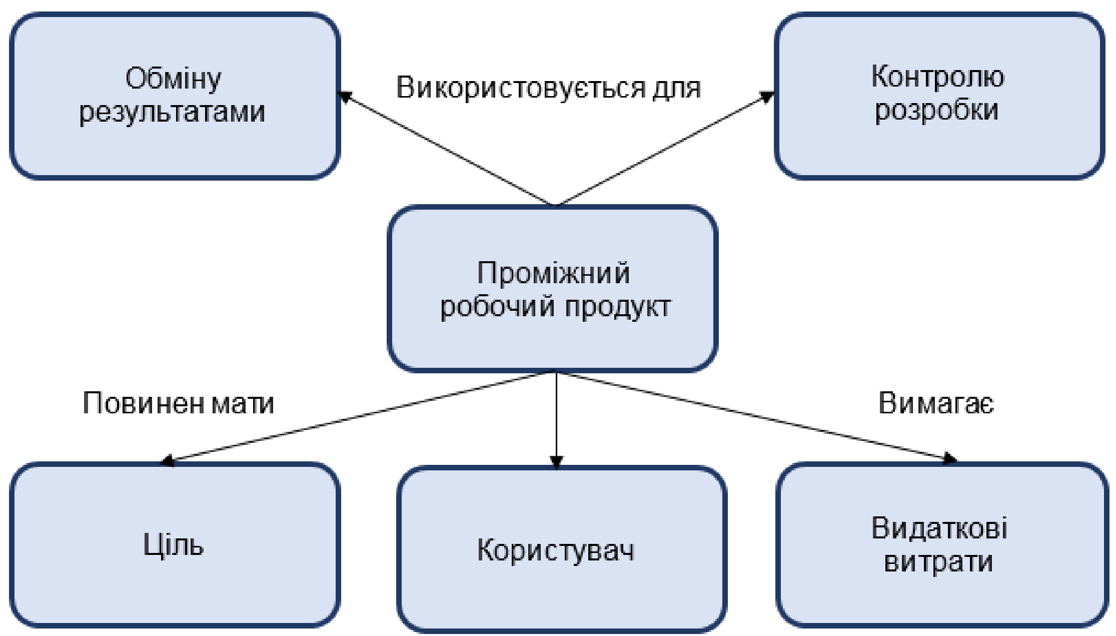

Рис. 3.2 Атрибути проміжного робочого продукту

3.2. Дисципліна зобов'язань

В основі розподілу обов'язків в бізнесі і в промисловому виробництві, формування корпоративних правил і норм лежить певна ділова етика або форма відносин – дисципліна зобов'язань. Вона широко використовується на практиці і є однією з можливих форм соціального взаємовідношення між людьми. Привнесення в бізнес і промисловість інших моделей людських відносин – сімейних, сексуальних, дружніх і так далі, часто наносить справам серйозні збитки, породжує конфліктність, знижує ефективність.

Основою такої форми відносин, як дисципліна зобов'язань,є зобов'язання, які:

- даються добровільно;

- не даються легко – робота, ресурси, розклад повинні бути ретельно враховані;
- між сторонами включає те, що буде зроблене, ким і в які терміни;
- відкрито і публічно сформульовані (тобто це не "таємне знання").

Крім того:

- відповідальна сторона прагне виконати зобов'язання, навіть якщо потрібна допомога;
- *до* настання *deadline*, як лише стає очевидно, що робота не може бути закінчена у визначений термін, обговорюються нові зобов'язання.

Відзначимо, що дисципліна зобов'язань не є якимсь зведенням правил і законів, вона також відрізняється і від корпоративної культури. Це – певний груповий психічний феномен, що існує в товаристві сучасних людей. Наведені вище пункти не є вичерпним описом цього феномену, але лише проявляють і позначають його, так би мовити, викликають потрібні спогади.

Дисципліна зобов'язань, хоч і здається цілком очевидною, деколи, не просто реалізується на практиці, наприклад, в творчих галузях людської діяльності, в галузі навчання і так далі. Існують окремі люди, яким ця дисципліна внутрішньо є неприйнятною незалежно від роду їх діяльності.

З іншого боку, люди, які освоїли цю дисципліну, часто прагнуть застосовувати її в інших галузях життя і людських відносин, що виявляється не завжди виправданим. Підкреслимо, що дана дисципліна є далеко не єдиною моделлю відносин між людьми. Як приклад, можна розглянути відносини в сім'ї або дружбу, що, з очевидністю, не можуть бути виражені

дисципліною зобов'язань. Так, замість точності і пунктуальності в цих відносинах важливим є емоційно-психологічне співпереживання, без якого вони неможливі.

Дисципліні зобов'язань приділяється багато уваги в межах MSF (Microsoft Solutions Framework(MSF)— методологія розробки програмного забезпечення, запропонована корпорацією Microsoft. MSF спирається на практичний досвід Microsoft і описує управління людьми і робочими процесами в процесі розробки рішення. MSF є узгоджений набір концепцій, моделей і правил), оскільки там в моделі команди немає лідера, начальника. Ця дисципліна реалізована також в Scrum (підхід управління проектами для гнучкої розробки програмного забезпечення. Скрам чітко робить акцент на якісному контролі процесу розробки): Scrum-команда має багато свобод, і через це – велику відповідальність. Регламентуються також правила дій, коли зобов'язання не можуть бути виконані такою командою.

3.3. Проект

Класичний операційний розподіл праці, що йде ще від Адама Сміта, (Ада́м Сміт *(англ. Adam Smith; *5 червня 1723 Керкколд, Шотландія — †17 липня 1790 Единбург, Шотландія) — шотландський економіст і філософ-етик; засновник сучасної економічної теорії. Головна праця — «Дослідження про природу і причини багатства народів» (англ.* ***An Inquiry into the Nature and Causes of the Wealth of Nations),*** *видана 1776 року. У цій праці Сміт з високою деталізацією систематизує сучасні йому ідеї. «Поділ праці повинен сприяти та розвивати продуктивність робочої сили найкраще», так розпочинає Сміт свою працю об'ємом понад 1000 сторінок. Особлива увага приділяється розгляду* ***Переваги поділу праці.*** *Причиною багатства окремого народу автор вважає розповсюдження поділу праці. Отже, на відміну від фізіократів, які вважали продуктивність властивістю лише землі/природи, саме працю Сміт вважає тим, що забезпечує добробут народу, оскільки забезпечення товарами та послугами стало кращим, проте, якість землі та клімат не змінилися протягом довгого часу. Такий феномен пояснює економічне зростання (як згодом і модель Солоу). Кращими стають уміння, розуміння під час здійснення певного виду роботи. Це, в свою чергу, умож-*

ливлює саме поділ праці. Він наводить приклад з голками, згідно з яким один працівник може максимально в день виготовляти 20 голок, а 10 працівників, кожний з яких спеціалізується на певному відрізку праці — 48 000 за той самий проміжок часу. Переваги поділу праці можна реалізувати лише в обмінній економіці і настільки ефективно, наскільки функціонує ринок. Сміт резюмує: чим більший розмір ринку, тим більш розвинутим є поділ праці.) і є суттю масового індустріального виробництва. Тобто, існує чітко налагоджений процес роботи і є галузі спеціалізації – один цех точить, інший калібрує, третій збирає, четвертий фарбує і так далі. Продуктивна спроможність такого виробництва набагато перевершує виконання всієї роботи однією людиною або однією групою. Таким чином в XIX столітті операційний розподіл праці став основою мануфактури, що витіснила індивідуальне, ремісниче виробництво. На початок XX століття цю структуру робіт перенесли і на управління – тобто численні менеджери контролювали окремі ділянки робіт.

Проте, високий рівень складності ряду завдань в промисловості і бізнесі не дозволяє (на щастя!) так працювати усюди. Існує багато творчих, нових завдань, де, мабуть, в майбутньому і з'явиться можливість створити конвеєри, але в даний момент для їх вирішення потрібна істотна концентрація сил і *енергії* людей, несподівані рішення, а також успіх і легка рука. Це і є галузь проектів.

Проект – це унікальна (у відмінності від традиційної післяопераційного промислового виробництва) *діяльність*, що має:

- початок і кінець у часі,
- спрямована на досягнення певного результату/цілі, створення певного унікального продукту або послуги,
- задані обмеженнях на ресурси і терміни,
- чіткі вимоги до якості і допустимого рівня ризику.

Зокрема, розробка програмного забезпечення, є, переважно, проектною галуззю.

Необхідно розрізняти проекти промислові і проекти творчі. У них різні принципи управління. Складність промислових проектів – у великій кількості різних організацій, компаній і відносної унікальності самих робіт. Приклад – будівництво багатоповерхового будинку. Сюди ж відносяться різні міжнародні проекти і не лише промислові – освітні, культурні і ін. Завдання в управлінні такими проектами – це все охопити, все проконтролювати, нічого не забути, все звести воєдино, добитися руху, причому рухи узгодженого.

Творчі проекти характеризуються абсолютною новизною ідеї – новий сервіс, абсолютно новий програмний продукт, якого ще не було на ринку, проекти в галузі мистецтва і науки. Будь-який початкуючий бізнес, як правило, є таким ось творчим проектом. Причому новизна в подібних проектах не лише абсолютна – такого ще не було. Таке, може, вже і було, але лише не з нами, командою проекту. Тобто присутній величезний об'єм відносної новизни для самих людей, які втілюють цей проект.

Проекти з розробки програмного забезпечення знаходяться між двома цими полюсами і займають в цьому просторі різне положення. Часто вони складні, тому що об'ємні, і знаходяться на стику різних дисциплін – того цільового бізнесу, куди повинен вбудуватися програмний продукт, і складного, нетривіального програмування. Часто сюди додається ще розробка унікального електронно-механічного устаткування. З іншого боку, оскільки програмування активно просувається в різні сфери людської діяльності, то відбувається це шляхом створення абсолютно нових, унікальних продуктів, і їх розробка і просування володіють всіма рисами творчих проектів.

Управління проектами (*project management*) – галузь *діяльності*, в ході якої, в межах певних проектів, визначаються і досягаються чіткі цілі при знаходженні компромісу між об'ємом робіт, ресурсами (такими як гроші, праця, матеріали, енергія, простір і ін.), часом, якістю і ризиками.

Відзначимо декілька важливих аспектів управління проектами.

Stakeholders – люди «з боку», які не беруть участь безпосередньо в проекті, але впливають на нього і/або зацікавлені в його результатах. Це можуть бути майбутні користувачі системи (наприклад, за ситуації, коли вони і замовник – це не одне і те ж), вище керівництво компанії-розробника і так далі. Ідентифікація всіх stakeholders і виважена робота з ними – важлива складова успішного проектного менеджменту.

Project scope – межі проекту. Це дуже важливе поняття для програмних проектів з причини мінливості вимог. Часто буває, що розробники починають створювати одну систему, а потім, поступово, вона перетворюється на іншу. Причому для менеджерів з продаж, а також замовника, нічого радикально не відбулося, а з погляду внутрішньої структури ПЗ, технологій, алгоритмів реалізації, архітектури – все радикально змінюється. За подібними тенденціями повинен стежити і ретельно з ними розбиратися проектний менеджмент.

Компроміси – найважливіший аспект управління програмними проектами в силу погодження ПЗ. Важливо не втратити всі параметри і сторони, що погоджуються, та знайти прийнятний компроміс.

Під час розробки програмних проектів, виходячи з досвіду і пропозицій ***MSF*** 3.1, важливими є галузі управління, які наведені в Табл. 3.1.

(***Модель процесів MSF - Microsoft Solutions Framework -*** *методологія розробки програмного забезпечення, запропонована корпорацією Microsoft. MSF спирається на практичний досвід Microsoft і описує управління людьми і робочими процесами в процесі розробки рішення. (MSF process model) представляє загальну методологію розробки*

та впровадження IT рішень. Особливість цієї моделі полягає в тому, що завдяки своїй гнучкості та відсутності жорстко нав'язуваних процедур вона може бути застосована при розробці досить широкого кола IT проектів.

Таблиця 3.1.

Важливі галузі управління

Галузь управління проектами	**Опис**
Планування і моніторинг проекту, контроль за змінами в проекті (*Project planning*/tracking/*changeControl*).	Інтеграція і синхронізація планів проекту; організація процедур і систем управління і моніторингу проектних змін
Управління межами проекту (*Scope Management*).	Визначення і розподіл об'єму робіт (меж проекту); управління компромісними рішеннями в проекті
Управління календарним графіком проекту *(Schedule Management*).	Складання календарного графіку виходячи з оцінок трудовитрат, впорядковування завдань, співвідношення доступних ресурсів із завданнями, застосування статистичних методів, підтримка календарного графіку
Управління вартістю (*Cost management*).	Оцінка вартості виходячи з оцінок тимчасових витрат; звітність про перебіг проекту і його аналіз; аналіз витратних ризиків; функціонально-вартісний аналіз (*value analysis*)
Управління персоналом (Staff Resource *Management*).	Планування ресурсів; формування проектної команди; вирішення конфліктів; планування і управління підготовкою

Управління комунікаціями (*Communications Management*)	Комунікаційне планування (між проектною групою, замовником/спонсором, споживачами/користувачами, ін. зацікавленими особами); звітність про перебіг проекту
Управління ризиками (*Risk management*).	Організація процесу управління ризиками в команді і сприяння йому; забезпечення документообігу управління ризиками
Управління постачанням (Procurement).	Аналіз цін постачальників послуг і/або апаратного/програмного забезпечення; підготовка документів про ініціацію пропозицій (requests for proposals – RFPS), вибір постачальників і субпідрядників; складання контрактів і переговори про їх умови; договори; замовлення на постачання і платіжні вимоги
Управління якістю (Quality management).	Планування якості, визначення вживаних стандартів, документування критеріїв якості і процесів його вимірювання

Ця модель поєднує в собі властивості двох стандартних виробничих моделей: каскадної (waterfall) і спіральної (spiral). Модель процесів в MSF 3.0 була доповнена ще одним інноваційним аспектом: вона покриває весь життєвий цикл створення рішення, починаючи з його відправної точки і закінчуючи безпосередньо впровадженням. Такий підхід допомагає проектним групам сфокусувати свою увагу на бізнес-віддачі (business value) рішення, оскільки ця віддача стає реальною лише після завершення впровадження та початку використання продукту.

Процес MSF орієнтований на «віхи» (milestones) - ключові точки проекту, що характеризують досягнення в його рамках будь-якого істотного (проміжного або кінцевого)

результату. Цей результат може бути оцінений і проаналізований, що має на увазі відповіді на питання: «Прийшла чи проектна група до однозначного розуміння цілей і рамок проекту?», «Чи достатньою мірою готовий план дій?», «Чи відповідає продукт затвердженої специфікації?», « чи задовольняє рішення потреби замовника? »і т. д.

Модель процесів MSF враховує постійні зміни проектних вимог. Вона виходить з того, що розробка рішення повинна складатися з коротких циклів, що створюють поступальний рух від найпростіших версій рішення до його остаточного вигляду.

Модель процесів MSF тісно пов'язана з базовими принципами MSF, розглянутими вище. Взагалі кажучи, трьома особливостями моделі процесів MSF є:

- *Підхід, заснований на фазах і віхи.*
- *Ітеративний підхід.*
- *Інтегрований підхід до створення та запровадження рішень.*

Модель процесів включає такі основні фази процесу розробки:

- *Вироблення концепції (Envisioning)*
- *Планування (Planning)*
- *Розробка (Developing)*
- *Стабілізація (Stabilizing)*
- *Впровадження (Deploying))*

Контрольні запитання

1. Дати визначення робочого продукту.
2. Навести приклади робочого продукту.
3. Чим відрізняється робочий продукт від компоненти ПЗ?
4. Що таке нематеріальний робочий продукт.
5. Як "працює" дисципліна зобов'язань.
6. Навести приклади видів відносин між людьми, що відрізняються від дисципліни зобов'язань.
7. Які межі застосування дисципліни зобов'язань.
8. Що таке проект?
9. Чим проект відрізняється від інших форм організації бізнесу і виробництва?

Глава 4. Архітектура ПЗ

Поняття архітектури ПЗ. Точка зору і характеристики точок зору. Множинність точок зору при розробці ПЗ.

4.1. Обговорення

Якось один менеджер пояснював основні ідеї свого достатньо крупного проекту, яким він керував. Він накреслив на дошці три кубики: ***frontend***, ***backend***, ***tools***. І сказав, що це і є головна структура проекту. І в сенсі внутрішнього устрою продукту, і в сенсі розподілу робіт в команді за трьома дистанційно рознесеним центрам розробки. Завдання backend складні, ресурсоємні, виконуються пакетно. Вони відокремлені від графічного інтерфейсу продукту (frontend), який також досить не простий. Frontend – це призначений для користувача інтерфейс: складний, такий, що параметризується, з вбудованими сервісами користувача (зокрема, браузером інформації), з можливістю персонального налагоджування. Обидві ці підсистеми взаємодіють одна з однією через добре і детально описаний програмний інтерфейс: алгоритми ***backend*** розбиті на методи, які ***frontend*** може викликати з параметрами за допомогою особливих правил, об'єднати в ланцюжок для досягнення своїх завдань. "Збоку" від всього цього знаходяться додаткові засоби – ***tools***. Вони інтегруються у frontend, не користуються методами backend, та реалізують свої завдання самостійно. Ці завдання не вимагають складної пакетної обробки, а націлені на інтерактивну взаємодію з користувачем. Для їх реалізації особливо багато уваги приділялося usability (зручності і простоті використання).

Кожна з трьох підсистем вимагала від розробників особливих навиків. У разі backend це було уміння і досвід з реалізації такого роду пакетних алгоритмів, у випадку з frontend – уміння створювати складний призначений для користувача інтерфейс, у випадку з tools вимагалось мистецт-

во в проектуванні і реалізації інструментів, які надають користувачам системи додаткові сервісні можливості. Для того, щоб розділити роботи таким чином, були ще і політичні аспекти. Зокрема, керівництво проекту хотіло мати процес розробки призначеного для користувача інтерфейсу поруч з собою, в одному з трьох центрів розробки, який територіально співпадав з штаб-квартирою. Вважалося, що зовнішній вигляд продукту дуже важливий для його успішного продажу і вимагає особливої уваги.

В результаті виконання проекту (а він розвивався більше 15 років, досягаючи в апогеї до 150 чоловік, які одночасно були зайняті в ньому) така чітка структура дещо змістилася: так географічно *інтерфейс* майже "переїхав" в той центр, де розроблявся backend. Але в цілому такий розподіл проекту на частини залишався багато років і був основним скелетом всієї розробки. Це і є приклад архітектури програмного проекту.

4.2. Визначення

Під **архітектурою ПЗ** слід розуміти внутрішню структуру продукту (компоненти і їх зв'язки), основи призначеного для користувача інтерфейсу продукту, а також квінтесенцію знань і рішень, що є інструментом розробки і *управління проектом*. Тобто *архітектура* – це крізна концепція або набір концепцій для подолання ентропії і хаосу, які прагнуть "проковтнути" розробку з причини складності, нематеріальності, погоджуваності і мінливості *ПЗ.* При цьому не слід розділяти продукт і проект, оскільки на практиці це, як правило, одне ціле, причому це "поєднання", якщо воно існує, є "сильною" стороною даної розробки.

Часто під архітектурою розуміють, наприклад, лише внутрішній устрій *ПЗ,* який виражений в *UML-діаграмах.* Ось жарт на тему того, що архітектуру не можна розуміти односторонньо. Одного з відомих розробників трансляторів запитали, чому в його відомому трансляторі рівно 21 відображення. Чекали почути *перерахування алгоритмічних* проблем, які у та-

кий спосіб вдалося подолати, щось про особливу *ефективність алгоритмів*, які були організовані таким чином, і так далі. Всіх здивувала відповідь метра. Він сказав, що саме стільки людей (тобто рівно 21), було у нього в команді розробників...

Отже, *архітектура* продукту є *інваріантом* проекту. Вона зустрічається і несподівано виникає в різних його частинах. Це і є аналогом "простих" природно-наукових постулатів і законів, відсутність яких в розробці *ПЗ, на думку* Фредеріка Брукса (Фредерік Філіпс Брукс (Frederick Phillips Brooks, Jr.; народився 19 квітня, 1931) — інженер програмного забезпечення та вчений-інформатик, найбільш відомий за управління розробкою ОС System/360 для IBM, та її наступників, а пізніше за опис цього процесу в книжці Міфічний людино-місяць. Брукс отримав за своє життя багато нагород, включаючи Національну медаль технологій та інновацій США у 1985, та премію Тюрінга у 1999), є причиною складності ПЗ (у сенсі хаосу, тобто "поганої" складності). Створювати такі структури – непроста справа, що вимагає великого мистецтва. Але саме це є *шляхом* до управління хаосом та зростаючою *ентропією,* у вигляді вимог до системи, які змінюються, для запобігання втрати розробниками ясного розуміння, яку ж саме систему вони створюють. І саме розробка таких структур надає *дійсно* творчу насолоду підчас розробки програмних систем. Добре "працюють" прості моделі, які не легко створювати. Такі моделі надають розробнику можливість розповідати про проект дуже довго, красиво його оформити та повісити на стінку.

В межах багатьох проектів не створюється оригінальна архітектура, оскільки вони є типовими і/або невеликими та ґрунтуються на готових технологіях, архітектурних зразках, моделях команди і оргструктурах проектів.

Проте, часто перед колективами, які добре себе зарекомендували в таких проектах, виникає завдання побудувати дійсно оригінальну нову ар-

хітектуру, що ґрунтується на колишніх розробках. Тут, перш за все, важливо відмітити цей перехід, усвідомити, що старі методи роботи не підходять і потрібний принципово новий *досвід*, якого, дуже часто, немаєу колективу та його лідерів...

4.3. Множинність точок зору

Під час розробки архітектури ПЗ важливим є поєднання *безлічі* точок зору. ПЗ виявляється настільки складним, що його архітектуру не можна побудувати як єдину модель: безліч окремих аспектів повинні бути представлені в архітектурі, їх зв'язки є складними і тому погано відображаються у явному вигляді. Тому корисним виявляється створення *безлічі* моделей, які спираються на різні точки зору.

Причина множинності точок зору підчас розробки ПЗ. Уміння розглядати предмет з різних точок зору є найважливішою філософією успішної практики у разі роботи з великими об'ємами різнорідної і складної інформації. Розглянемо розробку ПЗ і те, чому там затребувані різні погляди на процес, систему і так далі.

Це відбувається, перш за все, через різні види діяльності процесу розробки ПЗ (див. **Рис.** 4.1). Під час складання функціональних *вимог* до ПЗ звертають увагу на те, яка саме функціональність повинна бути реалізованою, але при цьому опускаються принципи і деталі реалізації. На етапі проектування, навпаки, на перше місце виходять принципи реалізації ПЗ. А підчас тестування деталі реалізації знову неважливі – на ПЗ дивляться як на *чорний ящик*, що реалізовує (не важливо яким чином) деякий набір призначеної для користувача функціональності. На етапі розгортання ПЗ у замовника на ПЗ дивляться як на набір файлів, сховищ даних і так далі.

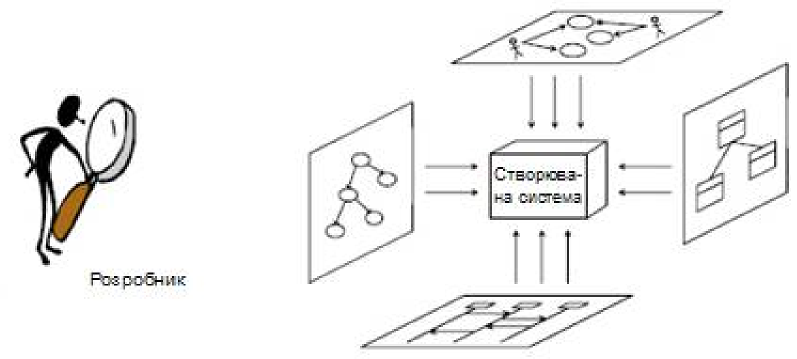

Рис. 4.1. Різні види діяльності – різні погляди на систему

Далі, у розробку/експлуатацію ПЗ залучено велика кількість дуже різних фахівців: програмісти, інженери, тестувальники, технічні письменники, менеджери, замовники, користувачі, продавці-маркетологи і так далі (див. **Рис.**4.2). Для всіх цих фахівців потрібна різна інформація про програмну систему. Уявіть, що відбудеться, якщо, наприклад, продавцеві або замовникові-непрограмістові у відповідь на прохання трохи краще ознайомитися з ПЗ ви дасте почитати програмний код.

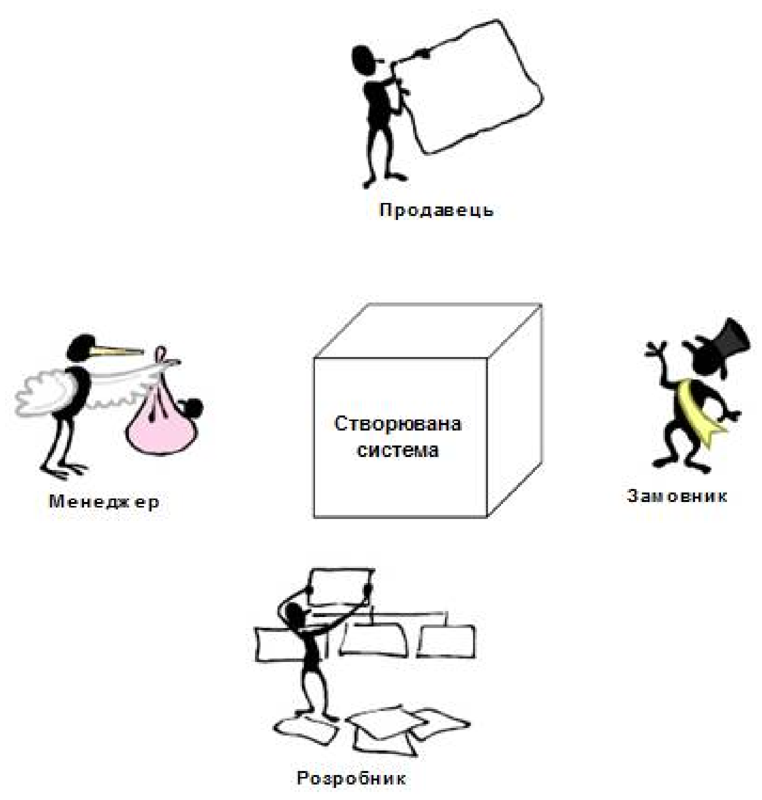

Рис. 4.2. Різні фахівці – різні погляди на систему

Множинність точок зору походить також від того, що немає єдиних стандартів і норм розробки ПЗ. Тобто розробка набагато у чому є "stateofart" («стан мистецтва»). Часто доводиться винаходити нову точку зору моделювання відповідно до ситуації – щоб саме цей експерт тебе зрозумів, щоб саме ці особливості системи були підкреслені. Часто тут – як в лотереї: створюється декілька описів системи з різних точок зору, якийнебудь з описів виявляється вдалим і саме його надалі всі використовують.

Отже, різні види діяльності в процесі розробки ПЗ, різні категорії фахівців, що задіяні в програмному проекті, і унікальність кожної конкретної ситуації підчас розробки – все це призводить до створення і використання різних моделей, які виконані з різних точок зору.

Точка зору (***view point***) – ***це певний погляд на систему, який здійснюється для виконання якогось певного завдання ким-небудь з учасників***

проекту. Точку зору слід ясно усвідомлювати на етапі створення візуальних моделей, наприклад, *варіантів використання*. Важливо розуміти, що модель може бути у кожному конкретному випадку своя. Найважливішими характеристиками точки зору моделювання є **мета** (навіщо створюється модель) і **цільова аудиторія** (тобто, для кого вона призначається).

Важливим питанням, на яке потрібно чесно собі відповісти на сам початок моделювання, – це «Навіщо ви використовуєте діаграми?» (зокрема, UML). Це і є визначення мети моделювання. Бо так створювати моделі правильно? І всі проблеми (навіть ті, про які нічого ще не відомо) чарівним чином зникнуть? Дуже часто, наприклад, підчас створення моделі випадків використання присутня саме така "мета" моделювання. А потім виявляється, що жодні проблеми не "вилікувалися", а навпаки, виникли нові (наприклад, створені нами діаграми ніхто не розуміє і не приймає). Навіть сам аналітик відчуває, що діаграми вийшли якісь дивні.

А може все відбувається зовсім не так. Наприклад, аналітик дійсно задався метою виявити вимоги до системи – не нав'язати своє власне бачення іншим, а з'ясувати потрібну інформацію, змоделювати і викласти її доступно. Для цього він і використовує діаграми випадків використання. Йому важливо, щоб майбутні користувачі системи могли брати участь в цьому процесі, діаграми малюються для них, вони зрозумілі і не надмірні. І ці ж діаграми структурують і прояснюють інформацію для самого аналітика.

Подібних сюжетів на практиці відбувається безліч. Тут важливо розуміти, що мета моделі – це не якесь гіпотетичне завдання типу "опису архітектури, тому що так потрібно, так правильно", а цільова аудиторія – це не абстракція типу "люди, які хочуть ознайомитися з ПЗ". І те і інше – щось дуже конкретне, що реально існує в проекті або поряд з ним. Адже розробники ПЗ не можуть дозволити собі за гроші замовника створювати щось на всі століття і для всіх народів. І мета моделювання, і аудиторія,

яка працюватиме з діаграмами, завжди існують, важливо лише ясно розуміти, які вони.

Існує корисний практичний прийом для орієнтації на цільову аудиторію, для якої призначена створювана вами модель. Можна вибрати одного представника такої аудиторії – конкретну і відому вам людину – і створювати діаграми, які зрозумілі саме йому. При цьому важливо не обговорювати надмірно з ним ваші моделі, оскільки це може створити додатковий контекст, якого інші користувачі моделей будуть позбавлені. Корисно уявляти собі цю людину підчас роботи з моделями – його реакції, питання, здивування і ін. І, виходячи з цього, корегувати, виправляти створене. І, звичайно ж, корисно перевірити свої припущення, коли показуєш йому, що вийшло.

Крім того, важливо, щоб точка зору була "живою", а не вигадувалася аналітиком або бездумно копіювалася з книжок і присвячених UML-тренінгів. Непомітно для себе аналітик може придумати свій власний проект, своїх власних користувачів системи, замовника і так далі. Тобто аналітик поволі, нав'язує самому собі певне сприйняття реально існуючих людей, завдань, сильно спотворюючи реальне положення справ. І саме в контексті цієї уявної ситуації він створює свої моделі. Але ж реальні люди, реальні ситуації володіють своєрідністю, великим діапазоном варіативності. Відповідно, аналітик повинен володіти гнучкістю свідомості, великим діапазоном техніки, а також чуйністю і щирим прагненням до того, щоб зробити кожен конкретний проект, де він бере участь, більш гармонійним та адекватним.

4.4. Мова UML

Часто поняття архітектури сильно звужують, і розуміють під ним лише опис основних, важливих аспектів ПЗ, які створюються, наприклад, архітектором підчас розробки дизайну системи. Для цих цілей використовується мова моделювання UML (Unified Modeling Language).

Ця мова є підсумком розвитку засобів схематичного опису програмних систем, які розвивалися з блок-схем, запропонованих ще Джоном фон Нейманом в кінці 40-х років (***Джон фон Нейман** (John von Neumann), **Нейман Янош Лайош** (угор. Neumann János Lajos), **Йоганн фон Нойман** (нім. Johann von Neumann) нар. 28 грудня 1903 — пом. 8 лютого 1957) — американський математик угорського походження, що зробив значний вклад у квантову фізику, функціональний аналіз, теорію множин, інформатику, економічні науки та в інші численні розділи знання. Він став засновником теорії ігор разом із Оскаром Морґенштерном у 1944 році. Розробив архітектуру (так звану «архітектуру фон Неймана»), яка використовується в усіх сучасних комп'ютерах.*). Він припускав, що ці схеми стануть високорівневою мовою введення алгоритмів в обчислювальні машини, але еволюція мов програмування пішла шляхом розвитку текстових мов. Проте,блок-схеми набули поширення при специфікації і документуванні ПЗ, були стандартизовані, але широкого практичного застосування не отримали. В кінці 60-х років, у зв'язку з пошуком нових засобів розробки ПЗ,народженням програмної інженерії і загальним розвитком в галузі проектування і розробки штучних систем з'явився термін структурний аналіз (structure data analysis) систем. Термін був введений вченим з MIT, Дугласом Россом, який також запропонував діаграмний метод аналізу і проектування великих штучних систем (***Дуглас Тейлор «Дуги» Росс** (21 грудня 1929 - 31 грудень 2007) американський вчений піонер IT, голова SOFTECH Inc. Найбільш відомий як автор терміну CAD (САПР) для автоматизованого проектування, і вважається батьком APT (Automatically Programmed Tools – Засобів автоматичного програмування) мови для управлінням виробництвом з цифровим управлінням*). Метод отримав назву SADT (Structured Analisysand Design Technique), став основою серії військових стандартів США серії IDEF і широко розповсюдився в індустрії. Проте, діаграмна мова в SADT була дуже скромною – набір блоків і

зв'язків між ними, з підтримкою декомпозиції блоків. У 70-х роках, у зв'язку з масовим виходом ПЗ на вільний ринок (тобто програмні системи почали створюватися не лише у військовій галузі, для крупного бізнесу, але також для середнього і малого бізнесу) структурний аналіз став бурхливо еволюціонувати – набір діаграм збагатився діаграмами станів і переходів, суть-зв'язків, потоків даних і так далі. З розвитком об’єктно-орієнованих засобів розробки (кінець 80-х – середина 90-х)структурний аналіз перетворився на об'єктно-орієнтований аналіз і проектування. З'явилася велика кількість методологій, поступово склалася єдина мова моделювання, яка і була закріплена в стандарті UML. Відбулося це в 1997 році.

З тих пір вийшло декілька версій стандарту UML. Поточна версія UML 2.1.

4.5. Види діаграм

"Скелетом" UML є діаграмна структура. Кожен вид діаграм є типом моделей, що реалізовує певну точку зору на програмну систему. Види діаграм не є строго обов'язковими в UML – їх можна перемішувати, створювати свої власні види діаграм. Проте, стандартні види діаграм є певним надбанням програмної інженерії, оскільки відображають досвід багатьох дослідників і практиків.

- ***Структурні діаграми:***
  - **діаграми класів** (***class diagrams***) призначені для моделювання структури об'єктно-орієнтованих додатків класів, їх атрибутів і заголовків методів, спадкоємства, а також зв'язків класів один з одним;
  - **діаграми компонент** (***component diagrams***) використовуються при моделюванні компонентної структури розподілених

додатків; усередині кожна компонента може бути реалізована за допомогою безлічі класів;

- **діаграми об'єктів** (***object diagrams***) застосовуються для моделювання фрагментів працюючої системи і відображають саме ті фрагменти, що реально існують (runtime) в екземплярах класів,та значення їх атрибутів;
- **діаграми композитних структур** (***composite structure diagrams***) використовуються для моделювання складових структурних елементів моделей – кооперацій, композитних компонент і т.д.;
- **діаграми розгортання** (***deployment diagrams***) призначені для моделювання апаратної частини системи, з якою ПЗ безпосередньо зв'язано (розміщено або взаємодіє);
- **діаграми пакетів** (***package diagrams***) служать для розбиття об'ємних моделей на складові частини, а також (традиційно) для групування класів модельованого ПЗ, коли їх дуже багато.

- Поведінкові діаграми:

- **діаграми активності** (***activity diagrams***) використовуються для специфікації бізнес-процесів, які ПЗ,що розробляється, повинно автоматизувати, а також для опису складних алгоритмів;
- **діаграми випадків використання** (***use case diagrams***) призначені для з’ясування і погодження вимог з замовником,користувачами і експертаминаочної галузі;
- **діаграми кінцевих автоматів** (***state machine diagrams***) застосовуються для визначення поведінки реактивних систем;

- **діаграми взаємодій** (***interaction diagrams***):
    - **діаграми послідовностей** (***sequence diagrams***) використовуються для моделювання часових аспектів внутрішніх і зовнішніх протоколів ПЗ;
    - **діаграми схем взаємодії** (***interaction overview diagrams***) служать для організації ієрархії діаграм послідовностей;
    - **діаграми комунікацій** (***communication diagrams***) є аналогом діаграм послідовностей, але іншим чином зображені (у звичній графовій манері);
- **часові діаграми** (***timing diagrams***) є різновидом діаграм послідовностей і дозволяють в наочній формі показувати внутрішню динаміку взаємодії деякого набору компонент системи.

4.6. Приклади.

Центральним видом діаграм є ***діаграми класів*** (Рис. 4.3).

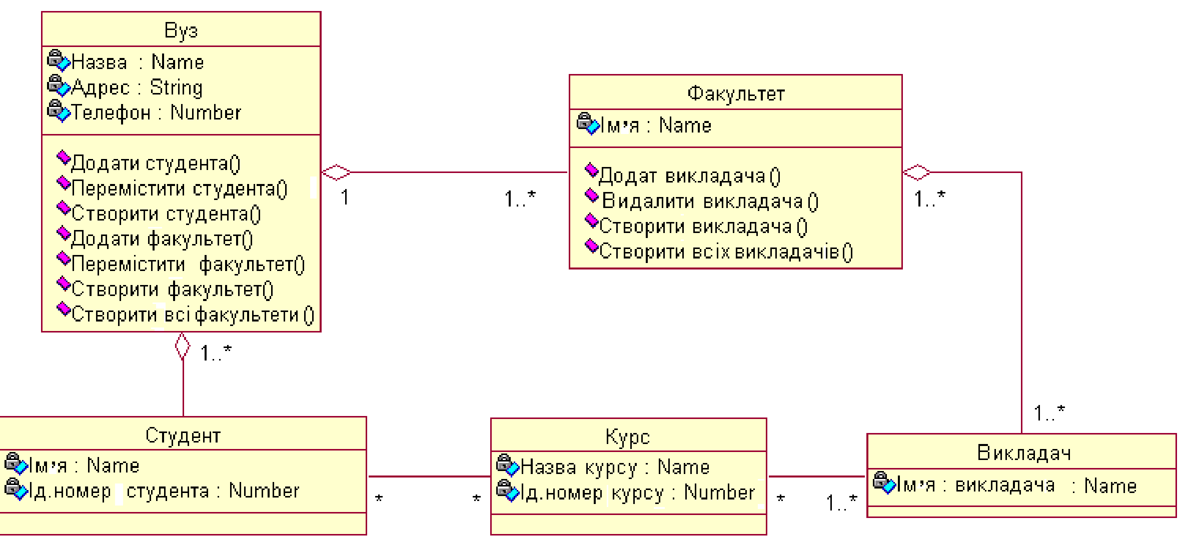

Рис. 4.3. Приклад діаграми класів

Ще один вид структурних діаграм – ***діаграми розгортання***, приклад наведений нижче.

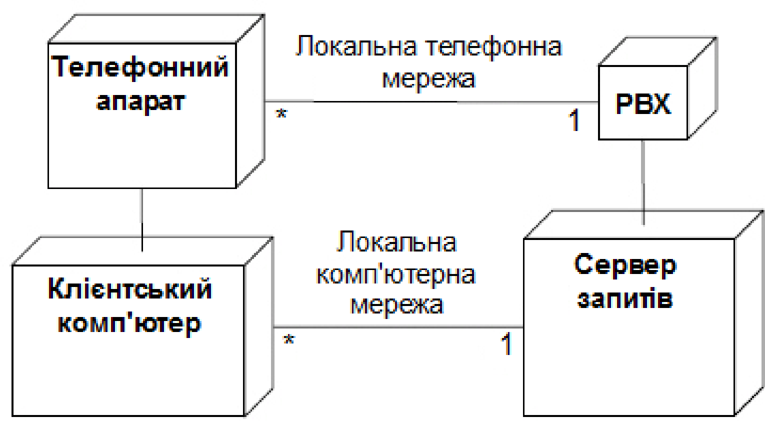

Рис. 4.4. Приклад діаграми розгортання

Відзначимо також ще один важливий вид діаграм UML – ***діаграми компонент***(приклад на **Рис**. 4.5).

Цікавий також варіант діаграм композитних структур – складні компоненти для систем реального часу і телекомунікації. Приклад наведено на **Рис**. 4.6.

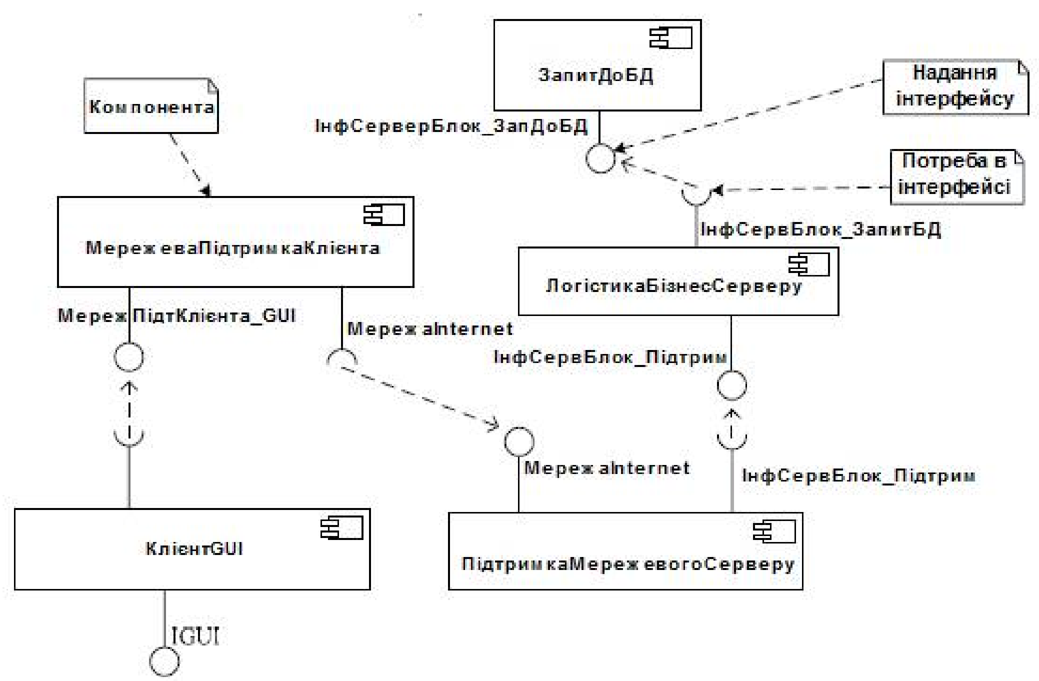

Рис. 4.5. Приклад діаграми компонент

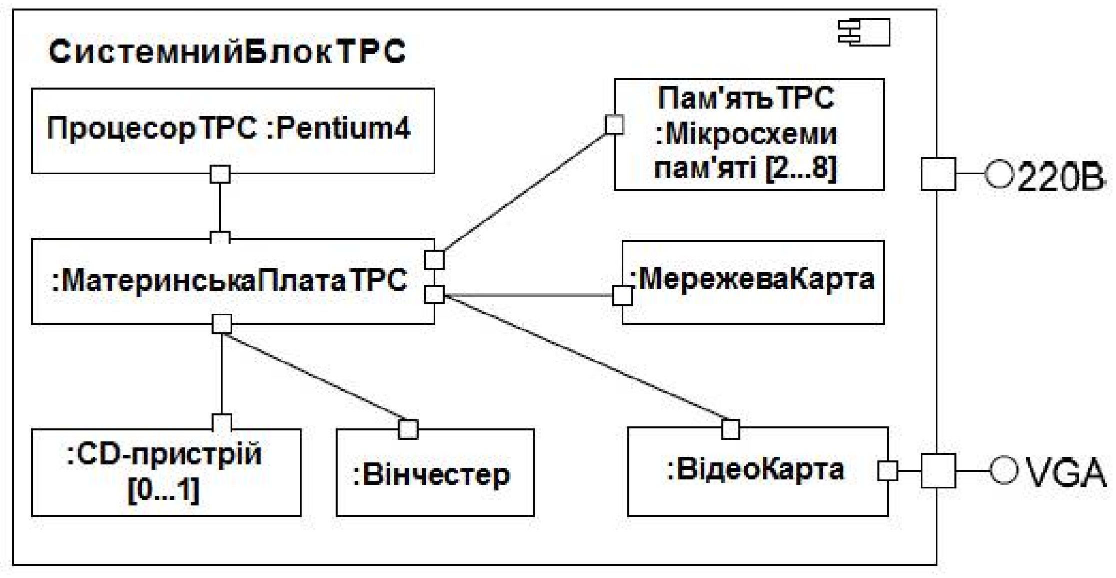

Рис. 4.6. Приклад діаграми композитних структур

Нижче наводяться приклади використання поведінкових діаграм UML. **Діаграми кінцевих автоматів** дозволяють створювати вичерпні специфікації поведінки телекомунікаційних, подіє-керованих алгоритмів і

автоматично генерувати за цими описами програмний код. Приклад такої діаграми для класу «Підключення» наведений нижче.

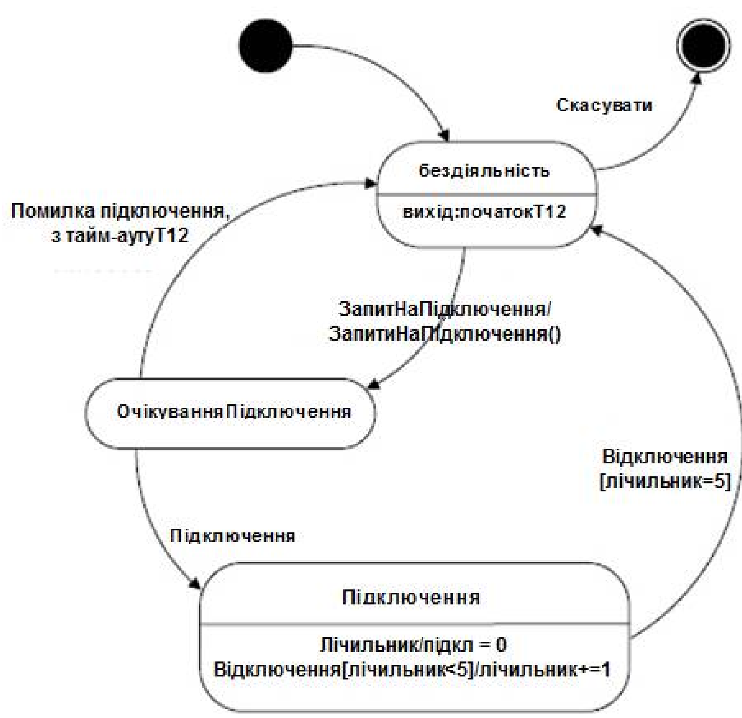

Рис. 4.7. Приклад діаграм кінцевих автоматів

Ще один важливий вид діаграм – **діаграми послідовностей**. Вони дозволяють задавати головні гілки складних телекомунікаційних алгоритмів, а також малювати ланцюжки викликів для об'єктно-орієнтованих застосувань, які програмуються в термінах об'єктів, але проектуються часто в термінах ланцюжків викликів. Приклад наведений нижче.

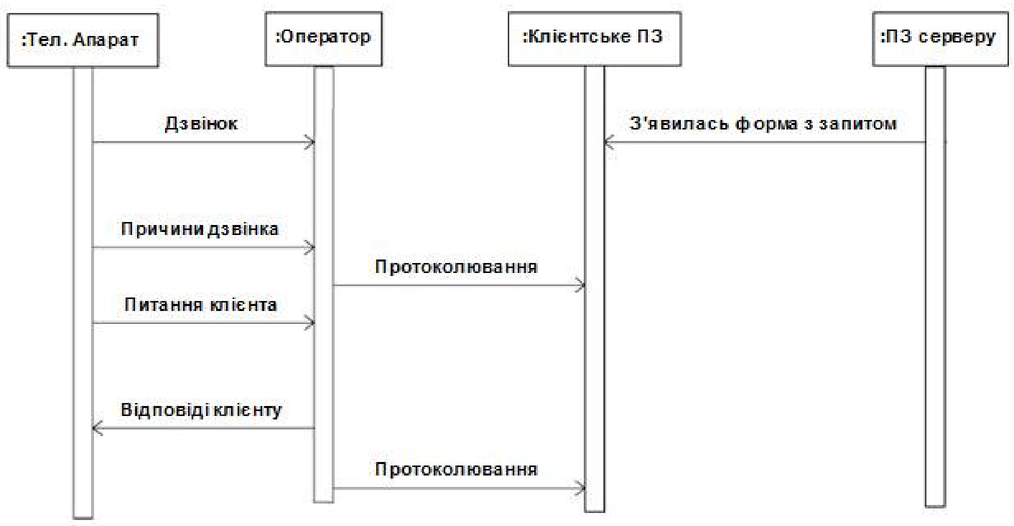

Рис. 4.8. Приклад діаграм послідовностей

Контрольні запитання

1. Дати визначення архітектури ПЗ.
2. На які аспекти розробки спирається поняття архітектури ПЗ.
3. Які причини множинності точок зору підчас розробки ПЗ.
4. Як множинність точок зору впливає на розробку ПЗ?
5. Перерахуйте і коротко прокоментуйте різні види діаграм UML.

Глава 5. Управління вимогами

Види вимог: функціональні вимоги, нефункціональні вимоги. Властивості вимог: ясність і недвозначність, повнота і несуперечність, необхідний рівень деталізації, можливість відслідковування, тестування і перевірки, модифікується. Формалізація вимог. Цикл роботи з вимогами.

5.1. Проблема

Для прикладу розглянемо роботу будівельної компанії. Будівельники будують будинки, нехай різні: багатоповерхові, окремі котеджі, офісні будівлі і ін. – проте, весь цей спектр цілком може охопити одна компанія. Будівельній компанії не доводиться будувати тарілку, що літає, гіперболоїд інженера Гаріна, місяцехід, систему миттєвої телепортації і ін. А розробники ПЗ, багато в чому, знаходяться саме в такій ситуації.

Велике розмаїття систем, які створює одна компанія, одна команда. Хоча зараз і намічаються тенденції до спеціалізації ринку розробки ПЗ, проте, чудасії світової економіки і багато інших причин приводять до того, що строго спеціалізованих компаній не так багато, як хотілося б. Багато галузей відчувають великий дефіцит окремих програмістів і цілих колективів і компаній, що добре розбираються в їх специфіці. Прикладом такої галузі може служити телебачення, де про дану проблему відкрито говорять на засіданнях різних міжнародних співтовариств.

Крім того, ПЗ продовжує проникати у все нові і нові галузі людської діяльності. Сформулювати адекватні вимоги в цьому випадку взагалі виявляється суперважким завданням.

Але навіть якщо мова йде про одну, певну галузь, то відсоток нових, унікальних рис систем, що належать до цієї галузі, високий: за поєднанням призначених для користувача характеристик, за особливостями середови-

ща виконання і вимогами до інтеграції, за розподіленністю інформації про вимоги серед працівників компанії-замовника. Все це несе на собі дуже великий відбиток індивідуальності замовника – персонально або його компанії, – сильно пов'язано із специфікою його бізнесу або устаткування, яке використовується в цій галузі.

Крім того, існують труднощі в розумінні між замовником і програмістами, а ще – в мінливості ПЗ (вимоги мають тенденцію змінюватися підчас розробки).

У результаті, далеко не очевидно, що та система, яку хоче замовник, взагалі може бути створена. Важко знайти чорну кішку в темній кімнаті, особливо якщо її там немає. Або те, як зрозуміли і втілили завдання розробники, виявиться зручним та затребуваним на ринку.

Помилки і різночитання, які виникають під час з’ясування вимог до системи, виявляються одними з найдорожчих. Вимоги – це те початкове розуміння завдання розробниками, яке є основою всієї розробки.

Декілька слів про важкість порозуміння замовника і розробників (Рис. 5.1). Тут позначається великий розрив між програмістами і іншими людьми. По-перше, щоб добре розібратися, якою повинна бути система автоматизації лікарні або система підтримки хімічних експериментів, слід попрацювати у відповідній галузі достатньо часу. Інший підхід – це навчитися якимось іншим способом бачити проблеми даної наочної галузі зсередини. По-друге, позначається специфічність програмування як сфери діяльності. Для більшості користувачів і замовників украй не просто точно сформулювати знання, які необхідні програмістам. На питання, скільки типів аналізів існує у вашій лабораторії, доктор, подумавши, відповідає – 43. І вже потім, випадково, програміст уточнив, а чи немає інших типів аналізів? Звичайно, є, відповів доктор, проте вони трапляються не часто і можуть бути в деякому розумінні, якими завгодно. З першого разу він на-

звав лише типові. Але, звичайно ж, інформаційна система повинна зберігати інформацію про всі аналізи, які проводяться в лабораторії.

Тепер трохи докладніше про мінливість програмного забезпечення і про її причини.

- Змінюється ситуація на ринку, для якого призначалася система, або змінюються вимоги до системи через перспективи продажу ще неготової системи.
- Потягом розробки виникають проблеми і труднощі, через які підсумкова функціональність змінюються (видозмінюється, урізується).
- Замовник може міняти своє власне бачення системи: чи то він краще розуміє, що ж йому насправді треба, чи то з'ясовується, що він щось упустив із самого початку, чи то з'ясовується, що розробники його не так зрозуміли. Загалом, буває різне, важливо лише, що тепер замовник безумовно хоче іншого.

Годі і говорити, що мінливість вимог під час розробки дуже хворобливо позначається на продукті. Були випадки, коли ще не створену систему відділ продажу починає активно продавати, через що надходить величезний потік додаткових вимог. Всі їх реалізувати у повному обсязі не вдається, як результат, система виявляється набором демо-функціональності.

5.2. Види і властивості вимог

Розділимо вимоги на дві великі групи – **функціональні** і **нефункціональні**.

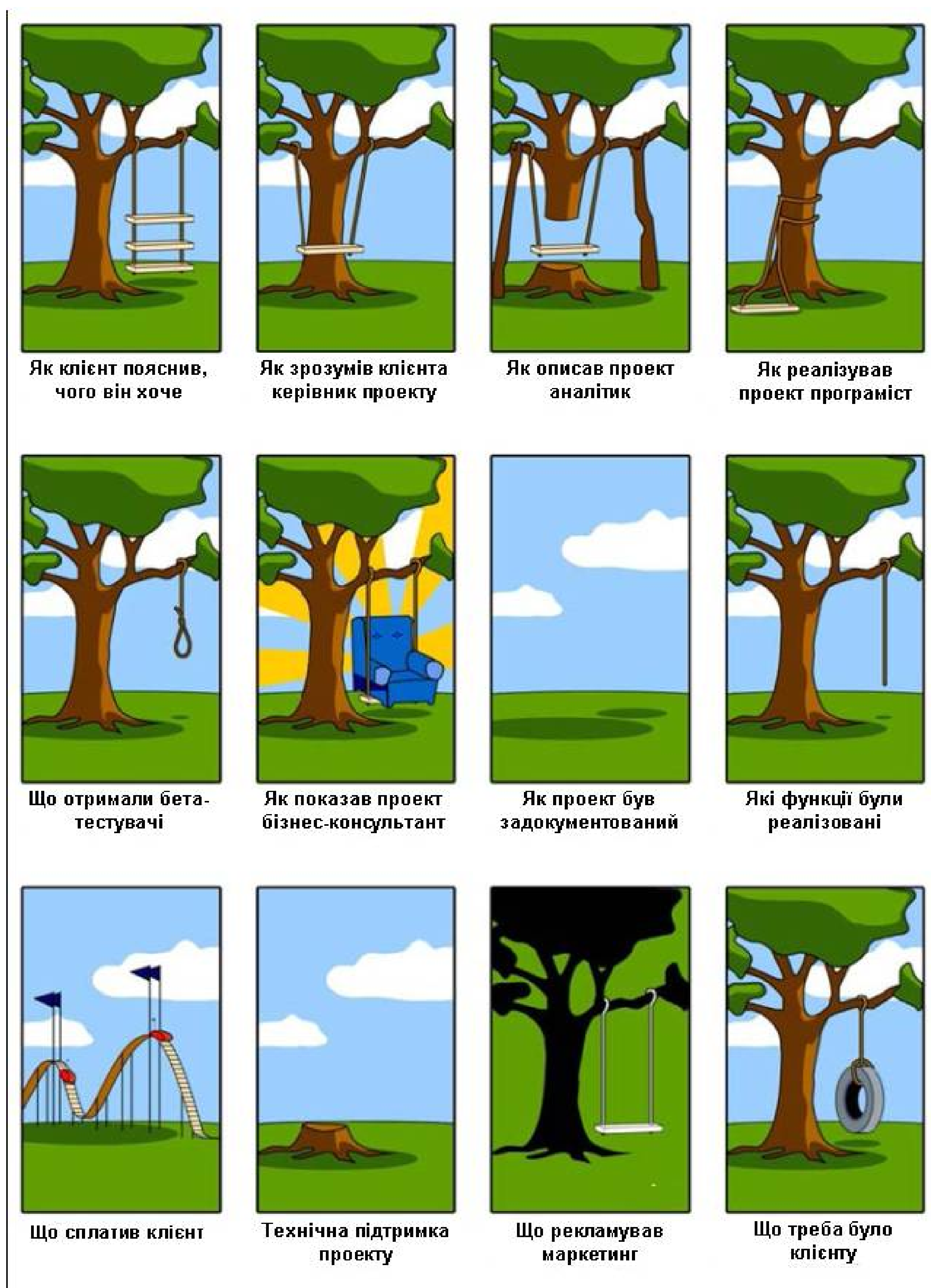

Рис. 5.1 Результати проекту через непорозуміння замовника і розробників

Функціональні вимоги є детальним описом поведінки і сервісів системи, її функціоналу. Вони визначають те, що система повинна уміти робити.

Нефункціональні вимоги не є описом функцій системи. Цей вид вимог описує такі характеристики системи, як надійність, особливості постачання (наявність інсталятора, документація), певний рівень якості (наприклад, для нової Java-машини це буде значити, що ПЗ має задовольняти вимогам щодо набору тестів, які підтримуються компанією Sun). Сюди ж можуть відноситися вимоги до засобів і процесів розробки системи, вимоги до перенесення, відповідності стандартам і так далі. Вимоги цього виду часто відносяться до всієї системи в цілому. На практиці, особливо початківці, часто забувають про деякі важливі нефункціональні вимоги.

Сформулюємо ряд важливих властивостей вимог.

- **Ясність, недвозначність** – однозначність розуміння вимог замовником і розробниками. Часто цього важко досягти, оскільки кінцева формалізація вимог, яка виконана з погляду потреб подальшої розробки, є важкою для сприйняття замовником або фахівцем наочної галузі, які повинні проінспектувати адекватність формалізації.
- **Повнота і несуперечність.**
- **Необхідний рівень деталізації**. Вимоги повинні мати чітко визначений рівень деталізації, стиль опису, спосіб формалізації:
    - або це опис властивостей наочної галузі, для якої призначене ПЗ;
    - або це технічне завдання, яке додається до контракту;

- або це проектна специфікація, яка повинна бути потім уточнена підчас детального проектування.

Важливо також ясно бачити і розуміти тих, для кого даний опис вимог призначений, інакше не уникнути непорозуміння і подальших труднощів. Адже в розробці ПЗ задіяно багато різних фахівців – інженерів, програмістів, тестувальників, представників замовника, можливо, майбутніх користувачів – і всі вони мають різну освіту, професійні навики і спеціалізацію, часто говорять різними мовами. Тут також важливо, щоб вимоги були максимально абстрактні і незалежні від реалізації.

- **Можливість відслідковування** – важливо бачити ту або іншу вимогу в різних моделях, документах, нарешті, в коді системи. Іноді виникають питання типу – "Хто знає, чому ми вирішили, що такий-то модуль повинен працювати саме таким чином?". Можливість відслідковування функціональних вимог досягається шляхом їх розбиття на окремі, елементарні вимоги, надання їм ідентифікаторів і створення моделі трасування, яка в ідеалі повинна сягати програмного коду. Важливо знати, де потрібно змінити код, якщо певна вимога змінилася. На практиці формальна можливість цілковитого відслідковування є важко досяжною, оскільки логіка і структура реалізації системи можуть сильно відрізнятись від останніх для моделі вимог. Як результат, одна вимога виявляється сильно "розмазаною" за програмним кодом, а та або інша ділянка коду може впливати на багато вимог. Але слід прагнути можливості відслідковування стану вимог, розумно поєднуючи формальні і неформальні підходи.
- **Можливість тестування та перевірки** – необхідно, щоб існували способи тестування і перевірки даної вимоги.

Причому, важливі обидва аспекти, оскільки часто перевірити замовник може, а ось тестувати дану вимогу дуже важко, або неможливо з причини обмеженості доступу до оточення системи для команди розробника (наприклад, з міркувань безпеки). Отже, необхідні процедури перевірки-виконання тестів, проведення інспекцій, проведення формальної верифікації частини вимог і ін. Потрібно також визначати "планку" якості (чим вище якість, тим воно дорожче коштує!), а також критерії повноти перевірок, щоб керівники проекту, які їх виконують, чітко усвідомлювали, що саме перевірене, а що ще ні.

- **Можливість модифікації**. Визначає процедури внесення змін у вимоги.

5.3. Варіанти формалізації вимог

Взагалі кажучи, вимоги як такі – це деяка абстракція. У реальній практиці вони завжди існують у вигляді якогось уявлення – документа, моделі, формальної специфікації, списку і так далі. Вимоги важливі як такі, тому що існують у вигляді розуміння розробниками потреб замовника і майбутніх користувачів створюваної системи. Але оскільки в програмному проекті багато різних аспектів, видів діяльності і фаз розробки, то це розуміння може набувати дуже різних форм. Кожне унаочнення вимог виконує певне завдання:

- служить "містком", фіксацією угоди між різними групами фахівців,
- використовується для оперативного управління проектом (відслідковується, в якій фазі реалізації знаходиться та або інша вимога, хто за неї відповідає і ін.),

- використовується для верифікації і модельно-орієнтованого тестування.

І у першому, і в другому, і в третьому прикладі ми маємо справу з вимогами, але формалізовані вони будуть по-різному.

Отже, формалізація вимог в проекті може бути дуже різною – це залежить від його величини, прийнятого процесу розробки, які інструментальні засоби використовуються, а також тих завдань, які вирішують формалізовані вимоги. Більш того, може існувати паралельно декілька формалізацій для вирішення різних завдань. Розглянемо варіанти.

1. **Неформальна постановка вимог в листуванні електронною поштою.** Добре працює в невеликих проектах, у випадку залучення замовника в розробку (наприклад, команда виконує субпідряд). Може також застосуватись, коли є взаєморозуміння між замовником і командою, тобто зайві формальності не потрібні. Проте, електронні листи в такій ситуації часто опиняються важливими документами – важливо уміти вести ділове листування, підводити підсумки, зберігати важливі листи і користуватися ними у разі розбіжностей. Важливо також вчасно зрозуміти, коли такий спосіб перестає працювати і необхідні більш формальні підходи.

2. **Вимоги у вигляді документа** – опис наочної галузі і її властивостей, технічне завдання як додаток до контракту, функціональна специфікація для розробників і так далі.

3. **Вимоги у вигляді графа** із залежностями в одному із засобів підтримки вимог (IBM Rational Requisitepro, DOORS, Borland CALIBERRM і деякі інші). Таке уявлення зручне у разі частої зміни вимог, для відслідковування виконання вимог, у випадку організації "прив'язки" до вимог завдань, людей, тестів, коду. Важливо також, щоб була можливість легко створювати такі графи з текстових документів, і навпаки, створювати презентаційні документи за такими графами.

4. **Формальна модель вимог** для верифікації, модельно-орієнтованого тестування і так далі.

Отже, кожен спосіб подання вимог повинен відповідати на питання: хто споживач, користувач цього подання, як саме, з якою метою це подання використовується.

5.4. Деякі помилки підчас документуванні вимог.

- Перерахуємо ряд помилок, що зустрічаються при складанні технічних завдань і інших документів з вимогами.
- Опис можливих рішень замість вимог.
- Нечіткі вимоги, які не допускають однозначну перевірку, залишають недомовленості, мають відтінок порад, обговорень, рекомендацій: "Можливо, що має сенс реалізувати також...", "і так далі".
- Ігнорування аудиторії, для якої призначено подання вимог. Наприклад, якщо специфікацію складає інженер замовника, то часто зустрічається надлишок інформації про устаткування, з яким повинна працювати програмна система, відсутній глосарій термінів і визначень основних понять, використовуються численні синоніми і так далі. Або допущений дуже великий ухил у бік програмування, що робить дану специфікацію незрозумілої всім непрограмістам.
- Відсутність важливих аспектів, які пов'язані з нефункціональними вимогами, зокрема, інформації про оточення системи, про терміни готовності інших систем, з якими повинна взаємодіяти ця система. Останнє трапляється, наприклад, коли ця програмна система є частиною більшого проекту. Типовою проблемою підчас створення програмно-

апаратних систем є ситуація, коли апаратура вчасно не постачається і ПЗ неможливо тестувати, а в термінах і вимогах це не передбачено.

5.5. Цикл роботи з вимогами

У зведенні знань щодо програмної інженерії SWEBOK визначаються такі види діяльності для роботи з вимогами.

- **Виділення вимог** (***requirements elicitation***), націлене на виявлення всіх можливих джерел вимог і обмежень на роботу системи та отримання вимог з цих джерел.
- **Аналіз вимог** (***requirements analysis***), метою якого є виявлення і усунення суперечностей і неоднозначностей у вимогах, їх уточнення і систематизація.
- **Опис вимог** (***requirements specification***). В результаті цієї діяльності вимоги повинні бути оформлені у вигляді структурованого набору документів і моделей, які можуть систематично аналізуватися, оцінюватися з різних позицій. Як результат, такий набір документів повинен бути затверджений як офіційне формулювання вимог до системи.
- **Валідація вимог** (***requirements validation***), яка вирішує задачу оцінки зрозумілості сформульованих вимог і їх характеристик, що необхідні для розробки на їх основі ПЗ, в першу чергу, несуперечності і повноти, а також відповідності корпоративним стандартам на технічну документацію.

Контрольні запитання

1. Охарактеризувати полягає особливості управління вимогами?
2. Які існують способи формалізації вимог?
3. Які існують способи фіксації формалізації?

4. Що мається на увазі як спосіб формалізації проміжної фіксації?
5. Що мається на увазі як спосіб формалізації фінальної фіксації?
6. Які є способи "витягування" вимог?
7. Яка існує техніка "витягування" вимог?
8. Які види документів формалізують вимоги?
9. В чому полягає відмінність функціональних і нефункціональних вимог.
10. Що таке типовий цикл роботи з вимогами?
11. Які існують етапи циклу роботи з вимогами?
12. Які існують типові помилки при роботі з вимогами?

Глава 6. Конфігураційне управління

Поняття конфігураційного управління. Управління версіями. Поняття "гілки" проекту. Управління збірками. Засоби версійного контролю. Одиниці конфігураційного управління. Поняття baseline.

6.1. Проблематика

Всім відомо, що на великих промислових підприємствах, в магазинах, книжкових видавництвах і ін. існують склади. Основне завдання складу – забезпечити зберігання і *доступ* до матеріальних *активів*: *товарам*, виробам, книгам і ін. Тобто різні матеріальні *активів* стає так багато, що необхідна спеціальна *служба по* їх обліку. Виявляється, що не досить складати, наприклад, все, наявні в книговидавництві книги в спеціальну кімнату і видавати їх власникам тиражу, коли вони за ними прийдуть. Книг виявляється дуже багато, а процедура видачі тиражу – не зовсім тривіальною. Потрібно, щоб власник приніс велику кількість супровідних документів, і всі вони повинні бути перевірені перед видаванням книг. А на самому складі необхідно підтримувати порядок, щоб було можливо швидко знайти потрібні книги (як показує *досвід*, вони можуть там досить довго знаходитися). Ще складніша процедура роботи з книгами в бібліотеці – там додаються ще каталоги, розподілені книжкові сховища, необхідність підтримувати хороший стан книг, а також контролювати повернення їх в бібліотеку після певного терміну. Аналогічним чином працює склад на будь-якому заводі, фабриці і так далі

Розглянемо тепер проект *по розробці програмного забезпечення*. Що в нім є аналогом матеріальних *активів* на звичайному виробництві? Безумовно, не столи і стільці, якими користуються розробники. І навіть не комп'ютери, запчастини до них і інше устаткування. Обліку і контролю, схожий на складський, вимагають **файли** проекту. У програмному проекті

їх дуже багато – сотні і тисячі навіть для відносно невеликих проектів. Адже створити новий *файл* дуже легко. Багато технологій програмування підтримують стиль, коли, наприклад, для кожного класу створюється свій окремий *файл*.

Файл – це віртуальна інформаційна *одиниця*. У чому головна відмінність файлу від матеріальних одиниць обліку? У тому, що у файлу може бути **версія**, і не одна, і породити ці версії дуже легко – досить скопіювати даний *файл* в інше місце на диску. Тоді як матеріальні предмети існують на складі самі *по* собі, і для них немає поняття версії. Так, може бути декілька однотипних предметів, різних заготовок виробу різного ступеня готовності. Але все це не те... А версія файлу – це дуже непростий *об'єкт*. Чим одна версія відрізняється від іншої? Декількома строчками тексту або повністю оновленим змістом? І яка з двох і більш за версії головніше, краще? До цього додається ще і те, що багато *робочих продуктів* можуть складатися з набору файлів, і кожен з них може мати *по* декілька версій. Як зібрати *коректну версію продукту*?

У результаті в програмному проекті починають відбуватися містичні і загадкові події.

- Програма, яку ретельно тестували, на показових *випробуваннях* не працює
- Функціональність, про яку довго просив замовник і яка була, нарешті, додана в продукт, і нова версія урочисто відіслана замовникові, таємничим чином зникла з продукту.
- На комп'ютері розробника програма працює, а у замовника – ні..

Розгадка проста – вся справа у версіях файлів. Там, де все добре, присутні файли однієї версії, а там, де все погано – інший. Але біда в тому, що "версія всього продукту" – це абстрактне поняття. На ділі є версії окре-

мих файлів. Один або декілька файлів в постачанні продукту мають не ту версію – все, справа погано. Необхідно управляти версіями файлів, а то подібна містика може стати величезною проблемою.

Вона серйозно гальмує внутрішню роботу. То розробники і тестери працюють з різними версіями системи, то підсумкове *збирання системи* вимагає спеціальних зусиль всього колективу. Більш того, можливі неприємності на рівні управління. Різні курйозні ситуації, коли заявлена функціональність відсутня або не працює (знову не ті файли послали!), можуть сильно погіршувати відносини із замовником. Незадоволений замовник може зажадати навіть грошової компенсації за те, що виникаючі помилки дуже довго виправляються. А буде тут не довге, коли розробники не можуть відтворити і виправити помилку, оскільки не можуть точно визначити, з яких же початкових текстів була зібрана дана версія!

Отже, стає зрозуміло, що в програмних проектах необхідна спеціальна *діяльність по* підтримці файлових *активів проекту* в порядку. Вона і називається **конфігураційним управлінням**.

Виділимо два основні завдання в *конфігураційному управлінні* – **управління версіями** і **управління збірками**. Перше відповідає за управління версіями файлів і виконується в проекті на основі спеціальних програмних пакетів – **засобів версійного контролю**. Існує велика кількість таких засобів – *Microsoft Visual* Sourcesafe, *IBM* Clearcase, cvn, subversion і ін. Управління *збірками* – це автоматизований процес трансформації початкових текстів ПЗ в пакет виконуваних модулів, що враховує численні налагоджування проекту, налагоджування *компіляції*, і інтегрований з процесом *автоматичного тестування*. Ця процедура є могутнім засобом *інтеграції* проекту, основою *ітеративної* розробки.

6.2. Одиниці конфігураційного управління

Так чим же ми управляємо в рамках цієї *діяльності*? Чи будь-якими файлами, які є в проекті? Ні, не будь-якими, а лише тими, які змінюються. Наприклад, файли з використовуваним в проекті купувальним ПЗ винен собі спокійний покоїтися на *CD-дисках* або в *локальній мережі*. Книги, документи із зовнішніми стандартами, використовуваними в проекті (наприклад, в теле-комунікаціях дуже багато різних стандартів на *мережеві інтерфейси*) і ін. також повинні просто зберігатися там, де кожен охочий їх може узяти. Як правило, така інформації в проекті небагато, але, зрозуміло, вона повинна бути в порядку. Проте ради цього спеціальний вид *діяльності* в проекті не потрібний.

Отже, *конфігураційне* управління має справу із змінними в процесі продуктами, що складаються з наборів файлів. Такі продукти прийнято називати **одиницями конфігураційного управління** (*configuration management items*). Ось приклади:

1. призначена для користувача документація;
2. проектна документація;
3. початкові тексти ПЗ;
4. пакети тестів;
5. інсталяційні пакети ПЗ;
6. *тестові звіти*.

У кожної одиниці конфігураційного управління повинно бути наступне.

1. Структура – набір файлів. Наприклад, призначена для користувача документація в *html* повинна включати індекс-файл і набір *html-файлів*, а також набір винесених картинок (*gif* або *jpeg-файли).* Ця структура повинна бути добре визначена і відстежуватися при *конфігураційному*

управлінні – що всі файли не втрачені і присутні, мають однакову версію, коректні посилання один на одного і так далі

2. Відповідальну особу і, можливо, групу тих, хто їх розробляє, а також ширшу і менш відповідальну групу тих, хто користується цією інформацією. Наприклад, визначеним *програмним компонентом* можуть в проекті користуватися багато розробників, але відповідати за його розробку, виправлення помилок і ін. винен хтось один.

3. Практика конфігураційного управління – хто і в якому режимі, а також в яке місце викладає нову версію елементу конфігураційного управління в засіб управління версіями, правила іменування і коментування елементу в цій версії, подальші маніпуляції з ним там і ін. Більш високорівневі правила, зв'язані, наприклад, з правилами зміни тестів і тестових пакетів при зміні коду. Проте, десь тут лежить вододіл між *конфігураційним управлінням* і іншими видами *діяльності* в проекті

4. Автоматична процедура *контролю цілісності* елементу – наприклад, збірка для початкових текстів програм. Є не у всіх елементів, наприклад, може не бути у документації, тестових пакетів.

Елементи конфігураційного управління можуть утворювати *ієрархію*. Приклад наведений на Рис. 6.1.

6.3. Управління версіями

Управління версіями файлів. Оскільки програмісти мають справу з величезною кількістю файлів, багато файлів водночас можуть бути необхідні декільком людям і важливо, щоб всі вони постійно складали єдину, як мінімум, версію продукту, що компілюється, необхідно, щоб була налагоджена робота з файлами з початковим кодом. Також може бути налагоджена робота і з іншими *типами файлів*. У цій ситуації файли виявляються

самими молодшими (*за* ієрархією включення) елементами конфігураційного управління.

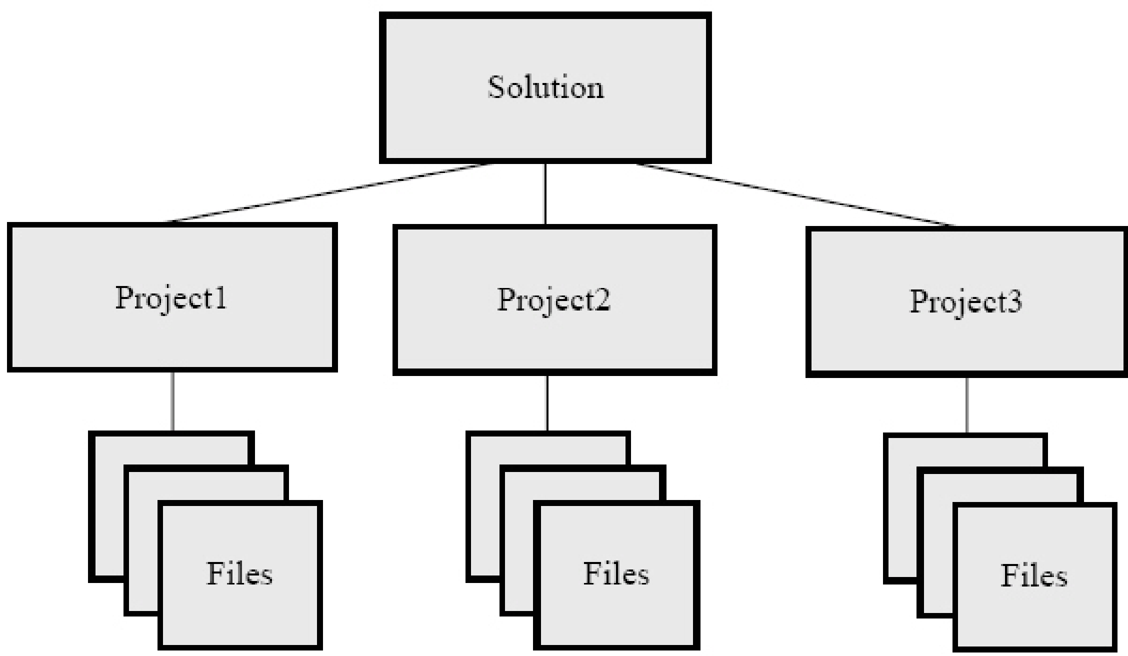

Рис. 6.1. Ієрархічна структура конфігураційного управління проектом

Управління версіями складених конфігураційних об'єктів. Поняття "гілки" проекту.

Одночасно може існувати декілька версій системи – і в сенсі для різних замовників і ін. (так би мовити, у великому, справжньому сенсі), і в сенсі одного проекту, одного замовника, але як різний набір початкових текстів. І в тому і в іншому випадку в засобі управління версіями утворюються різні **гілки**. Зупинимося трохи докладніше на другому випадку.

Кожна *гілка* містить повний образ початкового коду і інших *артефактів*, що знаходяться в *системі контролю версій*. Кожна *гілка можете* розвиватися незалежно, а може в певних крапках інтегруватися з іншими гілками. В процесі *інтеграції зміни*, які були *виконані* в одній з гілок, напівав-

томат переносяться в іншу. Як приклад можна розглянути наступну структуру розділення проекту на гілки.

- · V1.0 – гілка, відповідна випущеному *релізу*. Внесення змін до таких гілок заборонені і вони зберігають образ коду системи на момент випуску релізу.
- · Fix V1.0.1 – гілка, відповідна випущеному пакету виправлень до певної версії. Подібні гілки відгалужуються від початкової версії, а не від основної гілки і заморожуються відразу після виходу пакету виправлень.
- · Upcoming (V1.1) – гілка, відповідна *релізу*, що підготовлюваному до випуску і знаходиться у стадії стабілізації. Для таких гілок, як правило, діють строгіші правила і робота в них ведеться формальніше.
- · Mainline – гілка, відповідна основному напряму розвитку проекту. У міру дозрівання саме від цієї гілки відходять гілки підготовлюваних релізов.
- · WCF *Experiment* – гілка, створена для перевірки деякого технічного рішення, переходу на нову технологію, або внесення великого пакету змін, що потенційно порушують працездатність коду на тривалий час. Такі гілки, як правило, стають доступними лише для певного круга розробників і убиваються по *завершенню робіт* після інтеграції з основною гілкою.

6.4. Управління збірками

Отже, чому ж процедура *компіляції* і створення exe *dll* файлів *за* початковими файлами проекту – така важлива процедура? Тому що вона ба-

гато разів в день виконується кожним розробником на його власному комп'ютері, з його власною версією проекту. При цьому відрізняється:

- набір підпроектів, що збираються розробником; він може збирати не весь проект, а лише якусь його частина; інша частина або їм не використовується зовсім, або не збирається заново дуже давно, а по факту вона давно змінилася;
- відрізняються параметри компіляції.

При цьому якщо не збирати регулярно підсумкову версію проекту, то загальна *інтеграція* може виявити багато різних проблем:

- невідповідність один одному різних частин проекту;
- наявність специфічних помилок, що виникли через те, що окремі проекти розроблялися без урахування параметровкомпіляциі (зокрема, перехід в *Visual Studio* з *debug* на *release* версію часто супроводжується появою численних проблем).

У зв'язку з цим процедуру збірки проекту часто автоматизують, тобто виконують не з *середовища розробки*, а із спеціального скрипта – *build-скрипта*. Цей *скрипт* використовується тоді, коли розробникові потрібна повна *збірка* всього проекту. А також він використовується в процедурі безперервної *інтеграції* (*continues integration*) – тобто регулярній збірці всього проекту (як правило – щоночі). Як правило, процедура безперервної *інтеграції* включає і *регресійне тестування*, і часто – створення інсталяційних пакетів. Загальна схема автоматизованої збірки представлена на Рис. 6.2.

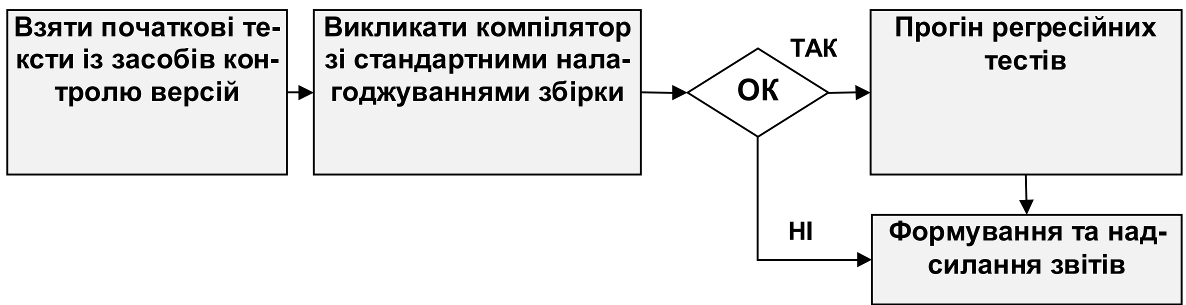

Рис. 6.2 Загальна схема автоматизованої збірки

Тестувальники повинні тестувати *по можливості* підсумкову і цілісну версію продукту, так що результати регулярної збірки виявляються дуже затребувані. Крім того, наявність базової, актуальної, цілісної версії продукту дозволяє організувати розробку в ітеративно-інкрементальному стилі, тобто на основі внесення змін. Такий стиль розробки має назву baseline-метод.

6.5. Поняття baseline

Baseline – це *базова*, остання цілісна версія деякого продукту розробки, наприклад, документації, програмного коду і так далі Мається на увазі, що розробка йде не суцільним потоком, а з фіксацією проміжних результатів у вигляді поточної офіційної версії активу, що розробляється. Ухвалення такої версії супроводжується додатковими діями *відносно оформлення*, згладжування, тестування, включення лише закінчених фрагментів і так дал.і Цей результат можна подивитися, віддати тестувальникам, передати замовникові і так далі. *Baseline* є хорошим засобом синхронізації групової роботи.

Baseline може бути зовсім простою – гілкою в засобі управління версіями, де розробники зберігають поточну версію своїх початкових код. Єдиною вимогою в цьому випадку може бути лише загальна компілірує-

місць проекту. Але *підтримка baseline* може бути формально складною процедурою, як показано на Рис. 6.3.

Baseline може також підтримуватися безперервною інтеграцією.

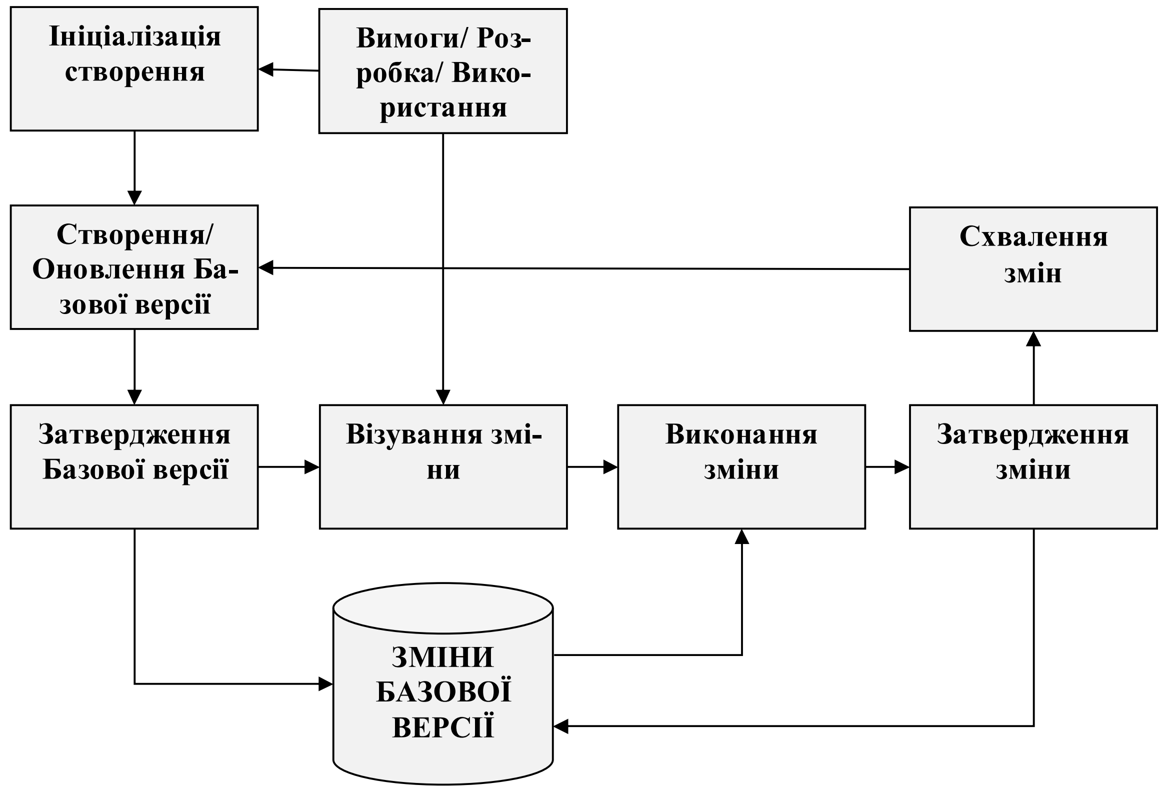

Рис. 6.3 Процедура формальної підтримки ***baseline***

Важливо, що *Baseline* (особливо у випадку з програмними активами) не повинна встановлюватися дуже рано. Спочатку потрібно написати якусь кількість коду, щоб було що інтегрувати. Крім того, спочатку багато уваги приділяється розробці основних архітектурних рішень, і цілісна версія виявляється не затребуваною. Але починаючи з якогось моменту вона просто необхідна. Який цей момент – вирішувати членам команди. Нарешті, існують проекти, де автоматична *збірка* не потрібна зовсім – це прості прое-

кти, що розробляються невеликою кількістю учасників, де немає великого кількість початкових текстів програм, проектів, складних параметрів *компіляції*.

Контрольні запитання

1. Які існують проблем в проектах, де немає хорошого конфігураційного управління?
2. Які завдання виконує конфігураційне управління в проекті?
3. Дати формальне визначення конфігураційному управлінню.
4. В чому полягає відома суперечність між абсолютним збереженням і зручним доступом?
5. Які артефакти проекту можуть "підпадати" під конфігураційне управління?
6. Які артефакти проекту можуть не "підпадати" під конфігураційне управління?
7. Що є головним артефактом конфігураційного управління?
8. Які основні функції версійного контролю?
9. Що таке управління збірками?
10. Що таке безперервна інтеграція?
11. Що таке baseline?

Глава 7. Управління якістю та тестування

Стандартизація якості. Методи забезпечення якості ПЗ. Поняття тестування. Тестування чорного ящика. Тестування білого ящика. Інструменти тестування. Критерії тестування. Види тестування. Робота з помилками. Засоби контролю помилок (bug tracking systems).

7.1. Стандартизація в сучасному бізнесі і промисловості.

Розвиток світового ринку призвів до того, що багато товарів і послуг почали розповсюджуватися в усьому світі, набули розвитку глобальні сервіси, зокрема, телекомунікаційні, банківські. Для того, щоб усунути технічні бар'єри в промисловості, торгівлі і бізнесі, які виникли унаслідок того, що в різних країнах для одних і тих самих технологій і товарів діяли різнорідні стандарти, почали створюватися національні і міжнародні комітети із стандартизації. Зупинимося на **найвідоміших міжнародних комітетах**.

1. 1865 рік – створений комітет, який зараз має назву ***ITU*** **(*International* Telecommunication *Union*)**. На сьогоднішній день його штаб-квартира знаходиться в Женеві (Швейцарія), а ITU є частиною ООН. Його основне завдання – стандартизація телекомунікаційних протоколів і інтерфейсів з метою підтримки і розвитку глобальної світової *телекомунікаційної мережі*. Найвідомішими стандартами *ITU* є:

- ***ISDN*** (цифровий телефонний зв'язок, який об'єднує телефонні сервіси і передачу даних);
- ***ADSL*** (широко відома модемна технологія, що дозволяє використовувати телефонну лінію для виходу в Інтернет і при цьому не блокує звичайний телефонний сервіс);

- ***OSI*** (модель відкритого 7-рівневого мережевого протоколу, на якій базуються всі сучасні стандартні мережеві інтерфейси і протоколи; також є стандартом ISO);
- мови візуального проектування телекомунікаційних систем, SDL і MSC, які пізніше влилися в UML.

Багато стандартів ***ITU*** перекладаються державними мовами країн світу і перетворюються на національні стандарти.

2. 1946 рік – створена організація ***ISO*** (***International Organization* for *Standardization***). Мета організації – сприяння розвитку стандартизації в світі, а також суміжних видів діяльності з метою забезпечення міжнародного обміну товарами і послугами, сприяння і розвиток співпраці в інтелектуальній, науково-технічній і економічній сферах. До теперішнього часу створено близько 17 000 стандартів у самих різних галузях промисловості – продовольчі і інші товари, різне устаткування, банківські сервіси і так далі. Наведемо деякі стандарти.

- Серія стандартів ***ISO 9000***. Ця серія націлена на стандартизацію якості товарів і послуг. Стандарти надають визначення якості, визначення системи підтримки якості на всіх життєвих фазах виробу, товару, послуги (проектування, розробка, комерціалізація, встановлення і обслуговування), опис процедур з поліпшення діяльності компаній, промислового виробництва.
- ***ISO/IEC 90003:2004*** – адаптація стандартів ISO 9000 до виробництва ПЗ з метою забезпечення якості в життєвому циклі ПЗ.
- ***ISO 9126:2001*** – визначення якісного ПЗ і різних атрибутів, що описують цю якість.

Багато стандартів ISO перекладаються українською мовою і перетворюються на українські стандарти у вигляді ДСТУ. Є багато стандартів в галузі інформаційних технологій, а також декілька – в галузі програмної інженерії. На відповідність стандартам ISO існує сертифікація. Зокрема, компанії сертифікуються на відповідність стандартам ISO 9000, тобто на якісний процес розробки ПЗ.

3. 1988 рік, утворення організації ***ETSI (European Telecommunications Standards Institute)***, штаб-квартира в технологічному парку на північному заході від Антіб й на південному заході від Ніцци у Франції. ***ETSI*** є незалежною, некомерційною, організацією із стандартизації в телекомунікаційній промисловості (виробники устаткування і оператори мереж) в Європі. Найвідоміші стандарти – ***GSM***, система професійного мобільного радіозв'язку ***TETRA***.

Міжнародні комітети, які безпосередньо пов'язані з розробкою ПЗ.

1. 1984 рік – створення ***SEI*** (*Software Engineering* Institute) на базі університету Карнегі-Меллон в м. Пітсбурзі (США). Ініціатор і головний спонсор – міністерство оборони США. Основне завдання – стандартизація в галузі програмної інженерії, розробка критеріїв для сертифікації надійних і зрілих компаній (що в першу чергу цікавить Міноборони США для виконання його замовлень). Найвідоміші продукти – стандарт ***CMM***, ***CMMI***, розробки в галузі сімейства програмних продуктів (*product* lines). Ці продукти зробили крок далеко за межі військових розробок США, їх використання і розвиток стало міжнародною діяльністю. Деякі продукти SEI стандартизовані також і ISO. На відповідність CMM/CMMI виконується сертифікація.

2. 1963 рік – створення *IEEE* (Institute of *Electrical* and *Electronics Engineers*). Веде історію з кінця XIX століття, в контексті промислової стандартизації в США. Зараз *IEEE* міжнародна некомерційна асоціація фахівців в галузі техніки, світовий лідер в галузі розробки стандартів для радіоелектроніки і електротехніки. Штаб-квартира знаходиться в США, існують численні підрозділи в різних країнах, включаючи і Українську Секцію IEEE (IEEE Ukraine Section). IEEE видає третю частину світової технічної літератури, що відноситься до застосування радіоелектроніки, комп'ютерів, систем управління, електротехніки, зокрема (січень 2008) 102 реферативних наукових журналів і 36 галузевих журналів для фахівців, проводить в рік більше 300 великих конференцій, брала участь в розробці близько 900 стандартів, що діяли або діють і дотепер.
3. 1989 рік – група американських IT-компаній (зокрема Hewlett Packard, Sun Microsystems, Canon) організували ***OMG*** (*Object Management Group*). Зараз включає близько 800 компаній-членів. Основний напрямок – розробка і просування об'єктно-орієнтованих технологій і стандартів, зокрема для створення незалежних програмних додатків рівня підприємств. Відомі стандарти ***CORBA***, ***UML***, ***MDA***.

Всі ці комітети і організації включають програмну інженерію в сферу своєї діяльності, співробітничають, видають спільні стандарти, використовують напрацювання один одного і так далі.

7.2. Стандартизація якості.

З погляду тестування ПЗ в цих стандартах слід звернути увагу на стандартизацію якості (в контексті тестування) продукції, що спочатку випускається, а потім поширюються і на процеси з її розробки. Тут спрацьо-

ває ідея про те, що якісного результату неможна створити без якісного процесу. Забезпечення якості є більш загальним контекстом для тестування.

Якість продукту або сервісу, який призначений для споживача, визначається в стандарті ISO 9000:2005, як ступінь відповідності його характеристик вимогам – обов'язковим або таким, що мається на увазі.

7.3. Методи забезпечення якості ПЗ.

Не претендуючи на абсолютну повноту, перерахуємо різні способи контролю якості, які використовуються на практиці підчас розробки ПЗ.

- ***Налагоджування якісного процесу***, іншими словами – це вдосконалення процесу. Для комплексного поліпшення процесів в компанії (підхід ***technology push***) компаніями-розробниками ПЗ використовуються стандарти CMM/CMMI та стандарти серії ISO 9000 (з подальшою офіційною сертифікацією). Застосовуються і локальні стратегії, які є менш затратними і більш спрямовані на вирішення окремих проблем (підхід ***organization pull***).
- ***Формальні методи*** – використання математичних формалізмів для доказу коректності, специфікації, перевірки формальної відповідності, автоматичної генерації і т.д. Серед них:
  - доказ правильності роботи програм;
  - перевірка на моделях певних властивостей (model cheking);
  - статичний аналіз коду за деревом розбору програми (наприклад, перевірка коректності коду за певними критеріями – акуратна робота з пам'яттю, пошук мертвого коду і ін.);

- модельно-орієнтоване тестування (model-based testing): автоматична генерація тестів і тестового оточення за формальними специфікаціями вимог до системи і так далі.

На практиці застосовуються обмежено через необхідність серйозної математичної підготовки користувачів, складність в освоєнні, велику роботу з розгортання. Ефективні для систем, що мають підвищені вимоги до надійності. Також є випадки ефективного використання засобів, які засновані на цих методах, з боку висококваліфікованих фахівців.

- ***Дослідження і аналіз динамічних властивостей ПЗ***. Як приклад, широко використовується профілізація – дослідження використання системою пам'яті, її швидкодія і ін. характеристик шляхом запуску і безпосередніх спостережень у вигляді графіків, звітів і ін. Зокрема, цей підхід використовується у випадку розпаралелювання програм, під час пошуку "вузьких" місць. Ще приклад – галузь, яка має назву "моделювання і аналіз продуктивності" (performance modeling and analysis). В цьому випадку моделюється навантаження системи з боку оточення (кількість одночасних користувачів системи, мережевий трафік і ін.) та спостерігається поведінка системи.
- ***Забезпечення якості коду.*** Сюди відноситься цілий комплекс різних заходів і методів. Нижче наведені найбільш відомі з них.
  - *Розробка стандартів оформлення коду* в проекті і контроль за дотриманням цих стандартів. Сюди входять правила на створення ідентифікаторів змінних, методів і імен класів, на оформлення коментарів, правила використання стандартних для проекту бібліотек і так далі.

- Регулярний *рефакторінг* для запобігання створення з коду "вермішелі". Існує тенденція погіршення структури коду у випадку внесення до нього нової функціональності, виправлення помилок і ін. З'являється надмірність, утворюються невживані або слабо використовувані фрагменти, структура стає заплутаною і важкою для розуміння. Рефакторінг – це регулярна діяльність з переписуванню коду, але не з метою додавання нової функціональності, а для поліпшення його структури. Рефакторінг з'явився в контексті "гнучких" методів, в даний момент активно підтримується різними середовищами розробки ПЗ.
- Різні *варіанти інспекції коду*, наприклад, техніка *peer code review*. Остання полягає в тому, що код кожного учасника проекту, вибірково, розглядається і обговорюється на спеціальних зустрічах (code review meetings), і робиться це регулярно. Практика показує, що в цілому код поліпшується.
- Є ще такий підхід, як *"вичитування" коду*, що використовується, наприклад, під час розробки критичних систем реального часу. Ним також займаються розробники, але їхня роль в даному проекті – вичитування, а не розробка.

- ***Тестування***. Найпоширеніший спосіб контролю якості ПЗ, який існує, фактично, в кожному програмному проекті.

7.4. Тестування

Тестування – це перевірка відповідності між реальною поведінкою програми і її очікуваною поведінкою в спеціально заданих, штучних умовах. Розберемо це визначення за складовими.

Очікувана поведінка програми. Початковою інформацією для тестування є знання про те, як система повинна поводитися, тобто вимоги до неї або до її окремої частини. Найпоширенішим способом тестування є тестування методом **чорного ящика**, тобто коли реалізація системи є недоступною для тестувальників, а тестується лише її інтерфейс. Часто це закріплюється і організацією колективу – тестувальники виявляються окремими співробітниками і в деяких компаніях вони навіть принципово не спілкуються з розробниками, щоб мінімально знати деталі реалізації і максимально повно виступити в ролі перевіряючої інстанції. Існує тестування методом **білого ящика**, коли код програм є доступним тестувальників і використовується як джерело інформації про систему. Його схема наведена на Рис. 7.1.

На Рис. 7.1 видно, що на основі вимог до системи створюється реалізація і тестова модель системи. Тестування є зіставлення два цих уявлень з метою виявити їх невідповідності. Чим більш незалежні один від одного будуть ці уявлення, тим більше пуття від їх зіставлення. Інакше, якщо тестувальники істотно використовують інформацію про реалізацію системи підчас складання тестів, то вони можуть мимоволі ввести у тести помилки реалізації. Знайдена при тестуванні невідповідність – це ще не помилка, оскільки самі тестувальники могли неправильно зрозуміти вимоги, тому і в тестах, і в засобах тестування могли бути власні помилки.

Даний підхід закріплюється також і в організації колективів програмістів – тестувальники, як правило, відокремлені від розробників. Це різні люди, що виконують несумісні ролі в MSF. Існує приклад однієї американської компанії де розробники і тестувальники розміщувались на різних поверхах, ходили в різному одязі (тестувальники в костюмах, розробники – в светрах) і начальство не заохочувало неробочі відносини між цими групами. Це, звичайно ж, крайність, але вона ще раз підкреслює, як важливо, щоб точка зору на систему у тестерів відрізнялася від точки зору розроб-

ників. Але, звичайно, і та, і інша точки зору повинні виходити із загального бачення системи – її вимог.

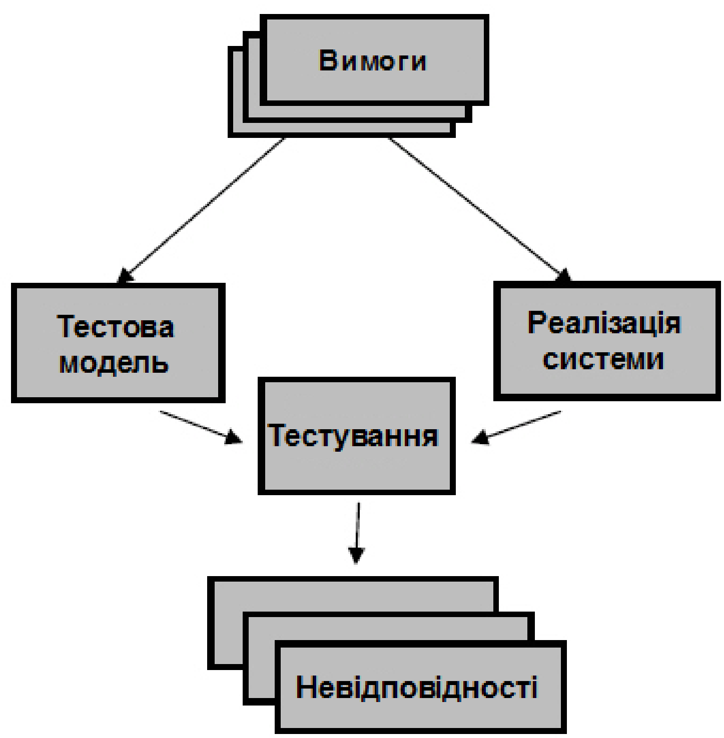

Рис. 7.1. Схема тестування методом білого ящика

Спеціально задані, штучні умови, – це ті умови, за яких виконується тестування. При цьому ключовим аспектом тут є наявність **тестів –** відтворених кроків маніпуляції з системою, які призводять до її некоректної роботи. Концепція тесту дуже важлива, оскільки необхідно не просто виявити некоректну поведінку системи, а створити і зафіксувати алгоритм відтворення помилки – щоб повторити його для розробника, або щоб розробник сам зміг відтворити помилку. Якщо помилка не відтворюється, то не існує можливості її виправити.

Тести можуть бути "ручними" і автоматизованими. *"Ручний" тест* – це послідовність дій тестувальника, яку він (або розробник) може відтворити і помилка станеться. Як правило, в засобах контролю помилками такі

послідовності дій містяться в описі помилки. *Автоматизований тест* – це деяка програма, яка впливає на систему і перевіряє ту або іншу її властивість. Автоматизований тест, в порівнянні з "ручним", можна легко відтворювати без участі людини. Можна створювати набори тестів і "проганяти" їх дуже часто, наприклад, в режимі регресійного тестування. Крім того, автоматизовані тести можна генерувати за більш високорівневими специфікаціями, наприклад, за формально описаними вимогами до системи. А тести для компіляторів можна генерувати за формальним описом мови програмування.

Таким чином, переваги автоматизованих тестів перед "ручними" є очевидними. Звернемо увагу на труднощі автоматизованого тестування.

По-перше, для того, щоб автоматично запускати тести, потрібні відповідні програмні продукти, які також є невід'ємною частиною спеціально заданих, штучних умов. Їх називають **інструментами тестування**. У їх завдання входить запуск тесту на системі, "прогін" цілого пакету тестів, а також аналіз отриманих результатів і їх обробка.

Крім того, важливим завданням інструментів тестування є забезпечення доступу тесту до системи через деякий її інтерфейс. Доступ до системи може виявитися складним, наприклад, через політичні обставини, коли сторонніми розробниками створюється підсистема деякої стратегічної системи, і доступ до цієї глобальної системи у субпідрядників сильно обмежений. Або через апаратні обмеження – важко "залізти" на "сервер", де працює цільовий код системи.

Крім того, часто важко виконати "безшовне" тестування системи, створюючи на неї мінімальне навантаження і добираючись при цьому до всіх аспектів її функціонування. В цілому, налагоджування і розгортання готових, сторонніх тестових інструментів часто виявляється дорогим і не-

тривіальним завданням. Розробка своїх власних тестових інструментів також є непростою.

По-друге, часто виникає проблема ресурсів для автоматичного тестування. Особливо при автоматичній генерації тестів: часто є можливість автоматично згенерувати дуже велику кількість тестів, і якщо їх ще й регулярно виконувати в режимі безперервної інтеграції, то може не вистачити наявних системних ресурсів. При цьому якість тестування може виявитися незадовільною – помилки знаходяться не часто, або взагалі не знаходяться. Річ у тому, що кількість всіх можливих станів програмної системи дуже велика, і тестування не може перевірити їх всіх. На практиці, в реальних проектах, визначають **критерії тестування**, які визначають ту "планку" якості, яку необхідно досягти в цьому проекті. Адже хороша якість коштує дорого і очевидно, що різне ПЗ має різну якість, наприклад, система управління ядерним реактором і текстовий редактор. На практиці, часто, якість ПЗ визначається бюджетом проекту при його розробці. Далі, через обмеженість ресурсів на тестування часто доцільно буває визначити ті аспекти ПЗ, які найбільш важливі - як для загальної працездатності системи, так і для замовника. Наприклад, у разі тестування WEB-додатку, що надає послугу із створення оголошень про продаж нерухомості, такими критеріями могли би бути:

- відповідність переходів складного майстра, зокрема, у зв'язку з можливістю переходів назад;
- цілісність введених користувачем даних про створювані оголошення.

Нарешті, окрім обмеження кількості тестів для відбору, важливим є їх прогін на деяких (не на всіх можливих!) вхідних даних. Часто тут застосовують **принцип факторизації –** безліч всіх можливих вхідних значень розбивають на значущі з погляду тестування класи і "проганяють" тести не

на всіх можливих вхідних значеннях, а беруть лише один набір значень з кожного класу. Наприклад, тестують деяку функцію системи на її граничні значення – дуже великі значення параметрів, дуже маленькі і ін. Часто факторизацію зручно виконувати, виходячи з вимог до даної функції. Також буває корисним подивитися на її реалізацію і "пройтися" тестами різними її логічними гілками (які породжуються, наприклад, умовними операторами).

7.5. Види тестування.

Не претендуючи на повноту, виділимо такі види тестування.

- **Модульне тестування –** тестується окремий модуль, у відриві від решти системи. Найпоширеніший випадок застосування – тестування модуля самим розробником, перевірка того, що окремі модулі, класи, методи виконують дійсно те, що від них очікується. Різні середовища розробки широко підтримують засоби модульного тестування – наприклад, є популярною вільно поширювана бібліотека Nunit для Visual Studio, Junit – для Java і так далі. Створені розробником модульні тести часто включаються в пакет регресійних тестів і можуть запускатися багато разів.
- **Інтеграційне тестування –** два і більше компонентів тестуються на сумісність. Це дуже важливий вид тестування, оскільки різні компоненти можуть створюватися різними людьми, в різний час, на різних технологіях. Цей вид тестування, безумовно, повинен застосовуватися самими програмістами, щоб, як мінімум, упевнитися, що все живе разом в першому наближенні. Далі тонкощі інтеграції можуть досліджувати тестувальники. Необхідно відзначити, що такого роду помилки – "помилки на стиках" – непросто виявляти і

усувати. Під час розробки всі компоненти одночасно можуть не бути готові, інтеграція відкладається, а в кінці проекту виявляються важкі помилки (в тому сенсі, що їх усунення вимагає істотної роботи). В такому випадку рішенням буває рання інтеграція системи і надалі використання практики постійної інтеграції.

- **Системне тестування –** це тестування всієї системи в цілому, як правило, через призначений для її користувача інтерфейс. При цьому тестувальники, менеджери і розробники акцентуються на тому, як ПЗ виглядає і працює в цілому, чи зручне воно, чи задовольняє воно очікуванням замовника. При цьому можуть відкриватися різні дефекти, такі як незручність у використанні тих або інших функцій, забуті або такі, що були не до кінця зрозумілими, вимоги.

- **Регресійне тестування –** тестування системи в процесі її розробки і супровід на регрес. Тобто перевіряється, що зміни у системі не погіршили вже існуючої функціональності. Для цього створюються пакети регресійних тестів, які запускаються з певною періодичністю, – наприклад, в пакетному режимі, який пов'язаний з процедурою постійної інтеграції.

- **Тестування навантаженням –** тестування системи на коректну роботу з великими об'ємами даних. Наприклад, перевірка баз даних на коректну обробку великого (граничного) об'єму записів, дослідження поведінка серверного ПЗ у разі великої кількості клієнтських з'єднань, експерименти з граничним трафіком для мережевих і телекомунікаційних систем, одночасне відкриття великого числа файлів, проектів і так далі.

- **Стресове тестування –** тестування системи на стійкість до непередбачених ситуацій. Цей вид тестування потрібний далеко не для кожної системи, оскільки має на увазі високу планку якості.

- **Приймальне тестування –** тестування, що виконується на етапі прийманні системи замовниками. Більш того, різні стандарти часто включають набори приймальних тестів. Наприклад, існує великий пакет тестів, що підтримуються компанією SUN Microsystems і є обов'язковими для прогону для всіх нових реалізацій Java-машини. Вважається, що лише після того, як всі ці тести успішно проходять, нова реалізація має право називатися Java.

7.6. Робота з помилками

Між програмістами і тестувальниками необхідний спеціальний **інтерфейс спілкування**. Адже помилок знаходиться багато, їх виправлення вимагає часу, і після їх виправлення розробниками тестувальники повинні упевнитися, що вони дійсно виправлені. Крім того, менеджерам потрібна статистика щодо знайдених і виправлених помилок – це хороший інструмент контролю проекту. Все це зображено на Рис. 7.2. Щоб справитися з цим потоком інформації і забезпечити необхідні в роботі зручні сервіси, існує спеціальний клас програмних засобів – **засоби контролю помилок** (*bug tracking systems*).

Як правило, опис помилки в системі контролю помилок має такі основні атрибути:

- відповідального за її перевірку – тестувальника, який її знайшов і який перевіряє, що виправлення, які були виконані розробником, дійсно усувають помилку;

- відповідального за її виправлення – розробника, якому помилка відправляється на виправлення;
- стан, наприклад, помилка знайдена, помилка виправлена, помилка закрита, помилка знов виявилася і так далі.

Цей список істотно доповнюється в різних програмних засобах контролю помилок, але це основні атрибути.

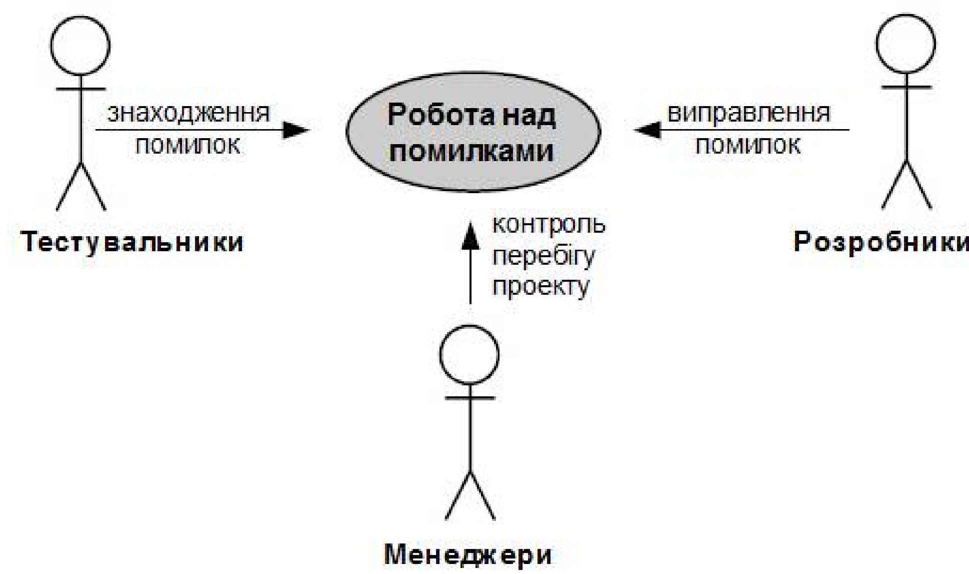

Рис. 7.2. Схема роботи з помилками ПЗ

Використання цих систем давно стало загальною практикою підчас розробки ПЗ, нарівні із засобами версійного контролю і багатьма іншими інструментами. Вони включають:

- базу даних для зберігання помилок;
- інтерфейс до цієї бази даних для внесення нових помилок і завдання їх численних атрибутів, для перегляду помилок на основі різних фільтрів – наприклад, всі знайдені помилки за останній місяць, всі помилки, за які відповідає даний розробник і т.д.;

- мережевий доступ, оскільки проекти все частіше виявляються розподіленими;
- програмний інтерфейс для можливостей програмної інтеграції таких систем з іншим ПЗ, що підтримує розробку ПЗ (наприклад, із засобами безперервної інтеграції – вони можуть автоматично вносити до бази даних знайдені підчас автоматичного прогону тестів помилки).

Дуже важливим для роботи з помилками виявляються різні звіти.

Контрольні запитання

1. Які існують міжнародні організації з питань стандартизації?
2. Які існують міжнародні організації з питань розробки ПЗ?
3. Які існують способи контролю якості ПЗ.
4. Дати визначення тестуванню.
5. Що означає в контексті тестування очікувану поведінку програми?
6. Що входить в штучні, спеціально задані умови дії на систему, які маються на увазі у визначенні тестування?
7. Охарактеризувати важливість концепції тесту?
8. Охарактеризувати переваги автоматичного тестування по відношенню до "ручного"?
9. Охарактеризувати труднощі автоматичного тестування?
10. Навести свої власні приклади проблем з інтерфейсами щодо систем, що тестуються.
11. Як прогін тестів може впливати на поведінку системи?
12. Охарактеризувати принцип факторизації вхідних значень під час тестування?
13. Охарактеризувати різні варіанти організації команди тестувальників.
14. Скільки існує основних видів тестування?
15. Що таке модульне тестування?
16. Що таке інтеграційне тестування?

17.Що таке системне тестування?

18.Що таке регресійне тестування?

19.Що таке тестування навантаженням?

20.Що таке стресове тестування?

21.Що таке приймальне тестування?

22.Що таке bug tracking system?

23.Які основні атрибути присутні в системі контролю помилок?

24.Які основні компоненти системи контролю помилок?

Глава 8. Діаграмна техніка в роботі із знаннями

Випадки використання. Робота з вимогами. Випадки використання в управлінні розробкою. Ітеративний цикл автор/рецензент. Карти пам'яті.

8.1. Метод «випадки використання»

Опис прикладу. Як приклад розглянемо "Телефонну службу прийому заявок". Замовником даної системи є компанія, що володіє *мережею* продуктових магазинів. Ця компанія, окрім звичайної роздрібної *торгівлі* і оптових постачань продуктів окремим їдальням і ресторанам, хоче надавати ще і сервіс *по* обслуговуванню клієнтів *по* телефонних заявках. Клієнт реєструється в компанії, а потім *по телефону*, в слушний для себе час, робить замовлення *товарів*, які до нього привозять додому, і він розплачується. Для цього компанія хоче організувати у себе локальний телефонний центр, що складається з офісної багатоканальної АТС, штату операторів і відповідного *програмного забезпечення*. При цьому в компанії вже існує інформаційна *система з*обробки заявок від постійних дрібнооптових клієнтів, і система, що замовляється, повинна бути об’єднаназ нею.

Робота з вимогами. Випадки або *варіанти використання* (usecases) були запропоновані в кінці 90-х років Айвером Якобсоном, одним з головних авторів мови *UML*, як діаграмний підхід для витягання і первинної формалізації вимог до систем. Вище вже мовилося про складність *по* формуванню єдиної і зв'язної картини вимог до *ПЗ*. Необхідно витягувати вимоги зі всіх можливих джерел, формалізувати в деякому вигляді і обговорити. Цей процес – витягання, формалізація, обговорення – є ітеративним, тобто все робиться не за один присід. Більш того, сам спосіб формалізації повинен бути зручний для обговорення, і в першу чергу, з потенційними користувачами системи, які можуть бути абсолютно не компетентні в ІТ. Їх коментарі, схвалення і незгоди часто є основою ітеративного *витягання ви-*

мог до системи. Крім того, цей спосіб роботи з інформацією повинен вести до створення моделей, зручних в подальшій реалізації системи. Іншими словами, ясно формулювати початкові завдання для розробки. Тобто спосіб формалізації повинен бути простий, зрозумілий і володіти достатньою строгістю. Цим вимогам задовольняють *діаграми випадків* використання, що є на сьогоднішній день складовою частиною стандарту *UML*.

Приклад *діаграми випадків* використання представлений на **Рис**. 8.1.

Отже, все починається з точної *ідентифікації користувачів* майбутньої системи. Це – основа хороших вимог і хорошої системи, адже основне завдання системи – задовольняти потреби майбутніх користувачів. Для цього потрібне їх знати в обличчя... У нашому випадку користувачами системи є *оператор*, *менеджер* і представники технічної підтримки і адміністрування. Система повинна також підтримувати зовнішній *інтерфейс* з системою обробки заявок. Це – четвертий користувач. Ще одним користувачем системи є Петров А.Б. – *директор* департаменту збуту *товарів*, який хоче періодично відстежувати *діяльність* телефонної служби прийому заявок. Для нього створене спеціальне призначене для користувача *місце с екранними формами статистики*.

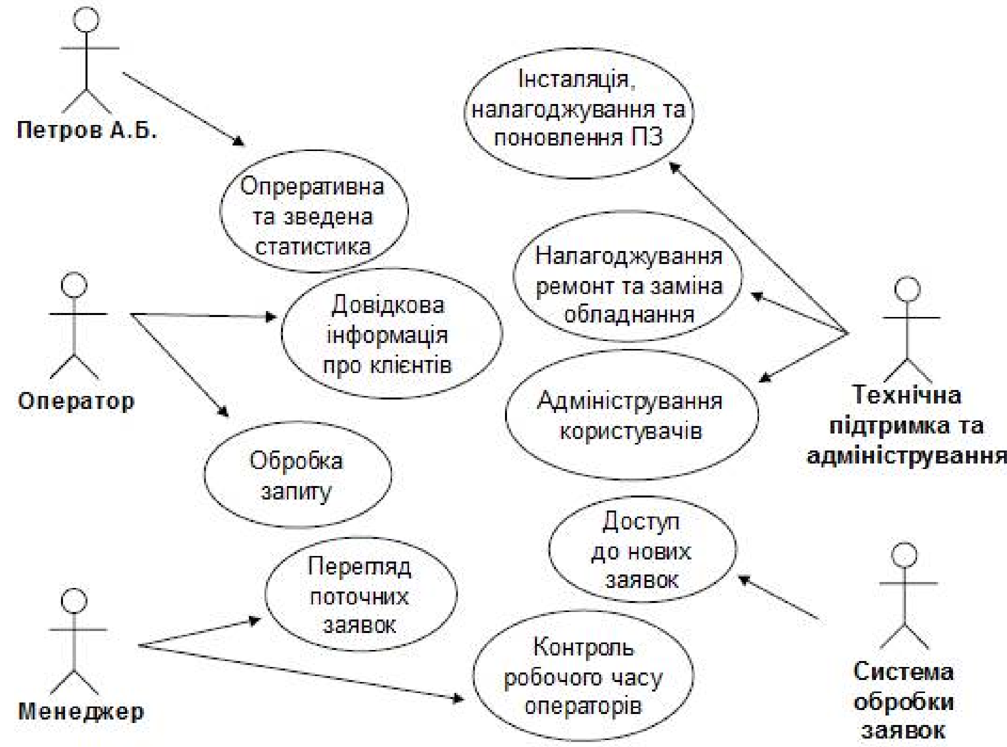

Рис. 8.1. Приклад діаграми випадків використання

Різні користувачі *ПЗ,* що зображені на діаграмах випадківвикористання, називаються **акторами** (*actors*). Актори можуть позначати:

- типових користувачів ("*Менеджер"*, "*Оператор"*, "Технічна підтримка") – працівників компанії, згрупованих по виконуваних обов'язках;
- інші системи, що взаємодіють з даною ("Система обробки заявок");
- виділеного користувача ("Петров А.Б.").

Відзначимо, що ***виділений користувач*** істотно відрізняється від ***типового користувача***. Він, як правило, Важлива Персона, і узгодження функціональності для нього узгоджується особисто з ним. Часто він впливає на *оплату* проекту, від його думки про систему, багато в чому, залежить її успішна здача. Такі персони, ради успіху проекту, потрібно уміти іденти-

фікувати і в рамках всієї системи створювати деяку функціональність спеціально для них і дуже при цьому старатися!

Після *ідентифікації користувачів* відбувається *визначення* випадків використання ними системи. Перш за все, визначається та функціональність системи, яка безпосередньо допомагає користувачам виконувати їх роботу, не пов'язану безпосередньо з *експлуатацією* системи. У нашому випадку, для оператора важливим плюсом від використання системи виявляється можливість діставати швидкий *доступ* до довідкової інформації про клієнтів, а також оперативно обробляти запити, що поступили *по телефону,* на покупки (*список товарів*, ціни, оформлення замовлення і ін.). Для менеджера важливою є можливість оперативного перегляду поточних заявок (виконаних, в роботі, відкладених, за певний період часу і ін.), а також облік *контроль робочого*часу операторів – хто і скільки часу витратив на різного виду роботи (телефонні розмови з клієнтами, оформлення заявки після закінчення розмови і так далі). При цьому важливо відзначити, що *функція* обліку робочого часу може зажадати певних дій з боку операторів – наприклад, натискати відповідну клавішу, йдучи на обід або на перекур. Проте ми не позначили відповідний зв'язок з цим випадком використання з боку оператора, оскільки ця функціональність не допомагає йому в безпосередній роботі, а допомагає його начальникові. Крім того, ми не включили у випадки використання ряд сервісів, пов'язаних з експлуатацією *системи*, наприклад, функцію логіна в систему. Наявність чіткої точки зору при складанні діаграм – застава їх корисності.

Отже, **випадок використання** (use*case*) – це незалежна частина функціональності системи, що володіє результуючою цінністю для її користувачів.

"*Незалежність"* означає, що якщо випадок використання завжди виконується разом з деяким іншим, то, *видно*, один з них потрібно включити

в іншій (який саме в якій, як назвати випадок використання, що вийшов у результаті, – залежить від обставин).

"Результуюча цінність" випадку використання для актора системи має на увазі, що він, даний випадок використання, повинен приносити акторові деякий закінчений і цінний з погляду його бізнесу результат. Будучи реалізований системою, цей випадок використання дійсно робить бізнес актора більш ефективним, *продуктивним*. Тим самим розробка системи фокусується на *бізнес-цілях*, а незначні випадки використання ігноруються, що важливе для компактності моделі. Адже будується не абстрактна модель функцій системи, а набір найважливіших (для замовника і користувачів) *сервісів*, щоб кожен з них правильно зрозуміти і не один не упустити. І надалі *контроль* розробки системи здійснюватиметься саме в термінах цього найважливішого – того, що потрібне замовникові і користувачам.

Випадки використання, відповідні акторам "Технічна *підтримка* і *адміністрування"* і "*Служба* обробки заявок" декілька не вписуються в представлене вище *визначення*. Перш за все, самі ці актори не є користувачами *ПЗ,* що беруть участь в основному бізнес-процесі*обробки* телефонних заявок. "Технічна *підтримка* і *адміністрування"* зайнята підтримкою ПЗ і устаткування системи обслуговування телефонних заявок, а також її *адмініструванням* (додаванням нових користувачів, призначенням ним відповідних прав і ін.). "*Служба* обробки заявок" є інформаційною системою, що вже існує в компанії, має базу даних і ряд *сервісів по* обробці заявок. *Ідентифікація* цих акторів і відповідних ним випадків використання важлива з погляду *визначення вимог* до системи. Для представників служби технічної підтримки необхідний спеціальний зручний *інтерфейс* з набором відповідних функцій. А всі заявки, що надійшли*з телефону*, повинні потрапити в єдину базу даних заявок і пройти єдиний цикл обробки. Упущення цих *чинників* може привести до серйозних недоліків і проблем. Крім того, вони ні звідки не слідують безпосередньо і тому потребують особливих

початкових вершин в *дереві* вимог – тобто ми вирішили, що доцільно помістити їх на головну діаграму випадків використання.

Відзначимо ще одну цікаву деталь. Клієнт магазина не є користувачем даного *ПЗ.* Він виявляється бізнес-користувачем всієї системи в цілому (включаючи відповідний *бізнес-процес*і устаткування). На **Рис**. 8.2 представлена бізнес-діаграма випадків використання.

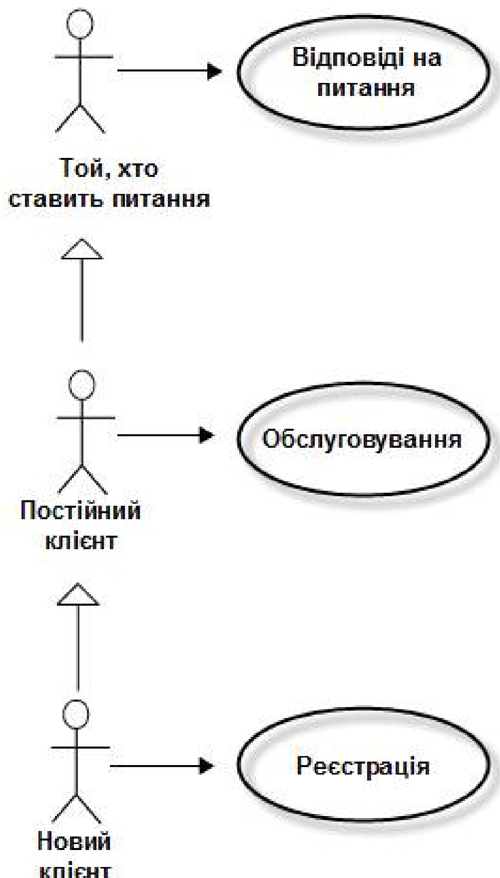

Рис. 8.2. Приклад діаграми бізнес-випадків використання

Її можна малювати окремо (класики на цьому наполягають), але можна примальовувати клієнта і на загальну діаграму, зв'язавши стрілкою з оператором. Часто буває, що затребуваний не дуже концептуальний, але компактний *запис*.

Кожен випадок використання супроводжується невеликим текстовим описом, а надалі може містити цілі розділи втехнічному*завданні. Діаграми випадків* використання можуть служити структурою *технічного завдання* або його окремих частин.

Інші версії. На практиці *діаграми випадків* використання створюються не лише у такий спосіб, як вказано вище. Багато практик вважають за краще будувати дуже детальні моделі, промальовував на них всі невеликі випадки використання, а також численні зв'язки між ними (використання, розширення і так далі). Хтось рішуче протестує проти включення в актори системи, що взаємодіють з даною. Інші вважають неприйнятними суміщати звичайні *діаграми* і бізнес-діаграми*випадків використання*і так далі. Яку саме ви виберете стратегію в конкретному випадку, яку точку зору поставите основною задачею – вам вирішувати самим. Рецепту тут немає.

Важливо лише відзначити, що добре певна точка зору потрібна. Вона дозволяє чітко сфокусуватися, вирішувати певні, добре усвідомлювані завдання. А також таку точку зору можна подекуди усвідомлено, на догоду практичній корисності, порушувати.

Випадки використання в управлінні розробкою. Отже, вище ми показали, як *діаграми випадків* використання можуть бути корисні при виявленні первинної формалізації вимог. Але вони можуть виявитися корисними і після того, як цей *процес завершений*. Результуючі *діаграми випадків* використання можна застосовувати при управлінні розробкою. *Менеджер проекту може* відстежувати прогрес проекту *по* тому, скільки реалізовано функціональності, необхідній користувачеві. Розробники можуть мати *діаграми випадків* використання десь перед очима, щоб не забувати про основну мету розробки. Ці ждіаграми можуть використовуватися в робочих зустрічах за проектом.

Здавалося б, що може бути простіше – реалізувати набір функцій, необхідних користувачеві. Проте на ділі програмний проект може непомітно втратити цю мету. Замість цього можна, наприклад, дуже довго займатися розробкою складної і багатофункціональної архітектури, після реалізації якої розробники обіцяють, що все *призначені для користувача функції будуть отримані* майже відразу ж і дуже легко. Проте, як правило, виявляється, що це "відразу ж" було сильним перебільшенням і проект вельми вибивається з розкладу, а багато замовлених користувачем функцій в цьому оточенні зробити важко або неможливо. Буває, що надмірна орієнтація на "внутрішню досконалість" ПЗ закінчується для проекту або великими проблемами, або повним крахом. Проте бувають і інші випадки, коли лише така орієнтація згодом і рятує проект. Останнє трапляється, коли система довго розвивається і супроводжується, або коли вимоги до неї раптово і сильно міняються, або коли на її основі робляться інші системи. Необхідний баланс між внутрішньою досконалістю *програмного забезпечення* і функціональністю, потрібною для замовника і доставленою йому в строк. Розробка ПЗ в термінах випадків використання – хороший спосіб контролювати, що процес створення системи рухається в потрібному напрямі.

8.2. Ітеративний цикл автор/рецензент

Опишемо одну цікаву і украй корисну техніку використання *візуального моделювання* при виявленні знань про яку-небудь *наочну галузь* через спілкування з експертами (фахівцями в цій *наочній галузі*). Ця техніка називається **цикл автор/рецензент** (*Reader/AuthorCyclereviewprocess*) і може застосовуватися, наприклад, при роботі з діаграмами випадків використання. при роботі як з *UML*, так і з будь-якою іншою мовою *візуального моделювання*. Ця техніка була визначена в рамках методології *SADT*.

Активний співробітник – *автор* візуальних моделей (*author*), – вивчає не цілком знайому йому галузь знань. При цьому авторові постійно

потрібний зворотний *зв'язок* з експертами в цій *наочній галузі*, щоб він усвідомлював, наскільки правильно він зрозумів і адекватно формалізував той або інший аспект знань, що вивчалися.

Як така галузь знань може виступати наочна галузь, для якої створюється *інформаційна система*. При розробці *інформаційної системи* її автори повинні добре розібратися в даній *наочній галузі*. Якщо майбутні користувачі або замовник системи не мали можливості детально ознайомитися з тим, як розробники зрозуміли і інтерпретували їх наочну галузь, то це неодмінно приведе до створення незатребуваної системи: дані будуть невірні або їх не хапатиме, формати *звітів* опиняться незручні і так далі

Отже, для того, щоб створити *адекватний опис системи*, необхідно своєчасно отримувати оцінку створюваних моделей від тих людей, які врешті-решт будуть нею користуватися. Для цього вводяться наступні ролі:

- ·*автор* (*author*) моделі – той, хто її створює;
- ·*експерт* (*commenter*) – це фахівець в тій *наочній галузі*, для якої будується дана модель; автор інтерв'ює експерта, отримуючи необхідну для моделювання інформацію; експерт переглядає і коментує створені авторомдіаграми; важливо, що експерт виражає свої коментарі письмово і розділяє з автором відповідальність за якість створюваних моделей; експерт може бути також архітектором системи, який активно бере участь в процесі розробки *моделі аналізу*, – але не як автор моделей (у нього вистачає інших турбот), а як активний критик (при розробці архітектури системи він активно використовуватиме цю модель);

- ·*читач* (*reader*) – у всьому схожий на експерта, але не зобов'язаний давати письмові коментарі до моделей і не несе відповідальності за якість моделювання.

Отримавши *діаграми автора*, експерт їх ретельно проглядає і пише свої коментарі (прямо на діаграмі, у вигляді приміток, червоною ручкою). *Автор*, отримавши назад свої *діаграми* з коментарями, зобов'язаний відреагувати на кожне зауваження – помітити синій ручкою на тій же копії, чи приймає він зауваження чи ні. Прийняті зауваження він враховує в наступній версії діаграм, неприйнятінадсилає назад експертові з мотивуванням. У разі виникнення нерозуміння організовується зустріч *автора* і експерта, на якій вони залагоджують всі непорозуміння.

Окрім *автора*, експерта і читача в циклі "читач/автор" є також наступні ролі:

- ·*бібліотекар* (librarian) – це головний *координатор* процесу моделювання; він стежить за тим, щоб всі учасники процесу вчасно отримували свіжі копії моделей, щоб ці копії не втрачалися і вчасно потрапляли в архів, а останній був би доступний; у його компетенцію входить також відстежувати, що всі зауваження експертів і читачів оброблені автором, не залишені без уваги; раніше, коли метод *SADT* лише з'явився, роль бібліотекара була велика – моделі будувалися на папері; тепер же для цього використовують різні графічні пакети, а для зберігання різних версій моделі – *програмні засоби* управління версіями;
- ·*комітет технічного контролю* (*technicalreview*committee) – це група людей, яка стежить за тим, наскільки процес моделювання відповідає *цілям проекту*, чи буде можливість використовувати в подальшій роботі створювані *діаграми*; цей

комітет стежить також за тим, коли моделювання потрібно завершити; адже час людей може коштувати істотних грошей, у проекту є терміни, а процес моделювання може продовжуватися дуже довго – наприклад, автор може захопитися, вивчаючи нову наочну галузь.

Слід відмітити, що цикл "читач/автор" може використовуватися в різних ситуаціях, коли необхідно ефективно витягувати інформацію з експертів деякої *наочної галузі*. Наприклад, така ситуація може скластися, коли технічний письменник створює документацію про програмне забезпечення, або *тестувальник* вивчає систему для того, щоб ефективно її тестувати, або новий *менеджер проекту* вивчає систему, яка вже давно розробляється і створенням якої йому потрібно буде керувати, і так далі

Крім того, цикл "*автор/рецензент"* може бути використаний і поза контекстом *витягання знань*, коли ми, під час створення візуальних моделей, хочемо отримувати регулярний і впорядкований *зворотний зв'язок*.

Різноманітність виробничих контекстів, де може застосовуватися дана техніка, а також особливості людських і організаційних відносин, приводять до того, що цикл "*автор/рецензент"* на практиці вимагає *адаптації*. Для його ефективного використання необхідне "тонке підстроювання" під особливості конкретної ситуації.

Зокрема, можуть варіюватися відповідальності різних ролей. Наприклад, експерт може відповідати за процес моделювання або зовсім не відповідати (вся відповідальність лежить на *авторові*). Само спілкування *автора* і експерта також може бути організоване *по-різному*. Наприклад, на відміну від приведених вище рекомендацій, експерт може висловлюватися лише усно, при особистих зустрічах з автором. На одній зустрічі експерт видає інформацію, на іншій перевіряє те, як вийшло у *автора* її формалізувати. Цей варіант представлений в прикладах нижче.

На **Рис. 8.3** показана початкова *діаграма*, яку намалював *автор* для першої зустрічі з експертом. На ній присутні лише інтерфейси з *діаграм компонент UML* і коментарі до них. На **Рис. 8.4** представлена завершальна *діаграма*, що вийшла у результаті численних *ітерацій*. На ній присутніба-гаточисленні компоненти системи, які згруповані *за*рівнями.

Рис. 8.3. Приклад початкової, першої діаграми.

Карти пам'яті

Карти пам'яті (Mind*Maps*) – техніка роботи з різними знаннями, запропонована і розвинена англійським психологом тоні Бьюзеном в кінці 70-х років минулого століття. Вона дуже проста і використовується при

роботі з інформацією будь-якого вигляду, для її структуризації, осмислення, кращого засвоєння і запам'ятовування. На листі паперу, в центрі, малюється *об'єкт*, що позначає ту тему або предмет, який ми розглядаємо. Далі малюються вторинні об'єкти, які пояснюють і уточнюють даний і з'єднуються з ним *дугами*. І так далі. Приклад наведений на **Рис. 8.5.**

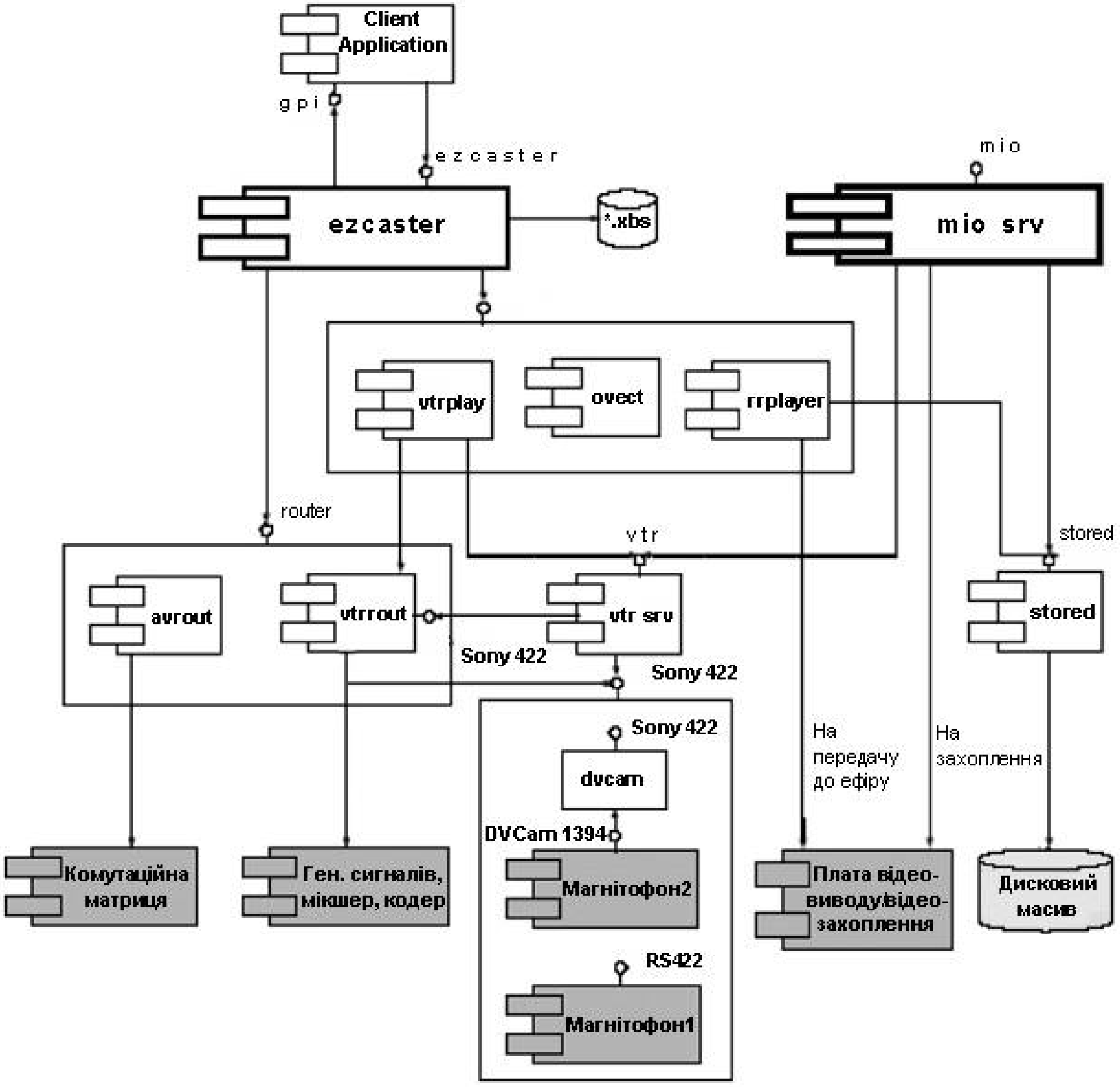

Рис. 8.4. Приклад підсумкової діаграми.

Рис. 8.5.Приклад використання карт пам’яті

Дизайн ідей. Карти пам'яті дозволяють виконати "*дизайн* ідей". Дуже часто ми, як слід не подумавши, починаємо щось робити – писати великий текст, з кимось зустрічатися, кимось керувати і ін. І виявляється, що розуміння *по ходу* справи виникає важко і болісно. Більш того, ми вимушені переробляти те, що вже зробили без цього розуміння. Хороша ілюстрація – робота над текстом (диплома, курсової роботи, статті, книги і ін.). Кардинально переробляти текст дуже важко. А якщо при цьому співавторів декілька? Карти пам'яті тут дуже добре працюють, оскільки дозволяють в компактному вигляді робити проби і помилки, бачивши всю картину перед собою. Її легко також обговорювати у такому вигляді з іншими людьми. Але тут не потрібно фанатизму. Можна і написати текст, якщо він легко "виливається" з вас. І знову повернутися до схеми – багато що на ній може прояснитися.

Планування детальної інформації. Метод дозволяє також виконати детальне планування великого об'єму інформації, що має величезну кількість важливих деталей. Наприклад, ми використовували карти пам'яті при проектуванні анкети – вона містила достатню кількість роз*галужень*, списки питань і ін. Все це в загальному вигляді, скорочено, було не представити, карти пам'яті, що підтримуються програмним інструментом Comapping1, можуть стати у нагоді. Приклад наведений на **Рис. 8.6.**

Реструктуризація. Карти пам'яті корисні при реструктуризації знань. Наприклад, при реструктуризації статті. У нас був випадок, коли результати були отримані, матеріал зібраний і викладений, але достатньо хаотично. Ми виконали реструктуризацію статті за допомогою карт пам'яті (модель представлена на **Рис. 8.7** і *за* цією моделлю швидко переписали текст. Виправлення безпосередньо за текстом затягнули б весь процес. Крім того, карти пам'яті дозволили розділити роботу між співавторами – один створив новий план, а другий його реалізував в новій версії тексту.

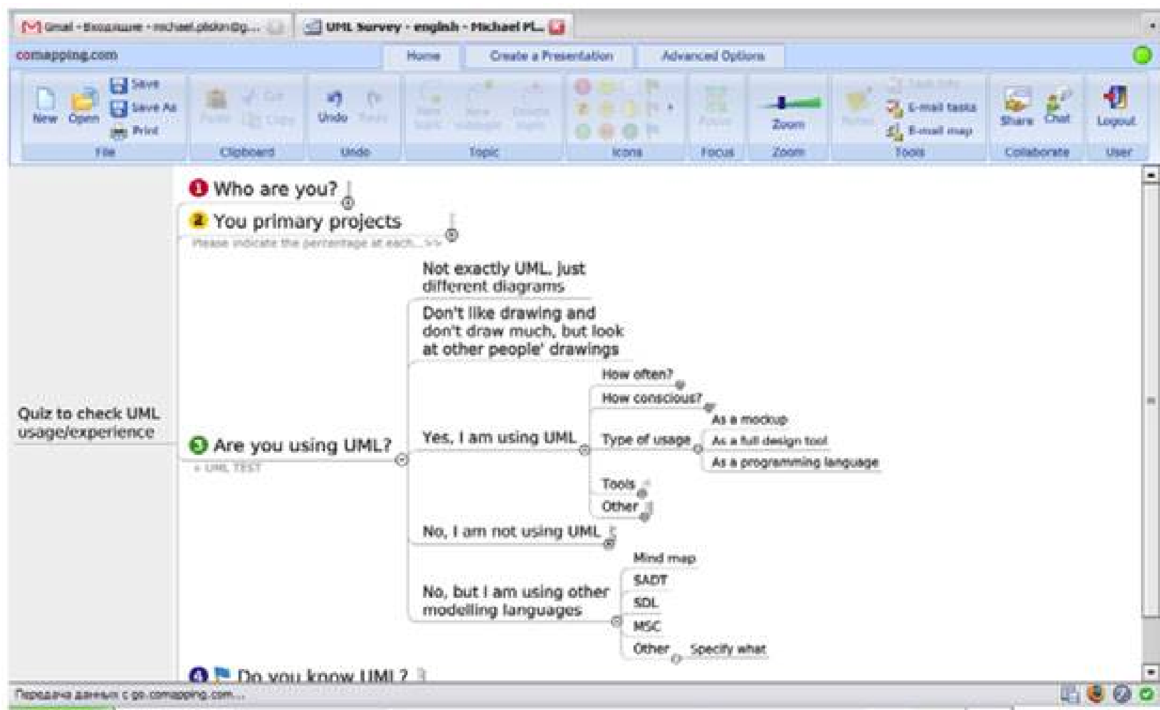

Рис. 8.6. Приклад карти пам'яті, що підтримуються програмним інструментом Comapping

Метод реструктуризації широко використовується при роботі із знаннями, наприклад, при виявленні і аналізі вимог. Побудувати подання тієї ж самої інформації, але з іншої точки зору – вірний спосіб знайти непомічені раніше суперечності, "темні кути", поглибити своє розуміння.

Робота з короткостроковою пам'яттю. Часто буває, що прослуховує лекцію, ми якийсь час пам'ятаємо її зміст (звичайно декілька днів), але *після* декількох місяців її зміст начисто випаровується з мозку. Так от, можна відразу, поки залізо ще гаряче, зробити собі нарис основних аспектів, і використанням карт пам'яті. Студенти говорять, що знайшовши потім такі конспекти-нагадування, вони швидко відновлюють учбовий матеріал в пам'яті. Це доцільно робити після лекції, коли цілісне враження від інформації ще є свіжим.

Колективна робота і продукт Comapping. Одне з головних достоїнств діаграм полягає в тому, що їх можна обговорювати з широким колом людей. Тест, наприклад, обговорювати важче – його потрібно спочатку прорахувати. А діаграму можна тут же дивитися і обговорювати. І виправляти. Більш того, за допомогою діаграм можна організовувати ефективні групові територіально розподілені процеси роботи з інформацією: планування, створення текстів, обмін результатами бесід, спілкування викладача і студента і ін.

Контрольні запитання

1. Яка роль акторів при побудові діаграм випадків використання?
2. Що таке випадок використання?
3. Чим випадок використання відрізняється від довільної функції системи?
4. Які бувають види акторів?

5. Охарактеризувати бізнес-діаграму випадків використання.
6. Яке основне призначення діаграм випадків використання?
7. Які існують варіанти застосування діаграм випадків використання?
8. Які існують варіанти застосування випадків використання в управлінні розробкою?
9. Яка основна ідея циклу автор/рецензент?
10. Як цей цикл автор/рецензент можна використовувати при витяганні знань з експерта?
11. Які додаткові особливості процесу витягання знань з експерта?
12. Що таке карта пам'яті?
13. Хто автор ідеї карт пам'яті?
14. В чому полягає використання карт пам'яті для планування детальної інформації?
15. В чому полягає використання карт пам'яті для реструктуризації?
16. В чому полягає ідея роботи з короткостроковою інформацією?
17. В чому полягає використання карт пам'яті для дизайну ідей?

Розділ 2. Програмні засоби підтримки процесу розробки програмного забезпечення

Глава 9. Microsoft Solutions Framework(MSF)

IT рішення. Основні принципи MSF. Модель команди: основні принципи, ролеві кластери. Масштабування команди MSF. Модель процесу. Управління компромісами.

1.1. Історія і поточний статус

У 90-х роках компанія Microsoft, що мала на меті досягти максимальної віддачі від реалізації замовлених IT-рішень і в цілях *покращення роботи* з субпідрядниками, узагальнила свій *досвід із* розробки, впровадженню, супроводу і консалтингу *ПЗ,* створивши методологію MSF. У 2002 році вийшла версія *MSF* 3.1, що складалася з п'яти документів-керівництв:

- модель процесів (*processmodel*)
- модель команди (*teammodel*)
- модель *управління проектами* (*projectmanagement*)
- дисципліна *управління ризиками* (*riskmanagement*)
- управління підготовкою (*readinessmanagement*).

IT рішення – це скоординоване постачання набору елементів (таких як *програмні засоби*, документація, навчання і супровід), які є необхідними для задоволення бізнес-потреб конкретного замовника. При цьому під його розробкою розумілося створення ПЗ, навчання користувачів і повна передача продукту команді супроводу. Завдання налагоджування повноцінного супроводу IT-рішення – важлива складова успішності проекту.

Основними нововведеннями *MSF* є:

1. Акцент на *впровадженні IT-рішень*.

2. *Модель процесу*, яка об'єднує спіральну і водоспадну моделі.

3. *Особлива організація команди* – не ієрархічна, а як група рівних, але таких, що виконують різні функції (ролі) працівників.

4. Техніка *управління компромісами*.

Нижче ми розглянемо ці положення детальніше.

У 2005 році *MSF* зазнало значних змін. Версія *MSF* 4.0. стала складовою частиною продукту *VisualStudioTeamSystem*(VSTS) і розділилася на дві гілки – *MSF forAgile* і *MSF forCMMI*. При цьому, якщо версії до 3.х були саме методологіями (там були викладені принципи, *MSF* вільно розповсюджувалася у вигляді Word-документів, які були також перекладені іншими мовами), то тепер *MSF* перетворилася на шаблони процесу для VSTS. Ці шаблони мають опис у вигляді *html-документів (*Word-документів вже немає) і визначають типи ролей, їх відповідальності, дії в межах цих відповідальностей, а також всі вхідні і вихідні *артефакти* цієї діяльності і інші формалізовані *атрибути процесу розробки*. Окрім цього "людського" опису *MSF forAgile* і *MSF forCMMI* мають *XML-налагоджування*, які дозволяють точно слідувати запропонованим вище описам з використанням VSTS. При цьому на процес накладаються достатньо жорсткі обмеження, *діяльність* розробників супроводжується набором автоматичних дій – все це задано у шаблонах. Дані шаблони можна частково використовувати (наприклад, без деяких ролей), а також змінювати (VSTS надає широкі засоби налагоджування шаблонів). Версія *MSF* 4.2 продовжила спрямованість версії *MSF* 4.0.

Можна вважати, що фактично, версії *MSF* 4.х є продуктами іншого класу, ніж *MSF* 3.х. *MSF* 3.х були націлені на розробку замовлених IT-рішень, *MSF* 4.0 – на розробку довільного *ПЗ.* Формально, документація цих версій не сильно перетинається і містить для 3.х більшою мірою загальні принципи, а для 4.х – формальні атрибути в термінах VSTS. В деяко-

му розумінні можна сказати, що *MSF* 4.x є реалізацією *MSF* 3.x для продукту VSTS. У цій лекції ми розглянемо основні принципи *MSF*, тобто, фактично, *MSF* 3.1, а в лекціях, присвячених VSTS будуть розглянуті *MSF forAgile* і *MSF forCMMI*.

1.2. Основні принципи MSF

Перерахуємо основні принципи MSF.

1. *Єдине бачення проекту*. Успіх колективної роботи над проектом немислимий без наявності у членів *проектної групи* і замовника єдиного бачення (*sharedvision*), тобто чіткого, і, найголовніше, однакового, розуміння цілей і *завдань проекту*. Як проектна група, так і замовник спочатку мають власні припущення про те, що повинне бути досягнуте в процесі роботи над проектом. Лише наявність єдиного бачення здатна надати ясність і забезпечити рух всіх зацікавлених в проекті сторін до загальної мети. Формування єдиного бачення і подальше проходження проекту у відповідності до цього бачення є такими важливими, що модель процесів *MSF* виділяє для цієї мети спеціальну фазу – "Вироблення концепції", яка закінчується відповідною віхою.
2. *Гнучкість – готовність до змін*. Традиційна дисципліна *управління проектами* і *каскадна модель виходять* з того, що всі вимоги можуть бути чітко сформульовані на початку роботи над проектом, і далі вони істотно не змінюватимуться. В протилежність цьому, *MSF* ґрунтується на принципі безперервної змінності умов проекту на разі незмінної ефективності управлінської діяльності.
3. *Концентрація на бізнес-пріоритетах*. Незалежно від того, чи націлений продукт, що розробляється, на організації або індивідуумів, він повинен задовольнити певні потреби споживачів і принести в деякій формі вигоду або віддачу. Відносно індивідуумів це може означати, наприклад, емоційне задоволення – як у разі комп'ютерних

ігор. Що ж до організацій, то незмінним цільовим *чинником* продукту є бізнес-віддача (*businessvalue).* Зазвичай продукт не може приносити віддачу до того, як він є цілком впроваджений. Тому модель процесів *MSF включає* у свій *життєвий цикл* не лише розробку продукту, але і його впровадження.

4. *Заохочення вільного спілкування*. Історично багато організацій будували свою *діяльність* на основі зведення інформованості співробітників до мінімуму, що є необхідним для виконання роботи (need-to-know). Часто такий підхід призводить до непорозумінь і знижує шанси команди на досягнення успіху. Модель процесів *MSF* припускає відкритий і чесний *обмін інформацією* як усередині команди, так і з ключовими зацікавленими особами. Вільний *обмін інформацією* не лише скорочує ризик виникнення непорозумінь, нерозуміння і невиправданих *витрат*, але і забезпечує максимальний внесок всіх учасників *проектної групи* в зниження невизначеності, що існує в проекті. З цієї причини модель процесів *MSF* пропонує проведення аналізу перебігу роботи над проектом в певних часових межах. *Документування* результатів робить зрозумілим прогрес, який був досягнутий в роботі над проектом, як для проектної команди, так і для замовника і інших зацікавлених в проекті сторін.

1.3. Модель команди

Основні принципи. Головна особливість моделі команди в *MSF* є те, що вона "плоска", тобто не має офіційного лідера. Всі відповідають за проект в рівній мірі, рівень зацікавленості кожного в результаті дуже високий, а комунікації усередині групи чіткі, ясні, дружні і відповідальні. Звичайно, далеко не кожна *команда* здатна так працювати – власне, начальники для того і потрібні, щоб нести основний вантаж відповідальності за проект і, багато в чому, звільнити від нього інших. Демократія в команді можлива у разі високого рівня усвідомленості і зацікавленості кожного, а також в си-

туації рівності в професійному рівні (хай і в різних галузях – див. різні ролеві *кластери* в команді, про які мова піде нижче). З іншого боку, в реальному проекті, в рамках даної моделі команди, можна варіювати ступінь відповідальності, зокрема аж до виділення, у разі необхідності, лідера.

Одній з особливостей відносин усередині команди є висока культура дисципліни зобов'язань:

- готовність працівників приймати на себе зобов'язання перед іншими;
- чітке визначення тих зобов'язань, які вони на себе беруть;
- прагнення докладати належні зусилля до виконання своїх зобов'язань;
- готовність чесно і негайно інформувати про погрози виконанню своїх зобов'язань.

Ролеві кластери. *MSF* заснований на постулаті про сім якісних цілей, досягнення яких визначає успішність проекту. Ці цілі зумовлюють *модель проектної групи* і утворюють **ролеві кластери** (або просто **ролі**) в проекті. У кожному рольовому *кластері* може бути присутнім *один* або декілька фахівців, деякі ролі можна надавати одному учасникові проекту. Кожен *ролевий кластер* визначає унікальну точку зору на проект, і у той самий час ніхто з членів проектної *групи* поодинці не в змозі успішно визначати всі можливі погляди, що відображають якісно різні цілі. Для вирішення цієї дилеми *команда* однодумців (*команда* рівних, *team*of*peers*), що працює над проектом, повинна мати чітку форму звітності перед зацікавленими сторонами (*stakeholders*) за умов розподіленої відповідальності за досягнення загального успіху. У *MSF* існують такі *ролеві кластери* (часто їх називають ролями) – див. Рис. 9.1 .

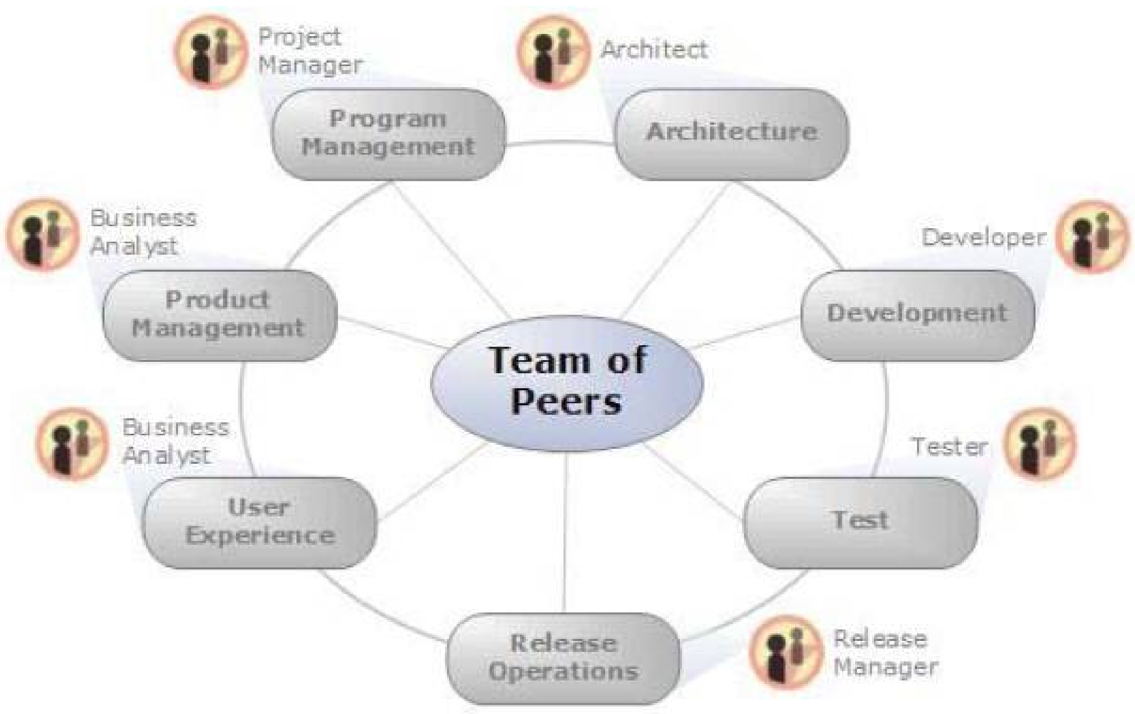

Рис. 9.1. **Ролеві кластери (ролі) у MSF**

- **Управління продуктом** (*product management*). Основне завдання цього ролевого кластера – забезпечити, щоб замовник залишився задоволеним в результаті виконання проекту. Цей ролевий кластер діє по відношенню до проектної групи як представник замовника і часто формується із співробітників організації-замовника. Він представляє бізнес-сторону проекту і забезпечує його узгодженість із стратегічними цілями замовника. О цієї ролі входить і контроль за повним розумінням інтересів бізнесу під час ухвалення ключових проектних рішень.
- **Управління програмою** (*program management*) забезпечує управлінські функції – відслідковування планів і їх виконання, відповідальність за бюджет, ресурси проекту, вирішення проблем і труднощів процесу, створення умов, за яких команда може працювати ефективно, забезпечення мінімуму бюрократичних перешкод.
- **Розробка** (*development*). Цей ролевий кластер займається безпосередньо програмуванням ПЗ.

- **Тестування** (*test*) – відповідає за тестування ПЗ.
- **Задоволення споживача** (*user experience*). Дизайн зручного *для користувача інтерфейсу* і забезпечення *зручності експлуатації ПЗ* (ергономіки), навчання користувачів роботі з ПЗ, створення призначеної для користувача документації.
- **Управління випуском** (*release management*). Безпосередньо відповідальний за безперешкодне впровадження проекту і його функціонування, несе відповідальність за зв'язок між розробкою рішення, його впровадженням і подальшим супроводом, забезпечує інформованість членів *проектної групи* про наслідки їх роботи.
- **Архітектура** (Architecture). Організація і виконання високорівневого проектування рішення, створення *функціональної специфікації ПЗ* і управління цією специфікацією в процесі розробки, визначення меж проекту і ключових компромісних рішень.

Таблиця 9.1

Ролеві кластери у MSF

	Управління продуктом	Управління програмою	Розробка	Тестування	Задоволення споживача	Управління випуском	Архітектура
Управління продуктом		–	–	+	+	+–	–
Управління програмою	–		–	+–	+–	+	+
Розробка	–	–		–	–	–	+
Тестування	+	+–	–		+	+	+–
Задоволення споживача	+	+–	–	+		+–	+–

Управління випуском	+–	+	–	+	+–		+
Архітектура	–	+	+	+–	+–	+	

Масштабування команди MSF. Наявність 7 ролевих кластерів не означає, що *команда* повинна складатися саме з 7 чоловік. Один співробітник може об'єднувати декілька ролей. Але при цьому деякі ролі не можна об'єднувати. У Таблиці 9.1 наведені рекомендації *MSF* щодо можливого *поєднання ролей* одним членом команди:

- "+" означає, що поєднання можливе,
- "+-" – що поєднання можливе, але небажане,
- "-" означає, що поєднання не рекомендується.

Зокрема, не можна суміщати розробку і тестування, оскільки, як обговорювалося вище, необхідно, щоб у тестувальників був сформований свій, незалежний погляд на систему, що базується на вивченні вимог.

Модель проектної групи MSF пропонує *розбиття* великих команд (більше 10 чоловік) на малі багатопрофільні **групи напрямів** (*feature*teams). Ці малі колективи працюють паралельно, регулярно синхронізуючи свої зусилля, кожна з яких побудована на основі моделі кластерів. Це компактні міні-команди, які створюють матричну організаційну структуру. До них входять *поодинці* або декілька членів з різних ролевих кластерів. Такі команди мають чітко сформульоване певне завдання і відповідальні за питання, яке цілком все відноситься до їх компетенції, починаючи від проектування і складання календарного *графіка*. Наприклад, може бути сформована спеціальна *група* проектування і розробки сервісів друку.

Крім того, коли ролевому кластеру потрібно багато ресурсів, формуються так звані **функціональні групи** (*functional*teams), які потім об'єднуються в *ролеві кластери*. Вони створюються у великих проектах, коли необхідно згрупувати працівників усередині ролевих кластерів *за* їх галузями компетенції. Наприклад, в Майкрософті *група* управління продуктом зазвичай включає фахівців *з* планування продукту і фахівців *з* маркетингу. Як перша, так і друга сфери діяльності відносяться до управління продуктом: одна з них спрямована на виявлення якостей продукту, які дійсно цікавлять замовника, а друга – на інформування потенційних споживачів про переваги продукту.

Аналогічно, в команді розробників можливе групування співробітників відповідно до призначення модулів, що розробляються ними: *інтерфейс* користувача, бізнес-логіка або об'єкти даних. Часто програмістів розділяють на розробників бібліотек і розробників рішення. Програмісти бібліотек зазвичай використовують мову низького рівня С і створюють компоненти, які повторно використовуються і можуть стати в нагоді всьому підприємству. Творці ж рішення зазвичай об'єднують ці компоненти і працюють з мовами більш високого рівня, такими як, наприклад Microsoft *VisualBasic*.

Часто функціональні групи мають внутрішню ієрархічну структуру. Наприклад, менеджери програми можуть бути підзвітними провідним менеджерам програми, які у свою чергу звітують перед головним менеджером програми. Подібні структури можуть також з'являтися усередині галузей компетенцій. Але важливо пам'ятати, що ці ієрархії не повинні затінювати модель команди *MSF* на рівні проекту в цілому.

1.4. Інші особливості

Модель процесу: Модель водоспаду – фази *робіт* і *віхи*.

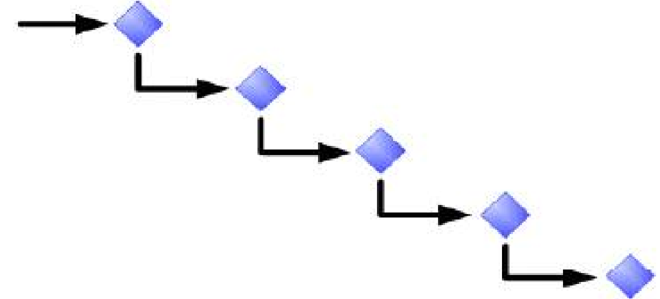

Рис. 9.2. Модель водоспаду – фази *робіт* і *віхи*

Спіральна модель – постійне уточнення вимог, активна взаємодія із замовником.

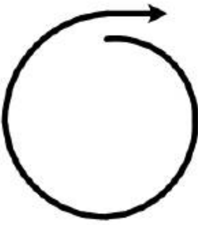

Рис. 9.3. **Спіральна модель**

У *MSF* об'єднуються модель водоспаду зі спіральною моделлю: зберігаються переваги впорядкованості моделі водоспаду, не втрачаючи при цьому гнучкості і творчої орієнтації спіральної моделі.

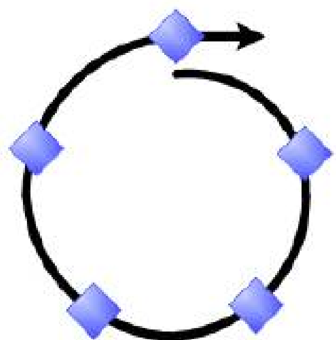

Рис. 9.4. **Об'єднання моделі водоспаду і спіральної моделі у *MSF***

Отже, процес *MSF* орієнтований на "віхи" (milestones) – ключові точки проекту, що характеризують досягнення в його межах якого-небудь істотного (проміжного або кінцевого) результату. Цей результат може бути оцінений і проаналізований. Мається на увазі отримання відповідей на питання: "Чи прийшла проектна *група* до однозначного розуміння цілей і меж проекту?", "Чи в достатньому ступені готовий план дій?", "Чи відповідає продукт затвердженій специфікації?", "Чи задовольняє рішення потреби замовника?" і так далі. А між віхами – ітерації, ітерації, ітерації...

Управління компромісами. Добре відома взаємозалежність між ресурсами проекту (людськими і фінансовими), його календарним графіком (часом) і можливостями, що реалізовуються (межами). Ці три змінні утворюють трикутник, який показаний на Рис. 9.5 .

Рис. 9.5. Трикутник взаємозалежності між ресурсами проекту, календарним графіком (часом) і можливостями

Після досягнення рівноваги в цьому трикутнику зміна на будь-якій з його сторін для підтримки балансу вимагає модифікацій на іншій (двох інших) сторонах і/або на початково зміненій стороні.

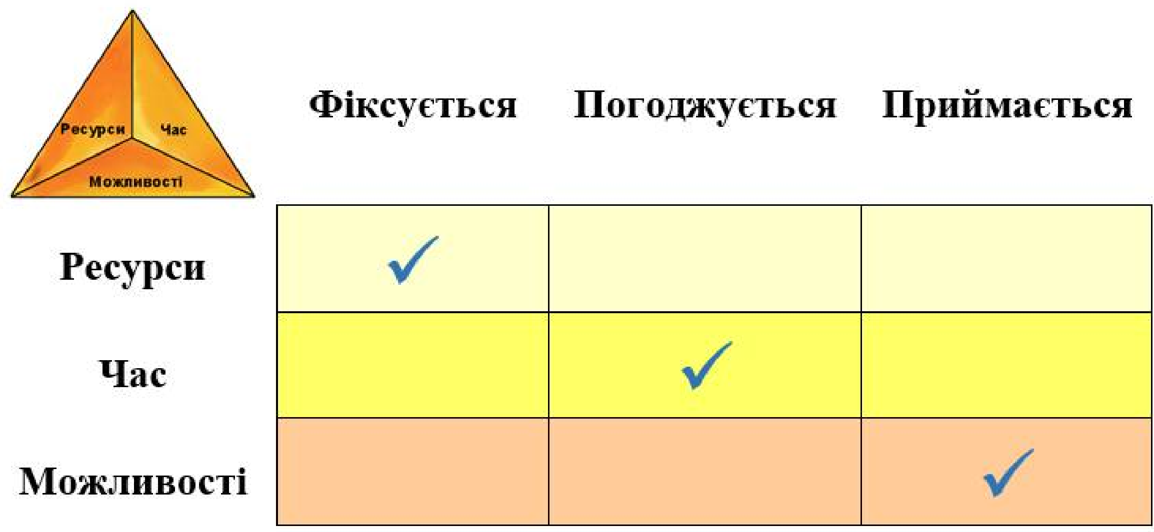

Рис. 9.6. **Зафіксувавши, ми погоджуємо і приймаємо рішення.**

Контрольні запитання

1. Які основні етапи створення MSF?
2. Які основні принципи MSF?
3. Охарактеризувати головні нововведення MSF?
4. Чим принципово відрізняються версії MSF 3.x та 4.x?
5. Що таке IT-рішення в концепції MSF?
6. Охарактеризувати принцип «Єдине бачення проекту».
7. Охарактеризувати принцип гнучкості.
8. Охарактеризувати принцип «Концентрація на бізнес-пріоритетах».
9. Охарактеризувати принцип «Заохочення вільного спілкування».
10. Які основні принципи побудови моделі команди в MSF?
11. Що являє собою модель команди MSF?
12. Охарактеризувати полягає свобода моделі команди MSF?
13. Де закінчується свобода моделі команди MSF?
14. В чому полягає культура дисципліни зобов'язань?
15. Що таке ролеві кластери MSF?
16. В чому суть ролевого кластеру «Управління продуктом»?

17. В чому суть ролевого кластеру «Управління програмою»?
18. В чому суть ролевого кластеру «Розробка»?
19. В чому суть ролевого кластеру «Тестування»?
20. В чому суть ролевого кластеру «Задоволення споживача»?
21. В чому суть ролевого кластеру «Управління випуском»?
22. В чому суть ролевого кластеру «Архітектура»?
23. Що таке групи напрямів в моделі проектної групи MSF?
24. Що таке функціональні групи в моделі проектної групи MSF?
25. Яка модель процесу розробки використовується в MSF?
26. Що таке управління компромісами?

Глава 10. Capability Maturity Model Integration (CMMI) (Інтеграція завершених працездатних моделей)

Поняття CMMI. Рівні зрілості процесів за CMMI. Галузі вдосконалення.

10.1. Що таке CMMI?

CMMI є деяким описом ідеального процесу розробки *ПЗ,* та пропонує певну *модель процесу*. **Capability Maturity Model Integration (Інтеграція завершених працездатних моделей) (CMMI)** — це підхід для вдосконалення процесів, який забезпечує розробників важливими елементами ефективних процесів. Він може використовуватись для покращення процесу як на рівні проекту чи відділу, так і на рівні цілої організації. CMMI дозволяє інтегрувати традиційно відокремлені організаційні функції, ставити цілі та пріоритети покращення процесів, забезпечує інструкцією зі створення якісних процесів, і дає контрольну точку для оцінки поточних процесів. Тобто, в процесі виділяються і ретельно описуються деякі складові частини, які є ключовими з погляду *CMMI*. Ця точка зору *CMMI* – вдосконалення процесів розробки. Тобто ці значущі частини процесу – **галузі вдосконалення**. У *CMMI* розрізняють такі групи галузей вдосконалення: управління *процесами, управління* проектами, інженерні галузі, службові галузі. При цьому всі галузі задаються у вигляді вимог, які визначають не те, як вони реалізовані, а у вигляді інтерфейсних вимог. У зв'язку з цим існують два наслідки.

Наслідок 1. *CMMI* допускає різні реалізації і не є *методологією розробки ПЗ,* подібно *MSF*, Scrum, *RUP* і ін. Останні можуть використовуватися в його реалізації. Існує, наприклад, спеціальний *шаблон* процесу в VSTS для *CMMI* під назвою *MSF* for *CMMI*.

Наслідок 2. *CMMI* використовується для *сертифікації* компаній на зрілість їх процесів. Спочатку, в кінці 80-х на початку 90-х років, *CMM* (тоді ще не *CMMI*) створювався саме як засіб *сертифікації* федеральних субпідрядників. І лише пізніше, набувши широкого поширення в світі, він почав використовуватися, а після і орієнтуватися на вдосконалення процесів.

Відзначимо ще одну важливу характеристику *CMMI*. Він призначений не лише для розробки програмних систем. Багато крупних компаній випускають не *ПЗ,* а цільові вироби, куди ПЗ входить як складова частина. Наприклад, авіаційна, аерокосмічна індустрії. Тобто розробка ПЗ відбувається разом з інженерними роботами інших видів. І часто буває, що в одному проекті бере участь більше двох різних видів інженерії. Завдання *CMMI* – надати таким проектам і компаніям єдину платформу організації процесу розробки.

10.2. Рівні зрілості процесів за CMMI

У відмінності від класичної моделі *CMM*, яка була жорстко ієрархічною і допускала лише послідовне *покращення процесів* у рівнях, модель *CMMI* має два виміри – послідовний, таке ж як і в *CMM*, і безперервний, такий, що допускає вдосконалення процесів в організації-розробника до деякої міри в довільному порядку. Зупинимося на послідовній моделі. Вона має 5 рівнів зрілості процесів, як показано на Рис. 10.1.

Рис. 10.1. **5 рівнів зрілості процесів послідовної моделі**

Початковий рівень (рівень зрілості 1) – це рівень, на якому, *за* визначенням, знаходиться будь-яка компанія. На цьому рівні розробка ПЗ ведеться більш-менш хаотично.

Керований рівень (рівень зрілості 2) – тут вже з'являються політика і процедури організації процесів, що затверджені на рівні компанії. Але повною мірою процеси існують лише в межах окремих проектів.

Певний рівень (рівень зрілості 3) – тут з'являється стандартний процес на рівні всієї компанії в цілому. Це великий набір *активів* процесу, що постійно поповнюється, – *шаблонів документів*, *моделей життєвого циклу*, *програмних засобів*, практик і ін. Будь-який конкретний процес є вирізкою, з цього стандартного.

Керований кількісно рівень (рівень зрілості 4) має на увазі появу системи вимірювань в компанії, які мають місце на основі стандартного процесу і дозволяють кількісно управляти розробкою.

Рівень, що оптимізується, (рівень зрілості 5) має на увазі постійне *поліпшення* процесів розробки, як поступових, покрокових, так і революційних. При цьому дані зміни виявляються не вимушеними, а такими, що

попереджують проблеми і труднощі. Процес сам і постійно удосконалюється, існують і реалізовані відповідні *механізми*.

10.3. Галузі вдосконалення

Рівень зрілості	Галузі удосконалення
Рівень зрілості 2	Управління вимогами Планування проекту Спостереження за проектом і контроль Управління домовленостями з постачальником Вимірювання та аналіз Перевірка процесів і продуктів на відповідність стандартам Конфігураційне управління
Рівень зрілості 3	Розробка вимог Технічне рішення Збірка і постачання продукту Перевірка продукту на відповідність вимогам (верифікація) Перевірка продукту на відповідність призначенню (валідація) Фокусування на процесах організації Визначення процесів організації організація навчання Комплексне управління проектом Управління ризиками Управління об'єднаною командою Комплексне управління роботою з постачальником

	Прийняття рішень: оцінка альтернатив Створення в організації умов для спільної роботи
Рівень зрілості 4	Встановлення показників виконання процесів в організації Управління проектами на основі кількісних показників
Рівень зрілості 5	Відбір і впровадження поліпшень в організацію Аналіз причин виникнення проблем та запобігання їх появи в майбутньому

Контрольні запитання

1. Що таке CMMI?
2. Що таке галузі вдосконалення CMMI?
3. Які групи галузей вдосконалення розрізняють в CMMI?
4. В якому вигляді задаються галузі вдосконалення?
5. Які наслідки зумовлює форма подання галузей вдосконалення в CMMI?
6. Чи є CMMI специфічною лише для розробки ПЗ платформою організації процесу розробки?
7. Скільки існує вимірів покращення процесів у моделі CMMI? Які?
8. Скільки існує рівнів зрілості для послідовної моделі покращення процесів за CMMI?
9. Чим характеризується рівень зрілості компанії 1?
10. Чим характеризується рівень зрілості компанії 2?
11. Чим характеризується рівень зрілості компанії 3?
12. Чим характеризується рівень зрілості компанії 4?
13. Чим характеризується рівень зрілості компанії 5?

Глава 11. "Гнучкі" (agile) методи розробки

Загальний опис "гнучких" методів розробки ПЗ. Extreme Programming: загальний опис, основні принципи організації процесу. Scrum: загальний опис, ролі, практики.

11.1. Загальні положення

"Гнучкі" (agile) методи розробки ПЗ з'явилися як *альтернатива* формальним і "ваговитим" методологіям, на зразок *CMM* і *RUP*. Талановиті програмісти не бажають перетворення розробки ПЗ в рутину, хочуть мати *максимум* свобод і обіцяють натомість високу ефективність. З іншого боку, практика показує, що "ваговиті" методології в значній кількості випадків є неефективними. Основними положеннями гнучких методів, що були закріплені в *Agile Manifesto* в 2007 році, є такі:

- замість процесів і програмних засобів – індивідуали і взаємодія;
- замість складної документації – ПЗ, що працює;
- замість жорстких контрактів – взаємодія із замовником;
- замість проходження плану – реакція на зміни.

Фактично, гнучкі методології спираються на невеликі самоорганізовані команди, що складаються з висококваліфікованих і енергійних людей, які орієнтовані на бізнес, тобто, наприклад, розробляють свій власний продукт для випуску його на ринок. У цього підходу є, очевидно, свої плюси і свої мінуси.

11.2. Extreme Programming

Найвідомішим гнучким методом є *Extreme Programming* (відома скорочена назва – XP). Він був створений талановитим фахівцем в галузі програмної інженерії Кентом Беком в результаті його роботи в 1996-1999 роках над системою контролю платежів компанії "Крайслер".

Модель процесу за XP виглядає як послідовність випусків (*releases*) продукту, настільки частих, наскільки це можливо. Але при цьому обов'язковою умовою є включення у випуск нової цільової функціональності. Нижче перераховані основні принципи організації процесу за XP.

1. Планування (*Planning Game*), засноване на принципі, що розробка ПЗ є діалогом між можливостями і бажаннями, при цьому зміняться і те і інше.
2. Простий дизайн (*Simple Design*) – проти надмірного проектування.
3. Метафора (*Metaphor*) – суть проекту повинна уміщатися в 1-2 ємких фразах або в деякому образі.
4. Рефакторинг (*Refactoring*) – процес постійного поліпшення (спрощення) структури ПЗ, який є необхідним у зв'язку з додаванням нової функціональності.
5. Парне програмування (*Pair Programming*) – один програмує, інший думає над підходом в цілому, про нові тести, про спрощення структури програми і так далі.
6. Колективне володіння кодом (*Collective Ownership*).
7. Участь замовника в розробці (*On-site Customer*) – представник замовника включений до команди розробника.
8. Створення і використання стандартів кодування (*Coding Standards*) в проекті – при написанні коду (створюються і) використовуються стандарти на імена ідентифікаторів, складання коментарів і так далі.
9. Тестування – розробники самі тестують своє ПЗ, перемежаючи цей процес з розробкою. При цьому рекомендується створювати тести до того, як буде реалізована відповідна функціональність. Замовник створює функціональні тести.

10. Безперервна інтеграція. Сама розробка уявляється як послідовність випусків.
11. 40-годинний робочий тиждень.

Проте в повному об'ємі XP не була використана навіть її авторами і тому є, швидше, філософією. Крім того, відомі і впроваджуються окремі практики XP, як, наприклад, *парне програмування*, *колективне володіння кодом*, і, звичайно ж, *рефакторинг коду*. Ідея простого, ненадмірного дизайну проекту також зробила значний вплив на світ розробників ПЗ.

Більш практичним "гнучким" методом розробки є *Scrum*.

11.3. Scrum

Історія. У 1986 японські фахівці Hirotaka Takeuchi і Ikujiro Nonaka опублікували повідомлення про новий підхід до розробки нових сервісів і продуктів (не обов'язково програмних). Основу підходу складала згуртована робота невеликої універсальної команди, яка розробляє проект на всіх фазах. Приводилася аналогія з регбі, де вся команда рухається до воріт супротивника як єдине ціле, передаючи (пасуючи) м'яч своїм гравцям як вперед, так і назад. На початку 90-х років даний підхід почав застосовуватися в програмній індустрії і знайшов назву Scrum (термін з регбі, що означає, – сутичка), в 1995 році Jeff Sutherland і Ken Schwaber представили опис цього підходу на OOPSLA '95 – одній з найавторитетніших конференцій в галузі програмування. З тих пір метод активно використовується в індустрії і багато разів описаний в літературі. Scrum також активно використовується і в Україні.

Загальний опис. Метод Scrum дозволяє ефективно розробляти проекти невеликими командами (7 чоловік плюс/мінус 2) в ситуації вимог, що змінюються. При цьому процес розробки є ітеративним і надає велику сво-

боду команді. Крім того, метод дуже простий – легко вивчається і застосовується на практиці. Його схема зображена на Рис. 11.1.

Спочатку створюються вимоги до всього продукту. Потім з них вибираються найактуальніші і створюється план на таку ітерацію. Протягом ітерації плани не міняються (цим досягається відносна стабільність розробки), а сама ітерація займає 2-4 тижні. Вона закінчується створенням працездатної версії продукту (робочий *продукт)*, яку можна пред'явити замовникові, запустити і продемонструвати, хай і з мінімальними функціональними можливостями. Після цього результати обговорюються і вимоги до продукту коректуються. Це зручно робити, маючи після кожної ітерації продукт, який вже можна якось використовувати, показувати і обговорювати. Далі відбувається планування нової ітерації і все повторюється.

Усередині ітерації проектом повністю займається *команда*. Вона є плоскою, ніяких ролей Scrum не визначає. Синхронізація з менеджментом і замовником відбувається після закінчення ітерації. *Ітерація* може бути перервана лише в особливих випадках.

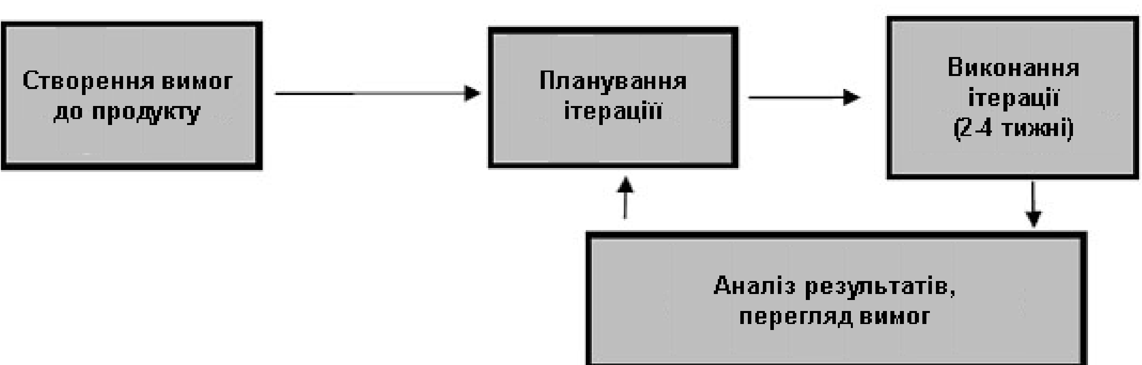

Рис. 11.1. Схема ітеративного процесу розробки

Ролі. У Scrum є всього три види ролей.

Власник продукту (***Product Owner***) – це менеджер проекту, який представляє в проекті інтереси замовника. У його обов'язки входить розробка початкових вимог до продукту (Product Backlog), своєчасна їх зміна, призначення пріоритетів, дат постачання і ін. Важливо, що він абсолютно не бере участь у виконанні самої ітерації.

Scrum-майстер (Scrum Master) забезпечує максимальну працездатність і продуктивну роботу команди – як виконання Scrum-процесу, так і вирішення господарських і адміністративних завдань. Зокрема, його завданням є огородження команди від всіх дій ззовні під час ітерації.

Scrum-команда (Scrum Team) – група, що складається з п'ятидев'яти самостійних, ініціативних програмістів. Першим завданням команди є постановка для ітерації реально досяжних і пріоритетних для проекту в цілому завдань (на основі Projectbacklog і при активній участі власника продукту і Scrum-майстра). Другим завданням є обов'язкове виконання цього завдання у відведені терміни і із заявленою якістю. Важливо, що команда сама бере участь в постановці завдання і сама ж його виконує. Тут поєднується свобода і відповідальність, подібно до того, як це організовано в MSF. Тут також "відчувається" дисципліна зобов'язань.

Практики. У Scrum визначені такі практики.

***Sprint Planning Meeting*.** Проводиться на початку кожного Sprint. Спочатку Product Owner, Scrum-мастер, команда, а також представники замовника і ін. зацікавлені особи визначають, які вимоги з Project Backlog найбільш пріоритетні і їх слід реалізовувати в межах даного Sprint. Формується Sprint Backlog. Далі Scrum-мастер і Scrum-команда визначають те, як саме буде досягнута визначена вище мета з Sprint Backlog. Для кожного елементу Sprint Backlog визначається список задач і оцінюється їх трудомісткість.

Daily Scrum Meeting – п'ятнадцятихвилинна щоденна нарада, метою якої є досягти розуміння того, що відбулося з часу попередньої наради, скоректувати робочий план згідно реаліям сьогоднішнього дня і позначити шляхи вирішення існуючих проблем. Кожен учасник Scrum-команди відповідає на три питання: що я зробив з часу попередньої зустрічі, мої проблеми, що я робитиму до наступної зустрічі? У цій нараді може брати участь будь-яка зацікавлена особа, але лише учасники Scrum-команди мають право ухвалювати рішення. Правило обґрунтоване тим, що вони давали зобов'язання реалізувати мету ітерації, і лише це дає упевненість в тому, що вона буде досягнута. На них лежить відповідальність за їх власні слова, і, якщо хтось з боку втручається і ухвалює рішення за них, тим самим він знімає відповідальність за результат з учасників команди. Такі зустрічі підтримують дисципліну зобов'язань в Scrum-команді, сприяють утриманню фокусу на цілях ітерації, допомагають вирішувати проблеми "у зародку". Зазвичай такі наради проводяться стоячи, протягом 15-20 хвилин.

Sprint Review Meeting. Проводиться в кінці кожного Sprint. Спочатку Scrum-команда демонструє Product Owner зроблену протягом Sprint роботу, а той у свою чергу веде цю частину мітингу і може запросити до участі всіх зацікавлених представників замовника. Product Owner визначає, які вимоги з Sprint Backlog були виконані, і обговорює з командою і замовниками, як краще розставити пріоритети в Sprint Backlog для поточної ітерації. У другій частині мітингу виконується аналіз минулого спринту, який веде Scrum-майстер. Scrum-команда аналізує в останньому Sprint позитивні і негативні моменти спільної роботи, робить виводи і ухвалює важливі для подальшої роботи рішення. Scrum-команда також шукає шляхи для збільшення ефективності подальшої роботи. Потім цикл повторюється.

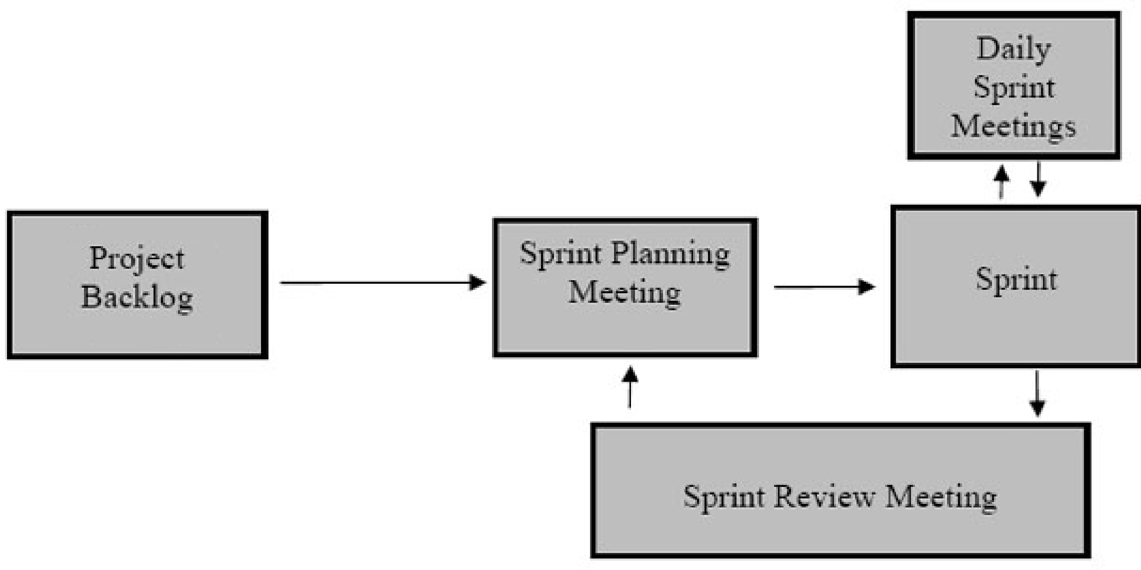

Рис. 11.2. **Визначені у Scrum практики**

Контрольні запитання

1. Що лежить в основі "гнучких" (agile) методів розробки ПЗ?
2. Як виглядає модель процесу розробки ПЗ за ХР?
3. Які основні принципи організації процесу за ХР?
4. Назвати основні принципи організації процесу за ХР.
5. Яка історія створення підходу до розробки методом Scrum?
6. Описати процес розробки продукту методом Scrum.
7. Які види ролей існують у методі Scrum?
8. Які види практики існують у методі Scrum?

Глава 12. Огляд технології Microsoft Visual Studio Team System (VSTS)

Склад продукту: огляд, клієнтська частина VSTS, серверна частина VSTS. Правила інсталяції. Пакет Team Explorer. Огляд

12.1. Інженерія програмного забезпечення (програмна інженерія)

Аналізуючи власний досвід розробки програмного забезпечення, а також досвід інших компаній, фахівці Microsoft прийшли до висновку, що істотна частина проблем, що виникають при розробці програмного забезпечення, викликана "людським чинником" – взаємодією різних фахівців в рамках однієї команди. Це люди різного віку, різної освіти, різних життєвих принципів і інтересів, вирішують різні завдання і переслідують різні цілі (хоча одна загальна мета у них все ж таки є – зробити зрештою якісне ПЗ), вимушені працювати разом волею долі або керівництва. Не дивно, що під час їх взаємодії часто виникають накладки і нерозуміння, а істинно злагоджені і ефективні команди зустрічаються не так часто, як хотілося б. Для вирішення цього завдання корпорацією Microsoft пропонується комплекс *VisualStudioTeamSystem* (VSTS), який забезпечує:

- "Нав'язування" процесу розробки. Інструменти VSTS дозволяють задати процес, який використовується в проекті (тобто створити конкретний процес, користуючись нашою термінологією), і тим самим обмежити дії учасників команди.
- Доступний опис процесу. VSTS припускає доступний опис процесу розробки.
- Єдине середовище розробки – комплекс інструментів, що підтримують всі етапи процесу розробки ПЗ і які використовуються всіма учасниками команди, і створюють не лише єдине інтегроване середовище розро-

бки, але і єдине культурне середовище, загальний базис для всіх учасників команди.

Ядром VSTS є засоби забезпечення *життєвого циклу елементів роботи* (*workitems*) – деяких дискретних характеристик проекту, навколо яких організовується вся робота команди (див. Рис. 12.1). Ось приклади елементів робіт:

- task – конкретне завдання, яке необхідно виконати в проекті;
- *bug* – помилка, яка знайдена, чекає свого виправлення, виправляється, наново перевіряється;
- *risk* – ризик проекту, у якого теж може бути різний стан; як правило, за ризиками їх станами стежать менеджери проектів.

Кожен елемент роботи має набір різних станів, перелік подій, які можуть змінювати ці стани, а також відповідальну особу. Таким чином, елемент роботи використовується для оперативного управління проектом. Кожен з учасників команди бачить пов'язані з ним елементи робіт і після виконання відповідної роботи міняє їх стани, а, можливо і відповідальну особу. Наприклад, програміст виправив помилку і після цього для елементу робіт, що позначає цю помилку, він міняє стан (наприклад, Fixed) і відповідального – відповідного тестувальника, щоб останній протестував зміни коду .

Окрім підтримки життєвого циклу елементів роботи в VSTS входять додаткові засоби – контролю версій, підтримки збірки, засоби інтеграції з офісними додатками (Project, Excel, Word), генератори різних звітів, засоби тестування і деякі ін. Крім того, через відкритий програмний інтерфейс VSTS можна надбудовувати і іншими сервісами, які є необхідними в процесі розробки. На Рис. 12.1 ці можливі сервіси представлені порожніми кубиками.

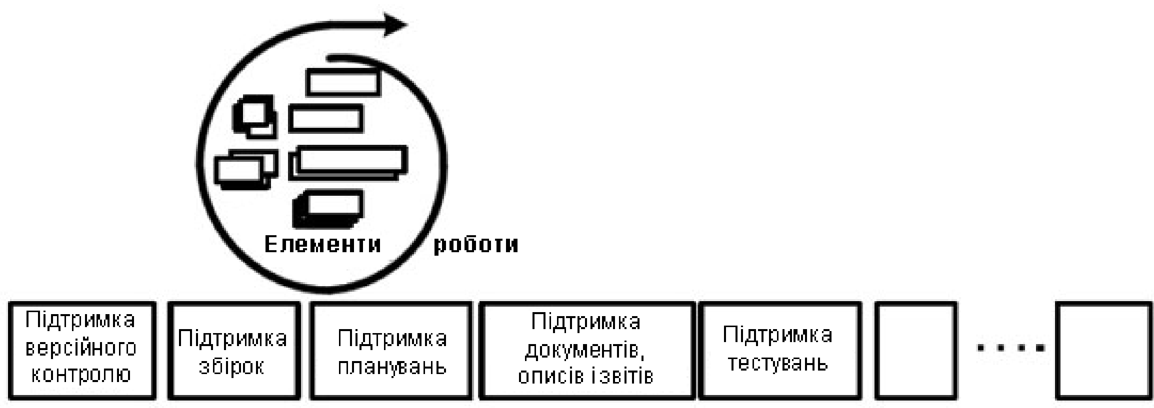

Рис. 12.1. Ядро VSTS

Перелік доступних типів елементів робіт, ролі в проекті, правила переходу елементів робіт з одного стану в інше, всілякі додаткові автоматичні дії, сервіси, а також обмеження, різні права ролей і членів команди на зміну елементів роботи і переклад їх в різні стани і так далі – все це є попереднім описом і налагоджуванням процесу розробки. Це налагоджування виконується перед початком проекту через механізм налагоджування шаблону процесу.

12.2. Склад продукту

Огляд. Тепер подивимося на VSTS як на *програмний продукт*. Він є складним, складеним продуктом і розділяється на клієнтське ПЗ і серверне ПЗ – див. Рис. 12.2.

Розглянемо докладніше клієнтську частину. Стандартним *клієнтом* від компанії Microsoft є продукт *VisualStudioTeamSuiteEdition*. Цей продукт є однією з редакцій середовища розробки *VisualStudio* з додатковим продуктом – *Team Explorer*. Останній служить для доступу до сервісів серверної частини VSTS і вбудовується в *VisualStudio*. Крім того, завдяки відкритому програмному інтерфейсу до серверної частини VSTS – бібліотеки TFS *Client API* – вона інтегрується з різними *середовищами розробки*, напри-

клад, з Eclipse. Також існує значна кількість різних клієнтських продуктів від сторонніх виробників (найбільш успішні з яких Microsoft намагається асимілювати).

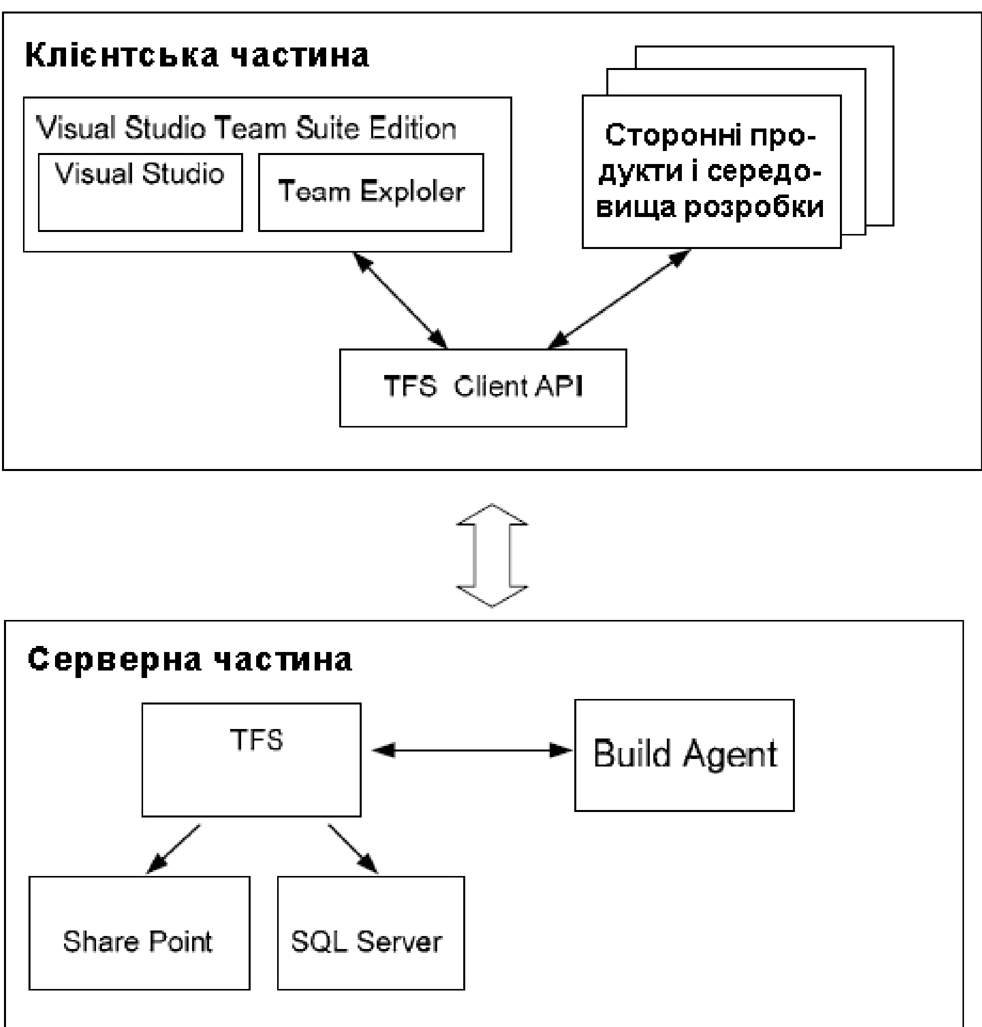

Рис. 12.2. Архітектура VSTS

Серверна частина VSTS складається з TFS (*Team Foundation Server*) – головної серверної компоненти, – а також компоненти *BuildAgent*. TFS реалізує головну функціональність серверної частини і використає два інших серверних продукти Microsoft – *SharePoint* (для організації Web-порталу з описом використовуваного шаблону процесу розробки, інших документів щодо процесу) і *SQL Server* (для *зберігання даних* TFS).

BuildAgent – це серверна компонента, яка відповідає за виконання збірок проектів. Винесення сервера *збірки* в окремий *серверний додаток* дозволяє прибрати процес *збірки* з основної, серверної машини, де розміщений TFS, на додаткову машину, що відповідає саме за проведення збірок. Подібне розділення дозволяє значно понизити навантаження на основний *сервер*, особливо у разі використання підходу безперервної інтеграції.

Зупинимося на клієнтській і серверній частинах VSTS детальніше.

Клієнтська частина VSTS. Зупинимося на стандартному клієнтському *ПЗ,* яке засноване на середовищі розробки *VisualStudio*. Остання випускається в декількох комплектаціях (*editions*), що орієнтовані на різних користувачів. При цьому видання, що включають інструменти комплексу VSTS мають в своїй назві *слово "Team"*. Ось перелік цих видань.

- *Microsoft VisualStudioTeamSystem 2008 ArchitectureEdition* розширений засобами управління повторним використанням, *засобами візуального моделювання* з *генераторами* кінцевого коду і нек. ін. можливостями.

- *Microsoft VisualStudioTeamSystem 2008 DevelopmentEdition* надання засобів аналізу коду з метою підвищення його якості, зокрема, виявлення складного, важкого в обслуговуванні шляхом оцінки відносин між класами, глубінинаследованія, цикломатіческой складності, рядків коду і індексу зручності обслуговування. Сюди ж входять різні засоби профіляциї застосувань.

- *Microsoft VisualStudioTeamSystem 2008 DatabaseEdition* включає засоби управління версіями всіх основних *об'єктів баз даних*, *модульного* тестірованієя баз даних, засоби підтримки *еволюції* схем, підтримка *синтаксису* SQL і багато що інше.

- *Microsoft VisualStudioTeamSystem 2008 TestEdition*надає повний набір засобів тестування Web-приложений і Web-сервисов, *інтегрований* в середу *VisualStudio*. За допомогою даних засобів тестеровщики можуть створювати, виконувати і управляти тестами і пов'язаними з ними елементами робіт VSTS безпосередньо з середовища *VisualStudio*. У це ж видання входять засоби *тестування навантаження*, управління тестовими пакетами і інші можливості.

Крім чотирьох "ролевих" видань, випускається і видання, що об'єднує функції всіх чотирьох блоків, – *Microsoft VisualStudioTeamSystem*2008 *TeamSuite*. Умовно взаємозв'язок різних видань відбитий на Рис. 12.3.

Кожне їх чотирьох "ролевих" видань серії VSTS розширює *ядро* (*TeamEditionCore*) додатковими інструментами, які призначені для певної *ролі (розробник*, тестер, *архітектор* або фахівець *з* баз даних), а видання *TeamSuite* є об'єднанням всіх чотирьох "ролевих" видань.

Ядро складається з *базової конфігурації VisualStudio – VisualStudio Professional*, – яка є найбільш поширеним виданням середовища *VisualStudio* і повсюдно використовується для розробки *програмного забезпечення*. Вона доповнюється *Team Explorer*, що призначений для *інтеграції* з TFS.

Серверна частина VSTS. Отже, ядром комплексу інструментів VSTS є TFS, який не є *цілісною системою*, а вдає із себе набір стандартних продуктів (зокрема, *SQL Server* і *SharePoint*), відповідним чином налагоджуваних і *об'єднаних* в єдине ціле за допомогою прошарку Web-сервисов. *Архітектура* серверної частини VSTS представлена на Рис. 12.4, де сірими *прямокутниками* показані компоненти VSTS, а білими – компоненти інших продуктів Microsoft. На цьому ж малюнку схемний позначена і клієнтська частина VSTS.

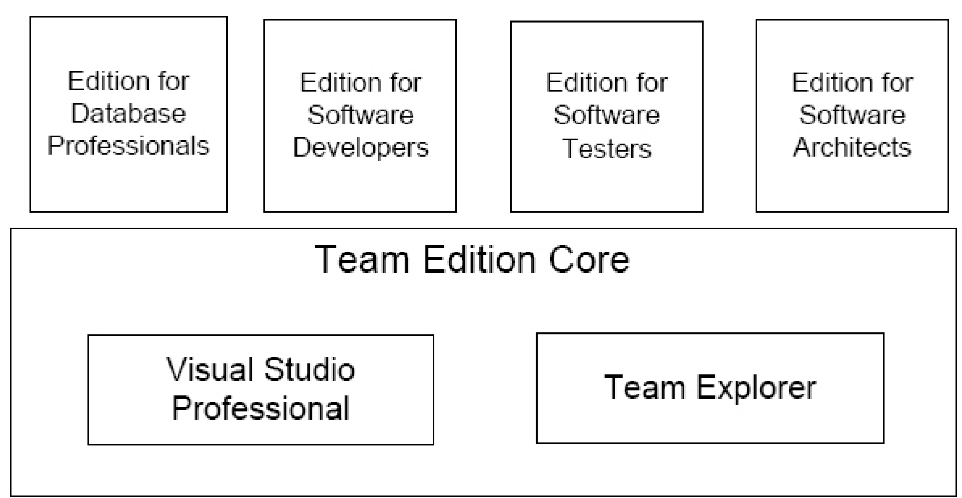

Рис. 12.3. Схема Microsoft VisualStudioTeamSystem 2008 TeamSuite

Рис. 12.4. Архітектура серверної частини VSTS

TFS, основна серверна підсистема VSTS, складається з двох основних рівнів: рівня додатків і рівня даних. Рівень додатків TFS включає такі компоненти.

- *SQL Server Analysis&Reporting* – компонента пакету SQL Server, що використовується TFS для побудови *звітів* аналізу статусу проектів. Доступ до цієї компоненті з клієнтського боку здійснюється не через компоненту TFS *Client API*, а безпосередньо, засобами web-браузера.
- *SharepointServices* – компонента з пакету *SharePoint*, що використовується для зберігання загальнодоступної інформації і опису використо-

вуваного процесу розробки. Доступ до цієї компоненті з клієнтського боку здійснюється не через TFS *Client API*, а безпосередньо, засобами web-браузера.

- *SharePointExtensionsfor TFS* – розширення *SharePoint* для TFS, яке забезпечує доступ до звітів і деяких функцій TFS безпосередньо з Web-порталу.
- *TeamFoundation Server* – головна компоненту TFS, яка складається з набору Web-сервісів, які є доступними через TFS *ClientApi клієнтському ПЗ* і що реалізовують основні сервісні функції TFS, зокрема:
  - версійний контроль;
  - управління елементами роботи;
  - роботам з шаблонами процесу;
  - *адміністрування* і так далі.
- *TeamFoundationBuildService у складі TFS* – призначена для *ініціації* процесу *збірки* і передачі відповідного завдання компоненті *BuildAgent*. Інший екземпляр цього додатку знаходиться в *BuildAgent* і виконує там системні функції.

Рівень додатків, що реалізовані на технології *ASP.NET* і працюють під управлінням *IIS* (*Internet*Infromation*Service*). *IIS* є Web-сервером, тобто середовищем для роботи Web-сервісів TFS, забезпечуючи *доступ* до функціональності сервера VSTS з боку його клієнтів.

Рівень даних складається з набору баз даних, де TFS зберігає свої дані. Він реалізований на основі продуктів MS SQL Server і SharePoint.

Залежно від розміру компанії-розробника ПЗ і передбачуваного навантаження ці два рівні TFS можуть бути встановлені на одному сервері (*single-serverdeployment*) або на двох різних серверах (*dual-serverdeployment*). Для дуже великих компаній можливе використання ме-

ханізмів *кластерізації*, які вбудовані в *Microsoft SQL Server* і InternetInformation Server.

BuildAgent – ще одна серверна підсистема VSTS. Як вже мовилося вище, вона призначається для виконання *зборки проектів*. Виконання *збірки* проекту відбувається засобами пакету *.NET Framework*, за допомогою стандартної *утиліти* цього пакету *Msbuild*, яка, отримавши завдання на збірку, викликає відповідний *компілятор* з *.NET Framework*. Цей же механізм використовується і для *збірки* проекту, яку запущено з *VisualStudio*.

У разі виконання *збірки* засобами BuildAgent, процедура відбувається *за* таким сценарієм. Компоненту *TFS BuildService* у складі TFS повідомляє такий же компоненті на комп'ютері, де розташований *BuildAgent*, що треба запустити виконання *збірки*. А та, у свою чергу, будучи *системним сервісом* і будучи запущеною, виявляється тим процесом *Windows*, в рамках якого і буде відбувається виконання *збірки* під управлінням компоненти Msbuild. При цьому весь *зв'язок* з TFS для виконання сценарію *збірки* здійснює компонента *CustomTasks*. У сценарії *збірки* указується, звідки потрібно брати початкові тексти збираного додатку, звідки брати регресійні тести і як їх запускати, як створювати звіти *за наслідками збірок* і так далі.

12.3. Правила інсталяції

Клієнтська частина встановлюється легко, або у якості розширення існуючої системи VisualStudio, або на чисту машину (в цьому випадку базова інфраструктура VisualStudio буде встановлена автоматично). Основна робота під час інсталяції VSTS – це розгортання серверної частини, тобто TFS. У версії 2008-го року інсталяція TFS значно покращена і спрощена в порівнянні з версією 2005-го року, проте, вимоги на програмне оточення як і раніше достатньо жорсткі:

- Microsoft Windows Server 2003 і вище;

- Microsoft SQL Server 2005 і вище;
- Internet Information Server 6 (для Windows Server 2003) або 7 (для Windows Server 2008);
- Active Directory Domain 5.0 і вище (TFS не працює з доменом 4.0). Важливо відзначити, що нормальне використання TFS поза доменом налагодити досить складно – для цього доводиться використовувати технологію VPN або Web-клиента, що веде до істотного збільшення витрат на адміністрування і накладних витрат при роботі.
- Sharepoint Server 3.0 або Microsoft Office Sharepoint Server 2007 і вище. При оновленні з TFS 2005 можна залишитися на Windowssharepoint Server 2.0.

12.4. Пакет Team Explorer

Даний пакет є найпоширенішим клієнтським додатком VSTS. Він вбудовується в середу *VisualStudio* у вигляді плаваючого вікна, а також ряду діалогових вікон і вікон-документів. Його зовнішній вигляд представлений на Рис. 12.5.

Основне *дерево Team Explorer* містить:

- список доступних TFS-серверів (1); кожен такий сервер є екземпляром серверної частини TeamSystem і, як правило, розташовується на окремому комп'ютері;
- список доступних проектів для кожного з підключених серверів (2);
- панелі інструментів інструментального вікна для того, щоб підключити/додати в TFS новий проект (3).

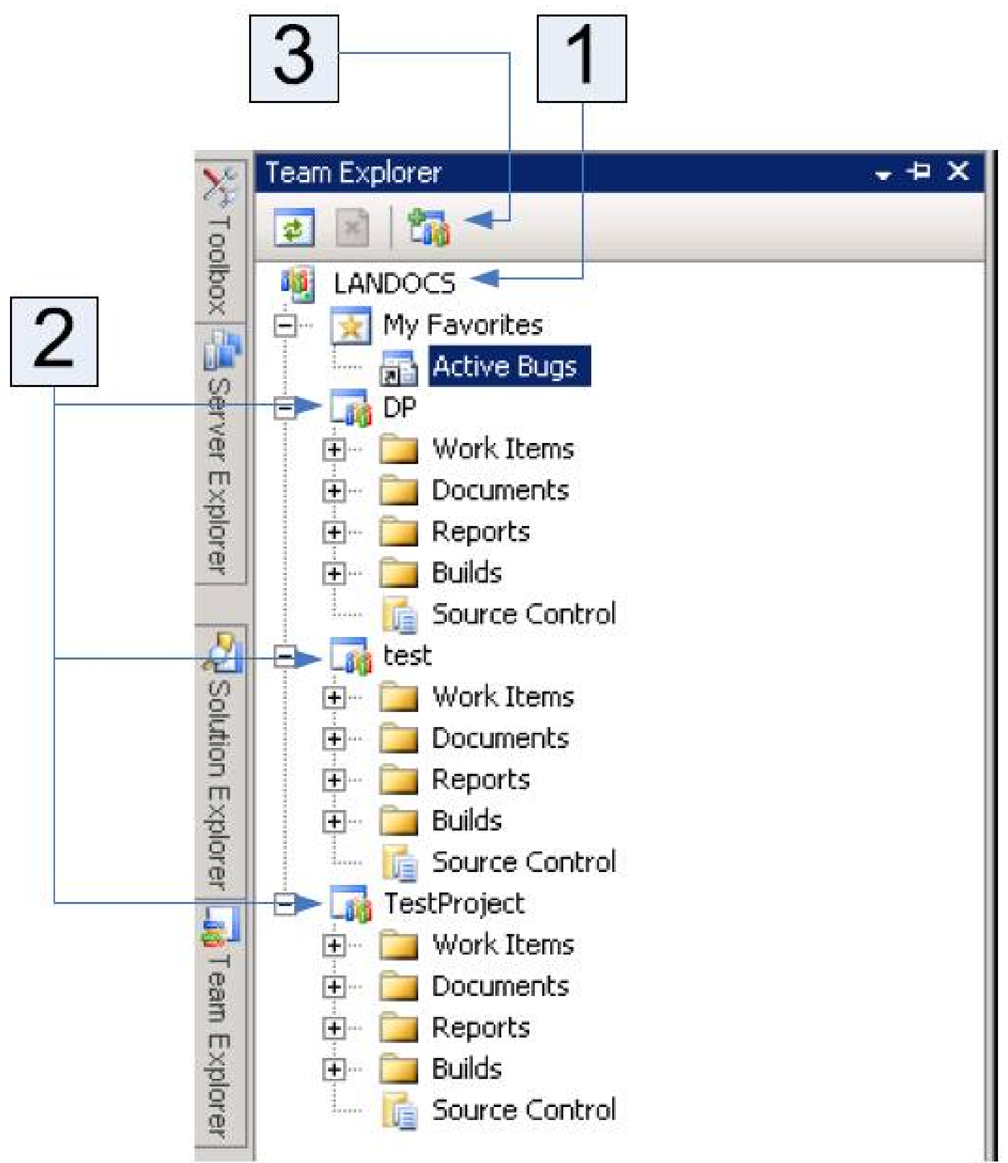

Рис. 12.5. Зовнішній вигляд Team Explorer

Для кожного з проектів в дереві *Team Explorer* відображається така *інформація*.

- Список елементів роботи (*WorkItems*) проекту, тобто всіх тих дискретних елементів роботи в проекті, які створюють менеджери і інші учасники проекту для того, щоб ні про що не забути, а також для комунікації один з одним.

- Список доступних документів (Documents). У цьому списку відображаються документи, що зберігаються на порталі проекту. Як, правило, це нормативні або допоміжні документи, що не вимагають зберігання в системі *контролю версій*.
- Список доступних *звітів* (Reports). У цьому списку представлені доступні для проекту звіти. Результат виконання звіту відкривається в окремому вікні документі.
- Список складок (Builds) проекту – описів і результатів.
- Система *контролю версій* (SourceControl). Дозволяє отримати доступ до версійного депозитарію з основними артефактами проекту (відкривається в окремому вікні-документі).

Крім того, через контекстні *меню* в дереві проектів можна виконувати такі *операції*:

- створити новий проект або підключиться до того, що існує;
- поміняти налагоджування сервера або проекту;
- створити/вилучити/змінити запит на елементи роботи;
- створити/вилучити/змінити звіт;
- створити/вилучити/змінити/запустіть процес *збірки*;
- створити/змінити/вилучити документ;
- підписатися на певні сповіщення або відмінити підписку.

Контрольні запитання

1. Для чого корпорацією Microsoft пропонується комплекс VisualStudioTeamSystem (VSTS)?
2. Що забезпечує комплекс VisualStudioTeamSystem (VSTS)?
3. Які засоби є ядром VSTS?
4. Навести приклади елементів робіт.
5. Для чого використовується елемент роботи?
6. Що має кожен елемент роботи?

7. На що розділяється VSTS як на програмний продукт?
8. З чого складається серверна частина VSTS?
9. З чого складається клієнтська частина VSTS?
10. З яких основних рівнів складається основна серверна підсистема VSTS?
11. Які особливості інсталяції VSTS?
12. Що містить основне дерево Team Explorer?
13. Яка інформація відображається в дереві Team Explorer для кожного з проектів?
14. Які операції можна виконувати через контекстні меню в дереві проектів?

Глава 13. VSTS: Управління елементами робіт (Work Items)

Визначення, властивості, життєвий цикл. Реквізити. Засоби використання (на прикладі елементу роботи task). Доступ до елементів роботи. Елементи роботи у разі планування. Елементи роботи в подальшій розробці. Елементи роботи у звітах.

13.1. Визначення, властивості, життєвий цикл

Огляд. Повернемося до елементів робіт VSTS – ключових дискретних характеристик проекту, таких як завдання (task), помилка (bug), ризик (risk) і так далі. Ці характеристики виділені в VSTS з метою конкретизувати об'єкти управління в проекті, зробити це управління прозорим у таких аспектах:

- забезпечити доступ до однієї і тієї ж інформації для різних учасників (і, головне, ролей!) в проекті; наприклад, доступ до помилок для менеджерів, розробників і тестерів;
- простежувати зв'язки одних елементів з іншими, наприклад, змін початкового коду і тими помилками, для виправлення яких ці зміни були виконані.

Завдяки єдиному середовищу, що включає засоби підтримки різних видів елементів роботи, в VSTS набагато простіше будувати зв'язки між елементами роботи різного вигляду та відстежувати їх зміни, ніж у разі використання окремих продуктів підтримки процесу. Наприклад, не потрібно чекати моменту, коли інформація про помилку або завдання буде перенесена з однієї системи в іншу. Адже, традиційно програмні засоби планування (там, де визначаються завдання), управління помилками (там, де відбувається облік помилок), засоби версійного контролю – це різні засоби. Крім того, через наявність єдиного інформаційного репозитарію в VSTS

можливі строгі посилання на такі об'єкти, певна збірка або тест, і отримання, за відповідним запитом, докладній інформації за різними фільтрами, на різну глибину деталізації. Можливо також набудувати автоматичну генерацію елементів роботи, наприклад, помилок у разі невдалої автоматичної збірки або у разі автоматичного прогону тестів.

Елементи роботи можна зв'язувати з іншими артефактами проекту – файлами з початковим кодом, збірками (як налагоджуваннями, так і результатами), документами (з процесу, проектними, призначеними для користувача і ін.), звітами, які можуть також зберігатися в VSTS. Все це показано на Рис. 13.1. Важливо відрізняти елементи роботи від цих артефактів – останні є робочими продуктами (точніше, з них робочі продукти формуються), а елементи роботи є інформацією в проекті, що управляє. Проте, і з них також можна створювати робочі продукти – генерувати звіти, документи і плани в продуктах Office і так далі. Але самі по собі елементи роботи є засобом оперативної роботи над проектом, а не результатами (у тому або іншому сенсі).

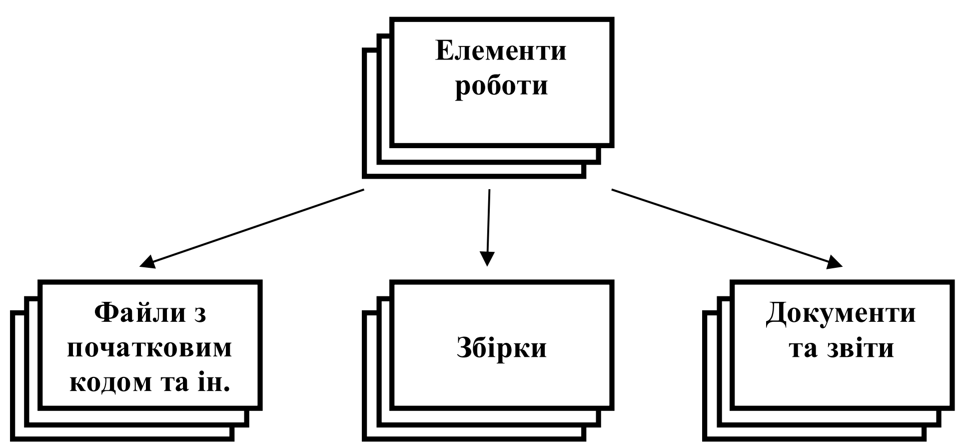

Рис. 13.1. Елементи роботи, що зберігаються в VSTS

Реквізити. Кожен елемент роботи належить певному типу. Тип елементу роботи визначає набір реквізитів, для кожного з яких можна задати:

- ім'я, що відображається в *звітах* і призначеному для користувача інтерфейсі TFS;

- ім'я для посилань (*Reference Name*), що використовується для вказування посилань на даний *реквізит* з інших місць шаблону процесу;

- тип – один із зумовлених типів для реквізитів: date, *int*, *string*, bool; типи можуть *системними*, тобто бути посиланнями в одне з типових сховищ TFS: наприклад, тип *build* results указує на інформацію про результати збірки;

- текстовий опис реквізиту, що відображається в спливаючих *підказках,* звітах і повідомленнях;

- режим використання в *звітах* – чи можна використовувати цей *реквізит* в *звітах* і як його слід використовувати.

Бувають також системні реквізити, імена, типи і обробка яких "закладена" в TFS. Це, наприклад, такі реквізити як стан (*state*), причини (*reasons*), зв'язки (*links*).

Окремої уваги заслуговують імена для посилань. Вони дозволяють *ідентифікувати* даний *реквізит* не лише в межах одного типу елементів роботи, але і в межах всього TFS-проекту, а також у разі перенесення елементів роботи з проекту в проект. Для підтримки унікальності цих імен рекомендується використовувати концепцію *просторів імен* (*namespaces*). Крім того, імена для посилань служать для організації свого роду пулу реквізитів – однакове ім'я для посилань, що використовується в різних типах елементів роботи, має на увазі однаковий сенс відповідних реквізитів, а також однакову їх роль з погляду формування звітів. Існує набір зумовлених імен для посилань, відповідних системним реквізитам. Реквізити з відповідними іменами для посилань можуть піддаватися особливій обробці з боку TFS.

Кожен *реквізит* може або не брати участь в *звітах* взагалі, або брати участь в режимах:

- як вимірювання (*Dimension*) – як вимірювання при побудові звітів; цей режим є допустимим для чисел, дат і *символьних полів* із зумовленим набором значень;
- у деталях (*Details*) – тобто як детальна інформація звіту; цей режим є допустимим для чисел, дат і довільних рядків;
- як *метрика* (*Measure*) – тобто в *звітах* використовується деяка *статистична функція*, що *обчислена* для значень даного реквізиту.

Життєвий цикл елементу роботи визначається двома системними реквізитами: станом і причиною.

Перший описує поточний стан елементу роботи і визначає його поточну роль в процесі. Кожен тип елементів роботи описує допустимий набір станів, наприклад, "активний", "завершений", "перевірений" і так далі.

Крім того, кожен тип елементу роботи має опис переходів між своїми станами, причини, що викликають ці переходи і дії, що виконуються в них. Причинами можуть бути, наприклад, "виконаний", "застарілий", "відкладений" і так далі. Переходи може здійснювати сама система TFS, автоматично, але в більшості випадків розробник сам ініціює перехід, вносячи до системи інформацію про те, що виконав завдання, виправив помилку і так далі.

Таким чином, *життєвий цикл* елементу роботи має вигляд *орієнтованого графу*, з навантаженням як *на* вузлах, так і *на дугах*. Використання такого графа замість навантаженого лише *по* вузлах, як наприклад, часто можна зустріти в системах управління помилками, дозволило різко скоротити розмір опису і підвищити інформативність.

Для налагоджування типів робочих елементів використовується продукт Team Foundation Power Tools. Він є вільно поширюваним продуктом і містить, зокрема, візуальний редактор, що дозволяє переглядати і редагувати життєвий *цикл* елементу роботи у візуальному вигляді (див. Рис. 13.2). На цьому рисунку показаний *граф життєвого циклу* для типу робочого елементу "Помилка". Ми можемо бачити три стани – *Active* (виявлена помилка), *Resolved* (помилка виправлена), *Closed* (помилка закрита) і переходи між цими станами. У кожному переході є *прямокутник Transition*, в якому містяться параметри переходу – причина, дії, які потрібно виконати, *список* реквізитів, які повинні бути змінені при переході, і деяку іншу інформацію.

Ще однією важливою складовою опису *життєвого циклу* реквізиту є **правила**, які описують різні обмеження на значення реквізитів елементу роботи, зокрема:

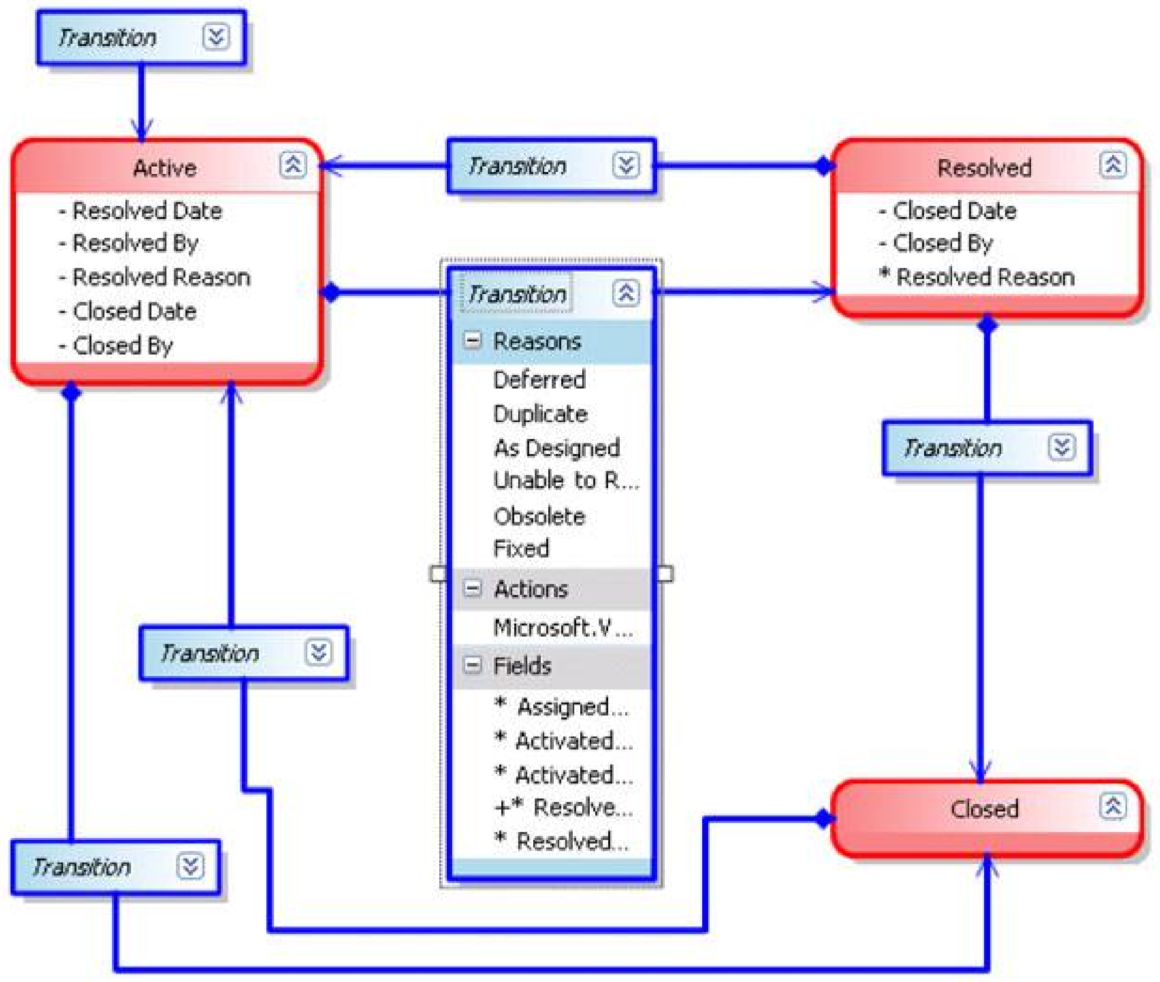

Рис. 13.2. Життєвий цикл елементу роботи типу "Помилка".

- базові обмеження: обов'язковий для заповнення чи ні, доступний лише на читання, порожньо, не може бути скинуто, може бути лише скинуто;
- обов'язковий збіг або неспівпадання за значенням з іншим реквізитом;
- є датою у минулому або в майбутньому;
- є ім'ям користувача, що входить до заданої групи, наприклад те, що за цей елемент роботи повинен відповідати працівник лише з групи тестувальників.

- значення повинне завжди задовольняти шаблону деякого регулярного виразу;

- значення є одним із зумовлених значень або навпаки, їх немає; для завдання таких правил є допустимим використання посилань на зовнішні списки, наприклад, на списки виконаних збірок; крім того, можна визначити список передбачуваних значень, які будуть запропоновані користувачеві, але не обов'язкові до обирання.

У описі кожного правила можна вказати *ім'я користувача* або групи, для яких це правило застосовуватиметься. *Виключенням* є лише правило "дозволити зберегти поточне *значення"*, яке є завжди вживаним для всіх користувачів.

Крім того, можна описати правила, які можуть бути вживані в різні моменти часу, зокрема:

- обмеження, що постійно діють, або вживані на разі створення правила;

- у разі зміни певного реквізиту, або навпаки, якщо реквізит залишився незмінний;

- у разі збігу або неспівпадання значення реквізиту із зумовленим значенням;

- у разі переходу або під час знаходження елементу роботи в певному стані;

- у разі здійснення певного переходу;

- у разі здійснення певного переходу з певної причини в життєвому циклі елементу роботи.

Система правил надає широкі можливості для специфікації різних тонкощів *бізнес-процесу*, проте *платити* за цю гнучкість доводиться скла-

дністю налагоджування. Саме опис коректних правил жадає від проектувальника найбільших зусиль при розробці власного шаблону і його використанні в реальному промисловому проекті.

Ще один важливий механізм налагоджування реквізитів в VSTS – це завдання способу подання реквізиту в *екранних формах редагування*, перегляду в *звітах*. Для цієї мети в TFS існує достатньо гнучкий діалект *XML*, що включає зумовлений набір елементів *призначеного для користувача інтерфейсу*, а також що дозволяє групувати їх, розбивати *на* колонки або розміщувати на закладках. Використовуючи вкладеність елементів і груп, можна описати строгий, зручний і красивий *інтерфейс* користувача, але іноді для цього потрібний багато часу. Деякі типи елементів *призначеного для користувача інтерфейсу* можна пов'язувати з реквізитами елементу роботи, використовуючи відповідне ім'я для посилань.

13.2. Засоби використання елементів роботи

Приклад: елемент роботи task. Розглянемо елемент роботи типу *task* (завдання). Як правило, на початку проекту деякий експерт (як правило, системний архітектор, провідний розробник і так далі) проводить *аналіз* всієї необхідної роботи *за* проектом і розбиває її на підзадачі, встановлюючи відповідальних, терміни і так далі. Ці підзадачі з відповідними атрибутами і є елементами роботи типу *task*.

Потім *менеджер проекту*, з урахуванням списку всіх завдань і їх взаємозв'язків, будує календарний план. На цьому етапі менеджерові можуть виявитися корисними засоби *Project* і *Microsoft Excel* – він користується ними на основі відповідних мостів, наявних в TFS.

Далі розробники починають реалізовувати відповідні завдання. Після того, як була внесена остання зміна і завдання виконані, розробник переводить елемент роботи в стан Resolved і інформація про нього увійде до

звіту за допомогою автоматичної збірки. При виявленні помилок реалізації тестер створить новий елемент роботи типу Bug і проставить йому зв'язок з початковим завданням. Якщо ж тестер виявить, що функціональність реалізована не в повному об'ємі, то він може вирішити перевести завдання назад в стан Active. Якщо ж реалізована функціональність достатньо стабільна, а наявні помилки не є критичними, тестер переводить завдання в стан Closed.

За всім цим процесом спостерігає *менеджер проекту*, який використовує як запити на елементи роботи, так і засоби *інтеграції* з офісними *додатками*, а також засоби побудови звітів. Таким чином, він дістає можливість максимально оперативно реагувати на виникаючі нештатні ситуації, відставання від плану і виникаючі додаткові незаплановані роботи.

Створення елементів роботи. Для створення нового елементу роботи можна скористатися пунктом *меню Team*, що додається в *VisualStudio* разом з *Team Explorer*, як показано на Рис. 13.3.

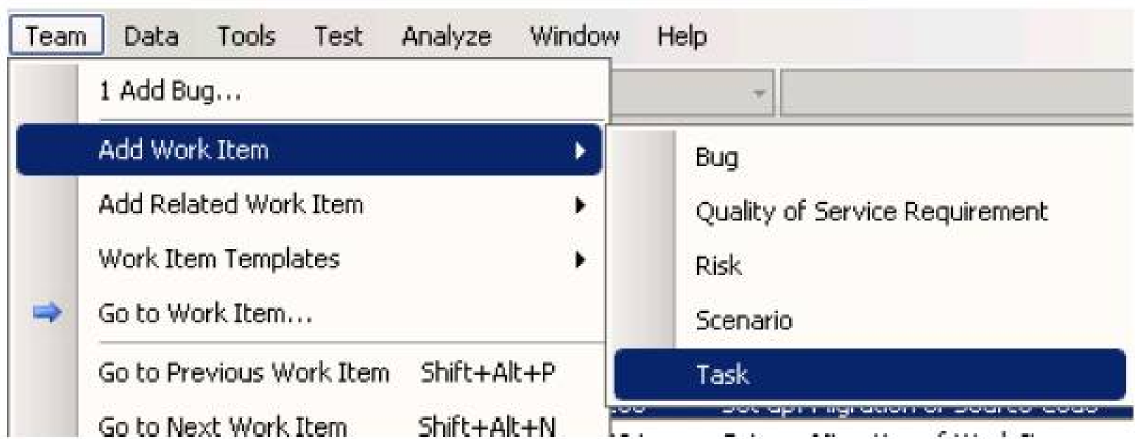

Рис. 13.3. Створення елементу роботи.

Після вибору пункту *меню Add Work Item* відобразився *список* з типів елементів роботи, що існували в даному проекті. Далі, після вибору відпо-

відного пункту *меню* буде відкрито вікно редагування нового елементу роботи, як показано на Рис. 13.4:

Це вікно дозволяє заповнити всі реквізити елементу роботи, а у разі помилок видасть відповідне попередження у верхній частині вікна. Після того, як всі поля заповнені, зберегти елемент роботи можна за допомогою кнопки на панелі інструментів. Після збереження елемент роботи автоматично отримає унікальний дентіфікатор{ і буде збережений в системі управління елементами роботи.Для додавання зв'язаних елементів роботи можна скористатися командою контекстного *меню Add* Related *Work Item*, як показано на Рис. 13.5:

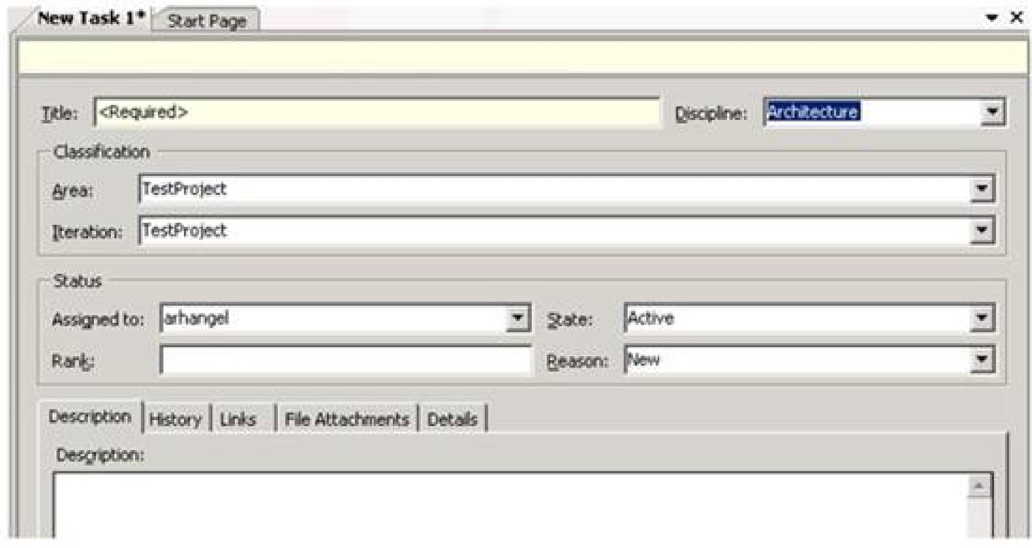

Рис. 13.4. Редагування елементу роботи.

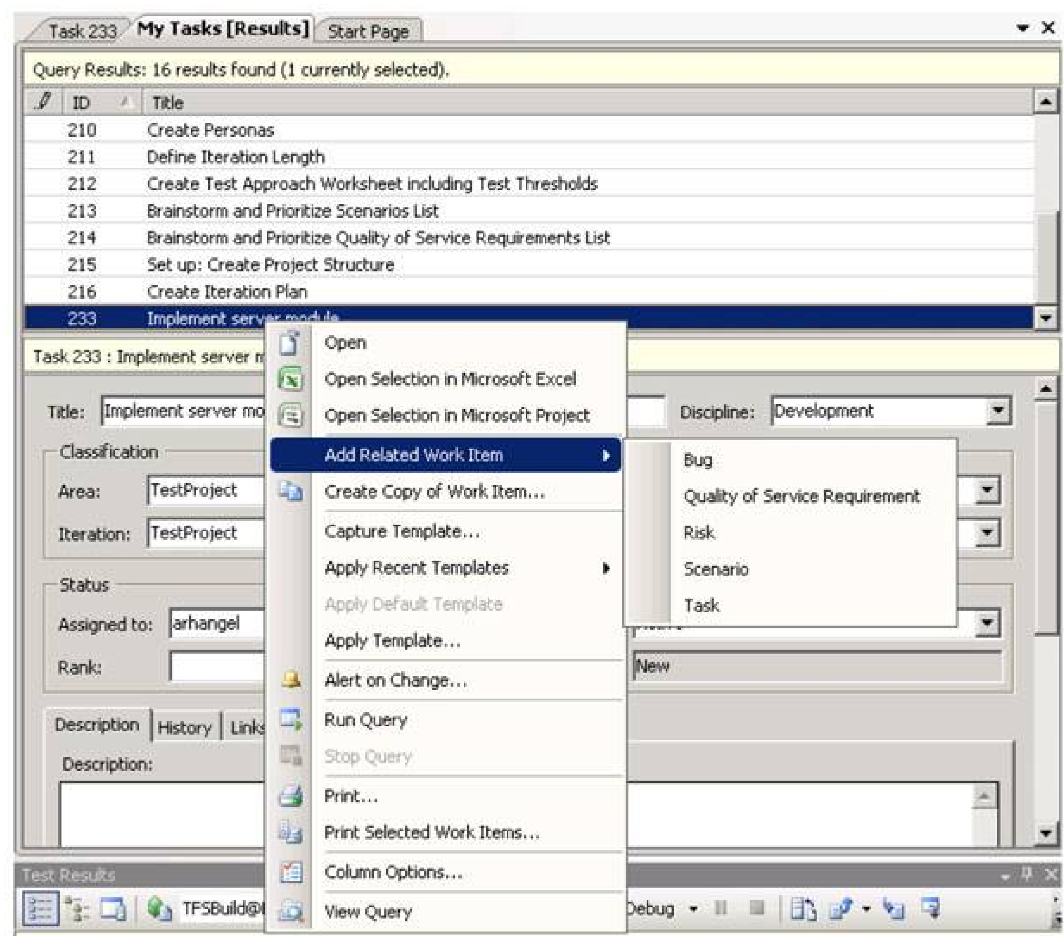

Рис. 13.5. Додавання зв'язаного елементу роботи.

Після додавання зв'язаного елементу роботи відкриється вікно редагування для знов створеного елементу роботи, при цьому зв’язок між двома елементами роботи буде додано автоматично, як показано на Рис. 13.6.

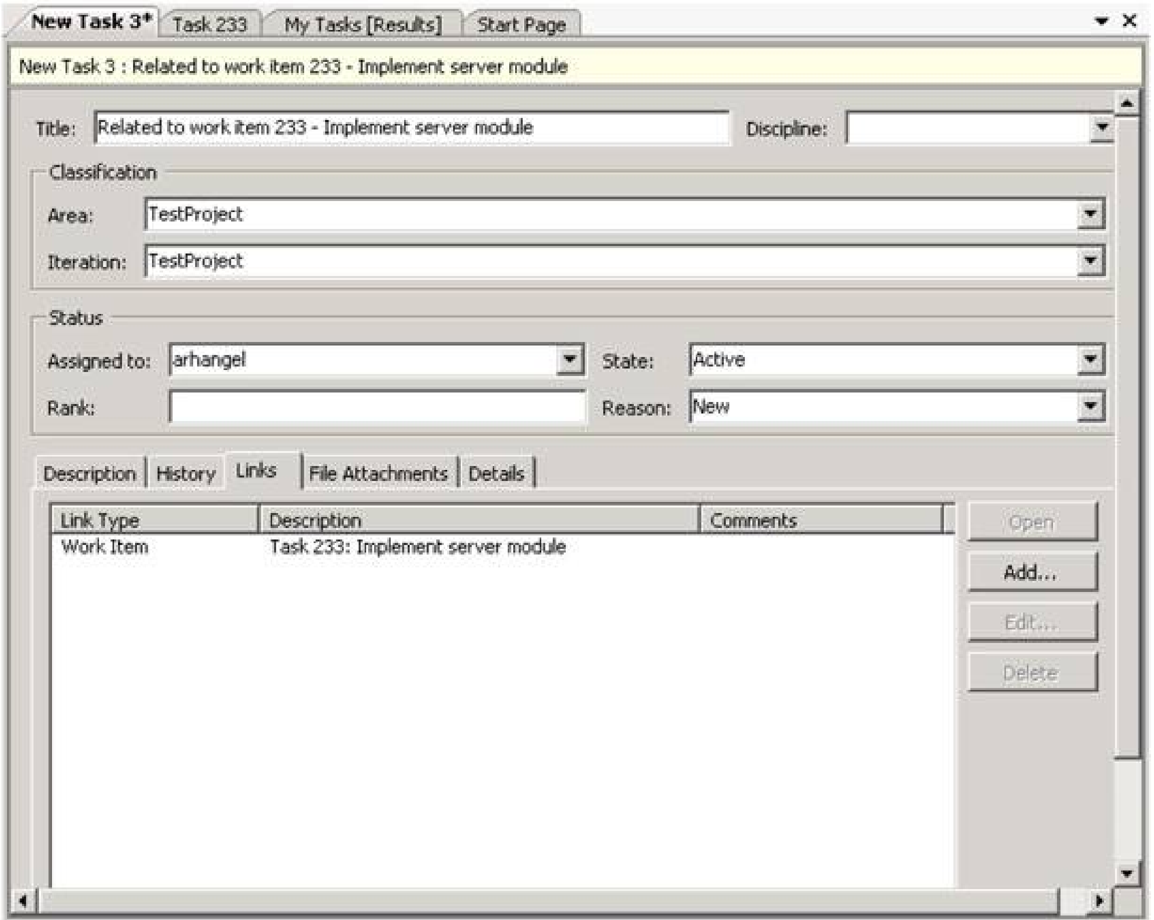

Рис. 13.6. Список зв'язків.

Створеному зв'язку можна приписати відповідний коментар. На жаль, він буде однаковим для записів про зв'язок в обох створених завданнях, що ускладнює ідентифікацію закінчень зв'язку.

Слід відмітити, що контекстне *меню* на Рис. 13.5 демонструє ще одну корисну під час створення елементів роботи можливість – шаблони елементів роботи. *Шаблон* визначається набором "наперед заповнених" атрибутів елементу роботи і може сильно полегшити життя учасникам проекту, яким доводиться створювати багато однотипних елементів.

Доступ до елементів роботи. Члени команди, що працюють з VSTS через *Team Explorer*, мають *доступ* до елементів роботи, відкривши в по-

точним проекті вкладку *Work Items* (див. Рис. 13.7). При цьому елементи *робіт* доступні не як величезна *купа* (легко зрозуміти, що їх може бути дуже багато в кожному проекті – сотні і навіть тисячі), а за допомогою спеціальних фільтрів – запитів (queries). Вони розподілені *по* теках *Team* Queries і My Queries. У першій теці розташовуються запити, які є видимими і використовувані всією командою. У другій теці розташовуються запити, які створені конкретним користувачам для себе особисто. Все це можна побачити на Рис. 13.7.

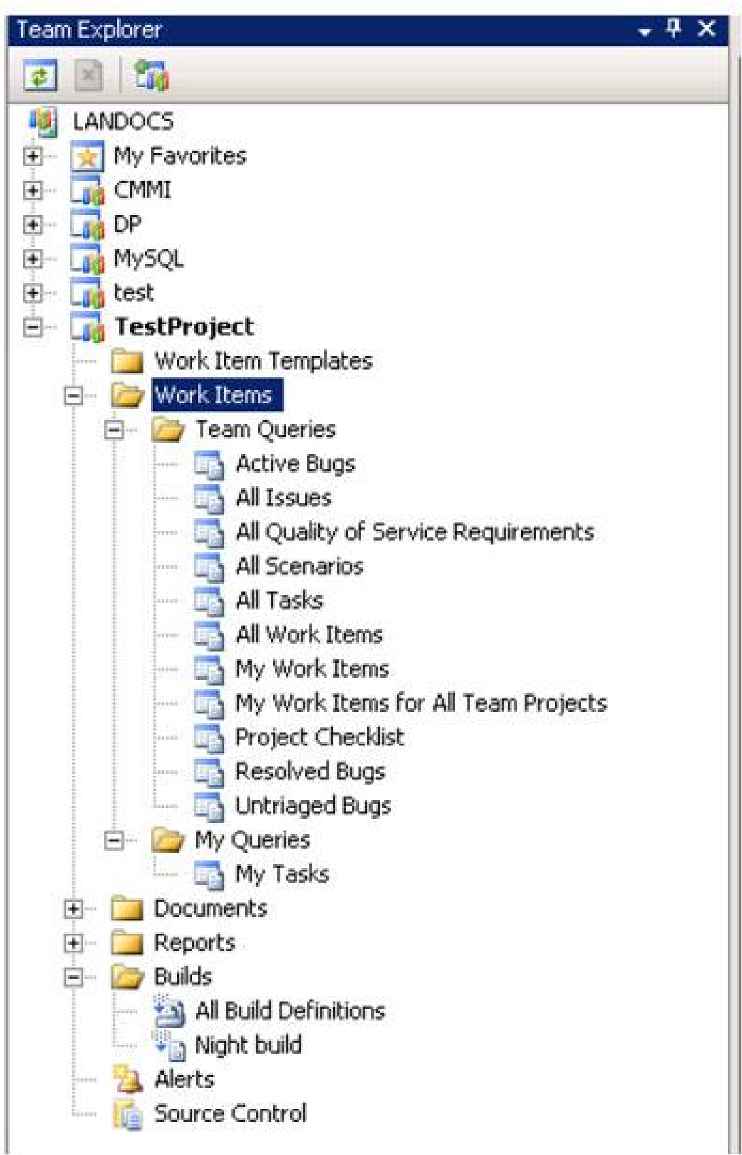

Рис. 13.7. Запити на елементи роботи.

Отже, коли розробникові знадобилося дістати *доступ* до певної групи елементів *робіт* (наприклад, до помилок, які йому потрібно виправити), він вибирає відповідний *запит* і виконує його. В результаті в спеціальному вікні буде відкритий список елементів роботи, що задовольняють даному запиту (Рис. 13.8), а у разі вибору певного елементу, в нижній частині вікна-списку відобразиться детальна *інформація* про цей елемент (а при подвійному клацанні елемент роботи буде відкритий в окремому вікні).

Спосіб відображення списку (набір колонок, *сортування*, і так далі) налагоджуються індивідуально для кожного користувача, а форма детальної інформації про елемент роботи налагоджуються для проекту в цілому для кожного типу елементів роботи окремо.

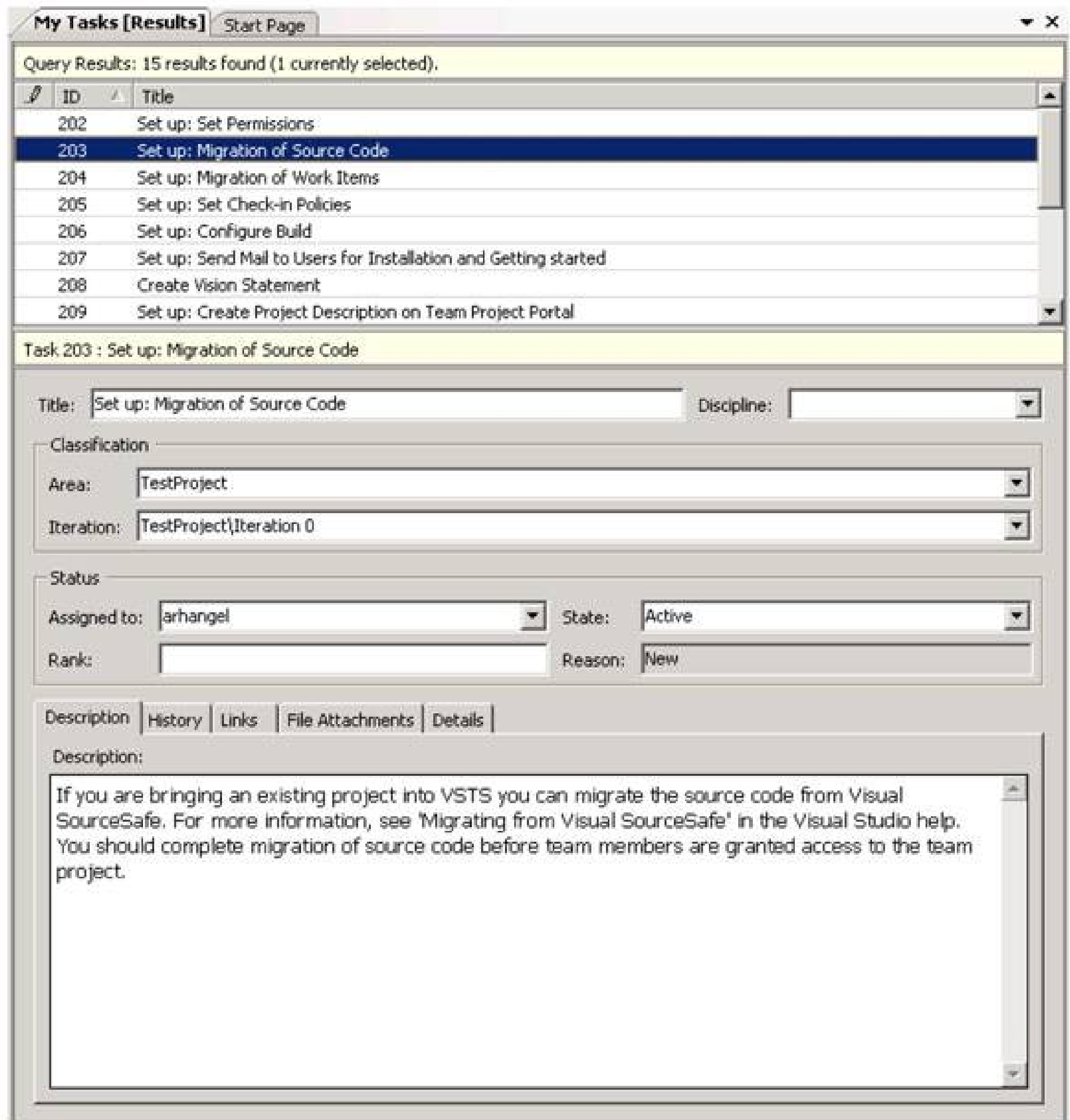

Рис. 13.8. Список елементів роботи.

Під час редагування і створення елементів роботи враховуються всі ті правила, які були задані в шаблоні процесу, де визначений даний тип елементів роботи. Це виражається в тому, що відповідні поля форми властивостей елементу роботи допускають або забороняють редагування, дозволяють вибір значень лише з певного списку і так далі.

Виділені (selected) елементи роботи можна експортувати в пакети *Microsoft Project*, *Word*, *Excel* за допомогою відповідних кнопок панелі ін-

струментів (та). Зміни, які проведені з елементами *робіт* в цих пакетах, можна потім завантажити назад в TFS використовуючи відповідні бібліотеки-розширення офісних додатків.

Елементи роботи під час планування. Не складно відмітити, що така важлива роль як *менеджер проекту*, не отримала власного розділу в *Visual Studio*. Зв'язано це з тим, що основна платформа *Visual Studio* погано пристосована для завдань, які доводиться вирішувати цій ролі. Набагато вдаліше для цього підходять *офісні додатки – Microsoft Excel* і *Microsoft Project*. Тому для повнішого залучення менеджера до інформаційного *простору* проекту *Team System* надає спеціальні мости.

Розглянемо приклад з пакетом *Project*. У цьому пакеті, за наявності на тому ж комп'ютері *Team Explorer*, з'являється *пункт меню Team*. У нім потрібно вибрати необхідний проект в VSTS, як показано на Рис. 13.9.

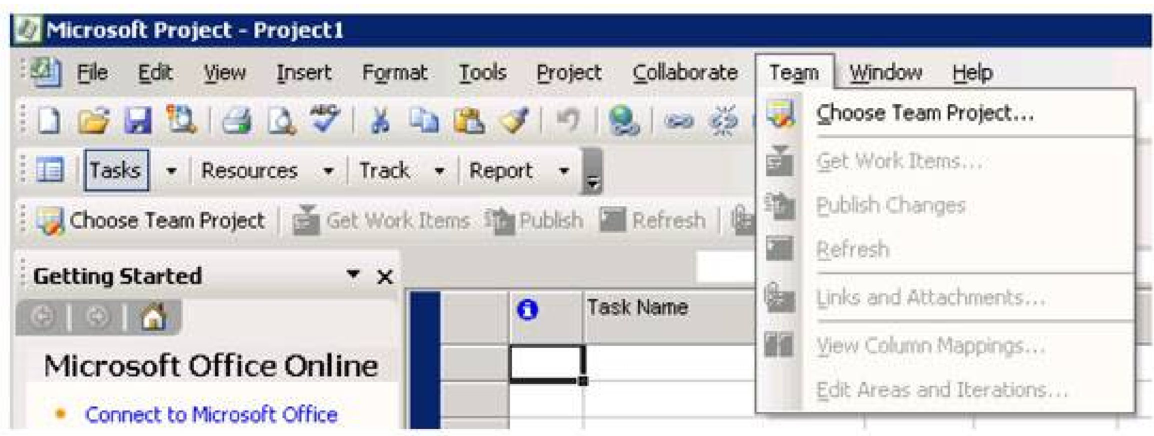

Рис. 13.9. Меню Team в Microsoft Project.

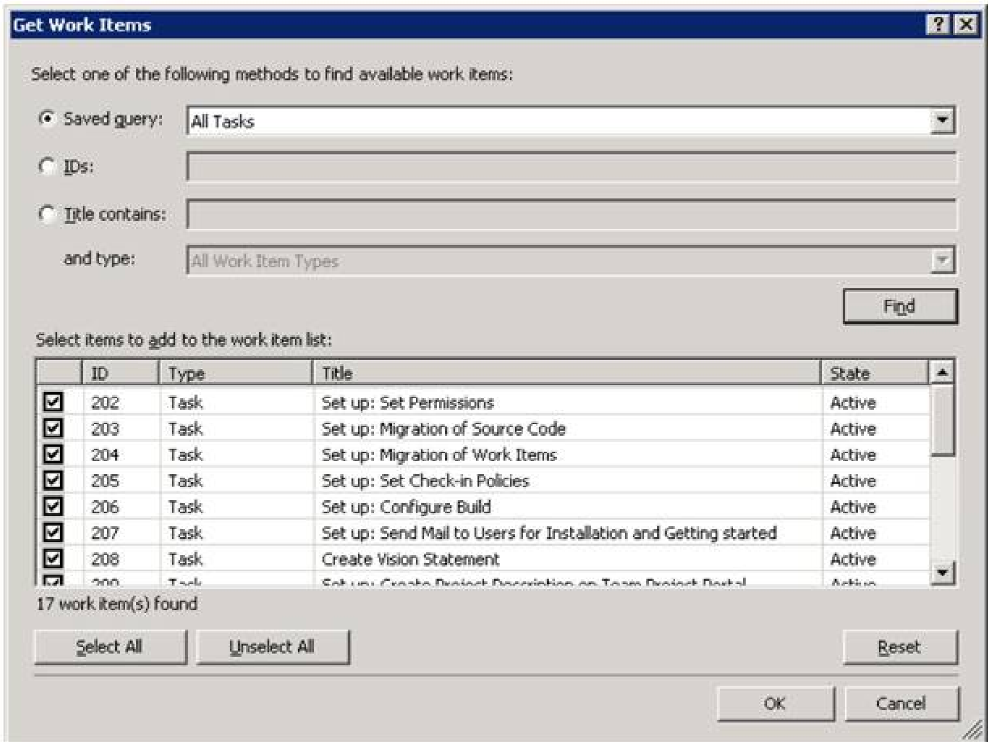

Рис. 13.10. Форма пошуку елементів роботи.

Після цього з'явиться можливість використовувати інші пункти *меню Team*, зокрема – *пункт меню Get Work Items*, що дозволяє рахувати необхідні елементи роботи з сервера. При виборі цього пункту *меню* з'явиться діалог, який показаний на Рис. 13.10 . *Пошук потрібних* елементів роботи можна здійснювати відповідно до існуючого запиту, *за* заданими ідентифікаторами або *за назвою* елементу роботи.

Після того, як потрібні елементи вибрані, вони будуть автоматично імпортовані в *Project* – див. Рис. 13.11:

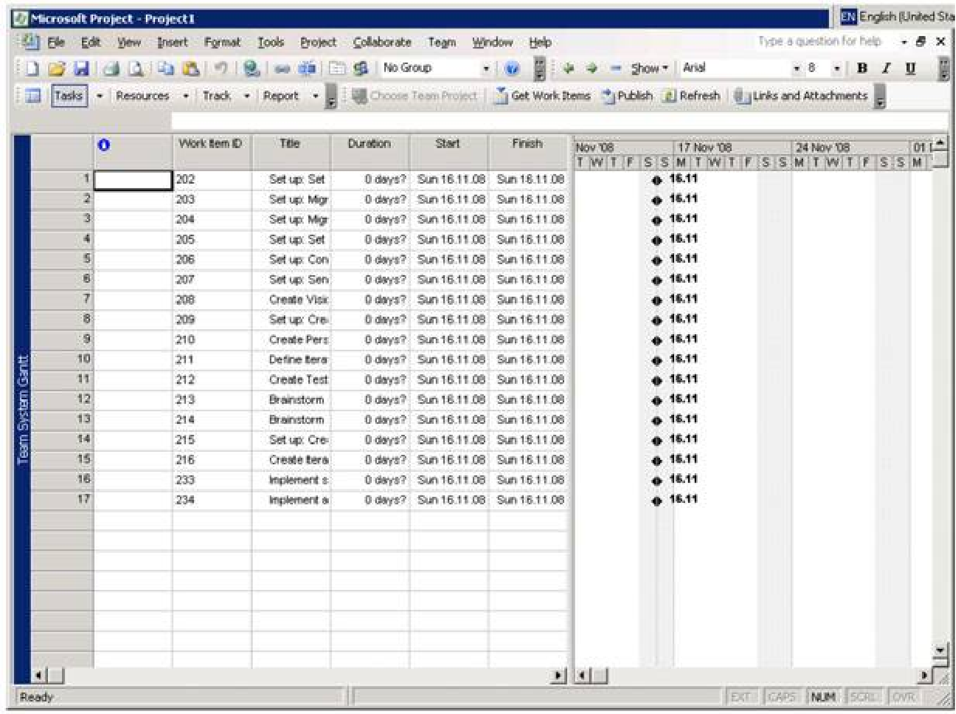

Рис. 13.11. Елементи роботи в Microsoft Project.

Після імпорту елементів роботи *менеджер* може проводити з ними всі дії, які він звик виконувати в *Project*. В даному випадку він створює повноцінний план *робіт*, як показано на Рис. 13.12 .

Відображення реквізитів елементів роботи *task* на атрибути завдань *Project* спочатку задається в шаблоні процесу розробки. Крім того, *менеджер* може дістати *доступ* з *Project* до решти всіх атрибутів завдання як до розширених полів, як показано на Рис. 13.13.

Після того, як всі дії в *Project* виконані, для внесення їх в VSTS скористатися командою **Publish**, а для отримання оновлень – командою **Refresh**. Крім того, сам *файл* з планом можна зберегти на диску або порталі Sharepoint. При цьому інформація про зв'язок з сервером TFS так само збережеться.

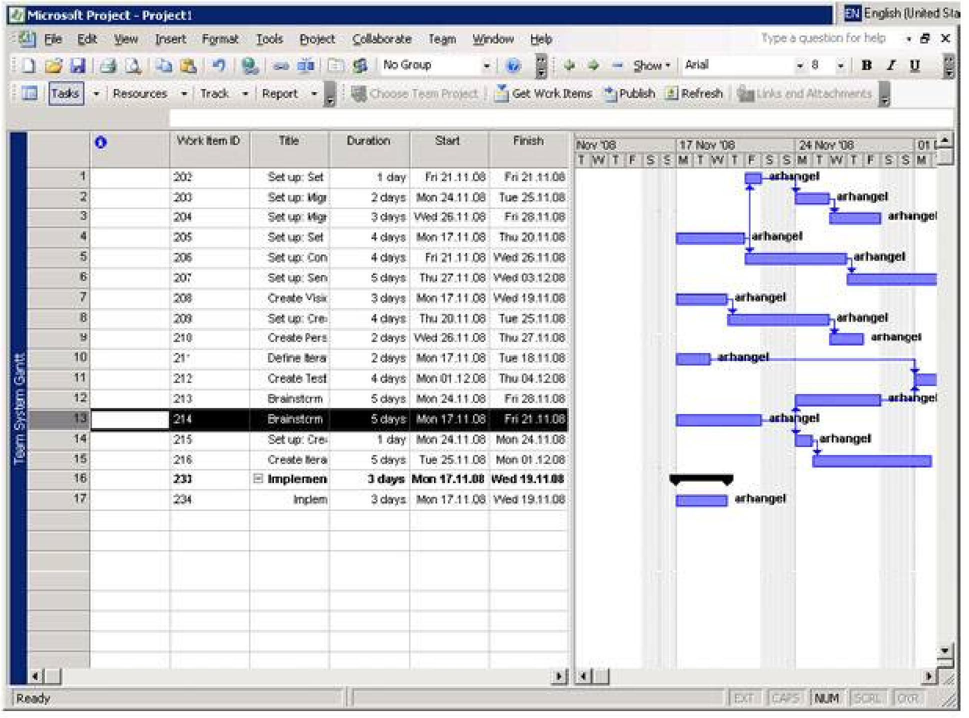

Рис. 13.12. Редагування елементів роботи в Microsoft Project.

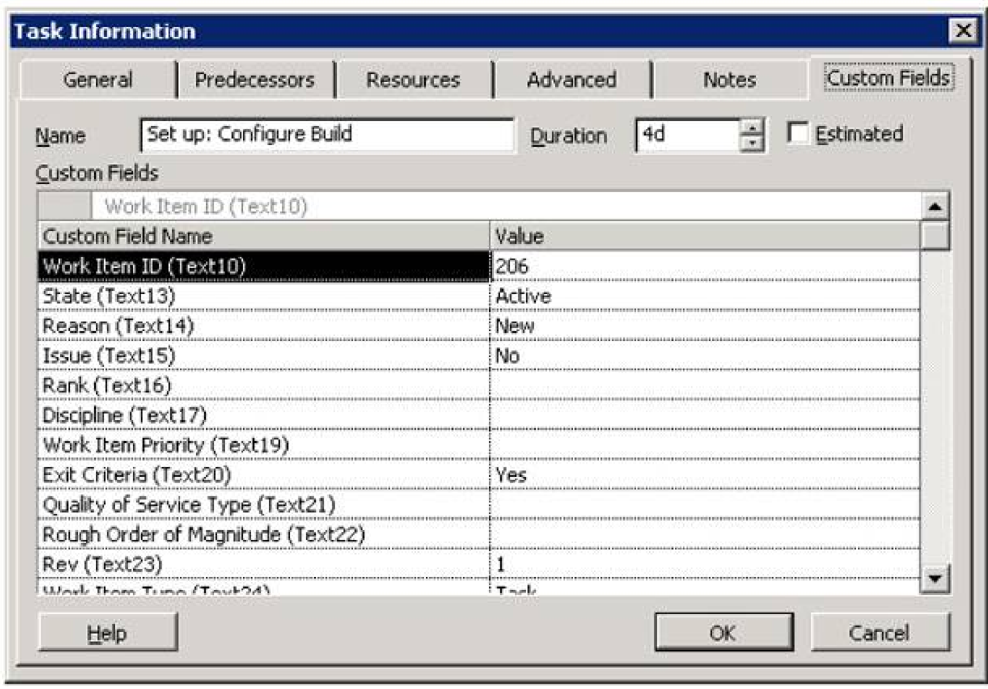

Рис. 13.13. Доступ до реквізитів елементу роботи.

Елементи роботи в подальшій розробці. Після того, як був побудований план і призначені виконавці певним завданням, відповідальні виконавці побачать їх в результатах відповідних запитів (типу My Tasks). І почнуть виконувати відповідну роботу. При цьому доведеться вносити деякі зміни до системи контролю версій, і у цей момент у них з'являється можливість вказати пов'язані з даними змінами елементи роботи – помилки, які виправляються і завдання, які виконуються в даній зміні коду, і так далі.

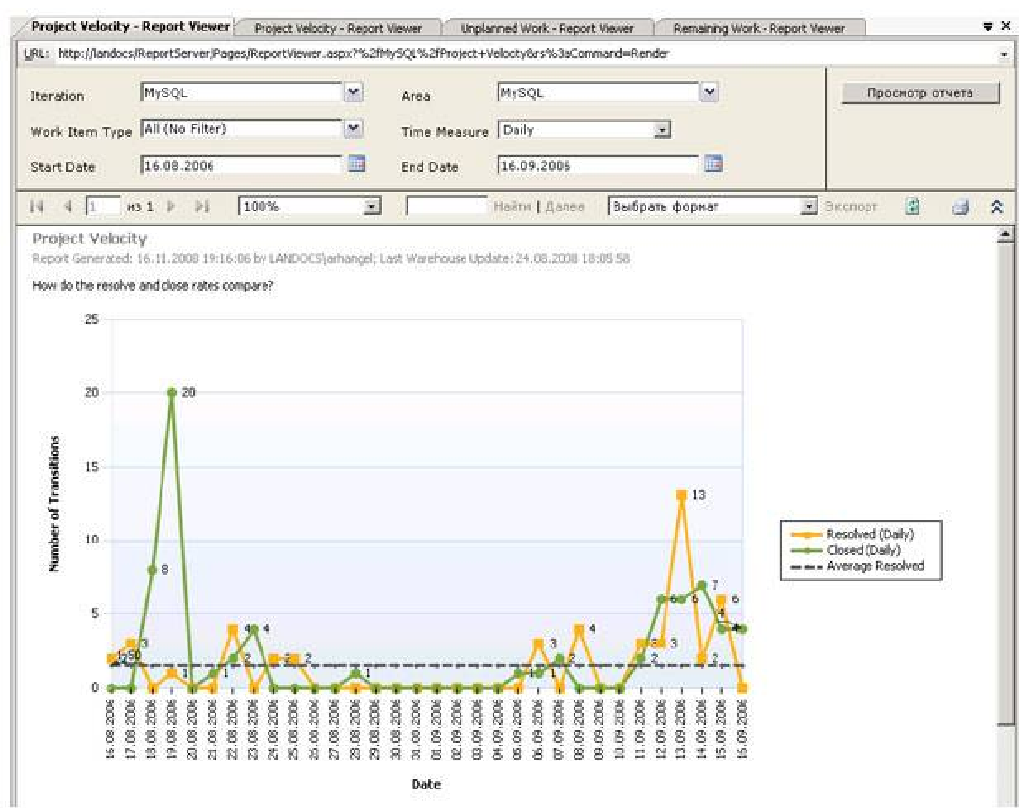

Рис. 13.14. Звіт Project Velocity (Швидкість проекту)

Елементи роботи в звітах. Одним із основних завдань підсистеми роботи із звітами є віддзеркалення реального актуального статусу проекту і *аналіз* його історії. Більшість звітів в TFS базуються саме на елементах роботи і відображають динаміку їх зміни. Зокрема, звіт *Project Velocity*, представлений на Рис. 13.14, відображає кількість закритих завдань відпо-

відно до днів (тижнів або місяців) і дозволяє судити про те, наскільки ефективно рухається проект.

На осі абсцис на цьому малюнку відкладається час, *на* осі ординат – кількість елементів роботи (в даному випадку – дефектів). Далі ми можемо бачити два *графіки* – зелений (скільки помилок було закрито), жовтий – скільки було знайдено. Сіра пунктирна лінія позначає середню інтенсивність роботи в проекті, яка вимірюється як кількість закритих помилок. З малюнка видно, що в проекті були сплески продуктивності (на початку і в кінці), а також спад в середині (в цей час розробник був у відпустці і тестувальник не тестував його компоненту).

Контрольні запитання

1. Для чого створені ключові дискретні характеристики проекту?
2. В чому переваги єдиного середовища VSTS, що включає засоби підтримки різних видів елементів роботи?
3. В чому різниця між елементами роботи та артефактами проекту?
4. Що таке реквізити елементу роботи і які реквізити існують в VSTS?
5. Для чого використовуються імена для посилань?
6. В яких режимах може брати участь будь-який реквізит в VSTS?
7. Чим визначається життєвий цикл елементу роботи?
8. Який вигляд має життєвий цикл елементу роботи?
9. Який продукт використовується для налагоджування типів робочих елементів?
10. Що описують правила, які є однією важливою складовою опису життєвого циклу реквізиту?
11. Які обмеження на значення реквізитів елементу роботи описують правила життєвого циклу реквізиту?
12. Які правила життєвого циклу реквізиту, що можуть бути вживані в різні моменти часу, додатково можна описати?
13. Як додається зв’язок між новими двома елементами роботи?

14. Що таке шаблони елементів роботи і навіщо він потрібен?
15. Яким чином члени команди, що працюють з VSTS, мають доступ до елементів роботи?
16. Як налагоджується спосіб відображення списку (набір колонок, сортування, і так далі) для кожного користувача?
17. Як налагоджується форма детальної інформації про елемент роботи для кожного типу елементів роботи?
18. Охарактеризувати зв’язок Visual Studio з офісними додатками – Microsoft Excel і Microsoft Project.

Глава 14. VSTS: Конфігураційне управління

Система контролю версій. Відслідковування змін в окремих файлах. Правила внесення змін. Управління гілками. Збереження без внесення. Автоматичні збірки.

14.1. Загальні положення

У VSTS є два типи інструментів для підтримки *конфігураційного управління – система контролю версій* і система управління *збірками*. Перша використовується для зберігання всіх основних *артефактів*, складових результату діяльності проектної команди (сюди входять початковий код додатку, *модульні* тести, тестові пакети і так далі). Друга система дозволить автоматизувати отримання образу кінцевого продукту у вигляді, що є готовим для тестування і відправки замовникові. Крім того, система управління збірками дозволяє безперервно контролювати якість кінцевого продукту завдяки *автоматичному тестуванню* і статичному аналізу коду . *В порівнянні з* іншими аналогічними системами в цьому аспекті робота у VSTS має декілька переваг:

- інтеграція з системою управління елементами роботи (а через неї і з системою звітів) дозволяють ефективніше відстежувати процес розробки і управляти ним;
- інтеграція з інтегрованим середовищем розробки є стандартом для будь-якої системи контролю версій, проте для систем управління збірками це не так – завдяки зручному призначеному для користувача інтерфейсу, який інтегрований у єдине середовище розробки, управління збірками стає значно простішим.

14.2. Система контролю версій

Функціональність системи контролю версій є стандартною для такого класу, тому детальніше слід зупинитися лише на її особливостях:

- відслідковування змін окремих файлів і їх "ув’язування" з елементами роботи;
- правила внесення змін (*check-inpolicies*);
- засоби управління "гілками";
- *збереження змін* без внесення.

Відслідковування змін окремих файлів. Основною характерною властивістю системи *контролю версій* в TFS є її інтеграція з іншими підсистемами TFS, а також більш тісна *інтеграція* з *VisualStudio*, ніж в багатьох інших системах контролю *версій.* Велика частина цих можливостей наочно демонструється самим зовнішнім виглядом check-in діалогу, який поданий на Рис. 14.1.

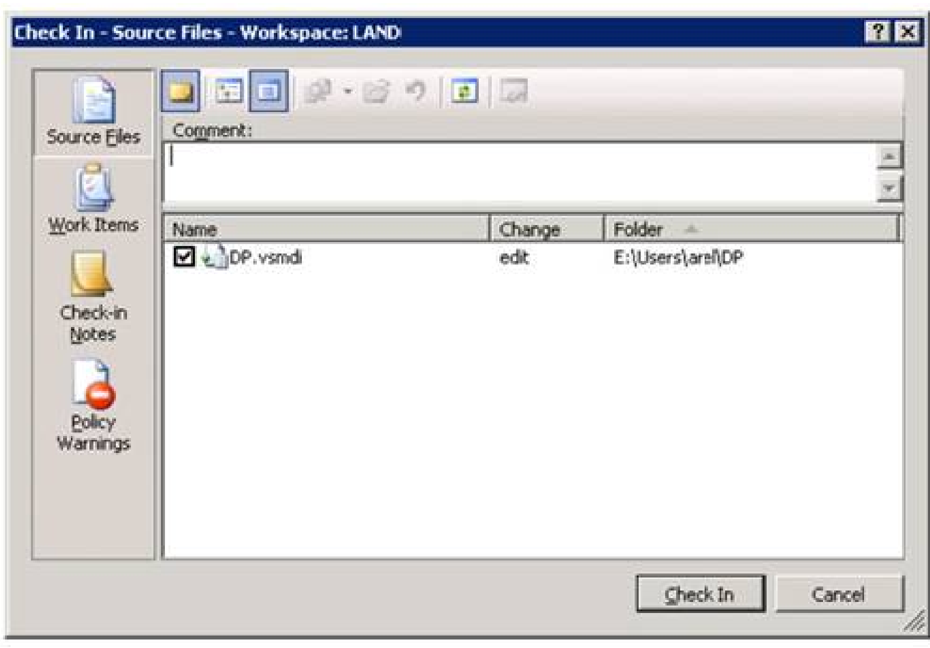

Рис. 14.1. Check-in діалог

На перший погляд діалог виглядає достатньо стандартно – *список* файлів і поля для внесення коментарів до файлів. Проте панель з додатковими закладками в лівій частині вікна і дозволяє отримати доступ до специфічної функціональності.

Найбільш затребуваною є *підтримка* в TFS можливості зв'язку змін, що вносяться, з елементами роботи, яку можна виконати на закладці *Work*Items, показаній на Рис. 14.2.

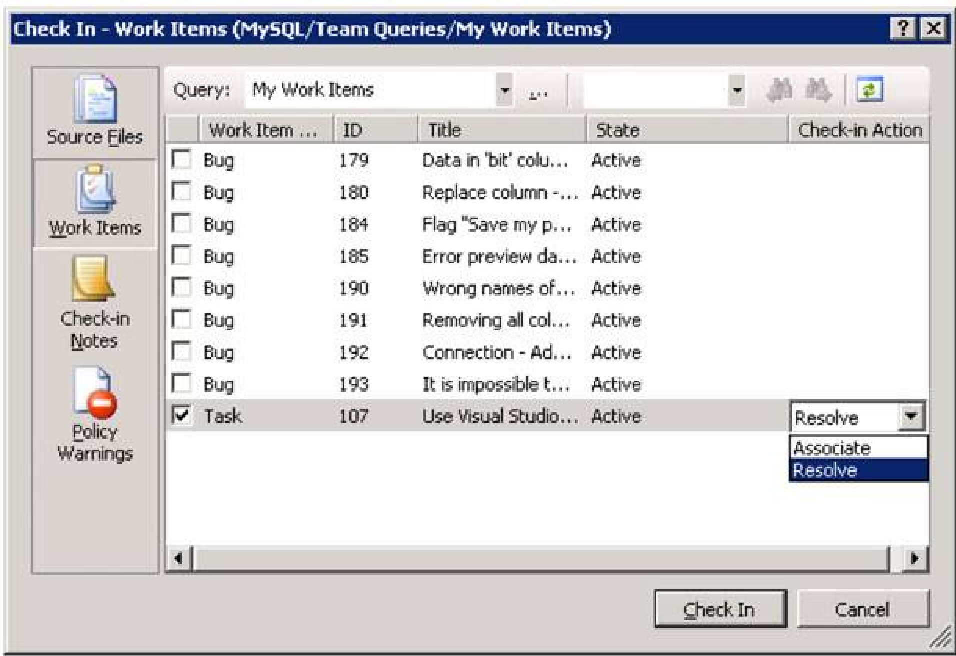

Рис. 14.2. Закладка WorkItems

Розробник, який вносить зміни до файлів з початковими текстами, може знайти відповідні цим змінам елементи роботи (адже він або виправляв яку-небудь помилку, або виконував завдання і тому подібне) за допомогою будь-якого з доступних запитів, а також текстового *пошуку*. *Запит* задається в секції *Query* – див. Рис. 14.2. Результат його виконання відображений в головному вікні діалогу на цьому рисунку. Для того, щоб зв'язати конкретний елемент роботи з цього запиту з даною зміною початкових файлів, треба обрати галочку в першому стовпці. На Рис. 14.2 вона обрана

для останнього елементу в списку – елементу роботи *Task*107. Буває так, що саме зміну початкових файлів "закриває" даний елемент *робіт*. Тоді в стовпці *Check-inAction* потрібно вибрати дію *Resolve* – це дія, як і інші, визначається в шаблоні процесу для всіх елементів роботи даного типу. Якщо ж дану зміну не "закриває" даний елемент роботи, то в цьому стовпці слід проставити дію *Associate*, і вона просто встановить *зв'язок* цієї зміни з даним елементом роботи.

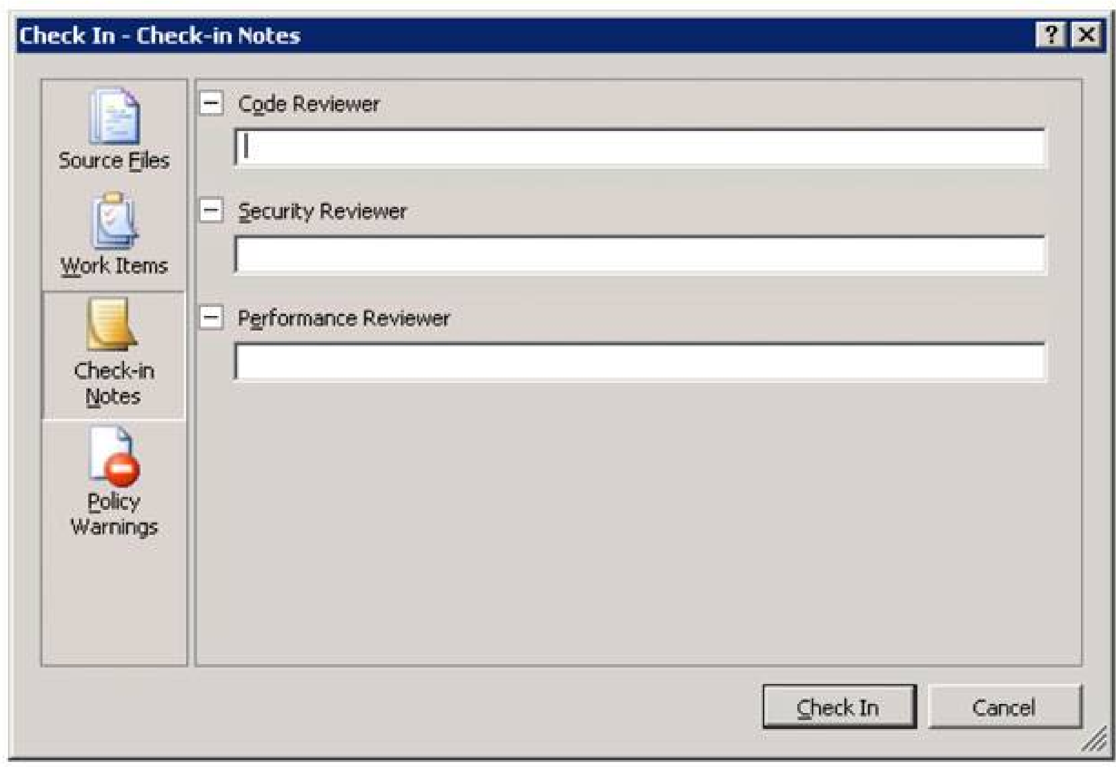

Рис. 14.3. Закладка Check-inNotes

У момент внесення змін, або після, до нього можуть бути приєднані додаткові коментарі людей, що проінспектували дану зміну (Рис. 14.3). *За замовчанням* TFS припускає три види інспекцій – інспекцію коду, інспекцію безпеки і інспекцію *продуктивності*. Інспекції є ефективним способом підвищення якості коду і припускають вивчення написаного коду іншою людиною. Нажаль, реалізація підтримки інспекцій в цьому вигляді не дає

можливість вказати, хто саме проводив інспекцію, що часто є важливою інформацією.

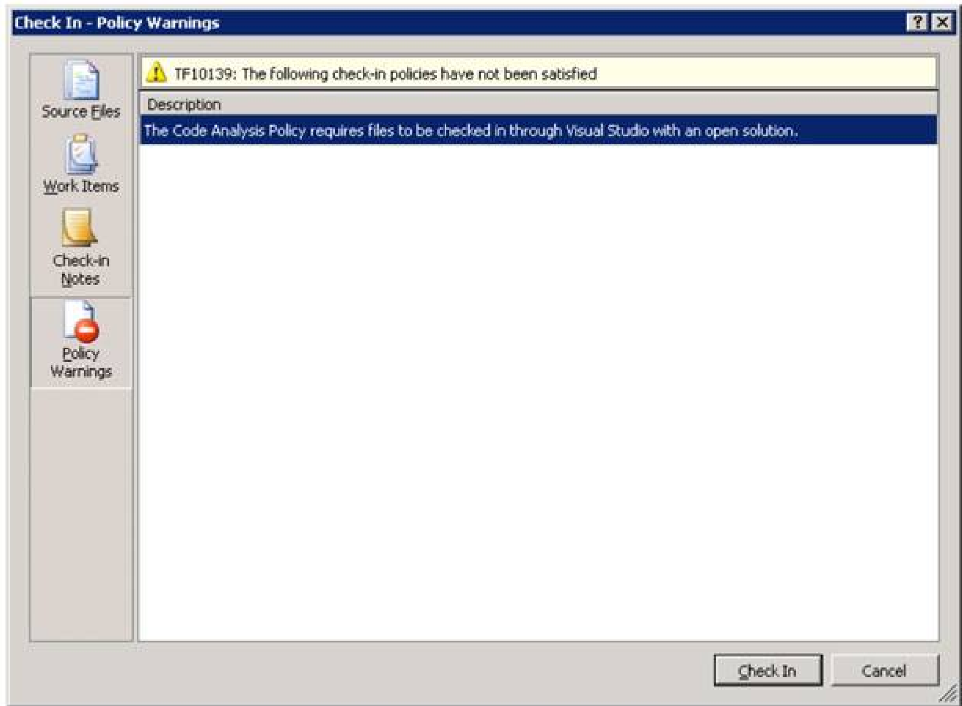

Рис. 14.4. Закладка PolicyWarnings

Цікавим нововведення для системи *контролю версій* TFS є гнучка система завдання правил внесення змін (*check-inpolicies*), про яку буде докладніше розказано пізніше. На закладці *PolicyWarnings* (Рис. 14.4) розробникові надається *перелік правил*, з якими увійшла у *конфлікт* його зміна (або процедура внесення змін).

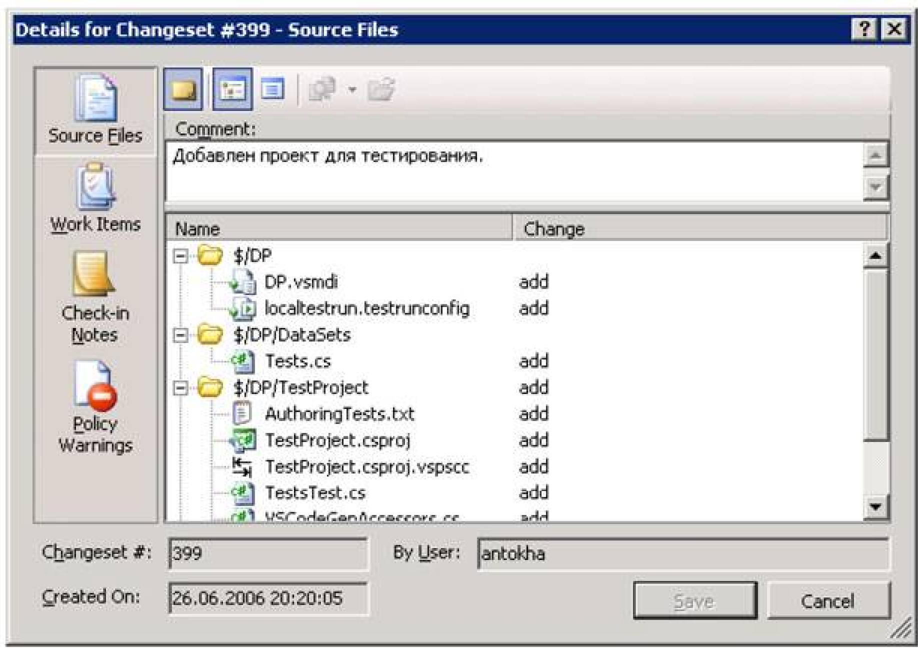

Рис. 14.5. Властивості набору змін

Більшість властивостей пакету змін надалі можна змінити у вікні редагування пакету змін (Рис. 14.5). Єдина властивість, яку не можна змінити з цього вікна – асоціації з елементами роботи. Для установки асоціацій елементу роботи і пакету змін необхідно звернутися до редактора елементу роботи.

Правила внесення змін. Однією з найбільш істотних переваг TFS як системи *контролю версій* є можливість завдання **правил внесення змін**. Ці правила застосовуються безпосередньо перед внесенням змін на комп'ютері розробника і в тому випадку, якщо правила не виконуються, розробникові відмовляється у внесенні змін.

Правила задаються за допомогою спеціального вигляду *.NET* збірок, що реалізовують певні інтерфейси. Декілька правил поставляється разом з самим TFS, величезна кількість правил реалізована співтовариством розробників і знаходиться у відкритому доступі. Якщо ж знайти ідеально від-

повідне правило так і не вдалося, в *Інтернеті* можна знайти величезну *кількість інформації про* написання власних правил.

У стандартне постачання TFS входять такі правила:

- *WorkItems* – припускає, що кожен пакет змін окрім файлів повинен мати асоціацію з елементом роботи;
- *Builds* – перевіряє, що перед внесенням змін розробник переконався в можливості зібрати проект;
- *Testing* – перевіряє, що перед внесенням змін розробником були виконані тести. На жаль, правило не може визначити, які саме тести треба запускати для перевірки даної зміни, тому дане правило не завжди ефективне;
- *CodeAnalysis* – виконує статичний аналіз коду перед внесенням змін.

У пакеті *TeamFoundationPowerTools* є такі додаткові правила:

- Правило *ForbiddenPatterns* дозволяє *заборонити* додавання файлів з певними шаблонами в іменах, наприклад, Form1.cs.
- Правило *CustomPath* дозволяє збільшити *гранулярність* застосування правил в проекті за допомогою конфігурування правил лише для певних частин структури папок системи *контролю версій*.
- Правило *ChangesetComments* перевіряє, чи супроводжуються зміни коментарями.
- Правило *WorkItemQuery* перевіряє, що всі елементи роботи, які повертаються в результаті певного запиту, є асоційованими з файлами (більш просунутий варіант правила *Work*Items).

Серед правил, які можна знайти у відкритому доступі, слід виділити:

- Правило *Code*CommentChecking перевіряє код на наявність коментарів перед внесенням змін (працює для коду на C#/vb.NET).
- Правило *CodeReviewWorkflow* перевіряє, чи є асоційованим кожен пакет змін з елементом роботи, який є завданням на інспекцію коду в стані "Виконано". Це правило дозволяє проводити процес інспекції змін, які вносяться, у більш формальному вигляді.
- Правило *Merge/branch*Only засвідчує, що всі зміни відбуваються в результаті *об'єднання* з іншою гілкою або відгалуження. Має особливий інтерес з погляду процесу *конфігураційного управління*. Дозволяє, наприклад, заборонити вносити зміни безпосередньо в одну з гілок (наприклад, гілка *реліза*). Вносити зміни до гілки, яка є захищеною таким правилом, можна лише через інтеграцію змін за допомогою інших гілок.

Вибрати *список* правил, які вживаються для командного проекту, можна за допомогою вікна налагоджування системи *контролю версій* (Рис. 14.6).

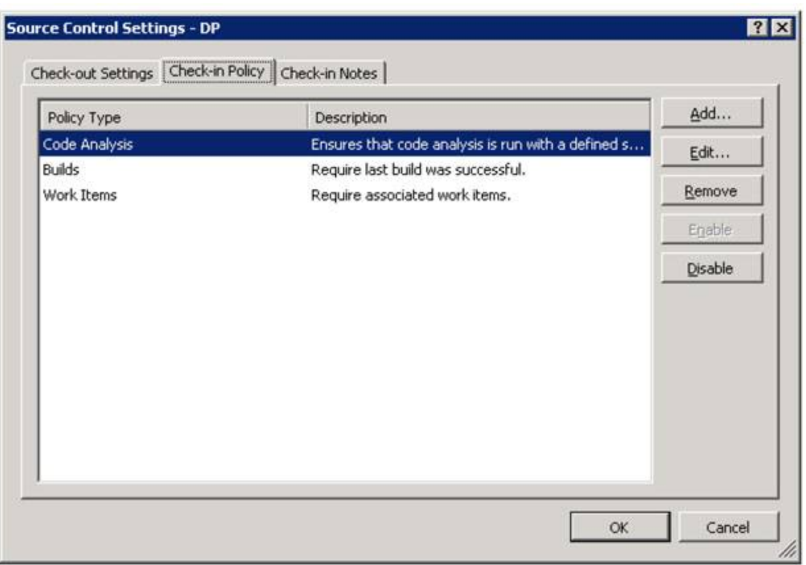

Рис. 14.6. Редагування правил внесення змін

Слід відмітити, що достатньо часто під час розробки трапляються ситуації, коли необхідно терміново ввести зміну і на задоволення всіх правил немає часу, або конкретне правило не може бути виконане *з* об'єктивних причин. В цьому випадку розробник має право скасувати правила для свого пакету змін і написати при цьому коментар з поясненням причин скасування (Рис. 14.7).

Управління гілками. Для підтримки *конфігураційного управління* в системі *контролю версій* TFS реалізовано дві команди: створення гілки (**Branch**) і *інтеграція* гілок (**Merge**). Ці команди є доступними на файлах і теках в системі *контролю версій*. У разі вибору команди створення гілки відкривається діалог (Рис. 14.8), що дозволяє вказати *шлях*, куди слід скопіювати (відгалузити) обрані файли. Після виконання цієї команди в системі *контролю версій за* вказаним шляхом буде створена повна копія обраних файлів.

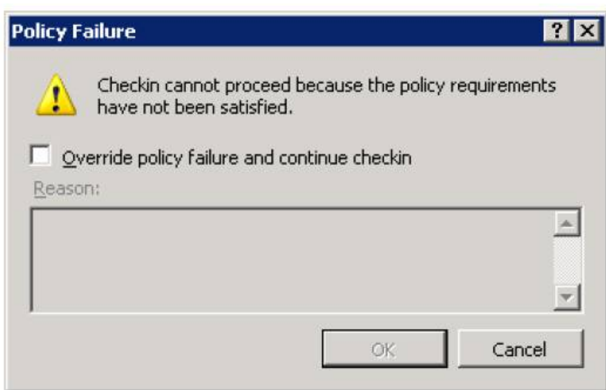

Рис. 14.7. Діалог PolicyFailure

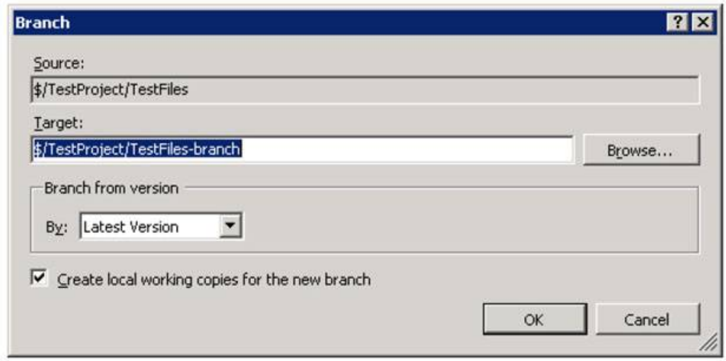

Рис. 14.8. Створення нової гілки

Слід відмітити, що після створення гілки вона не потрапляє на *сервер* автоматично. Для того, щоб *гілка* потрапила на *сервер* і стала доступною для всіх, необхідно виконати операцію внесення змін (Рис. 14.9).

Набагато складнішою, як правило, є операція перенесення змін з гілки в *гілку*. Для виконання цієї *операції* (*команда* **Merge**) використовується спеціальний майстер, який дозволяє розробникові задати необхідні параметри злиття за декілька кроків.

На першому кроці (Рис. 14.10) розробник задає звідки (**sourcebranch**) і куди (**targetbranch**), а також з якої зони він хоче перенести зміни (все аж до певної версії, або лише обрані пакети змін).

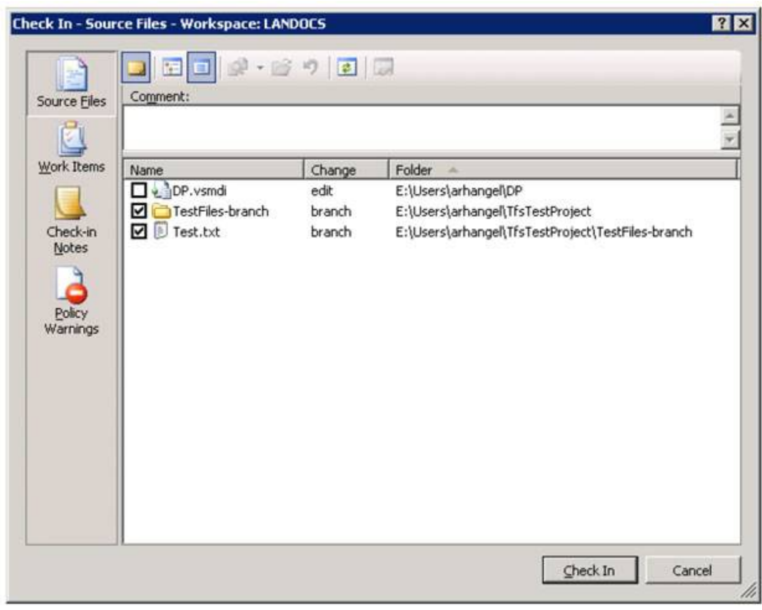

Рис. 14.9. Внесення змін відгалуження

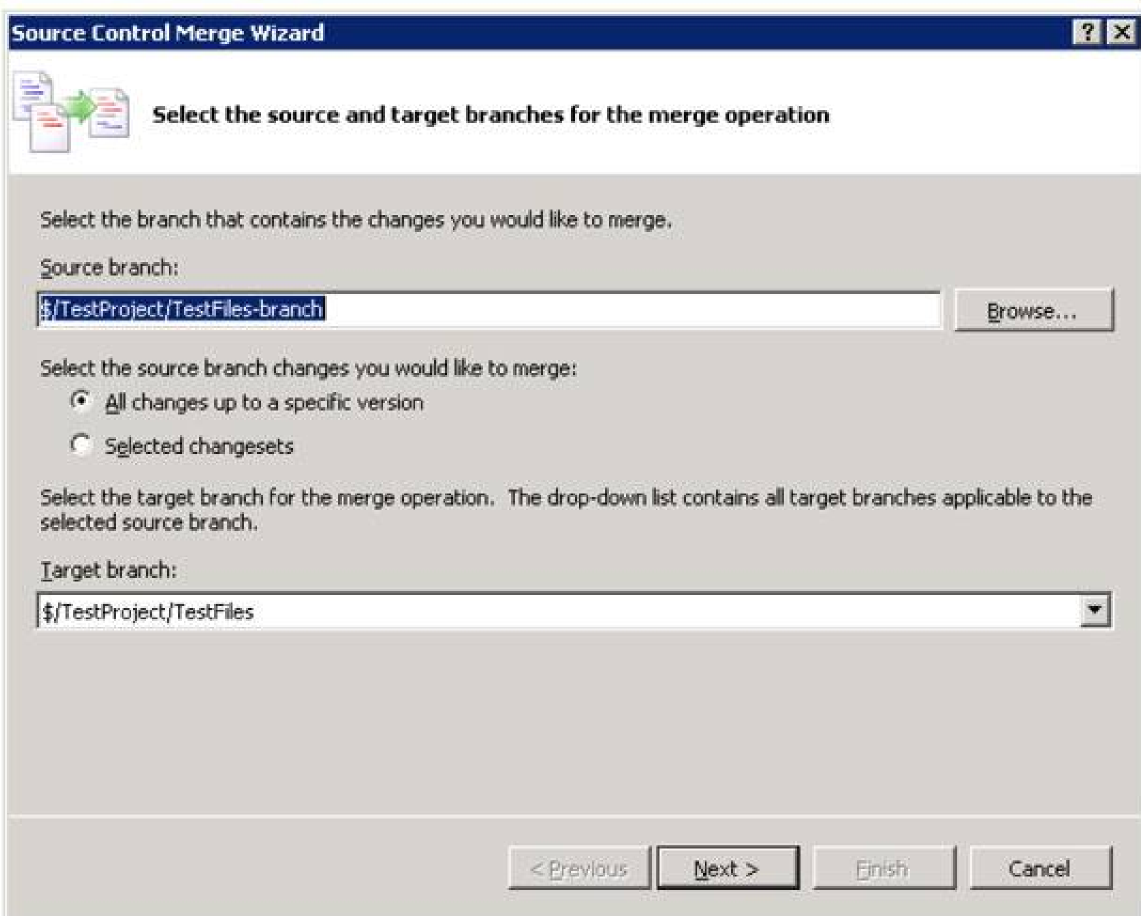

Рис. 14.10. Зона інтеграції

У випадку, якщо розробник вибрав перенесення всіх змін, йому пропонується вибрати версію аж до початкової гілки, до якої потрібно перенести зміни (Рис. 14.11). Розробник може вибрати цілковите перенесення, перенесення до певної дати, всі передуючі певному пакету зміни, або інтеграцію до тих версій, які знаходяться в поточному локальному робочому *просторі*.

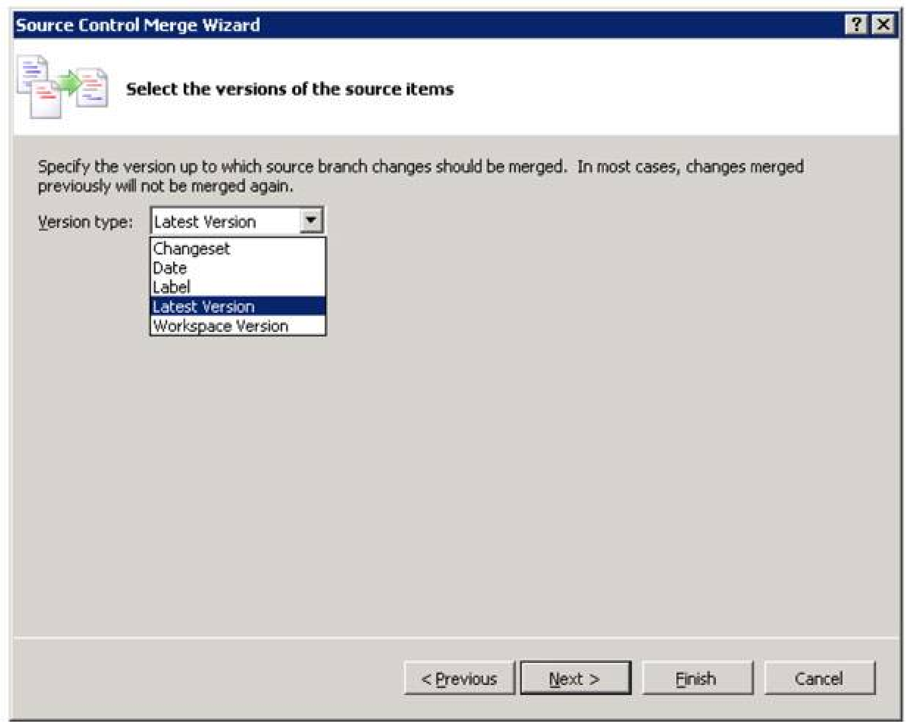

Рис. 14.11. Версія для інтеграції

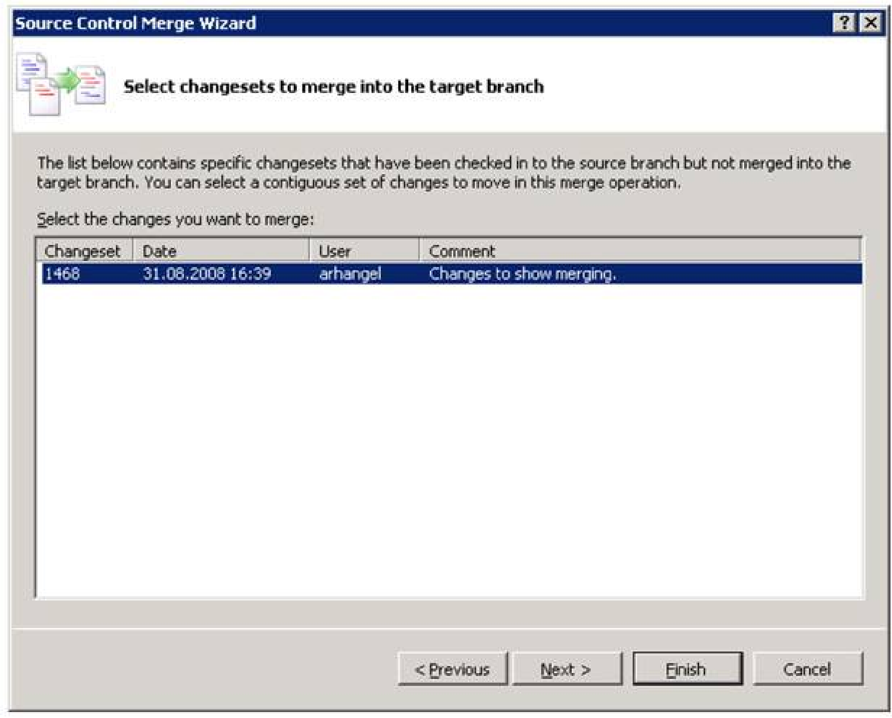

Рис. 14.12. Вибір пакетів для інтеграції

У випадку, якщо розробник вибрав інтеграцію *за* пакетами змін, йому надається можливість обирати серед ще не інтегрованих пакетів (Рис. 14.12), і він може вибрати один або декілька з них.

Так само, як і підчас створення нових гілок, у разі виконання інтеграції необхідно в явному вигляді ввести пакет змін, застосувавши до нього всі правила налагодження і виконавши асоціювання з відповідним елементом роботи.

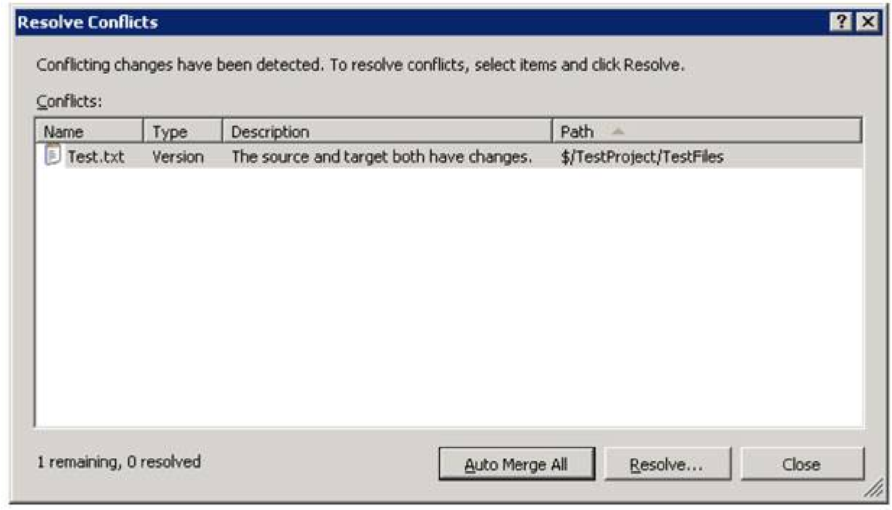

Рис. 14.13. Список конфліктів

Достатньо часто підчас інтеграції може виникнути ситуація конфлікту змін, коли інтегрований *файл* був змінений одночасно в обох гілках. В цьому випадку *система контролю версій* відкриває вікно зі списком виявлених конфліктів (Рис. 14.13) і пропонує обрати спосіб вирішення конфлікту.

Рис. 14.14. Вирішення конфлікту

Розробник може вибрати автоматичний спосіб дозволу, який спрацює лише для тих файлів, в яких були змінені різні частини. У випадку, якщо автоматичний дозвіл неможливий, система відкриє діалог дозволу власноруч (Рис. 14.14).

Для вирішення конкретного конфлікту розробник може вибрати декілька способів: автоматично об'єднати зміни (якщо можливо), прийняти зміни з початкової гілки, зберегти зміни цільової гілки або власноруч вирішити всі внутрішні конфлікти за допомогою спеціального інструмента (Рис. 14.15).

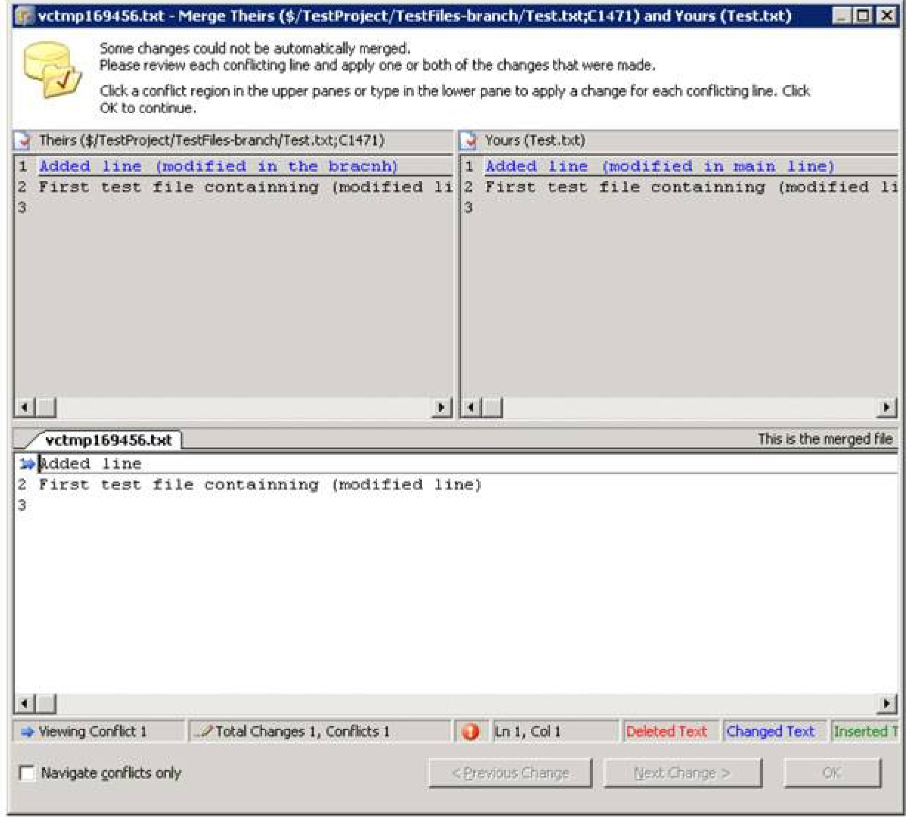

Рис. 14.15. Вирішення внутрішніх конфліктів

Слід зауважити, що даний інструмент має достатньо обмежену функціональність, тому деякі розробники віддають перевагу використанню аналогічних інструментів інших виробників.

Збереження без внесення. Корисною можливістю системи *контролю версій* є можливість зберегти зміни в спеціальному сховищі на сервері, не вносячи їх безпосередньо до системи *контролю версій*. Для тимчасово збереженої коду не перевіряються правила внесення змін (якщо про це не попросити явно) і він не є доступним іншим розробником (якщо вони про це явно не попросять).

Ця функціональність дозволяє захистити важливий пакет зміни від втрати, якщо розробник вимушений тимчасово припинити роботу (наприклад, щоб відпочити). Знаходячись на сервері ці зміни підпадають під політику створення резервних копій *бази даних* і, отже, *вірогідність* втрати цих змін стає мінімальною.

Ще одним способом застосування даної можливості є перенесення змін між різними машинами, в тому випадку, якщо розробник використовує декілька машин (наприклад, робочу та/або домашню). Збережені на сервері зміни розробник зможе отримати на іншій машині в тому ж вигляді, щоб продовжити роботу, де б він не знаходився.

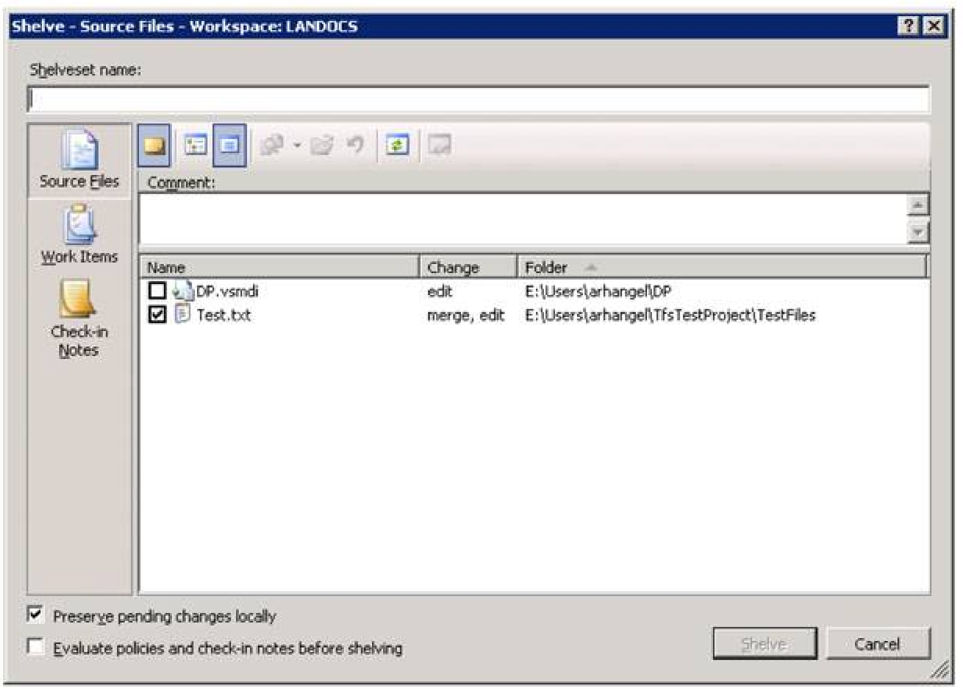

Рис. 14.16. Збереження без внесення

Для збереження змін служить *команда* **Shelve**, що відкриває діалог, який представлений на Рис. 14.16. Цей діалог є аналогічним діалогу внесення змін за винятком поля для введення імені збереженого пакету у верхній частині і двох наступних додаткових опцій в нижній частині.

- Зберегти локальні зміни. Якщо ця опція включена, то зміни залишаться локально в *робочому просторі користувача* (рекомендується у разі тимчасової зупинки роботи). Якщо ж ця опція не включена, то локальні зміни відкатуються і залишаються лише у вигляді збереженого на сервер пакету (рекомендується у випадку зміни робочої машини).
- Застосувати правила перед збереженням – дозволяє проаналізувати пакет із застосуванням всіх тих правил, які діють підчас внесення.

Для того, щоб відновити збережені зміни необхідно скористатися командою **Unshelve**, що є доступною для файлів і тек, а також в глобальному контексті (з вікна із списком невнесених змін – див. Рис. 14.17). Ця *команда* відкриває діалог відновлення змін (Рис. 14.18), який дозволяє вибрати один із збережених пакетів для відновлення. З цього ж діалогу пакети можна видалити, якщо вони втратять актуальність.

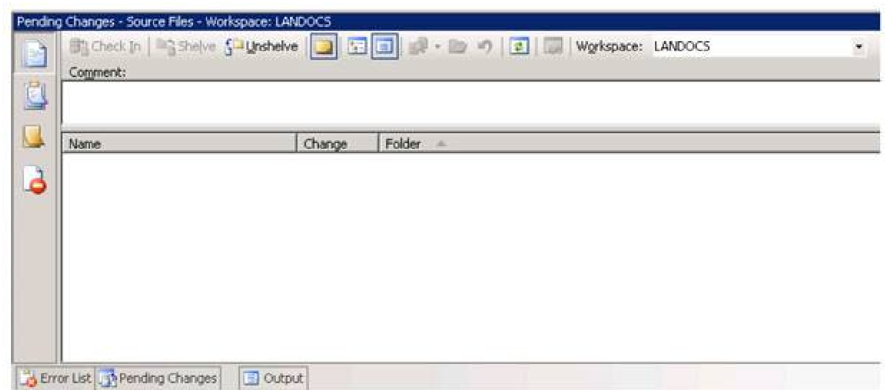

Рис. 14.17. Список поточних змін

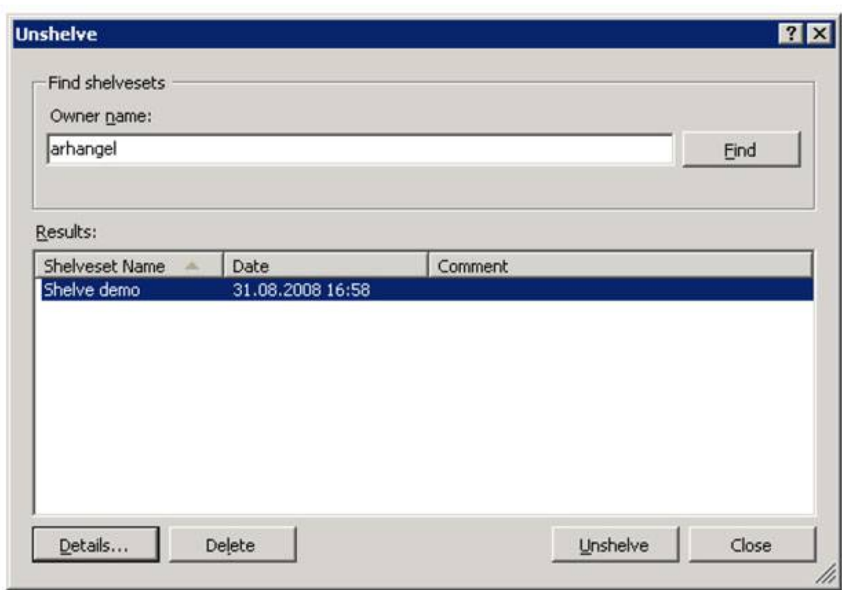

Рис. 14.18. Діалогове вікно відновлення змін

14.3. Автоматичні збірки

Загальні відомості. Однією з істотних переваг TFS *в порівнянні з* іншими системами управління *збірками* є простота, з якою він дозволяє створювати і налагоджувати процес автоматичної збірки. Не дивлячись на те, що в основі збірок TFS лежить давно і широко відома технологія Msbuild, саме TFS дозволяє вивести її на принципово новий рівень завдяки наведеним нижче особливостям.

- TFS поставляється разом з набором MsBuild-задач, що дозволяють значно спростити і прискорити налагоджування процесу збірки. Серед найбільш важливих завдань слід зазначити такі:
  - збірка проекту (при цьому початкові тексти програм автоматично беруться з системи *контролю версій*);

- автоматичний запуск тестових пакетів (як створених власноруч, так і ідентифікованих автоматично);
- застосування статичного аналізу коду;
- розміщення результатів збірки в мережевій теці;
- автоматична підтримка унікального ідентифікатора збірки і його реєстрація;
- виявлення приєднаних елементів роботи і так далі.

- TFS надає серверне середовище, що дозволяє запускати процес збірки в "чистому" оточенні, у відмінності від звичайних MsBuild-зборок, де організація відповідної інфраструктури вимагає значних зусиль.
- Можливість автоматичного запуску процесу збірки як в режимі безперервної інтеграції, так і за розкладом.
- Візуальне супроводження перебігу процесу збірки, результатів, а також історії раніше відпрацьованих зборок.

Значною перевагою TFS є те, що опис простого процесу збірки створюється менше ніж за хвилину, а опис складніших створюється не складніше, ніж за допомогою стандартного Msbuild.

Створення опису збірки. Для створення опису нової збірки проекту необхідно вибрати команду **NewBuildDefinition** у відповідному вузлу *Team*Explorer, і після цього відкриється вікно з декількома закладками (Рис. 14.19).

На першій закладці (**General**) знаходиться загальна *інформація* – назва і опис призначення цього сценарію збірки.

На другій закладці (**Workspace**) – див. Рис. 14.20 – описується те, які початкові тексти необхідно узяти з системи *контролю версій* для збірки, а

також те, як цей код розмістити на машині, де буде відбуватися *збірка*. Крім того, ці налагоджування впливають на поведінку у разі безперервної інтеграції – внесення змін саме до обраних областей в засобі *контролю версій буде* сигналом для запуску даного процесу збірки.

Найцікавішою є третя закладка **Project File** (Рис. 14.21), де розробник може вибрати один з існуючих MsBuild-сценаріїв збірки або створити новий.

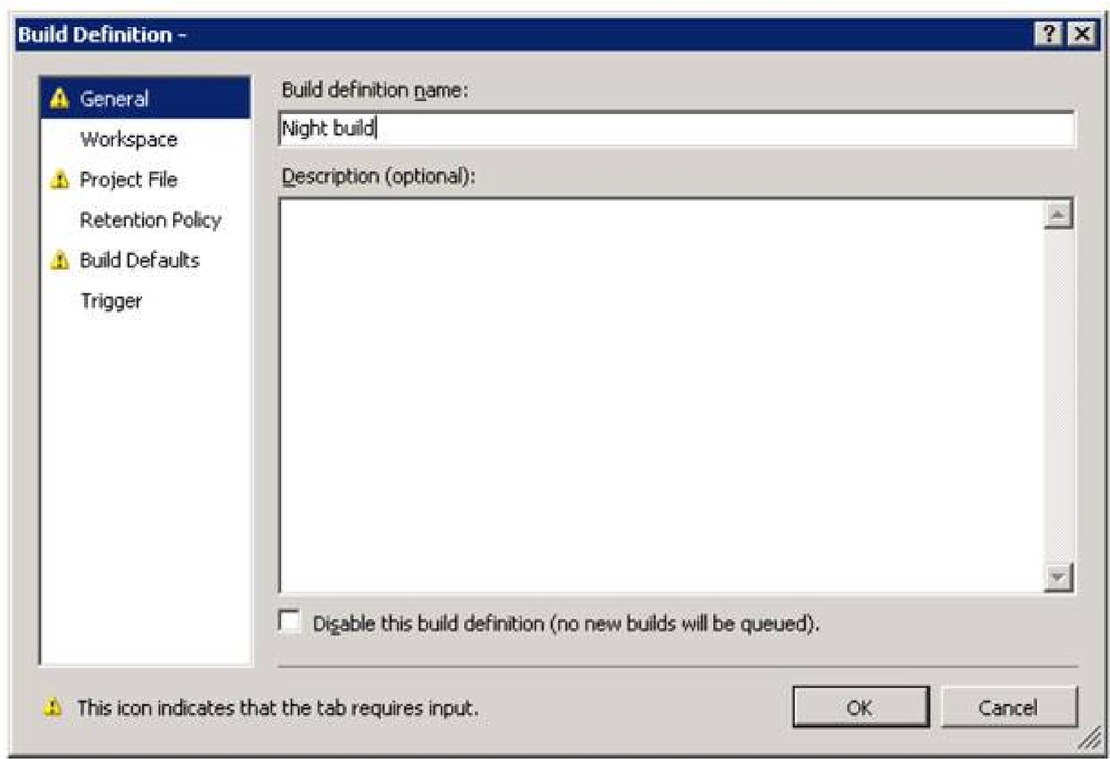

Рис. 14.19. Загальні налагоджування опису збірки

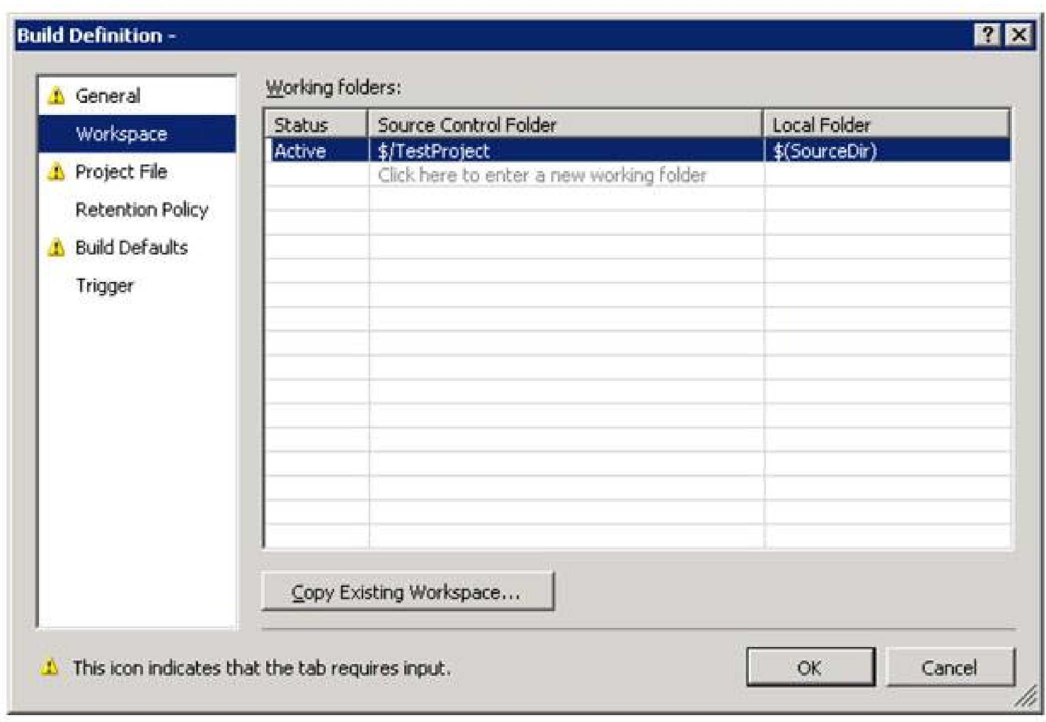

Рис. 14.20. Налагоджування робочого простору

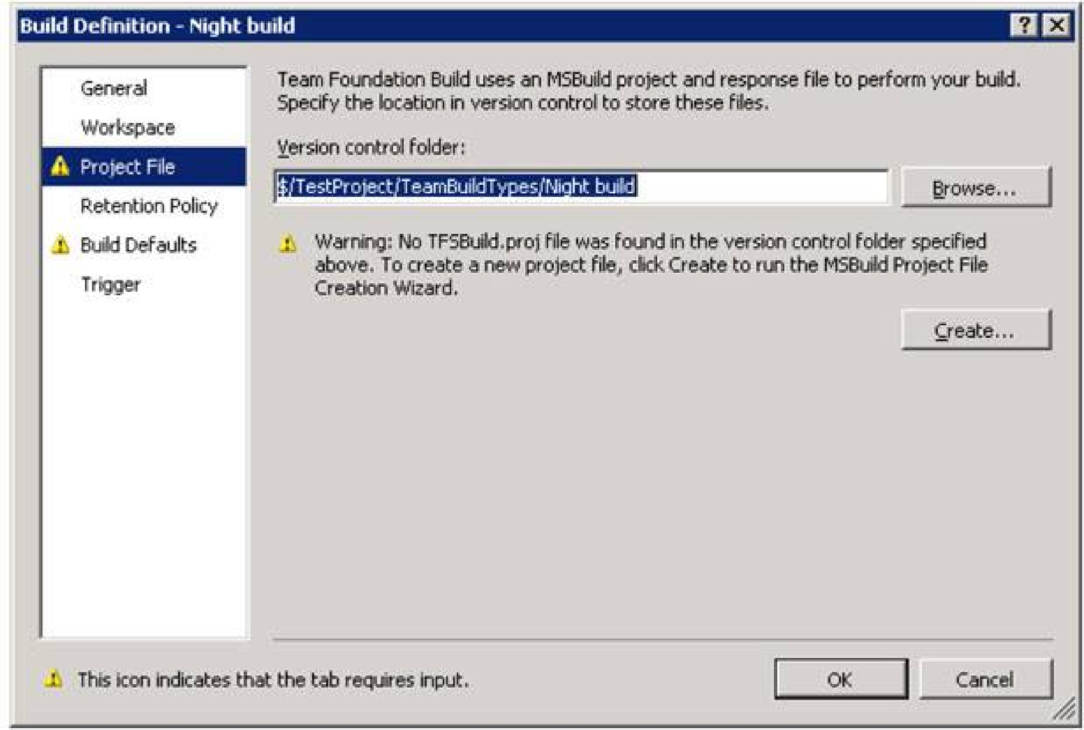

Рис. 14.21. Вибір або створення MsBuild-проекту

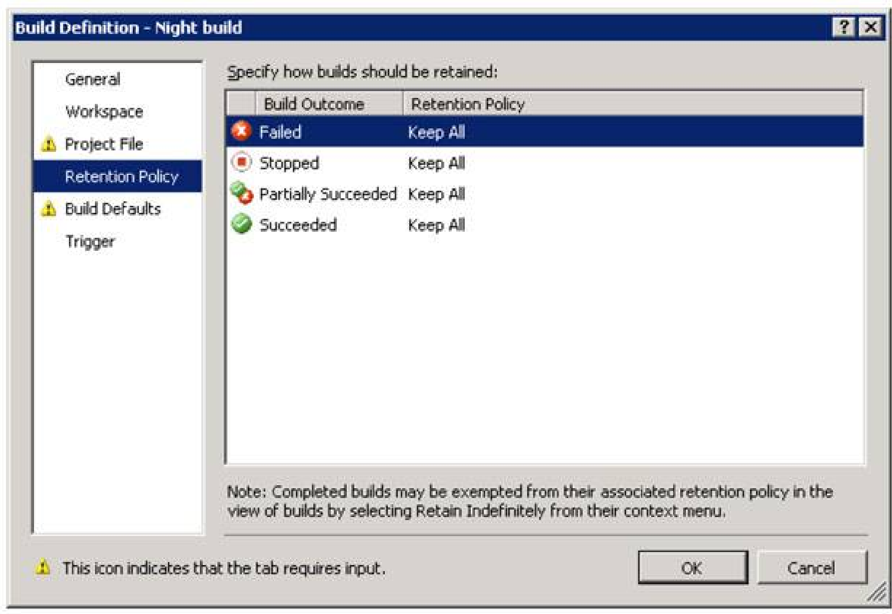

Рис. 14.22. Правила збереження

У TFS у зв'язку з реалізацією функціональності *для* безперервної інтеграції виникла проблема великої кількості описів збірок і результатів, що були збережені в системі і на диску. Для усунення цієї проблеми був реалізований механізм автоматичного очищення, що отримав назву *Retention Policy* (див. Рис. 14.22) і дозволив задавати, скільки останніх результатів треба зберігати. При цьому для кожного з типів результату (невдалий, зупинений, частково успішний, успішний) можна задати свою кількість.

Корисною функцією є те, що політику знищення результатів можна відключити для конкретної збірки за допомогою опції *Retain*Indefinitely. Результати, які були відміченими цією опцією, зберігатимуться в системі завжди (або доки *опція* не буде знята).

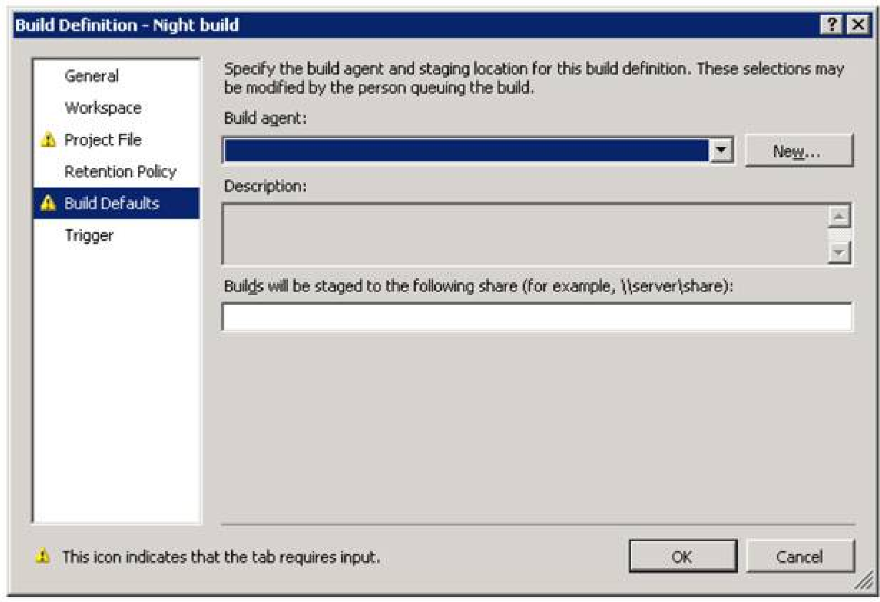

Рис. 14.23. Обирання агенту

На закладці **BuildDefaults** (див. Рис. 14.23) ми задаємо властивості оточення, яке буде використано для автоматичного запуску процесу збірки. Головним тут є *складальний агент* – процес, в межах якого виконуватиметься процес збірки.

Крім того, саме на цій закладці можна задати мережеву теку, в якій зберігатимуться результати процесу збірки. Під час обирання теки слід переконатися, що складальний *агент* має до неї *доступ* з правом на *запис*.

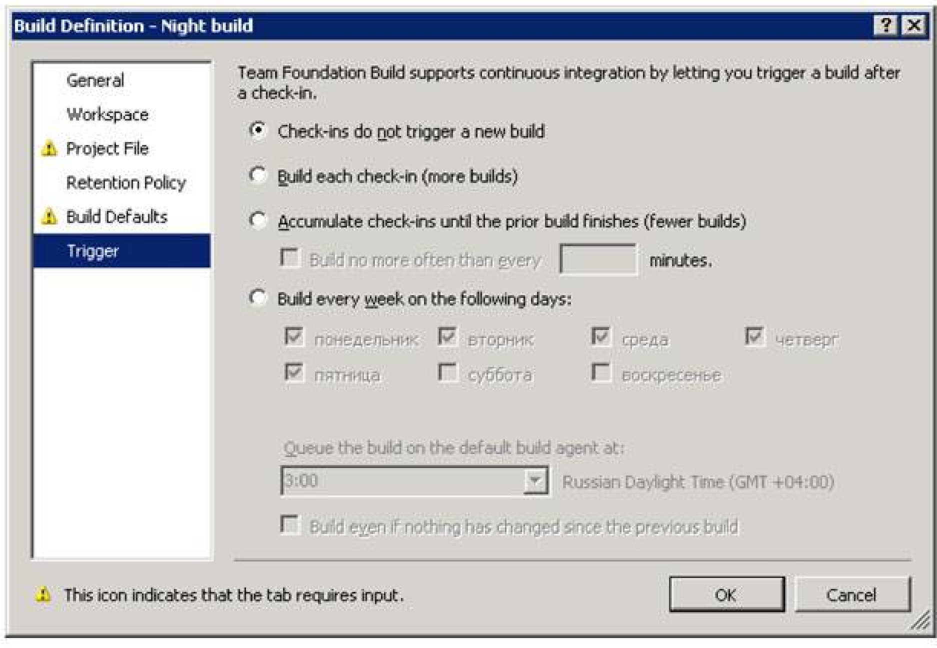

Рис. 14.24. Створення тригера

На завершальному етапі налагоджування процесу збірки (див. Рис. 14.24) ми задаємо умову, у разі виконання якої процес збірки, визначений даним описом, повинен виконуватися. Ця умова може бути однією з таких:

- не запускати процес збірки автоматично (тобто лише "власноруч");
- запускати після кожного внесення змін;
- акумулювати зміни, доки не закінчиться попередня збірка і не закінчиться певний інтервал часу;
- запускати процес збірки лише за розкладом, навіть в тому випадку, якщо нічого не мінялося.

Створення проекту Msbuild. Проект Msbuild, не дивлячись ні на що, все-таки складає основу системи автоматичних збірок TFS. Проте фахівцями Microsoft витрачено немало зусиль на те, щоб ми могли забути про необхідність підтримувати великі і складні XML-файли з описом збірок. У

TFS для створення проектів Msbuild реалізований достатньо зручний майстер (Рис. 14.25 – 14.27).

На першому кроці (Рис. 14.25) ми обираємо в системі *контролю версій* ті рішення, які повинні збиратися в даній збірці.

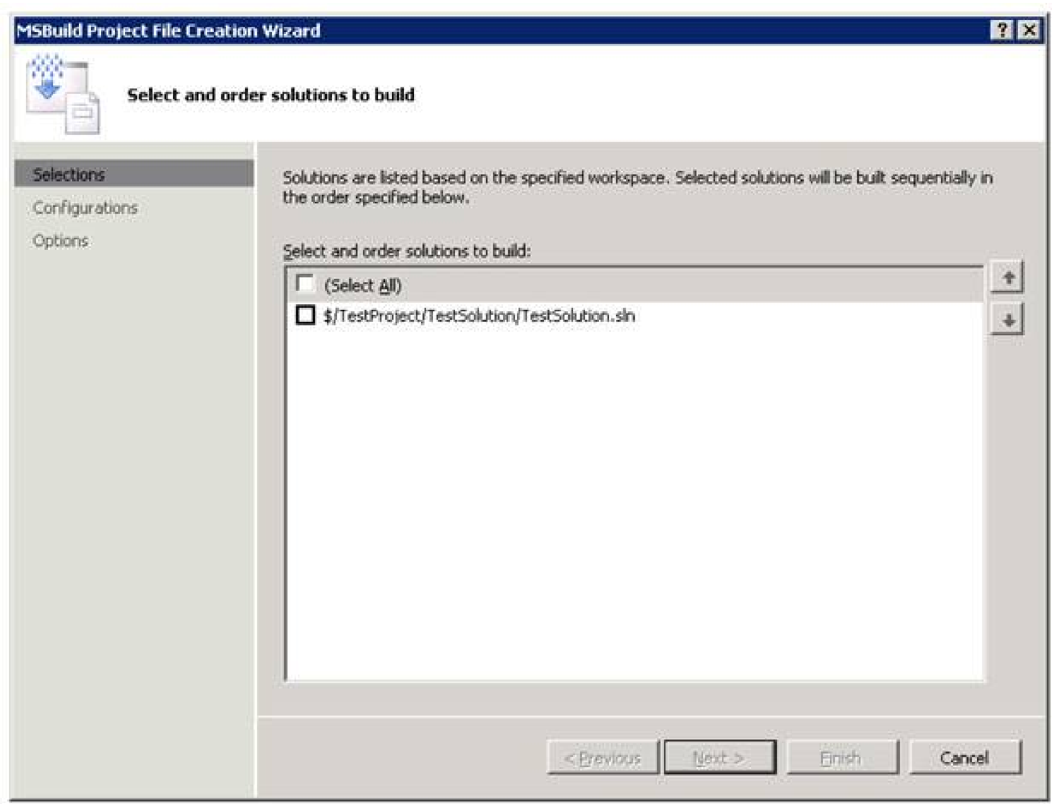

Рис. 14.25. Обирання рішень

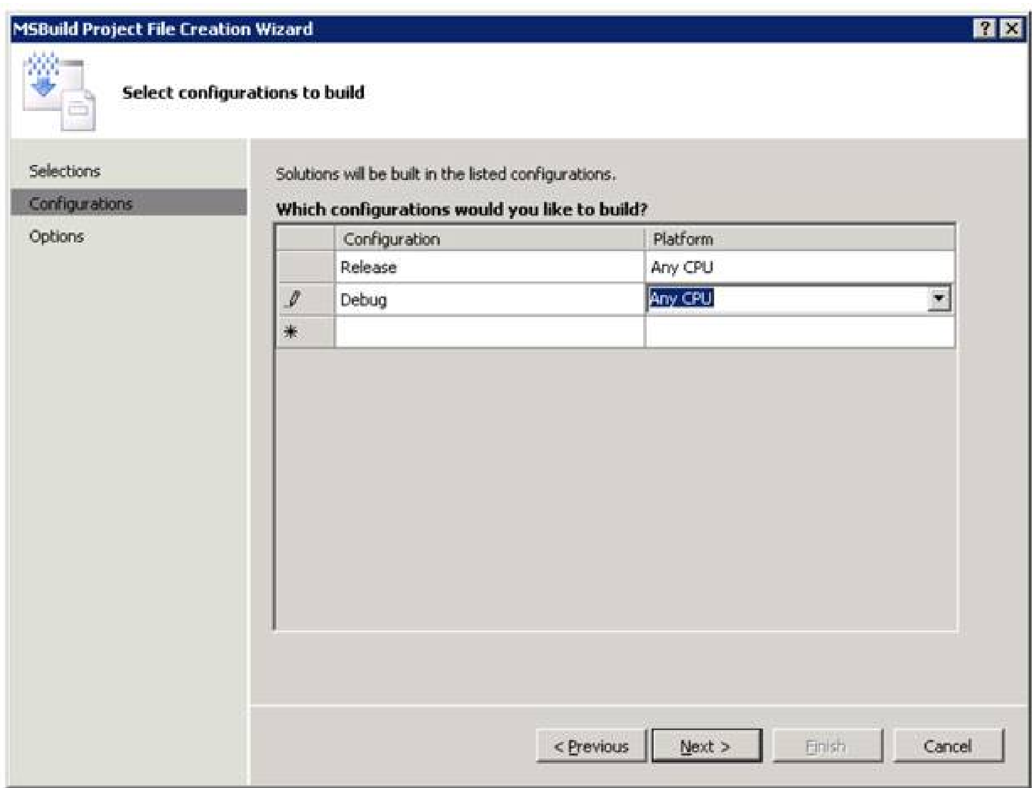

Рис. 14.26. Обирання конфігурацій

На другому кроці (див. Рис. 14.26) ми обираємо ті конфігурації в цих рішеннях, які потрібно збирати.

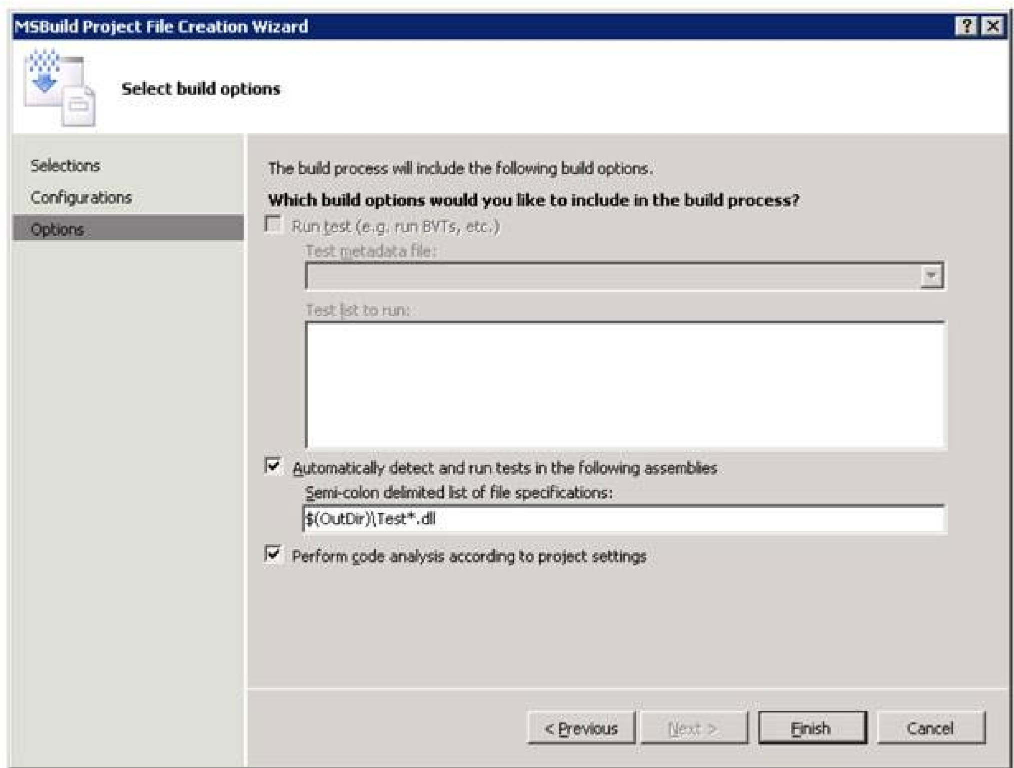

Рис. 14.27. Обирання тестів і правил аналізу

І, нарешті, на третьому кроці (Рис. 14.27) ми обираємо ті тести, які ми хочемо запустити і визначаємо, чи хочемо ми проводити статичний *аналіз* коду. На відміну від більш ранніх версій TFS, де для вибору тестів можна було використовувати лише заздалегідь підготовлені тестові пакети, в останніх версіях TFS з'явилася така можливість, як автоматичне підключення пакетів тестів *за міткою* імені (зазвичай збірки починаються або закінчуються на Test). Ця можливість дозволяє без додаткових витрат включити виконання модульних тестів в автоматичну збірку.

Створення складального агенту. Поява складальних агентів в TFS 2008 дозволила значно розширити можливості конфігурації автоматичного виконання процесу збірки. Складальний *агент* – це процес, що запущений на деякій виділеній машині, в рамках якого і відбувається автоматична *збірка*. Насправді, складальні агенти виконуються процесом Tfsbuild, що запущений у вигляді сервера або консольного додатка.

Одна машина може розміщувати у собі декілька процесів Tfsbuild, кожен з яких є доступним як Web-сервіс на певному порту. При цьому кожен процес Tfsbuild може містити декілька складальних агентів, які відрізняються іменами і робочими теками, де вони виконують збірку.

Не дивлячись на складність описаного процесу, налагоджування його достатньо просте і проводиться, в основному, за допомогою вікна властивостей агенту (Рис. 14.28).

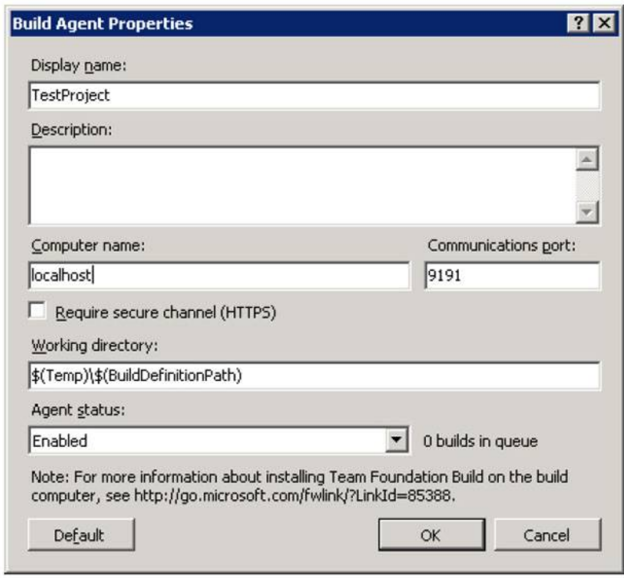

Рис. 14.28. Властивості складального агенту

Запуск процесу збірки і аналіз результатів. Отже, після того, як опис збірки створений і *інфраструктура* виконання налагоджена, ми можемо запустити процес збірки за допомогою команди **QueueNewBuild**, яка викликає появу вікна (Рис. 14.29).

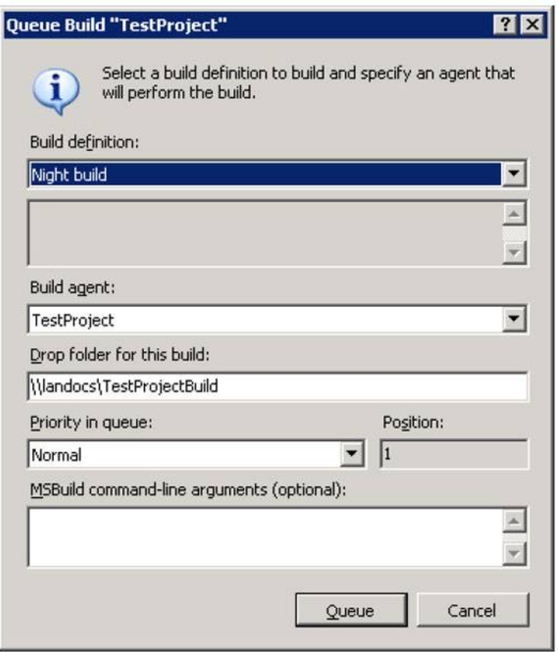

Рис. 14.29. Запуск нової збірки

Під час запуску нової збірки *користувач* може вибрати опис збірки, складальний агент (*за замовчанням* використовується *агент*, який був заданий в описі), теку для зберігання результатів (так само береться *за замовчанням* з опису), пріоритет і позицію в черзі (кожен складальний *агент* може виконувати не більше однієї збірки за раз), а також додаткові *параметри командного рядка* для Msbuild. Як правило, в наведеному вікні можна використовувати всі налагоджування *за замовчанням*, міняти які доводиться лише в особливих випадках.

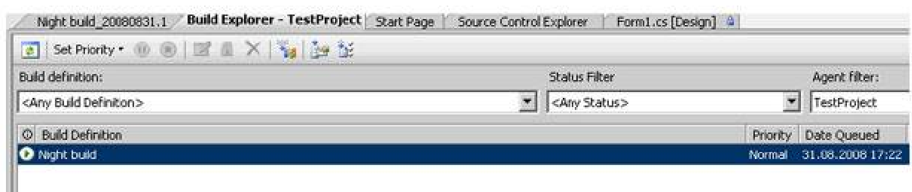

Рис. 14.30. Список описів збірок

Після того, як *збірка* потрапила в *чергу*, вона відображається в списку збірок (Рис. 14.30), звідки можна перейти до вікна з детальною інформацією про збірку (Рис. 14.31). У цьому вікні відображається перебіг збірки, або її результати, якщо вона вже завершена.

Приклад на Рис. 14.31 наочно демонструє засоби *Team*Explorer *щодо* візуалізації результатів. У відкритому вікні видно *список* тестів з помилками, а також є посилання на всі файли з логами, які можна відкрити одним клацанням в оточенні студії (за умови, що у вас є *доступ* до мережевої теки, куди вони були скопійовані). Крім того, в інформації про минулий процес збірки можна побачити, які зміни початкових текстів програм в неї потрапили, а також те, якими елементами роботи вони були обумовлені.

Управління процесом збірки. Всі продемонстровані нами засоби візуального опису збірок є доступними не лише під час створення такого опису, але і для будь-якого з існуючих описів збірок (*команда* **EditBuildDefinition**). Єдиним виключенням є *файл* Msbuild, який, після того як його одного разу згенерували за допомогою майстра, далі підтримується уручну.

Цей *файл* можна знайти в системі *контролю версій* (Рис. 14.32) в тій теці, яка була обрана під час його створення. Ця *тека містить* два файли:

- PROJ-файл – основний файл, що описує завдання Msbuild. Цей файл містить достатню кількість коментарів, що були згенеровані, дозволяють проводити нескладне налагоджування (додати або видалити рішення, включити або відключити статичний аналіз і так далі) достатньо легко, проте для складніших змін необхідне знайомство як з принципами роботи Msbuild, так і із специфікою MsBuild-задач, які використовуються в TFS.
- RSP-файл, який містить параметри командного рядка для передачі підчас запуску Msbuild.

- Для більшості простих проектів, зазвичай, вистачає налагоджувань, які є доступними через візуальні редактори, і змінювати ці файли доводиться не часто.

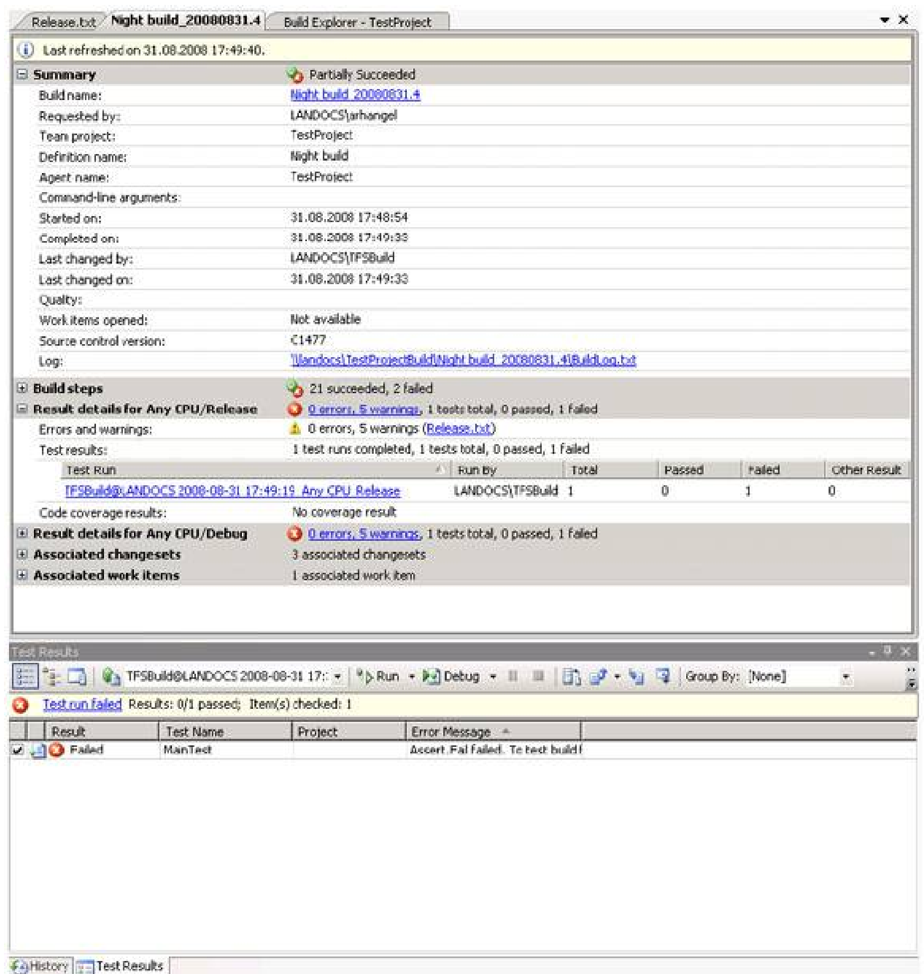

Рис. 14.31. Результати збірки

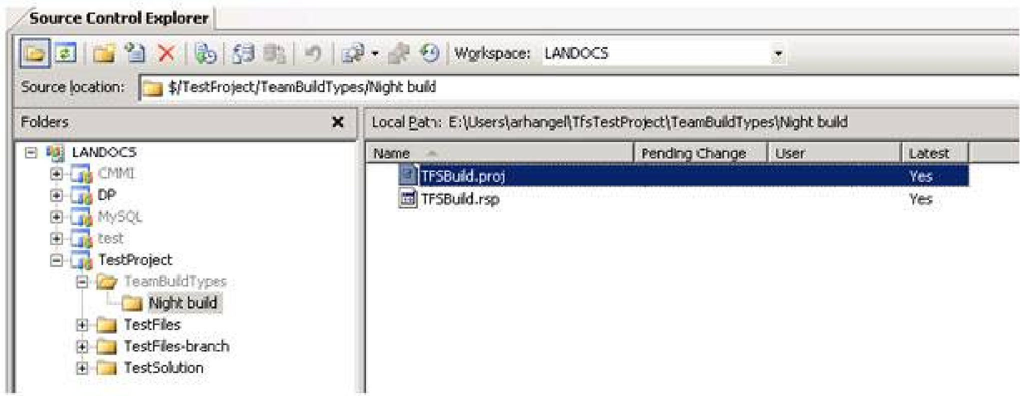

Рис. 14.32. Файл визначення проекту

Контрольні запитання

1. Які інструменти є у VSTS для підтримки конфігураційного управління?
2. Для чого використовується система контролю версій?
3. Для чого використовується система управління збірками?
4. Що дозволяє система?
5. Які переваги має робота з управління збірками у VSTS порівняно з іншими системами?
6. Які особливості функціонування системи контролю версій у VSTS?
7. Які особливості підсистеми відслідковування змін окремих файлів?
8. Які правила внесення змін у системі контролю версій у VSTS?
9. Які правила входять у стандартне постачання TFS?
10. Які правила входять у додаткові правила внесення змін у TFS?
11. Які правила можна виділити у відкритому доступі?
12. Як обійти ситуацію, коли необхідно терміново ввести зміну без задоволення всіх правил?
13. Які команди реалізовано в системі контролю версій TFS для підтримки конфігураційного управління?
14. Які особливості роботи команди створення гілки в системі контролю версій TFS?

15. Які особливості роботи команди перенесення змін з гілки в гілку в системі контролю версій TFS?
16. Як підчас інтеграції вирішується ситуація конфлікту змін, коли інтегрований файл був одночасно змінений в обох гілках незалежно?
17. Навіщо потрібне і як працює збереження без внесення?
18. В чому полягають особливості роботи системи управління збірками в автоматичному режимі в TFS?
19. Як відбувається створення опису збірки?
20. Як відбувається створення проекту Msbuild?
21. Яка процедура створення складального агенту?
22. Як відбувається запуск процесу збірки і аналіз результатів?
23. Як відбувається управління процесом збірки?

Глава 15. VSTS: Тестування

Система відслідковування помилок. Створення опису помилки. Зв'язок змін початкових текстів ПЗ і помилок. Система сповіщень. Модульні тести. Пакети тестів. Автоматичне тестування Web-додатків.

15.1. Загальні положення

З тестування слід виділити такі можливості VSTS:

- *інтегрована* система відслідковування помилок;
- *засоби розробки* модульних тестів;
- засоби організації тестових пакетів;
- *автоматичне тестування* Web- додатків (у тому числі і навантаження).

Крім перерахованих вище можливостей в процес тестування в VSTS до деякої міри залучена решта практично всіх систем – *система контролю версій* використовується для зберігання описів тестів, система управління збірок дозволяє автоматично виконувати тестові пакети, система *звітів* дозволяє стежити за зміною якості продукту, а *інтеграція* з офісними *додатками* дозволяє будувати плани *з* тестування і виправлення дефектів. Проте, кожна з вище перерахованих систем використовується лише в рамках своїх стандартних можливостей, тому детально в цьому розділі вони не розглядаються.

15.2. Система відслідковування помилок

Загальний опис. Система відслідковування помилок в VSTS реалізована на базі системи управління елементами *робіт*. Адже помилки (bugs) – особливий тип елементів *робіт*. *В порівнянні з* іншими системами відслі-

дковування помилок є *інтегрованою* системою на основі системи управління елементами роботи, що дає декілька важливих переваг.

- Можливість завдання зв'язку змін програмного коду з помилками, які вони призначені виправити, дозволяє легше підтримувати і розвивати систему, уникаючи при цьому значної регресії.
- Інтеграція з системою автоматичної збірки дозволяє легко відстежувати те, до якої збірки увійшло виправлення тієї або іншої помилки, не вимагаючи від розробників додаткових дій.
- Можливість легко будувати узагальнюючі звіти дозволяє легко відстежувати поточну якість продукту.
- Можливість інтеграції з офісними додатками і, зокрема, з продуктом Microsoft Project, дозволять простіше планувати і управляти процесом виправлення помилок, у той самий час система автоматичних сповіщень дозволяє зробити цей процес більш оперативним.

На Рис. 15.1 наведений опис *життєвого циклу* елементу роботи "помилка" (*Bug*) з шаблону процесу VSTS з назвою msf for Agile 4.2. У цього типу елементу роботи визначено три стани:

· Active – помилка потребує виправлення

· Resolve – помилка виправлена

· Close – помилка перевірена і виправлення прийняте.

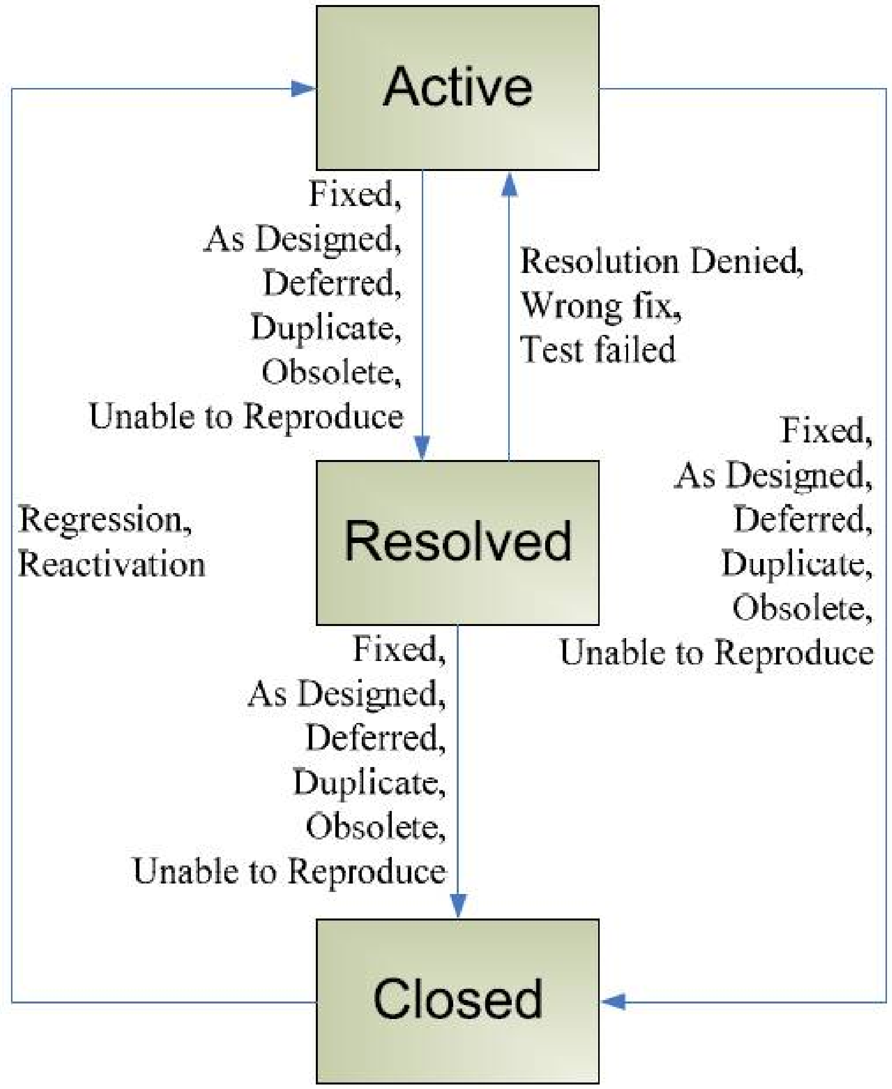

Рис. 15.1. **Життєвий цикл**.

На стрілках-переходах вказані причини, через які помилка перейшла в даний стан. Опишемо деякі, найбільш вживані переходи, що зустрічаються.

У стан *Active* помилка потрапляє, по-перше, після свого створення – тобто *тестувальник* знайшов нову помилку і створив відповідний елемент роботи (ця початкова дія на рисунку не показана). Далі, в цей стан помилка може потрапити із стан Resolved, після того, як *тестувальник* перевірив виправлення програміста і виявив, що тести все одно "падають" (*причина*

Test failed). Якщо виправлення помилки було виконано некоректно (поведінка системи не відповідає бажаному), то помилка переходить в стан *Active* внаслідок Wrong *Fix*. Якщо ж спосіб закриття помилки є неприйнятним (наприклад, тестер не згоден, що дана помилка є дублікатом іншої), то використовується *причина Resolutiondenied.* Нарешті, помилка може перейти в стан *Active*, якщо вона знов почала з'являтися – причини *Reactivation* і *Regression*. При цьому для тестувальника важливо не створювати нову помилку, а зрозуміти, що це стара, закрита помилка, яка знов виявилася. Ця інформація допоможе розробникам швидше розібратися з виправленнями – продивитися ті зміни початкових текстів, які закривали цю помилку і виправити їх. При цьому може дуже ефективно працювати *зв'язок*, який забезпечує VSTS для елементів *робіт* і змінами в засобі *контролю версій*.

У стан *Resolve* помилка переходить, по-перше, після того, як розробник її виправив. У цей стан розробник може перевести помилку ще і тому, що це не помилка, а властивість (*тестувальник* невірно зрозумів вимоги до системи або проектну специфікацію) – *причина As Designed*. А також тому, що помилка повторює іншу помилку (*Duplicate*), що була знайдену раніше, помилка не відтворюється у розробника (Unable to *reproduce*) і так далі.

У стан *Close* помилка переходить, по-перше, коли *тестувальник* прийняв її виправлення (*причина Fixed*). По-друге, коли він погодився з думкою розробника, що вона повторна (Duplicated), не відтворюється (Unable to *reproduce*) і ін. З цих же причин сам розробник може перевести помилку у стан *Close* прямо із стану *Active*. Але таким розробником може бути не будь-хто, а, наприклад, технічний *керівник проекту* або *архітектор*. Решта всіх розробників може не мати прав переводити помилки самостійно в стан *Close*, а зобов'язані діяти лише через тестувальників.

Як створюється опис помилки. Створення нової помилки може відбуватися або за допомогою пункту *меню* в *Team Explorer "team/add Bug*", або за допомогою додавання зв'язаних елементів роботи для завдань, під час реалізації яких помилки були виявлені. Крім того, провал автоматичної *збірки* або *прогону тестів* може служити *тригером* для автоматичного створення помилки. Вікно для опису помилки показане на Рис. 15.2.

Зв'язок змін початкових текстів ПЗ і помилок. У лекції про *конфігураційне управління* був детально розглянутий *зв'язок змін* початкового коду з елементами роботи. Слід розглянути, як помилка пов'язана з цими змінами (тобто ми дивимося на те ж саме завдання, але з іншого боку – з боку елементів роботи, і обираємо один специфічний тип елементу роботи – помилку). Всі зміни в коді, які пов'язані з виправленням певної помилки, легко можна відстежити за допомогою закладк "*Links"* в діалозі опису помилки. На Рис. 15.3 показано, що помилка пов'язана з пакетом змін, який є внесеним до системи *контролю версій*.

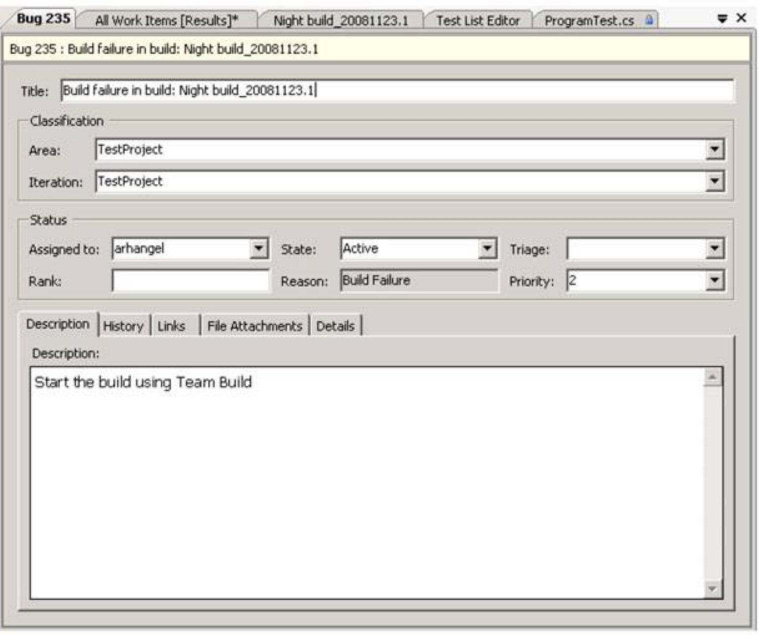

Рис. 15.2. **Автоматично створена помилка.**

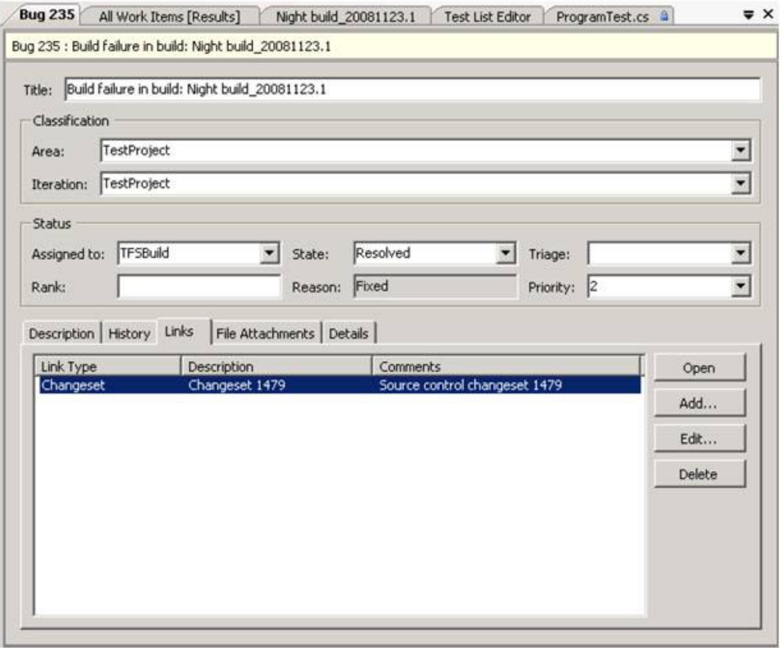

Рис. 15.3. **Відслідковування змін є помилковим**

Система сповіщень про події в проекті є окремою підсистемою TFS, що оповіщає членів команди про різні події за допомогою електронної пошти, наприклад, про *завершення процесу збірки* проекту або про зміну елементу роботи. Ця система використовується для оперативного моніторингу стану проекту, що є особливо важливим для тестування. Розсилка сповіщень здійснюється за допомогою електронної пошти.

Набудувати умови, за яких слід відправляти сповіщення, а також *список* одержувачів, можна за допомогою спеціальної команди з *меню* проекту, як показано на Рис. 15.4. Все це конфігурується на рівні проекту в цілому, але кожен учасник розробки може визначити свої правила відправки сповіщень.

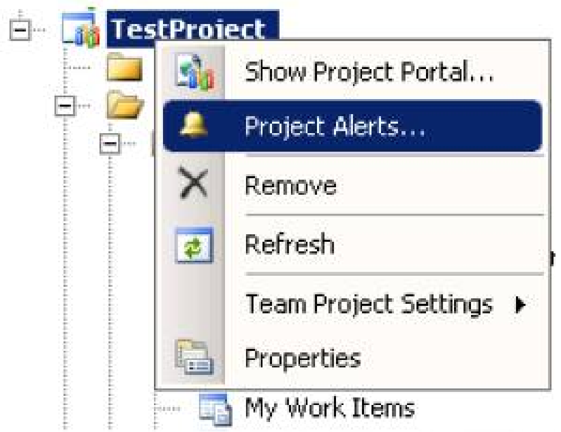

Рис. 15.4. **Управління підписками.**

У TFS, як показано на Рис. 15.5, підтримуваними є такі типи автоматичних сповіщень.

- У разі зміни елементу роботи (сповіщення відправляється у випадку зміни будь-якого реквізиту). Цей вид сповіщень дозволяє оперативно дізнаватися про появу нових елементів роботи і про зміну тих, які вже існують. Наприклад, розробник може оперативно отримувати повідомлення про надіслані йому помилки.
- Внесення змін до системи контролю версій. Як правило, цей тип сповіщення використовується архітекторами або технічними лідерами команди для контролю якості коду, що був розроблений, за допомогою регулярної перевірки змін, які вносяться.
- У разі автоматичної або ручної зміни атрибуту "якість" в описі результатів автоматичної зборки. Цей тип сповіщень дозволяє керівникам проекту дізнаватися про зміну стану проекту.
- У разі завершення процесу автоматичної збірки, незалежно від результатів. Даний тип сповіщень корисний для всіх учасників проекту.

Сповіщення, TFS, що розсилаються, містять лише базову інформацію про подію, яка відбулася, і *посилання*, яке дозволяє проглянути деталі про подію через *Internet Explorer*, на *Share Point* порталі проекту. Зокрема, *інформація* про результати автоматичної *збірки* і про ті помилки, вимоги на виправлення яких увійшли до відповідної версії початкового коду проекту, наведена на Рис. 15.6.

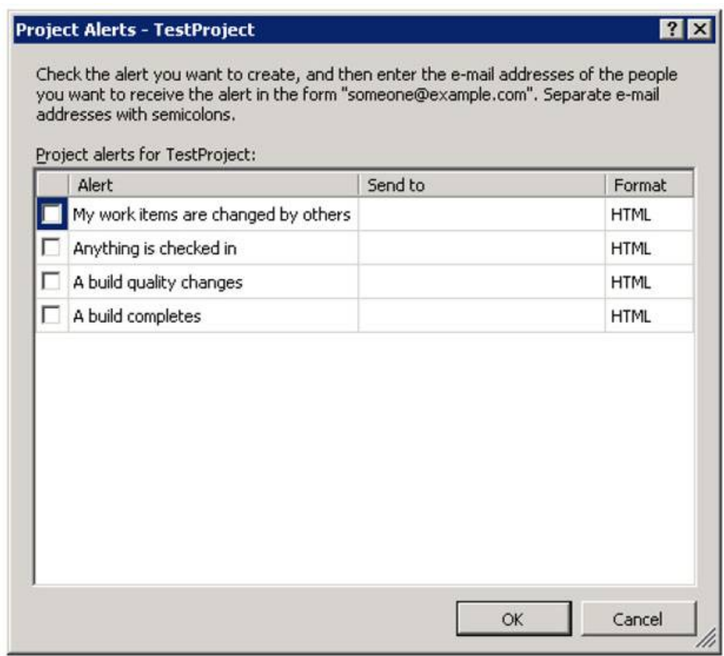

Рис. 15.5. **Налагоджування одержувачів сповіщень.**

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. 15.6. **Результати автоматичної збірки.**

15.3. Модульні тести

Модульні тести як засіб підвищення якості *програмного забезпечення* з'явилися достатньо давно, проте вони довгий час залишалися не підтримуваними продуктами *Microsoft*. Вперше *підтримка* модульних тестів з'явилася в *Visual* Studio 2005 і була доступною у версіях *Professional* і вище (зокрема, і у всіх версіях групи Team).

Основна ідея модульних тестів полягає в тому, що працездатність коду можна перевірити автоматично за допомогою написання *додаткового коду*, який викликає те, що слід тестувати, а потім аналізує результати. При цьому, якщо для системи в цілому такий підхід достатньо скрутний з причини складності системи, то для окремих частин системи (модулів), цей метод застосовується і дає добрі результати. Як правило, основною одиницею для *модульного тестування* є класи і *методи класів*.

Історично склалося, одним з перших популярних інструментів, який був орієнтований на організацію *модульного тестування*, став **jUnit** для

Java-додатків. (Потім він був *клонований* під багато інших платформ.) Для платформи *.NET* піонером тут був **nUnit**, який до цих пір займає лідируючу *позицію* в цій ніші. Проте, лідерство **nUnit** серйозно похитнулось з появою підтримки модульних тестів у *Visual Studio*. *В порівнянні з* класичними системами, *Visual Studio* має суттєві переваги:

- Підтримана повна *інтеграція* до *призначеного для користувача інтерфейсу*, включаючи запуск та *аналіз* результатів (для інших систем інтеграція є доступною за окремі гроші і не такою функціональною).
- Реалізовані можливості для легкої інтеграції до засобів автоматичної *збірки* (лише для TFS *Team Build*).
- Запропоновані додаткові засоби для опису процедури *розгортання* тесту (хворе місце для більшості з решти систем) і *конфігурації* інших аспектів виконання.
- Є засоби автоматичної генерації *сигнатур* тестів і засобів доступу до *приватних* частин класів, що тестуються.
- Підтримано управління тестовими даними, а також тестами, що використовують дані на рівні платформи.

У *версії Visual Studio* 2008 *підтримка модульного тестування* була перенесена з версій сімейства *Team* до версії *Professional*, що було цілком логічним кроком, оскільки *модульне тестування* є загальною практикою, що вживається як в особистій, так і в командній розробці. Модульне тестування більш не є чимось специфічним для *Visual Studio Team System*, а його реалізація відповідає більшості стандартних пакетів в цій галузі.

15.4. Пакети тестів

Як *правило*, *модульні* тести і тестові *конфігурації* розробляються самими розробниками, основне завдання тестера в цьому випадку – організу-

вати всі тести у впорядковану структуру пакетів і вказати, за яких умов які пакети повинні виконуватися. Вся *ієрархія* тестів зберігається у так званому файлі *метаданих*, який має розширення vsmdi. Цей *файл* знаходиться у системі контролю *версій* і може бути включений до рішення у якості окремого елементу (Рис. 15.7).

Для створення тестового пакету можна скористатися *меню "Create New Test List"*, як показано на Рис. 15.8, і після цього відкриється вікно для завдання імені нового *пакету* і визначення його місця в ієрархії тестових пакетів (див. Рис. 15.9). В тому випадку, якщо для вирішення вже *створено файл метаданих*, пакет тестів буде доданий до нього, якщо ж файл *метаданих* ще створений не був, він буде створений автоматично.

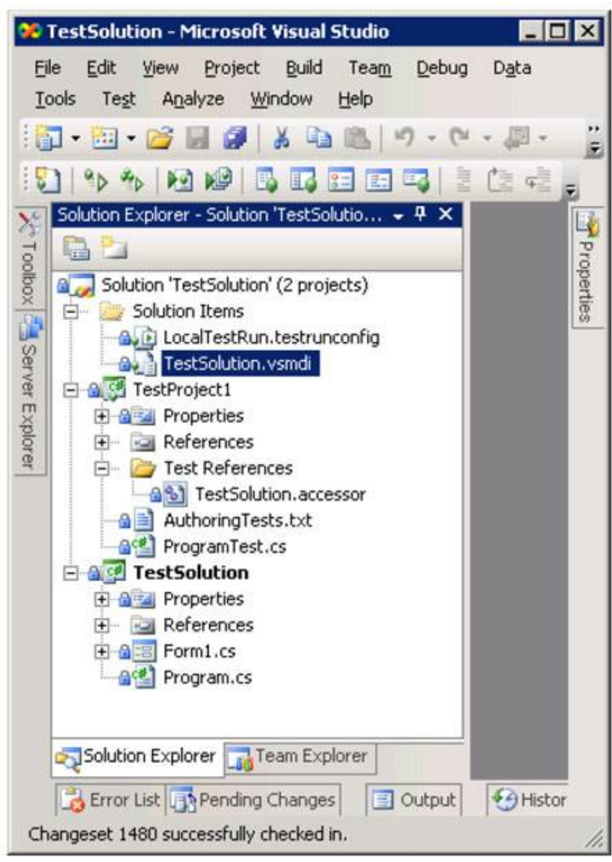

Рис. 15.7. **Пакет з ієрархією тестів**

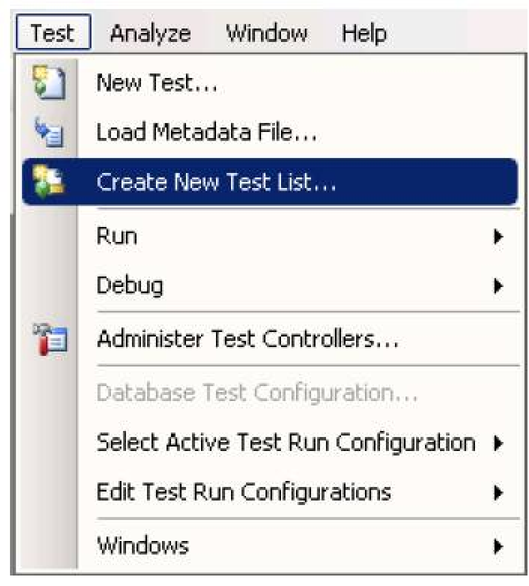

Рис. 15.8. Створення списку тестів.

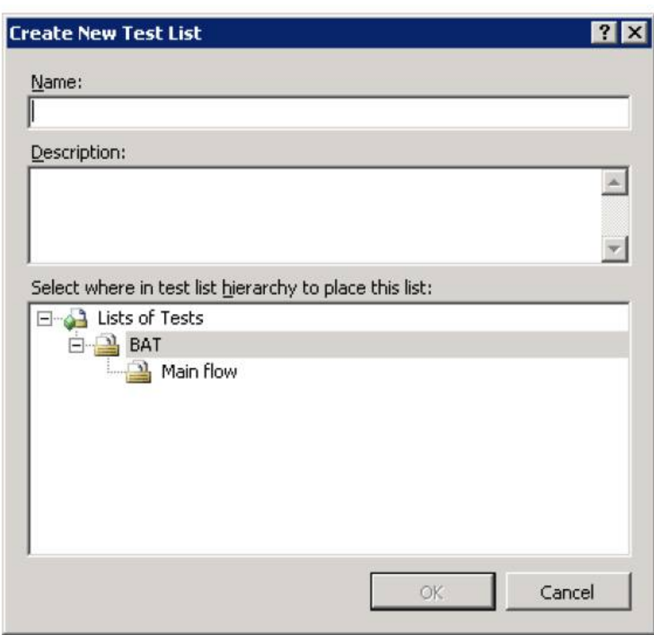

Рис. 15.9. Властивості нового списку тестів

Вміст *пакетів* тестів редагується за допомогою спеціального редактора, який показаний на Рис. 15.10. До пакету можуть бути включені тести, що знаходяться в одному з проектів поточного рішення.

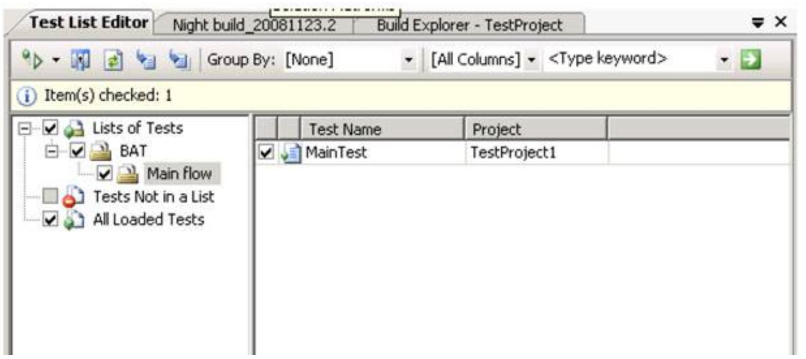

Рис. 15.10. Редактор списку тестів

Тестові *пакети* можуть використовуватися як для ручного *прогону тестів* певної тематики (*команда "***Run checked"**, Рис. 15.11), так і для автоматичного прогону в межах автоматичної *збірки*.

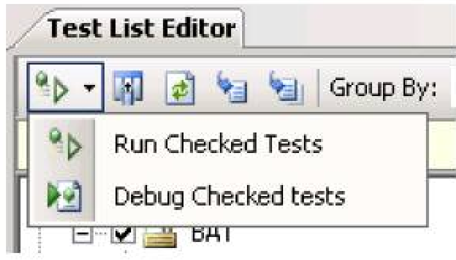

Рис. 15.11. "Ручний" запуск пакету тестів.

Вказати тести, які запускатимуться для певної збірки, можна під час створення файлу з описом *збірки MsBuild*, або надалі через модифікацію проекту *MsBuild*. У першому випадку достатньо на відповідному кроці майстра вибрати *файл метаданих* і позначити галочками пакети тестів, які цікавлять (Рис. 15.12). У другому випадку необхідно відкрити проект *MsBuild* в редакторі XML*,* знайти елемент MetaDataFile, або вписати необхідні пакети власноруч:

<MetaDataFile

Include="$(BuildProjectFolderPath)/../../TestSolution/TestSolution.vsmdi">

<TestList>BAT/Main flow</TestList>

</MetaDataFile>

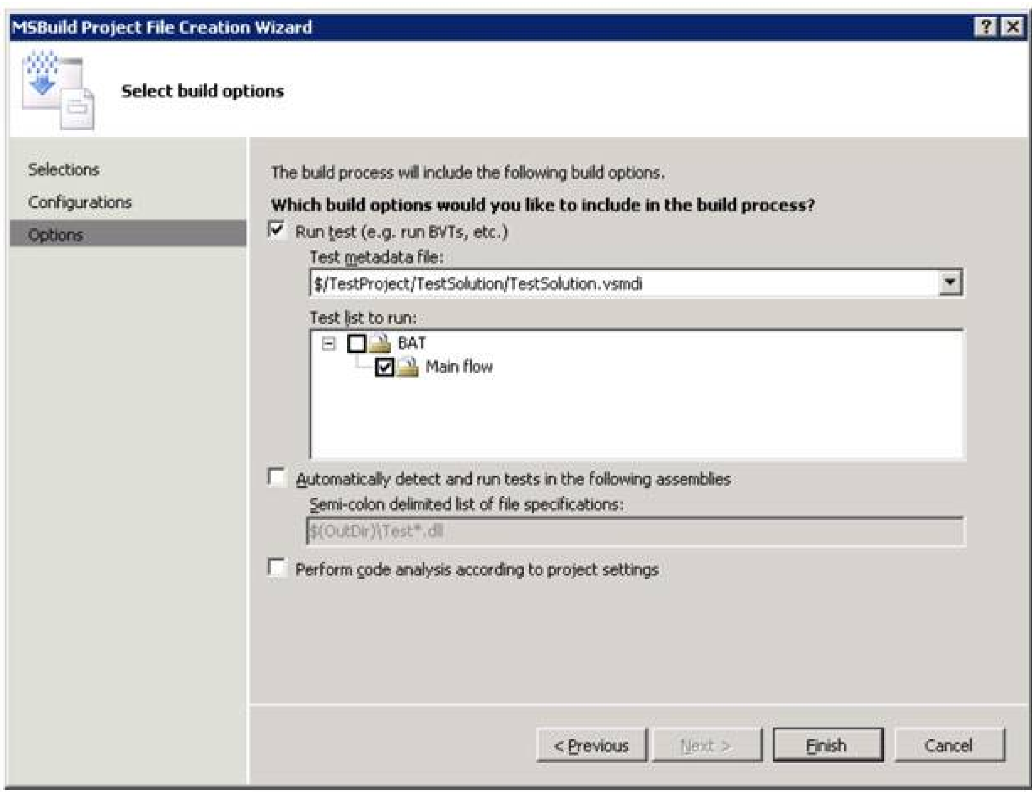

Рис. 15.12. Вибір пакету тестів для автоматичної збірки

15.5. Автоматичне тестування Web-додатків

Capture & Playback підхід. Цей підхід до тестування призначених для користувача інтерфейсів виглядає дуже ефектно. В його основі лежить таке: *тестувальник* проходиться мишею *по* вікнах, пунктах *меню* і інших елементах інтерфейсу; спеціальна програма записує його кроки, а потім їх відтворює в пакетному режимі. Тобто, так дуже просто створювати повторювані тести. Технічно це реалізовано таким чином.

Спеціальне тестове оточення з тією або іншою точністю розпізнає, де саме було натиснуто мишкою на екрані, і створює відповідний код у спеціальному скрипті. Потім цей *скрипт* виконується в пакетному режимі і відтворює дії тестувальників. Все питання полягає в тому, яким чином розпізнається *клік* тестувальника мишкою. Ідеально, коли цей *клік* зв'язується з відповідним елементом управління інтерфейсу. Тобто якщо *тестувальник* натиснув кнопку в якомусь діалозі, то в скрипті ця *інформація* зберігається в повному обсязі. Інша, більш складна ситуація має *місце* тоді, коли тестове оточення не може розпізнати, який саме елемент призначеного для користувача інтерфейсу активував *тестувальник*. Тоді в *скрипт* заноситься *інформація* про ті координати того місця на екрані, де був *клік* мишкою.

Чим погана остання ситуація? Річ у тому, що у разі щонайменшої зміни призначеного для користувача інтерфейсу (а це типова ситуація, адже ПЗ розвивається, допрацьовується і тестується одночасно) автоматичний тест-скрипт, що був створений таким чином, виходить з ладу. Там, куди раніше *клік* мишкою потрапляв (наприклад, на потрібну кнопку), тепер знаходиться зовсім інший елемент управління.

Якщо ж тестове оточення розпізнало елемент інтерфейсу, то такий тест-скрипт виявляється більш "живучим". Це відбувається на рівні перехоплення повідомлень на рівні операційної системи (зокрема, *Windows*). Але для того, щоб це було можливо, код додатку повинен бути написаний "правильним" чином. Далеко не всі інтерфейсні додатки написані "правильно".

То, наскільки успішно для конкретного додатку можна застосувати даний підхід, визначається декількома чинниками. Основним є те, яка платформа використовується (наприклад, *Java Swing*, *AWT*, *Windows* Forms, WPF, *і т.і.*) і додаткові бібліотеки з елементами призначеного для користувача інтерфейсу. Найбільш зрілою платформою з цієї точки зору на даний момент є MS *Windows* Forms.

Capture & Playback для тестування Web-інтерфейсів. У випадку з тестуванням *Web-інтерфейсу* ситуація з точністю розпізнавання елементів управління (*interface* controls) значно простіша, ніж у разі тестування довільного інтерфейсу користувача. Взаємодія з сервером відбувається *за* строго описаним протоколом *HTTP*, що дозволяє під час запису перехопити повідомлення, які відправляються і отримуються. Крім того, візуальне *подання* сторінки задане в структурованому форматі *HTML*, що дозволяє легко ідентифікувати окремі елементи на сторінці.

У версії *Visual Studio Team Edition* for *Software Testers* включений додатковий пакет, який полегшує автоматизацію тестування Web-додатків методом *Capture & Playback*. Він дозволяє, як автоматично генерувати прості тестові сценарії на основі запису дії користувача, так і писати точніші тести на будь-якій мові платформи *.NET*.

Додати новий *Web-тест* до рішення можна за допомогою команди "**Test/new Test"**, якщо обрати у виниклому діалозі (Рис. 15.13) тип тесту *Web Test*.

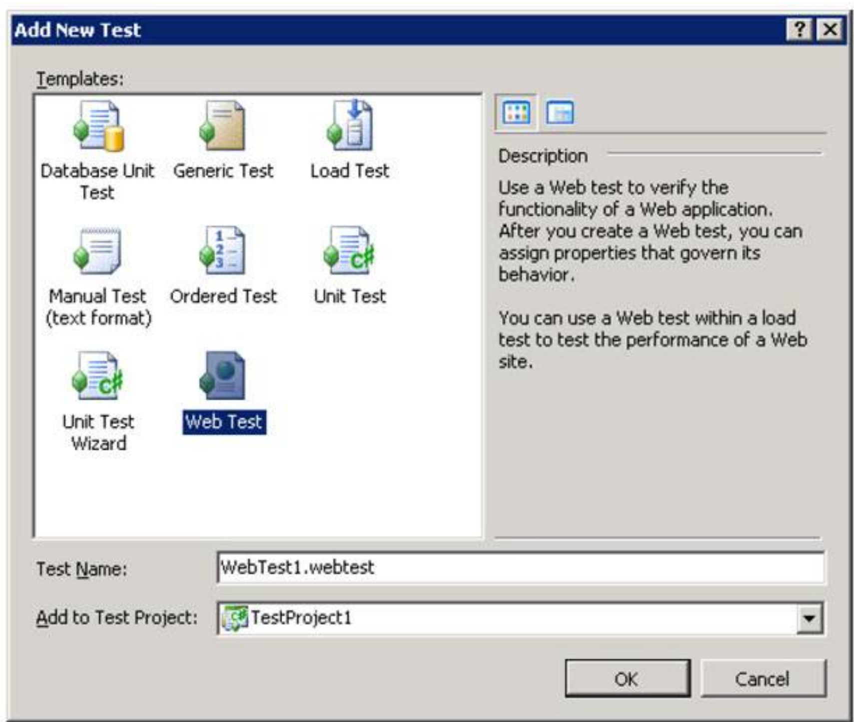

Рис. 15.13. **Створення Web-тесту**

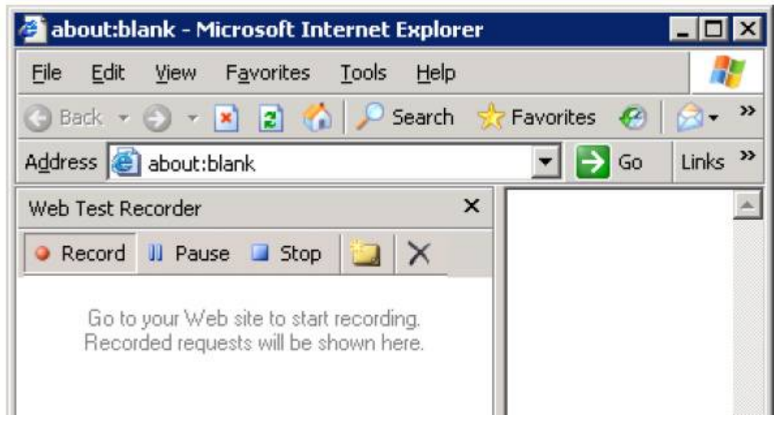

Рис. 15.14. **Запис кроків тестувальника Web-додатків в Internet Explorer**

Після створення нового тесту автоматично буде запущена процедура запису сценарію тесту в браузері (Рис. 15.14). На цьому етапі досить ввести www-адрес додатку, який слід протестувати, після чого виконати тестову перевірку *Web-інтерфейсу* безпосередньо у браузері.

Після закінчення запису автоматично буде згенерований тест, який містить всі відправлені на *сервер http-запити* і всі отримані відповіді (див. Рис. 15.15). У цьому випадку *генератор* автоматично додасть деякі правила, *за* якими буде перевірятися коректність роботи тесту.

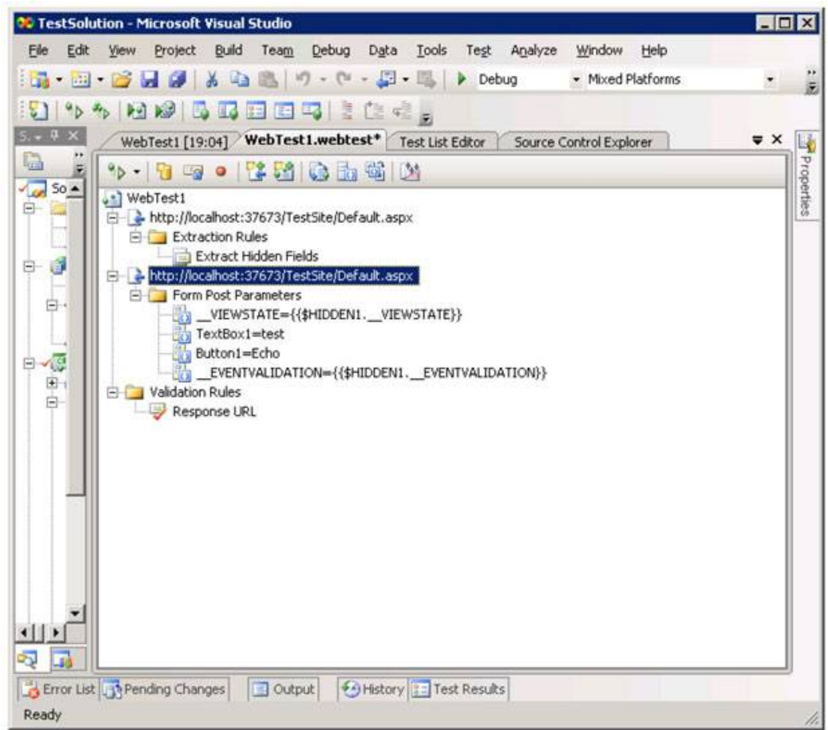

Рис. 15.15. **Редактор Web-тесту**

Для кожного з кроків тесту можна, за допомогою візуального редактора, додати додаткові перевірки або опції, що керують перебігом виконання: *пошук* підрядка у відповіді серверу (текст отриманого *HTML*), валидацію *HTML* за допомогою визначення регулярного виразу, перевірка наявності або відсутності певних тегів чи атрибутів на сторінці і так далі. Допускається також можливість розробки власних правил на будь-якій *.NET* мові. У тих же випадках, коли гнучкості редактора недостатньо, можна згенерувати C# код для даного тесту і реалізувати необхідну логіку "власноруч".

Під час написання правил валідації дуже важливо правильно обрати необхідний *рівень деталізації*. Чим детальніше сформульовано правило і чим більш воно є специфічним для даного *HTML*, тим більше *вірогідність* того, що тест доведеться змінювати у разі зміни коду тестованого додатку, навіть якщо ці зміни не стосувалися безпосередньо цієї частини. Гарне

правило *валідації* повинне перевіряти лише те, що є важливою частиною бізнес логіки додатку або те, що є ключовою властивістю даної *HTML-сторінки.* При цьому правило не повинне перевіряти деталі верстки і додаткові візуальні ефекти. На жаль, правила, що автоматично згенерували, далеко не завжди виявляються найбільш ефективними.

Створений в редакторі або власноруч тест є повноправним тестом і може бути включений в тестовий пакет, а отже, і в процедури автоматичної *збірки* (Рис. 15.16). Проте, для того, щоб *автоматичне тестування* було можливим, необхідно відповідним чином налагодити *сервер* автоматичних зборок – на ньому повинен бути розгорнутий сервер *IIS* з тестовим сайтом, а одним з етапів *збірки* повинне бути оновлення коду цього сайту.

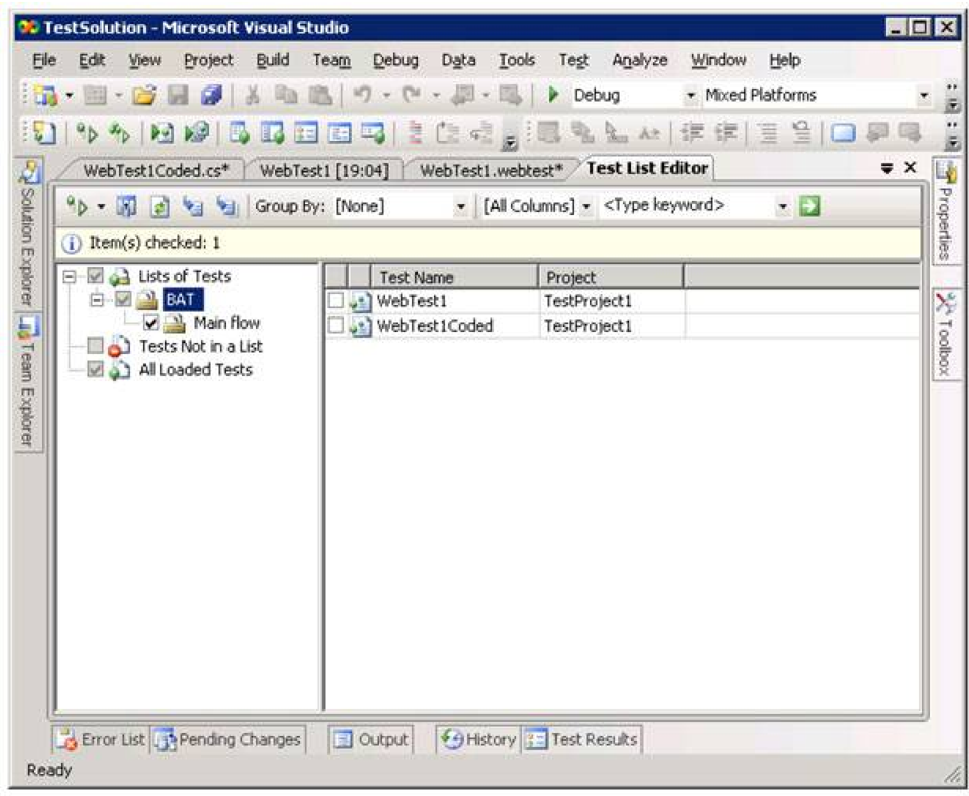

Рис. 15.16. **Web-тести в пакетах тестів**

Capture & Playback підхід. Цей підхід до тестування призначених для користувача інтерфейсів виглядає дуже ефектно і заснований на такій

ідеї. *Тестувальник* проходиться мишкою *по* вікнах, пунктах *меню* і інших елементах інтерфейсу. Спеціальна програма записує його кроки і потім їх відтворює в пакетному режимі. Тобто дуже просто виходять повторювані тести. Технічно це влаштовується так.

Спеціальне тестове оточення з тією або іншою точністю розпізнає, куди саме було натиснуто мишкою на екрані і створює відповідний код в спеціальному скрипті. Потім цей *скрипт* "проганяється" в пакетному режимі, відтворюючи дії тестувальників. Все питання в тому, яким чином розпізнається *клік* тестувальника мишкою. Ідеально, коли цей *клік* зв'язується з відповідним елементом управління інтерфейсу. Тобто якщо *тестувальник* натиснув кнопку в якомусь діалозі, то в скрипті ця *інформація* зберігається в повному об'ємі. Інша, грубіша ситуація має *місце* тоді, коли тестове оточення не може розпізнати, який елемент призначеного для користувача інтерфейсу активував *тестувальник*. Тоді в *скрипт* заноситься *інформація* про ті координати на екрані, куди був *клік* мишкою.

Чим погана остання ситуація? Річ у тому, що при щонайменшій зміні призначеного для користувача інтерфейсу (а це типова ситуація, адже ПЗ розвивається, допрацьовується і тестується одночасно) автоматичний тест-скрипт, створений таким чином, виходить з ладу. Там, куди раніше *клік* мишкою потрапляв, наприклад, на потрібну кнопку, тепер знаходиться зовсім інший елемент управління.

Якщо ж тестове оточення розпізнало елемент інтерфейсу, то такий тест-скрипт виявляється більш "живучим". Це відбувається на рівні перехоплення повідомлень на рівні операційної системи (зокрема, *Windows*). Але для того, щоб це було можливо, код додатку повинен бути написаний "правильним" чином. Далеко не всі інтерфейсні додатки написані "правильно".

То, наскільки успішно для конкретного застосування можна застосувати даний підхід, визначається декількома чинниками. Основним є те, яка використовується платформа (наприклад, *Java Swing*, *AWT*, *Windows* Forms, WPF, *etc*.) і додаткові бібліотеки з елементами призначеного для користувача інтерфейсу. Найбільш зрілою платформою з цієї точки зору на даний момент є MS *Windows* Forms.

Capture & Playback при тестуванні Web-інтерфейсів. У випадку з тестуванням *Web-інтерфейсу* ситуація з точністю розпізнавання елементів управління (*interface* controls) значно простіша, ніж при тестуванні довільного призначеного для користувача інтерфейсу. Взаємодія з сервером відбувається *по* строго описаному протоколу *HTTP*, що дозволяє при записі перехопити повідомлення, що відправляються і отримувані. Крім того, візуальне *представлення* сторінки задане в структурованому форматі *HTML*, що дозволяє легко пізнати окремі елементи на сторінці.

У видання *Visual Studio Team Edition* for *Software* Testers включений додатковий пакет, що полегшує автоматизацію тестування Web-додатків методом *Capture & Playback*. Він дозволяє, як автоматично генерувати прості тестові сценарії на основі запису дії користувача, так і писати точніші тести на будь-якій мові платформи *.NET*.

Додати новий *Web-тест* до рішення можна за допомогою команди **"Test/new Test"**, вибравши у виниклому діалозі (Рис. 15.13) тип тесту *Web Test*.

Після створення нового тесту автоматично буде запущена процедура запису сценарію тесту в браузере (див. Рис. 15.14). На цьому етапі досить ввести www-адрес додатку, який слід протестувати, після чого виконати тестову перевірку *Web-інтерфейсу* безпосередньо у браузері.

Після закінчення запису автоматично згенерує тест, що включає всі відправлені на *сервер http-запити* і всі отримані відповіді (див. Рис. 15.15).

При цьому *генератор* автоматично додасть деякі правила, *за* якими буде перевірятися коректність роботи тесту.

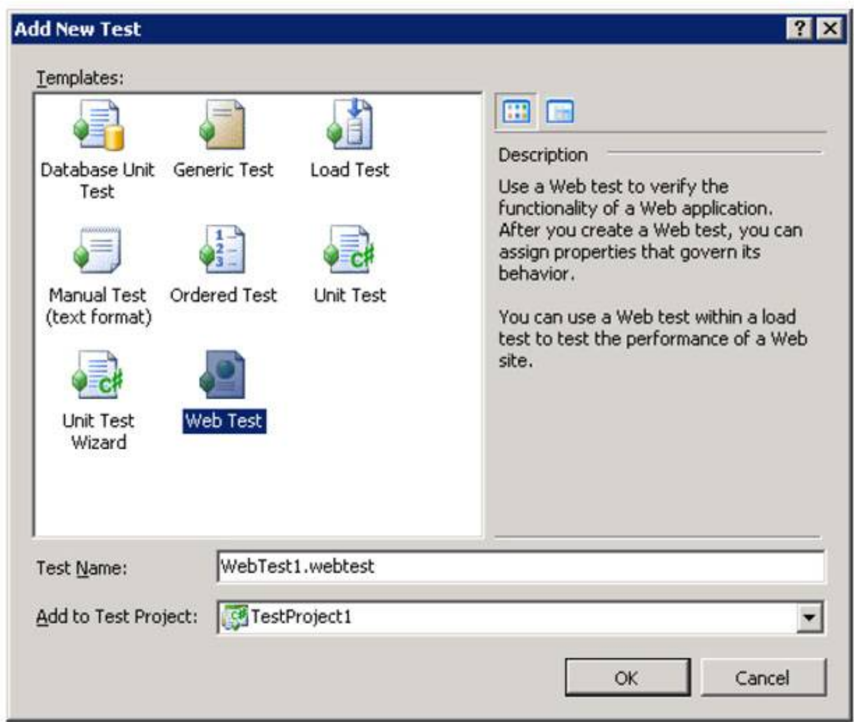

Рис. 15.13. **Створення Web-тесту.**

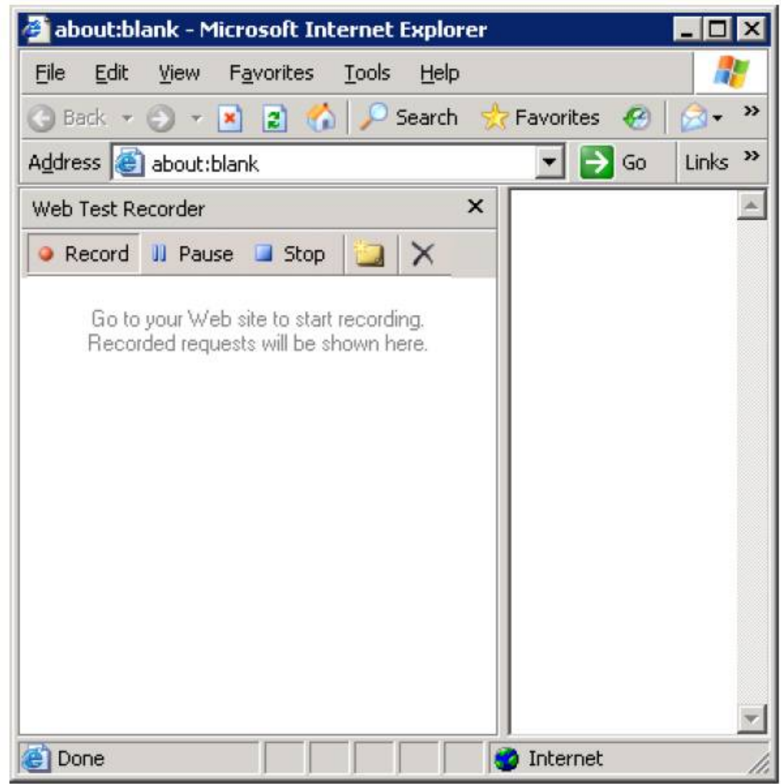

Рис. 15.14. **Запис кроків тестувальника Web-додатків в Internet Explorer.**

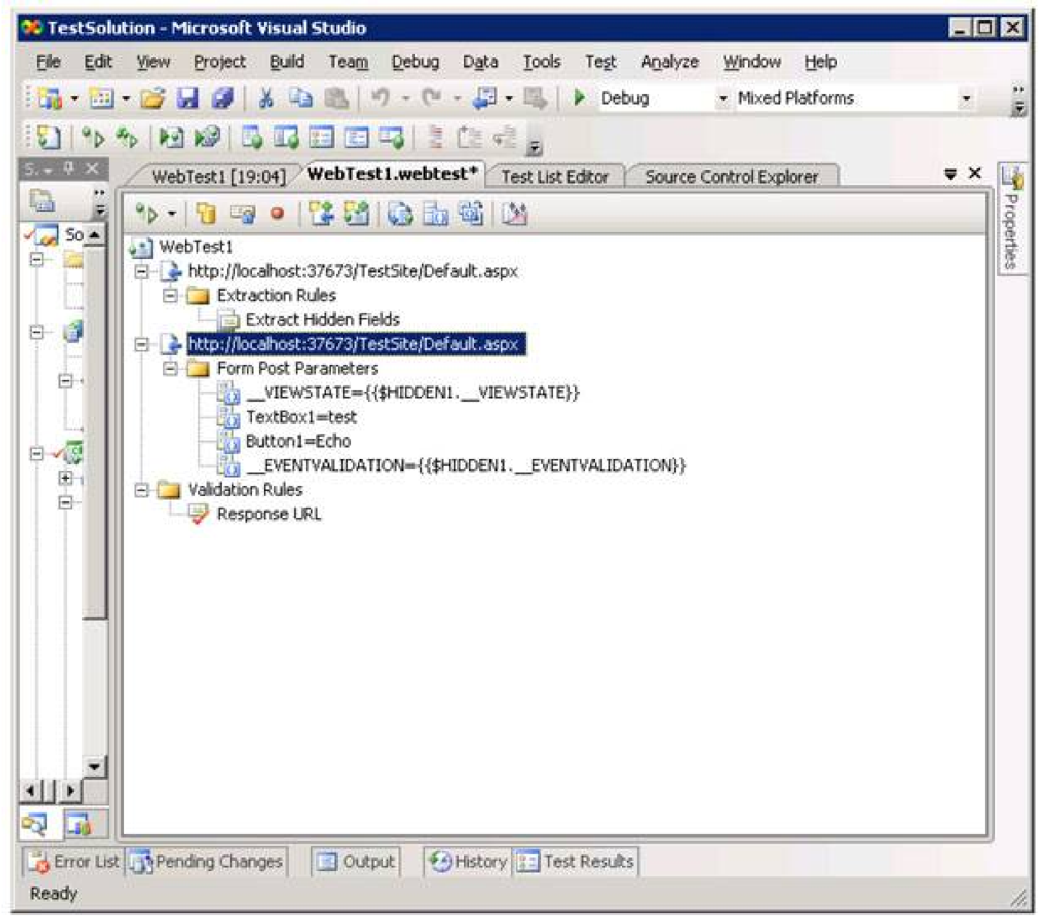

Рис. 15.15. **Редактор Web-тесту.**

Для кожного з кроків тесту можна, за допомогою візуального редактора, додати додаткові перевірки або опції-*керівники* перебігом виконання: *пошук* підрядка відповіді серверу (текст отриманого *HTML*), валідацию *HTML* через завдання регулярного виразу, перевірка наявності або відсутності певних тегів або атрибутів на сторінці і так далі. Допускається також можливість розробки власних правил на будь-якій *.NET* мові. У тих же випадках, коли гнучкості редактора недостатньо, можна згенерувати C# код для даного тесту і реалізувати необхідну логіку "власноруч".

При написанні правил валідації дуже важливо правильно вибрати необхідний *рівень деталізації*. Чим детальніше сформульовано правило і чим більш воно є специфічним для даного *HTML*, тим більше *вірогідність*

того, що тест доведеться змінювати при зміні коду тестованого додатку, навіть якщо ці зміни не стосувалися безпосередньо цієї частини. Гарне правило *валідації* повинне перевіряти лише те, що є важливою частиною бізнес логіки додатку або те, що є ключовою властивістю даної *HTML-сторінки.* При цьому правило не повинне перевіряти деталі верстки і додаткові візуальні ефекти. На жаль, правила, що автоматично згенерували, далеко не завжди виявляються найбільш ефективними.

Створений в редакторові або в ручну тест є повноправним тестом і може бути включений в тестовий пакет, а отже, і в процедури автоматичної *збірки* (див. Рис. 15.16). Проте, для того, щоб *автоматичне тестування* було можливим, необхідно відповідним чином набудувати *сервер* автоматичних складок – на нім повинен бути розгорнутий сервер *IIS* з тестовим сайтом, а одним з етапів *збірки* повинне бути оновлення коду цього сайту.

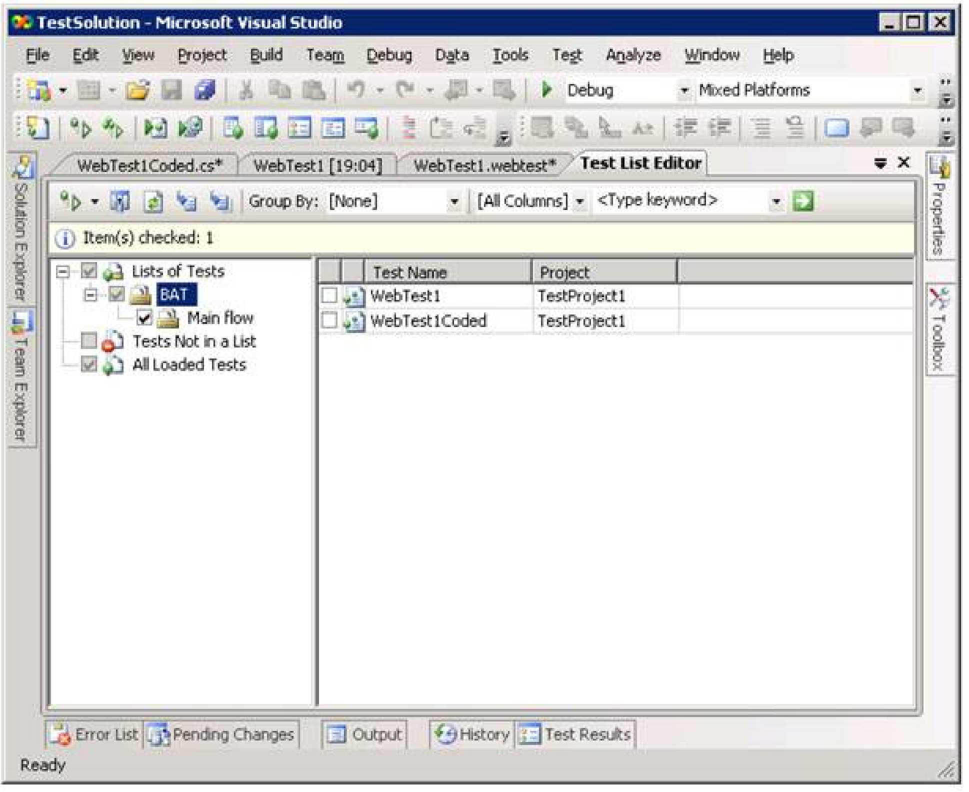

Рис. 15.16. **Web-тести в пакетах тестів.**

Контрольні запитання

1. Які можливості VSTS слід виділити з тестування?
2. Які основні переваги надає інтегрована система відслідковування помилок на основі системи управління елементами роботи у порівнянні з іншими системами?
3. Які стани визначено для елементу роботи типу "помилка" (Bug)?
4. Як помилка потрапляє у стан Active?
5. Як помилка потрапляє у стан Resolve?
6. Як помилка потрапляє у стан Close?
7. Як створюється опис помилки?
8. Як реалізований зв'язок змін початкових текстів ПЗ і помилок?
9. Як реалізований зв'язок початкових текстів ПЗ і помилок?
10. Як реалізований життєвий цикл помилки в шаблоні процесу MSF for Agile?
11. Як створюється опис помилки?
12. Що являє собою система автоматичних повідомлень в TFS?
13. Які цілі і завдання модульного тестування?
14. Як модульні тести, що були створені розробниками, можуть використовуватися в подальшому?
15. Які альтернативи MS VSTS існують для автоматичної підтримки модульного тестування для Visual Studio?
16. В чому полягає підтримка модульного тестування в MS VSTS?
17. Яка частина середовища реалізує підтримку модульного тестування?
18. Як реалізована підтримка роботи з пакетами тестів в MS VSTS?
19. В чому полягає підхід Capture & Playback для тестування користувальницького інтерфейсу?
20. Охарактеризувати труднощі підходу Capture & Playback для тестування користувальницького інтерфейсу?
21. Як ці проблеми вирішуються в разі тестування інтерфейсів Web-додатків?

22. В чому полягає підтримка Capture & Playback для тестування інтерфейсів Web-додатків в MS VSTS.

Глава 16. VSTS: Підтримка різних моделей процесу

Підтримка шаблонів процесу. Інструменти налагоджування. Огляд існуючих шаблонів. MSF for Agile Software Development. Scrum.

16.1. Підтримка шаблонів процесу

Загальні положення. Чим активніше *інструментарій* інтегрується у процес розробки, тим більше функціональності *з* контролю і підтримки бізнес-процесів розробки ПЗ він надає, тим більше затребуваними стають *механізми* налагоджування цього інструментарію. Природно, для такої системи як VSTS питання налагоджування стоїть особливо гостро.

У разі створення в VSTS кожного нового проекту, відразу після вибору імені проекту, *користувач* вибирає *шаблон* процесу розробки для цього проекту. Весь подальший процес створення проекту визначається цим шаблоном. *Шаблон* може бути окремо розроблений або підправлений що існує.

Шаблон процесу містить шість *основних розділів*, в рамках яких можна здійснювати налагоджування роботи TFS.

1. Класифікація (*Classification*) – опис галузей роботи, *ітерацій* і налагоджування *інтеграції* з *Microsoft Project*. *Галузі роботи* – це спосіб для категоризації робіт в проекті. Прикладом галузі роботи може бути як напрям *діяльності*(розробка, тестування, *документування* і так далі), так і робота над певною частиною проекту (серверна частина, *клієнт*, інфраструктура і так далі).
2. Відслідковування елементів роботи (*Work Item Tracing*) – опис типів елементів роботи, включаючи завдання їх *життєвого циклу*,

визначення набору елементів роботи і запитів, що створюються за умовчанням для нового проекту.

3. *Звіти* (*Reports*) – опис *звітів* проекту на спеціальній *XML-мові RDL* (*Report Definition Language*). Наявні в шаблоні звіти за замовчанням можна використовувати "As is", а також виправляти і доповнювати. Створення повністю нового вигляду звітів є досить трудомісткою роботою.
4. *Портал* (*Portal*) – налагоджування шаблону *порталу* в *Sharepoint* з тестовим описом процесу розробки, а також набором робочих документів проекту (планів, дизайн-специфікацій і ін.). На підставі цього шаблону автоматично формуватиметься *портал* для кожного нового проекту.
5. Групи і права доступу (*Groups* & *Permissions*) налагоджуються для використання системи управління версіями, а також для різних правил *життєвого циклу* елементів роботи.
6. *Контроль версій* (*Source Control*) – набір налагоджувань для системи *контролю версій*, включаючи набір політик внесення зміни, дозволу або заборони множинного узяття файлів на редагування і так далі.

Шаблон процесу розробки діє в двох таких основних напрямах: обмежує дії учасників процесу так, щоб вони максимально *відповідали шаблону*, і надає деяку *інфраструктуру*, що дозволяє легко вирішувати основні завдання, що виникають в рамках даного процесу. До обмежень можна віднести налагоджування *життєвого циклу елементів* роботи, а також суті і політики роботи з *системою контролю версій*, а до надання інфраструктури: списки звітів, запити на елементи роботи і основна частина – *портал Sharepoint*, що містить інформацію про використання процесу, а також необхідні адміністративні документи. Цей портал є важливою частиною з

"людської" точки зору, проте, з погляду автоматизації його роль мінімальна.

Природно, цих засобів недостатньо для того, щоб гарантувати, що всі учасники процесу будуть чітко його дотримуватися. Основним елементом в процесі *як і раніше* залишається *людський чинник*. Але, з іншого боку, впроваджуючи лише відносно невелике число обмежень, TFS дозволяє зберегти загальну гнучкість, що може принести величезну користь. Таким чином, відносно TFS, як і відносно більшості успішних систем цього класу, вірне твердження – "не стільки важливий сам інструмент, скільки те, як їм користуються".

Інструменти налагоджування. Для управління шаблонами процесу розробки використовується *утиліта Process Template Manager* (Рис. 16.1), яка є доступною з *меню* TFS *Settings* (ця *утиліта* встановлюється разом з *Team Explorer*). Цей *менеджер* дозволяє завантажувати і вивантажувати шаблони, а також визначати *шаблон*, що використовується для нових проектів *за замовчанням.*

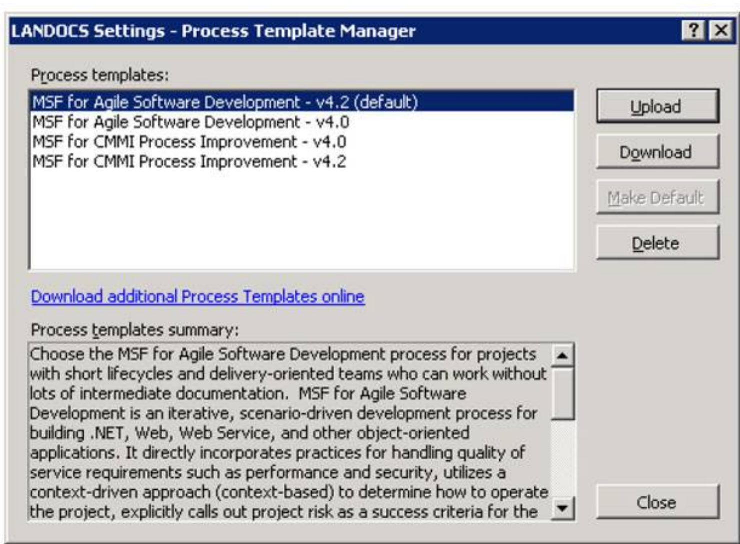

Рис. 16.1. **Менеджер шаблонів**

З погляду реалізації *шаблон* процесу розробки є набором *XML-файлів*, які, мабуть, нікому не захочеться редагувати "власноруч". На щастя, існує інструменти, що дозволяють значно спростити цей процес. Ці інструменти входять у вже згадуваний нами вище пакет *Power Tools*. Ці інструменти доступні з *меню* **Tools/process Editor** (Рис. 16.2). До інструментів належать:

- Редактор типів елементів роботи, який дозволяє редагувати визначення типів елементів роботи, включаючи набори реквізитів і життєвий цикл. Може бути використаний як для редагування файлів з описом типів елементів роботи, експортованих за допомогою Process Template Manager, так і для редагування типів, що знаходяться усередині одного шаблону процесу. Крім того, цей редактор може бути використаний для імпорту/експорту типів елементів роботи у/з шаблон процесу, що є особливо корисним у випадку розповсюдження зроблених змін у різні проекти. Детальніше цей редактор вже розглядався вище.
- Редактор шаблону процесу розробки дозволяє редагувати решту аспектів шаблону процесу розробки. Може бути використаний лише для редагування шаблону, вивантаженого з TFS у файлову систему.
- Редактор глобальних списків дозволяє визначити спискові типи для реквізитів елементів роботи, а також управляти безліччю значень в них. За умовчанням TFS автоматично підтримує глобальний список збірок, проте користувач може визначити і інші списки.
- Переглядач реквізитів елементів роботи – це невелика утиліта, що дозволяє переглянути всі використовувані реквізити всіх типів елементів роботи.

Слід відзначити, що розробка шаблону процесу є дуже трудомістким завданням, яке вимагає від виконавця як знання принципів роботи TFS, так і гарного знайомства з бізнес-процесами своєї компанії. *Витрати* на розробку власного шаблону будуть виправдані лише для достатньо великих

компаній і в довготривалій перспективі. Для невеликих компаній і проектів можна рекомендувати інший підхід – використання одного із стандартних, існуючих шаблонів, що найближче відповідає бізнес процесам, і поступове налагоджування необхідних параметрів.

16.2. Огляд існуючих шаблонів

На даний момент існує достатньо широкий вибір вільно поширюваних шаблонів процесу розробки, які можна було б використовувати як основу. Найбільш відомими є такі шаблони.

- *MSF for Agile Software Development* – один з двох шаблонів, що входять у постачання з TFS. Описує достатньо простий варіант методології MSF, що використовується для розробки невеликих проектів. Входить до стандартної комплектації TFS.

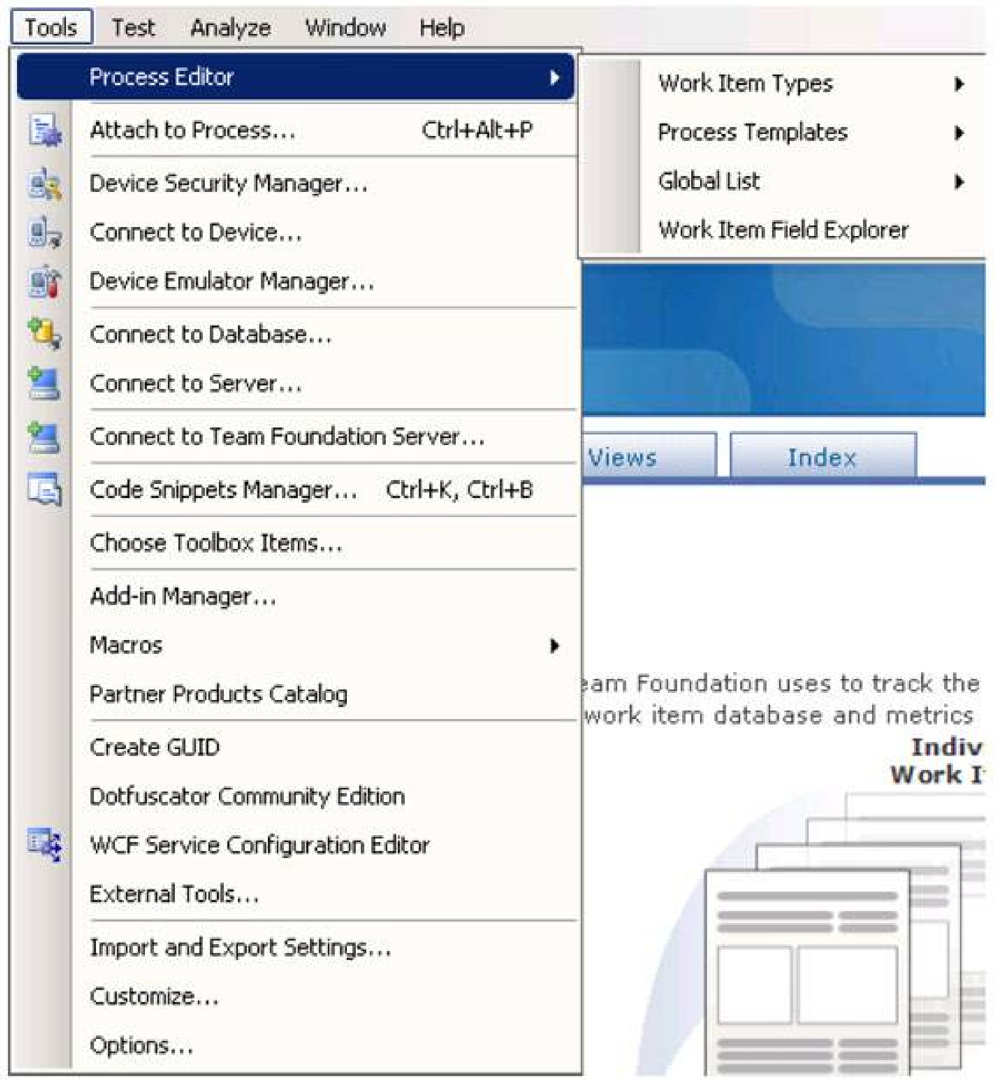

Рис. 16.2. **Інструменти редагування шаблону процесу**

- *MSF for CMMI* – шаблон, що використовується для більших компаній, має на увазі більшу кількість типів елементів роботи, а також більше формальних процедур розробки. Входить до стандартної комплектації TFS.
- *Conchango SCRUM* – шаблон, що описує широко відому гнучку методологію SCRUM. Не входить до стандартної комплектації TFS.

16.3. MSF for Agile Software Development

У цьому шаблоні використовуються стандартні ролі *MSF*, які детально обговорювалися вище. Для підтримки процесу *MSF* for Agile відповідний *шаблон* оголошує такі основні типи елементів роботи.

- Сценарій (*scenario*) – *функціональна вимога* до системи у вигляді деякого сценарію взаємодії користувача і системи, яка є описаною природною мовою як послідовність дій. Сценарій описує лише лінійний шлях розвитку подій, а для опису різних роз*галужень* використовуються додаткові сценарії.
- Вимога до якості *сервісу* (*Quality* of *Service Requirement*, *QOS*) – опис нефункціональної вимоги до системи (тобто вимоги, яка не може бути вираженою в термінах сценарію взаємодії), наприклад, швидкодія і ефективність використання пам'яті.
- Завдання (*Task*) – завдання на виконання деякої обмеженої в об'ємі роботи в проекті. Для кожної ролі завдання можуть мати свою специфіку: для розробника це написання коду, що реалізовує частину сценарію або направленого на досягнення певної якості, а для тестера – написання тестових сценаріїв.
- Помилка (*Bug*) – елемент роботи, який використовується для того, щоб відстежувати і усувати проблеми і помилки, які були виявлені в системі.
- Ризик (*Risk*) – деякий аспект *управління проектом*, який може вплинути на перебіг проекту (як правило, негативний).

За замовчуванням, разом з активацією цього шаблону на порталі *Share Point* розгортаються такі документи.

- План розробки в *Microsoft Project*, що є налаштованим на імпорт елементів роботи, які відносяться до галузі роботи "Розробка". Використовується для планування роботи розробників.
- План розробки тестів – те ж саме, лише спрямований на планування роботи тестерів.

- Список проблем – список виявлених проблем, які необхідно відстежувати (елементи роботи з проставленим прапорцем "Is *Issue*").
- Список (*Check List*) основних вимог до проекту і їх поточний статус.
- Список неврегульованих елементів роботи, яким слід встановити пріоритети і запланувати.

Крім того, для підтримки процесу використовуються такі *звіти*:

- *Помилки* за пріоритетом – показує процентне співвідношення в проекті помилок різного ступеня серйозності і, таким чином, дозволяє оцінити ступінь ефективності тестування.
- *Рейтинг* помилок – показує співвідношення знов знайдених помилок до закритих і знайдених, які ще залишилися. Дозволяє скласти уяву про "здоров'я" продукту.
- *Збірки* – дозволяє оцінювати зміну якості збірок.
- Швидкість проекту – звіт, що демонструє наскільки *активно* закриваються елементи роботи, тобто наскільки швидко команда вирішує поставлені перед нею завдання.
- *Індикатори* якості – об'єднує декілька *індикаторів*, включаючи кількість дефектів, рівень тестового покриття і так далі.
- Звіт про *тестування навантаженням* – показує результати *тестування навантаженням*.
- Регресія – показує тести, які раніше проходили, але тепер почали падати.
- Реактивація – показує те, скільки елементів роботи знову переходить в активний стан після виправлення.
- Зв'язані елементи роботи – ще один спосіб перегляду зв'язків елементів роботи один з одним.
- Робота, що залишилася, – показує кількість елементів роботи, які закриваються, а які залишаються і з'являються з часом.

- Незапланована робота – показує співвідношення зробленого до запланованого і до незапланованого.

Крім вище перерахованих є ще декілька *звітів*, що повертають списки елементів роботи, проте вони не часто використовуються для аналізу.

16.4. Scrum

Шаблон для роботи в методології Scrum був розроблений співтовариством розробників, зараз підтримується компанією Conchagp і доступний за адресою <http://scrumforteamsystem.com/>. У цьому шаблоні використовуються стандартні ролі Scrum, які детально обговорювалися вище. Визначаються такі елементи роботи:

- *Product Backlog Item* – високорівневий опис певної функціональної або нефункціональної вимоги. Містить опис, пріоритет, який встановлений Product Owner, а також попередню оцінку трудомісткості. Багато в чому є аналогічним до сценарію *MSF for Agile*.
- *Sprint* – містить інформацію про поточний або запланований Sprint, включаючи поточний стан і об'єм доступних для Sprint ресурсів.
- *Sprint Backlog Item* – більш низькорівневий опис завдання, який сформований самою командою. Є аналогічним до завдання *MSF*.
- *Bug* – помилка, яку виявлено під час тестування. Помилки стають частиною Product Backlog і отримують пріоритети від Product Owner.
- *Sprint Retrospective* – опис деякого елементу (наприклад, проблеми, вдалого нововведення), що ідентифікований в рамках Sprint Review Meeting і вимагає подальшого вивчення або виконання певних дій.
- *Impediment* – щось що, реально або потенційно заважає ефективній роботі команди. Багато в чому аналогічне до ризику з MSF.

Слід зазначити, що опис елементів роботи в шаблоні Scrum вигідно відрізняється від *MSF* відсутністю великої кількості додаткових, в більшо-

сті випадків непотрібних реквізитів, що дозволяє концентруватися на найбільш важливій інформації. Але, з іншого боку, в шаблоні для Scrum фактично не використовуються ні причини переходів (для кожного переходу визначена рівно одна причина), ні правила, що обмежують переходи. Таким чином, цей *шаблон* використовує лише частину можливостей, які надаються TFS.

Серед *звітів, що* поставляються з шаблоном, слід виділити такі:

- *Bugs Count* – показує кількість помилок, які відсортовані за станом і ступенем впливу на процес тестування.
- *Bugs Fixed and Found* – співвідношення знайдених помилок до закритих.
- *Bug History* – показує кількість знайдених помилок відповідно до їх впливу на процес тестування, а також зміну цієї кількості з часом.
- *Bug Priority* – показує розподіл помилок у відношенні до їх впливу на процес тестування.
- *Bug Resolution Time* – показує наскільки швидко виправляються знайдені помилки.
- *Development To Test cycle time* – демонструє, скільки проходить часу між виконанням роботи і початком тестування за цією роботою.
- *Product Burndown* – звіт, що демонструє, як швидко знижується кількість роботи, яку треба зробити в контексті Product Backlog. Може показувати прогрес у днях або спринтах, а також дозволяє побачити загальну тенденцію і передбачити дату завершення робіт.
- *Product Cumulative Flow* – показує загальний стан Product Backlog у вигляді співвідношення відкритих і закритих елементів, а також їх зміни з часом.
- *Sprint Burndown* і *Cumulative Flow* – теж саме, лише для Sprint Backlog.

Окремої згадки заслуговує система *Task Board for Team System*, яка дозволяє візуалізувати представлення основного засобу управління і моні-

торингу в Scrum, – дошки із завданнями. Ця система отримує інформацію про всі елементи Sprint Backlog з сервера і візуалізує їх у вигляді стікерів на дошці, що дозволяє змінювати їх стани шляхом перетягування (Рис. 16.3.). Цей продукт можна безкоштовно викачати *за* посиланням <http://www.scrumforteamsystem.com/en/TaskBoard/default.aspx>.

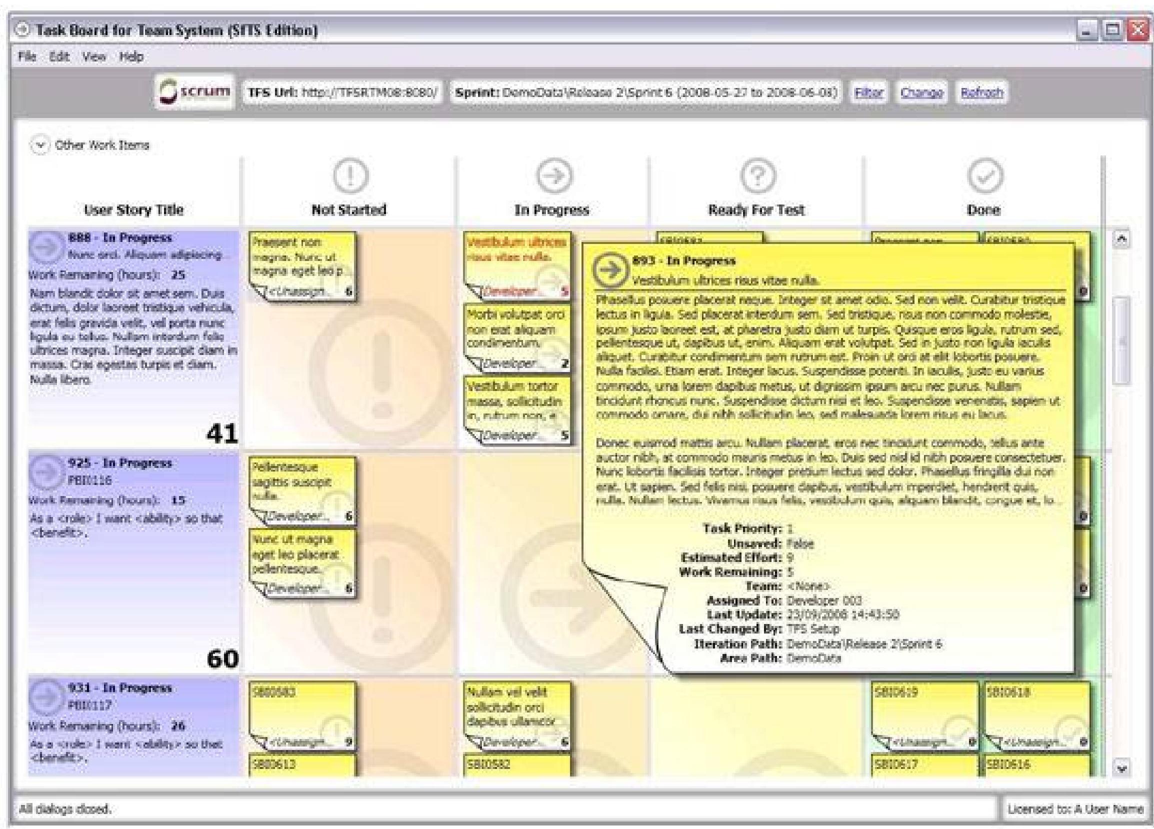

Рис. 16.3. **Дошка завдань**

Контрольні запитання

1. Навіщо потрібні різні шаблони процесів в MS VSTS?
2. Що визначають шаблони процесів в MS VSTS, що задають, і як обмежують розробників? І як їм допомагають?
3. Яка на ваш погляд, трудомісткість створення власного шаблону процесу "з нуля"?

4. З якою темою курсу пов'язані шаблони процесів в MS VSTS? Який термін з курсу в точності може замінити термін "шаблон процесу"?
5. Перерахувати і охарактеризувати розділи опису шаблону процесу.
6. Які існують шаблони процесу в MS VSTS?
7. Описати шаблон MSF for Agile Software Development.
8. Описати шаблон Scrum.
9. Чим відрізняються шаблон MSF for Agile Software Development від шаблону Scrum?

Литература

1. Barker R. CASE-method. Entity Relatiionship Modeling Copyrigth ORACLE Corporation UK Limited New York: Publ., 1990. – 312 p
2. Blaha V., Premerlani W, Rumbaugh J. Object-Oriented Modelling and Design Englewood Cliffs, NJ: Prentice Hall. – 1991. – 451 p
3. Booch G., Jacobson I., Rumbaugh J. The Unified Software Development Process N.-Y. –Addison-Wesley, 1999. – 463 p
4. Coad P., Yourdan E. Object-oriented analysis.-Second Edition Prentice Hall. – 1991. – 296 p
5. Constantine L, Yourdan E. Structured Design. Yourden Press Engwood Cliffs.N.J. – 1983
6. Curtis B., Paulk M. Capability Maturity Model for Software, version 1.1 CMU–SEI–93–24, Soft. Engin. Institute, Pittsburg PA 15213, Feb. – Pittsburg. – 82 p
7. DeMarko D.A., McGovan R.L. SADT: Structured Analysis and Design Technique New York: Mcgray Hill, 1988. – 378 p
8. Gibbs W.W., «Software's Chronic Crisis», Scientific Am., 1994, Sept.
9. Goguen J.A., Pinheiro F.A.C. An Object-Oriented tool for Tracing Requirements Software, Mach 1996, № 3.
10. Griss M., Jacobson I., Jonsson P. Software Reuse N.-Y. – Addison-Wesley, 1997. – 497 p
11. Guckkenheimer S., Peter J. Software Engineering With Microsoft Visual Studio. Team System Adison Wesley, 2006. – 273 p.
12. IEEE Std. 1012-1998, Standard for Software Verification and Validation, IEEE Press, Piscataway, N.J., 1998
13. IEEE Std. 1016-1998, Recommended Practice for Software Design Descriptions, IEEE Press, Piscataway, N.J., 1998
14. IEEE Std. 1058-1998, Standard for Software Project Management Plans, IEEE Press, Piscataway, N.J., 1998

15. IEEE Std. 1220-1998, Standard for Application and Management of the System Engineering Process, IEEE Press, Piscataway, N.J., 1998
16. IEEE Std. 610.12-1990, Standard Glossary of Software Engineering Terminology, IEEE Press, Piscataway, N.J., 1990
17. IEEE Std. 830-1998, Recommended Practice for Software Requirements Specifications, IEEE Press, Piscataway, N.J., 1998
18. IEEE, Software Engineering Standards Collection, vols. 1-4, IEEE Press, Piscataway, N.J., 1999
19. ISO/IEC 12207: 1995. Information technology – Software life cycle processes. Информационные технологии – Процессы жизненного цикла программного обеспечения
20. ISO/IEC 9126, Information Technology – Software quality characteristics and metrics (Part 1–4) 1997
21. ISO/IEC TR 15504, Information Technology – Software Process Assessment (Part 1–9)
22. Jackson M. Software requirement & specifications Wokingham, England: Addison–Wesley, ACM Press Books, 1995. – 228 p
23. Jackson M., Zave P. Four Dark Corners of Requirements Engineering ACM Transactions on Software Engineering, January 1997, № 1.
24. Jacobson I. Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing New York: AddisonWesley Publ. Co., 1994. – 529 p
25. Jotterbarn D., Miller K., Rogerson S. Software Engineering CODE of Ethic is Approved Com. of the ACM.V. 42. – N 10. – 1999. – P. 102–107
26. Koznov D.V., Pliskin M.Y.. 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.
27. Martin J., Odell J.J. Object-oriented analysis and design Prentice Hall. – 1992. – 367p
28. McConnel S., Tripp L. Professional Software Engineering: Fact or

Fiction? IEEE Software. – Nov.– Dec. 1999. – P. 13–18

29. Pfleeger S.L. Software Engineering. Theory and practice Printice Hall: Upper Saddenle River, New Jersey, 1998. – 576 p
30. Royce W.W. «Software Systems Engineering», seminar presented as part of the course titled Management of Software Acquisition, Defense Systems Management College, Fort Belvoir, Va., 1981-1988
31. Schardt J.A. Assentials of Distributed Object Design M.S.E. Advanced Concepts Center. – 1994. – p. 225 –234
32. Thayer R.H. Software Engineering Project Management, 2 nd. Ed IEEE CS Press, Los Alamitos, Calif., 1997. – 391 p
33. Thayer R.H., «Software System Engineering: A Tutorial», Software Engineering Volume 1: The Development Process, 2nd ed., R.H. Thayer and M. Dorfman, eds., IEEE CS Press, Los Alamitos, Calif., 2002
34. Yourdan E. Modern Srtucrured Analysis New York: Yourdan Press / Prentice Hall, 1988. – 297 p
35. Андон Ф.И., Коваль Г.И., Коротун Т.М., Лаврищева Е.М. Суслов В.Ю. Основы инженерии качества программных систем Киев: Академпериодика.– Второе изд., 2007. – 680 с
36. Андон Ф.И., Резниченко В.А, Яшунин А.Е. Логические модели интеллектуальных информационных систем Киев: Наук.думка, 1999. – 396 с
37. Бабенко Л.П., Лаврищева Е.М Основы программной инженерии. Учебник Киев: Знание, 2001. – 269 с
38. Бабич А.В. Введение в UML. − Електронний ресурс. − http://www.intuit.ru/
39. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. — Пер. с англ. — СПб.: Символ-Плюс, 2001. — 304 с.
40. Брукс Ф.П. Как проектируются и создаются программные комплексы: мифический человеко-месяц М.: Наука, 1979. – 321 с

41. Буч Г, Джекобсон А, Рамбо Дж. UML: специальный справочник СПб.: Питер. – 2002. – 656 с
42. Буч Г. Объектно-ориентированное проектирование. – 3-е издание М.: Бином, 1998. – 560 с
43. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++. 2-е изд.: Пер. с англ. − М.: Издательство Бином, СПб.: Невский диалект, 1999.
44. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя.: Пер. с англ. − М.: ДМК, 2000. − 432с.
45. Вендеров А.М. Проектирование программного обеспечения экономических информационных систем. Учебник М.: Финансы и статистика, 2000. – 347 с
46. Вигерс К.И. Разработка требований к ПО М.: Русская редакция Microsoft, 2004. – 575 с.
47. Гамма Э. Приемы объектно - ориентированного проектирования. Паттерны проектирования. СПб. Питер, 2006. - 366 с.
48. Грищенко В.Н., Лаврищева Е.М. Области знаний программной инженерии – SWEBOK и подход к обучению этой дисциплины Управляющие системы и машины.– 2005. – №1.– С.38–54
49. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. − М.: Вильямс, 2003. − 192 с.
50. Коберн А. - Современные методы описания функциональных требований к системам. Лори, 2011 г. – 288 с.
51. Кознов Д.В, Кириленко Я.А. Опыт сочетания теории и практики в обучении программной инженерии . Труды III Международная научно-практическая конференция "Современные информационные технологии и ИТ-образование" 2008г. http://2008.itedu.ru/pages/Conference-works.
52. Кознов Д.В. Визуальное моделирование: теория и практика. − Електронний ресурс. − http://www.intuit.ru/department/se/vismodtp/

53. Кознов Д.В.. Методика обучения программной инженерии на основе карт памяти. Системное программирование. / Вып. 3, под ред. А.Н.Терехова и Д.Ю.Булычева. СПб.: Изд. СПбГУ, 2008. С. 121-140. http://www.sysprog.info.
54. Коротун Т.М, Лаврищева Е.М. Построение процесса тестирования программных систем. Проблемы программирования. – 2002. – № 1–2. – С. 272–281
55. Лаврищева Е.М. Методы и средства инженерии программного обеспечения. − Електронний ресурс. − https://www.intuit.ru/studies/courses/2190/237/info
56. Лаврищева Е.М. Основные направления исследований в програ-ммной инженерии и пути их развития Проблемы програмирования. – 2003. – № 3–4. – С. 44–58
57. Лаврищева Е.М. Проблематика программной инженерии Киев: Знание, 1991. – 19 с
58. Ларман К., Применение UML 2.0 и шаблонов проектирования (3-е издание) Вильямс. 2006. – 496 с.
59. Леоненков А. В. Самоучитель UML / А. В. Леоненков. − СПб. : БХВ-Петербург, 2004. − 432 с.
60. Леоненков А.В. Нотация и семантика языка UML. − Електронний ресурс. − http://www.intuit.ru/department/pl/umlbasics/
61. Леонов И.В. Введение в методологию разработки программного обеспечения при помощи Rational Rose Ескейп, 2004. – 301 с.
62. Леффингуэлл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. Вильямс 2002. – 448 с.
63. Орлов С. Технологии разработки программного обеспечения: Учебник / С. Орлов. — СПб.: Питер, 2002. — 464 с.: ил.
64. Орлов С.А. Технологии разработки программного обеспечения. Учебник для Вузов СПб.: Питер, 2002. – 463 с

65. Орфали Р., Харки Д. Эдвардс Дж. Основы CORBA Из-во «Малип», М.: 1999. –317с
66. Программы следующего десятилетия. Открытые системы. – Декабрь, 2001. – С.60 –71
67. Соммервил И. Инженерия программного обеспечения. 6-е издание М. – СПб. – Киев, 2002. – 623 с
68. Фреге Г. Логика и логическая семантика М.: Аспект–пресс, 2000. – 512 с
69. Чернецки К., Айзенекер У. Порождающее программирование. Методы, инструменты, применение Издательский дом «Питер». – М. – СПб. – Харьков – Минск, – 2005. – 730 с
70. Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях Киев: – Диалектика, 1993. – 240 с
71. Эммерих В. Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI М.: Мир, 2002. – 510 с.