ФАКУЛЬТЕТ АВТОМАТИКИ, ТЕЛЕМЕХАНІКИ ТА ЗВ'ЯЗКУ

Кафедра спеціалізованих комп'ютерних систем

В.С. Коновалов, К.Є. Радоуцький

СУЧАСНІ ПРИНЦИПИ І МЕТОДИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій
з дисципліни
«СИСТЕМНЕ ПРОГРАМУВАННЯ»
Частина 2

Харків – 2015

Коновалов В.С., Радоуцький К.Є. Сучасні принципи і методи проектування програмного забезпечення: Конспект лекцій. – Харків: УкрДАЗТ, 2015. – Ч. 2. – 109 с.

У конспекті лекцій викладено сучасні принципи і методи проектування програмного забезпечення, описано проблеми розроблення програмного забезпечення (ПЗ), складність його розроблення, вимоги до ПЗ.

Розглянуто стратегії й методи проектування ПЗ, основи конструювання і техніки тестування, супроводження програмного забезпечення.

Конспект лекцій може використовуватися при підготовці до виконання лабораторних робіт і самостійного вивчення матеріалу.

Призначено для студентів факультету АТЗ спеціальності "Спеціалізовані комп'ютерні системи" і спеціалізації "Комп'ютерні інформаційно-управляючі системи" спеціальності "Автоматика і автоматизація на транспорті" при вивченні дисциплін "Системне програмування", "Системне програмне забезпечення", "Проектування програмного забезпечення комп'ютерних систем" і "Мікропроцесорні пристрої".

Іл. 6, табл. 2, бібліогр.: 14 назв.

Конспект лекцій розглянуто і рекомендовано до друку на засіданні кафедри спеціалізованих комп'ютерних систем 16 травня 2014 р., протокол № 16.

Рецензент

проф. С.В. Лістровий

В.С. Коновалов, К.Є. Радоуцький

СУЧАСНІ ПРИНЦИПИ І МЕТОДИ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Конспект лекцій
з дисципліни
«СИСТЕМНЕ ПРОГРАМУВАННЯ»
Частина 2

Відповідальний за випуск Коновалов В.С.

Редактор Ібрагімова Н.В.

Підписано до друку 19.09.14 р.

Формат паперу 60x84 1/16. Папір писальний.

Умовн.-друк.арк. 5,0. Тираж 50. Замовлення №

Видавець та виготовлювач Українська державна академія залізничного транспорту,
61050, Харків-50, майдан Фейєрбаха, 7.
Свідоцтво суб'єкта видавничої справи ДК № 2874 від 12.06.2007 р.

Зміст

Вступ. Класифікація програмного забезпечення

Програмне забезпечення – це найважливіша зі складових частин комп'ютерного забезпечення, яка включає комп'ютерні програми і дані, що призначені для розв’язання певного кола задач і зберігаються на машинних носіях. Програмне забезпечення є або даними для використання в інших програмах, або алгоритмом, реалізованим у вигляді послідовності інструкцій для процесора.

У комп'ютерному жаргоні часто використовується слово «софт» від англійського software, яке в цьому значенні вперше застосував у статті American Mathematical Monthly математик з Прінстонського університету Джон Тьюки (John W. Tukey) у 1958 р. У галузі обчислювальної техніки і програмування програмне забезпечення – це сукупність всієї інформації, даних і програм, які обробляються комп'ютерними системами.

Програмне забезпечення класифікують, як правило, за способом розповсюдження і призначенням. У класифікації за способом розповсюдження виділяють такі види програмного забезпечення (ПЗ):

- комерційне програмне забезпечення (Commercial Software) – програмне забезпечення, створене комерційною організацією з метою отримання прибутку;
- умовно-безкоштовне програмне забезпечення (Shareware або Trial Software) – програмне забезпечення, яке має ті або інші обмеження на його використання. Основний принцип Shareware – «спробуй, перш ніж купити» (try before you buy). Програма, поширювана як shareware, надається користувачам безкоштовно. Протягом певного терміну (що становить звичайно 30 днів) користувач може використовувати програму, тестувати її, освоювати її можливості, але після закінчення цього терміну він зобов'язаний або купити її, або видалити зі свого комп'ютера;
- безкоштовне програмне забезпечення (Freeware) – це безкоштовне програмне забезпечення, поширюване без вихідних кодів;
- вільне програмне забезпечення (Free Software) – це програмне забезпечення, до складу якого входять і вихідні тексти даного ПЗ. Відносно вільного ПЗ користувач володіє «чотирма

свободами», які сформулював Річард Столлман у 1970-х рр.: запускати, вивчати, поширювати і покращувати програму. Слід підкреслити, що ці принципи оговорюють тільки доступність програм для загального використання, критики і поліпшення, але ніяк не обумовлюють пов'язані з розповсюдженням програм грошові відносини, у тому числі не припускають і безкоштовності. Відкритий доступ до вихідних текстів програм є ключовою ознакою вільного ПЗ, тому запропонований дещо пізніше Еріком Реймондом термін «open source software» (ПЗ з відкритим вихідним текстом) декому здається навіть більш вдалим для позначення феномена вільного програмного забезпечення, ніж спочатку запропонований Столлманом «free software»;

- шпигунське програмне забезпечення (Spyware) – програмне забезпечення, що звичайно розповсюджується разом з іншим корисним і займається збиранням інформації на комп'ютері користувача і відсиланням її творцю;
- застаріле програмне забезпечення (Abandonware) – програмне забезпечення, яке більше не виставляється на продаж компанією–виробником. Частіше за все Abandonware розповсюджується безкоштовно (як freeware), іноді — платно (як shareware);
- Adware – вид програмного забезпечення, при використанні якого користувачу примусово показується реклама (приклад – «рідний клієнт» ICQ, який, на відміну, наприклад, від Miranda IM або Quip, під час своєї роботи рекламує ті або інші товари і послуги). Як правило, засоби, отримані від реклами, ідуть на подальший розвиток цього ПЗ. До даної категорії можуть потрапляти програми з будь-якої іншої категорії – freeware, commercial software і т. ін.;
- Careware (charityware) – вид умовно-безкоштовного програмного забезпечення shareware. Автор даного виду ПЗ вимагає, щоб оплата за нього йшла на добродійність. Автором концепції Careware вважається Поль Лютус (Paul Lutus).

За призначенням програмне забезпечення класифікують так:

- системне;
- прикладне;
- спеціалізоване.

Системне програмне забезпечення — це набір програм, які управляють компонентами обчислювальної системи, такими як процесор, комунікаційні і периферійні пристрої, а також які призначені для забезпечення функціонування і працездатності всієї системи.

Прикладне програмне забезпечення – програми, призначені для виконання певних призначених для користувача задач і розраховані на безпосередню взаємодію з користувачем. На відміну від прикладного, системне програмне забезпечення використовується для забезпечення роботи комп'ютера самого по собі і виконання прикладних програм.

Спеціалізоване ПЗ можна представити, наприклад, такими розділами.

Промислове програмне забезпечення (Enterprise software) – служить для управління процесами і потоками даних у крупних організаціях. Часто промислове ПЗ є цілим пакетом окремих програм. Основними виробниками промислового ПЗ у світі є, як правило, крупні транснаціональні софтверні гіганти, такі як Microsoft, Adobe Systems, Oracle Corporation, SAP.

До промислового ПЗ належать:

1 Інформаційне програмне забезпечення (information worker software) – служить для створення і управління різного роду інформаційними даними в організаціях. До такого ПЗ належать:

- системи управління ресурсами (СУР) – різні аккаунтні системи (наприклад, WebMoney), системи табельного обліку (облік робочого часу по турнікетах) і т. ін.;
- системи управління персональними даними (СУПД) – електронні органайзери (MS Outlook);
- системи підготовки документації (СПД) – текстові процесори (OpenOffice Writer. MS Word), програми для побудови різного роду схем і діаграм (ConceptDraw, MS Visio), програми для підготовки публікацій (Adobe InDesign, Apple Pages, MS Publisher) і презентацій (OpenOffice Impress, MS PowerPoint) і т. ін.;
- науково-аналітичні системи (НАС) – різні CAD-системи (AutoCAD, MathCAD), системи для проведення інженерних розрахунків (Maple, MATHLAB), системи статистичного аналізу (Statistical Lab, SHAZAM) і т. ін.;

• системи колективної роботи (СКР) – різні програми обміну повідомленнями (ICQ, Skype), засоби забезпечення колективного розроблення (MS VSS, CVS), wiki-інструментарій і т. ін.

2 Програмне забезпечення для доступу до контенту (content access software) – служить для перетворення і відображення різного роду цифрової інформації (контенту) у зручному для людського сприйняття вигляді. До даного ПЗ належать:

• веб-браузери – програми для проглядання Інтернет-сторінок (MS Internet Explorer, MyIE, Mozilla Firefox, Opera);

• медіа-плеєри – програми для програвання мультимедіа-файлів, тобто файлів, які містять у собі різні види контенту, наприклад відео і аудіо (MS Media Player, Media Player Classic, WinAMP, WinDVD, Macromedia Flash Player);

• всі види розважального програмного забезпечення – ігри, скринсейвери, електронні тварини і т. ін.

3 Освітнє програмне забезпечення (educational software) – служить для навчання і підвищення рівня знань користувачів. До даної категорії ПЗ належать:

• навчальні розважальні програми (edutainment) – дані програми дозволяють в ігровій манері розвивати в користувачів ті або інші навики чи знання (клавіатурні тренажери, програми поліпшення пам'яті);

• програми, що перевіряють отримані знання, програми для проведення різного роду електронних атестаційних тестів і т. ін.;

• електронні книги – електронні словники і перекладачі (Lingvo, Prompt), електронні підручники, електронні енциклопедії.

4 Імітаційне програмне забезпечення (simulation software) – служить для імітації фізичних або абстрактних систем з дослідницькою, навчальною або промисловою метою. До даного ПЗ відносять:

• програми тривимірної візуалізації – дозволяють відображати в тривимірному вигляді ті або інші фізичні об'єкти (3D Max Studio, Maya);

• програми моделювання соціальних процесів – дозволяють прогнозувати розвиток різних соціальних процесів (EcoLab, Multi-Agent Simulation Suite);

- програми-симулятори управління – імітують управління тим або іншим транспортним засобом, виробничим процесом (імітація пульта управління) і т. ін.

5 Інструментальним програмним забезпеченням є програми або цілі пакети програм, призначені для розроблення нових програм. Сучасне інструментальне ПЗ – це цілі інтегровані середовища (англ. IDE, Integrated Development Environment) або системи програмних засобів, що використовуються програмістами для розроблення програмного забезпечення. Звичайно середовище розроблення включає текстовий редактор, компілятор і/або інтерпретатор, засоби автоматизації збирання і налагоджувач. Іноді також містить систему управління версіями і різноманітні інструменти для спрощення конструювання графічного інтерфейсу користувача. Багато сучасних середовищ розроблення також включають браузер класів, інспектор об'єктів і діаграму ієрархії класів – для використання при об'єктно-орієнтованому розробленні ПЗ. Хоча і існують середовища розроблення, призначені для декількох мов, такі як Eclipse або Microsoft Visual Studio, звичайно середовище розроблення призначається для однієї певної мови. Приклади середовищ розроблення – Sun Studio, Turbo Pascal, Borland C++, GNU toolchain, DrPython, Borland Delphi, Dev-C++, Lazarus. Окремий випадок ICP – середовища візуального розроблення, які включають можливість візуального редагування інтерфейсу програми.

У минулому широко застосовувалася класифікація програмістів як прикладних і системних. Прикладним називають програміста, програми якого призначені для розв’язання прикладної задачі, що задовольняє потреби кінцевого користувача і, за задумом класифікації, лежить поза комп'ютерною сферою. Системним називається програміст, програми якого призначені для забезпечення роботи комп'ютера і використовуються іншими комп'ютерними фахівцями.

На сьогодні дана класифікація значною мірою втратила актуальність, оскільки експлуатація комп'ютерів вийшла за межі кола фахівців, забезпечення роботи комп'ютера перетворилося на одну з основних потреб його користувачів і, таким чином, змістовне розгалуження системного і прикладного програмування здебільшого зникло.

1 Складність розроблення ПЗ

Як і інші традиційні інженерні дисципліни, розроблення програмного забезпечення має справу з проблемами якості, вартості й надійності. Деякі програми містять мільйони рядків вихідного коду, які, як очікується, повинні правильно виконуватися в мінливих умовах. Складність ПЗ порівнянна зі складністю найбільш складних із сучасних машин, таких як літаки.

Коли Грейс Хоппер працювала з комп'ютером Гарвард Марк II у Гарвардському університеті, її колеги виявили моль, що застрягла в реле і в такий спосіб перешкодила роботі пристрою, після чого вона відзначила, що вони «налагоджували» (debug) систему. У такий спосіб почав набувати популярності термін «Баг» — помилка програмного забезпечення.

Розроблення програмного забезпечення може бути розділене на кілька розділів:

1 Вимоги до програмного забезпечення: добування, аналіз, специфікація й ратифікація вимог для програмного забезпечення.

2 Проектування програмного забезпечення: проектування програмного забезпечення засобами Автоматизованого Розроблення Програмного Забезпечення (CASE) і стандарти формату описів, такі як Уніфікована Мова Моделювання (UML).

3 Інженерія програмного забезпечення: створення програмного забезпечення за допомогою мов програмування.

4 Тестування програмного забезпечення: пошук і виправлення помилок у програмі.

5 Обслуговування програмного забезпечення: програмні системи часто мають проблеми сумісності й перенесення, а також потребують наступних модифікацій протягом довгого часу після того, як закінчена їхня перша версія. Підгалузь має справу з цими проблемами.

6 Управління конфігурацією програмного забезпечення: оскільки системи програмного забезпечення дуже складні й модифікуються в процесі експлуатації, їхні конфігурації повинні управлятися стандартизованим і структурованим методом.

7 Управління розробленням програмного забезпечення: управління системами програмного забезпечення має

запозичення з управління проектами, але є нюанси, що не зустрічаються в інших дисциплінах управління.

8 Процес розроблення програмного забезпечення: процес побудови програмного забезпечення гаряче обговорюється серед практиків, основними парадигмами вважаються agile (з англ. моторний, гнучка модель розроблення) або waterfall (модель водоспаду, каскадний метод).

9 Інструменти розроблення програмного забезпечення, методика оцінки складності системи, вибору засобів розроблення й застосування програмної системи.

10 Якість програмного забезпечення: методика оцінки критеріїв якості програмного продукту й вимог щодо надійності.

1.1 Процес і методологія

Протягом декількох десятиліть стоїть задача пошуку повторюваного, передбачуваного процесу або методології, яка б поліпшила продуктивність, якість і надійність розроблення. Одні намагалися систематизувати й формалізувати цей, очевидно, непередбачений процес. Інші застосовували до нього методи управління проектами й методи програмної інженерії. Треті вважали, що без постійного контролю з боку замовника розроблення ПЗ виходить з-під контролю, забираючи зайвий час і кошти.

Досвід управління розробленням програм відображається у відповідних стандартах. Якщо при розробленні використовується кілька стандартів і нормативних документів, то має сенс скласти профіль.

Інформатика як наукова дисципліна пропонує й використовує на базі методів структурного програмування технологію надійного розроблення програмного забезпечення, використовуючи тестування програм і їхню верифікацію на основі методів доказового програмування для систематичного аналізу правильності алгоритмів і розроблення програм без алгоритмічних помилок.

Дана методологія спрямована на розв’язання задач на ЕОМ, аналогічна технології розроблення алгоритмів і програм, використовуваній на олімпіадах з програмування вітчизняними

студентами й програмістами з використанням тестування й структурного псевдокоду для документування програм у корпорації IBM з 1970-х років.

Методологія структурного проектування програмного забезпечення може використовуватися з застосуванням всіляких мов і засобів програмування для розроблення надійних програм різного призначення. Одним з таких проектів було розроблення бортового програмного забезпечення для космічного корабля «Буран», у якому вперше використовувався бортовий комп'ютер для автоматичного управління апаратом, що виконав успішний старт і посадку космічного корабля.

При виборі методології розроблення програмного забезпечення слід керуватися тим, що складність методології порівнянна зі складністю структури програмного продукту, невиправдана для такого продукту і тільки невиправдано збільшить вартість розроблення.

1.2 Учасники процесу розроблення ПЗ

Учасниками процесу розроблення є:

– користувач;
– замовник;
– розробник проекту;
– розробник;
– постачальник.

1.3 Проблеми розроблення ПЗ

Найпоширенішими проблемами, що виникають у процесі розроблення ПЗ, вважають:

– нестачу прозорості. У будь-який момент часу складно сказати, у якому стані знаходиться проект і який відсоток його завершення. Дана проблема виникає при недостатньому плануванні структури (або архітектури) майбутнього програмного продукту, що найчастіше є наслідком відсутності достатнього фінансування проекту: програма дійсно потрібна, але скільки часу займе розроблення, які етапи, чи можна якісь етапи

виключити або заощадити — наслідком цього процесу є те, що етап проектування скорочується;

– нестачу контролю. Без точної оцінки процесу розроблення зриваються графіки виконання робіт і перевищуються встановлені бюджети. Складно оцінити об'єм виконаної роботи і об'єм, що залишився. Дана проблема виникає на етапі, коли проект, завершений більш ніж наполовину, продовжує розроблятися після додаткового фінансування без оцінки ступеня завершеності проекту;

– нестачу трасування;

– нестачу моніторингу. Неможливість спостерігати хід розвитку проекту не дозволяє контролювати хід розроблення в реальному часі. За допомогою інструментальних засобів менеджери проектів приймають рішення на основі даних, що надходять у реальному часі. Дана проблема виникає в умовах, коли вартість навчання менеджменту володінню інструментальними засобами порівнянна з вартістю розроблення самої програми;

– неконтрольовані зміни. У споживачів постійно виникають нові ідеї щодо розроблюваного програмного забезпечення. Вплив змін може бути суттєвим для успіху проекту, тому важливо оцінювати пропоновані зміни й реалізовувати тільки схвалені, контролюючи цей процес за допомогою програмних засобів. Дана проблема виникає внаслідок небажання кінцевого споживача використовувати ті або інші програмні середовища. Наприклад, коли при створенні клієнт-серверної системи споживач висуває вимоги до операційної системи не тільки на комп'ютерах-клієнтах, але й на комп'ютері-сервері;

– недостатню надійність. Найскладніший процес — пошук і виправлення помилок у програмах на ЕОМ. Оскільки кількість помилок у програмах заздалегідь невідома, то заздалегідь невідома й тривалість налагодження програм і відсутність гарантій відсутності помилок у програмах. Слід зазначити, що залучення доказового підходу до проектування ПЗ дозволяє виявити помилки в програмі до її виконання. У цьому напрямку багато працювали Кнут, Дейкстра і Вірт. Професор Вірт при розробленні Паскаля і Оберона за рахунок строгості їхнього синтаксису добився математичного доказу завершеності й

правильності програм, написаних цими мовами. Особливо великий внесок у дисципліну програмування вніс Дональд Кнут. Його чотиритомник «Искусство программирования» є необхідною для кожного серйозного програміста книгою. Проблема недостатньої надійності виникає при неправильному виборі засобів розроблення. Наприклад, при спробі створити програму, що вимагає засобів високого рівня, за допомогою засобів низького рівня, або при спробі створити засіб автоматизації з СУБД на асемблері. У результаті вихідний код програми виходить занадто складним і таким, що погано піддається структуруванню;

– відсутність гарантій якості й надійності програм через відсутність гарантій відсутності помилок у програмах аж до формальної здачі програм замовникам. Дана проблема не є проблемою, що стосується винятково розроблення ПЗ. Гарантія якості — це проблема вибору постачальника товару (не продукту).

2 Вимоги до програмного забезпечення

Вимоги до програмного забезпечення (сукупність тверджень щодо атрибутів, властивостей або якостей програмної системи, що підлягає реалізації) створюються в результаті аналізу вимог, що висуваються до кінцевого продукту.

Вимоги можуть виражатися у вигляді текстових тверджень і графічних моделей.

У класичному технічному підході сукупність вимог використовується на стадії проектування ПЗ. Вимоги також використовуються в процесі перевірки ПЗ, тому що тести ґрунтуються на певних вимогах.

Етапу розроблення вимог, можливо, передувало техніко-економічне обґрунтування, або концептуальна фаза аналізу проекту. Фаза розроблення вимог може бути поділена на виявлення вимог (збір, розуміння, розгляд і з'ясування потреб зацікавлених осіб), аналіз (перевірка цілісності й закінченості), специфікацію (документування вимог) і перевірку правильності.

2.1 Види вимог за рівнями

Бізнес-вимоги визначають призначення ПЗ, описуються в документі про бачення (vision) і границі проекту (scope).

Користувацькі вимоги визначають набір користувацьких задач, які повинна розв’язувати програма, а також способи (сценарії) їх розв’язання в системі. Користувацькі вимоги можуть виражатися у вигляді фраз тверджень, у вигляді способів застосування (use case), користувацьких історій (user story), сценаріїв взаємодії (scenario).

Функціональні вимоги охоплюють передбачувану поведінку системи, обумовлюючи дії, які система здатна виконувати, описуються в системній специфікації (англ. system requirement specification, SRS).

2.2 Види вимог за характером

Функціональний характер — вимоги до поведінки системи:

– бізнес-вимоги;

– користувацькі вимоги;

– функціональні вимоги.

Нефункціональний характер — вимоги до характеру поведінки системи:

– бізнес-правила — визначають обмеження, що з'являються з предметної області й властивостей об'єкта, що автоматизується (підприємства);

– системні вимоги і обмеження — визначення елементарних операцій, які повинна мати система, а також різних умов, які вона може задовольняти. До системних обмежень належать обмеження на програмні інтерфейси, вимоги до атрибутів якості, вимоги до застосовуваного обладнання й ПЗ;

– атрибути якості;

– зовнішні системи й інтерфейси;

– обмеження.

2.3 Джерела вимог

Джерелами вимог є:

– федеральне й муніципальне галузеве законодавство (конституція, закони, розпорядження);

– нормативне забезпечення організації (регламенти, положення, статути, накази);

– поточна організація діяльності об'єкта автоматизації;

– моделі діяльності (діаграми бізнес-процесів);

– представлення й очікування споживачів і користувачів системи;

– журнали використання існуючих програмно-апаратних систем;

– конкуруючі програмні продукти.

2.4 Характеристики якісних вимог

Характеристики якісних вимог по-різному визначені різними джерелами. Загальновизнаними є наведені в таблиці 2.1 характеристики.

Таблиця 2.1

Характеристика	Пояснення
1	2
Одиничність	Вимога описує одну й тільки одну річ
Завершеність	Вимога повністю визначена в одному місці й уся необхідна інформація присутня
Послідовність	Вимога не суперечить іншим вимогам і повністю відповідає зовнішній документації
Атомарність	Вимога «атомарна», тобто вона не може бути розбита на низку більш детальних вимог без втрати завершеності
Відслідковуваність	Вимога повністю або частково відповідає діловим потребам, які заявлено зацікавленими особами, і документована
Актуальність	Вимога не стала застарілою з часом
Виконуваність	Вимога може бути реалізована в межах проекту
Недвозначність	Вимога коротко визначена без звертання до технічного жаргону, акронімів та інших прихованих формулювань. Вона виражає об'єктивні факти, а не суб'єктивні думки. Можлива одна й тільки одна інтерпретація.

Продовження таблиці 2.1

1	2
	Визначення не містить нечітких фраз. Використання негативних тверджень і складених тверджень заборонене
Обов'язковість	Вимога представляє визначену зацікавленою особою характеристику, відсутність якої призведе до неповноцінності рішення, яке не може бути зігнороване. Необов'язкова вимога — протиріччя самому поняттю вимоги
Перевірність	Реалізованість вимоги може бути визначено через один із чотирьох можливих методів: огляд, демонстрація, тест або аналіз

2.5 Методи виявлення вимог

Методами виявлення вимог є:

– інтерв'ю, опитування, анкетування;

– мозковий штурм, семінар;

– спостереження за виробничою діяльністю, «фотографування» робочого дня;

– аналіз нормативної документації;

– аналіз моделей діяльності;

– аналіз конкурентних продуктів;

– аналіз статистики використання попередніх версій системи.

2.6 Перевірка вимог

Усі вимоги повинні бути такими, що піддаються перевірці. Найбільш загальноприйнята методика перевірки — тести. Якщо перевірка тестами неможлива, тоді повинен використовуватися інший метод перевірки (аналіз, демонстрація, огляд вимог або огляд дизайну).

Певні вимоги, по своїй суті, є такими, що не піддаються перевірці. Вони включають вимоги, які говорять, що система ніколи не повинна або завжди повинна показувати специфічну

властивість. Належне тестування цих вимог потребувало б нескінченного циклу тестування. Такі вимоги повинні бути перевизначені так, щоб вони стали такими, що піддаються перевірці. Як зазначено вище, всі вимоги повинні бути такими, що піддаються перевірці.

Нефункціональні вимоги є такими, що не піддаються перевірці на програмному рівні, однаково повинні бути збережені як документація намірів клієнта. Такі вимоги до продукту можуть бути перетворені у вимоги до процесу. Наприклад, нефункціональна вимога, щоб ПЗ не містило «таємних ходів», може бути задоволена заміною на вимогу використовувати парне програмування.

2.7 Аналіз вимог

При розробленні вимог часто виникають проблеми двозначності, неповноти і непогодженості окремих вимог. Усунення цих проблем на етапі розроблення вимог коштує на кілька порядків менше, ніж усунення цих самих проблем на пізніших стадіях розроблення. Для вирішення й усунення цих проблем існує процес розроблення вимог.

При розробленні вимог існує технічний компроміс між занадто невизначеними вимогами й вимогами настільки деталізованими, що вони:

– вимагають багато часу для розроблення, іноді навіть ризикують застаріти до кінця розроблення;

– обмежують можливі способи реалізації;

– є занадто дорогими.

2.8 Документування вимог

Вимоги звичайно використовуються як засіб комунікації між різними зацікавленими особами. Це означає, що вимоги повинні бути простими й зрозумілими для звичайних користувачів і розробників. Один загальний спосіб задокументувати вимогу — це написати твердження про те, що саме повинна зробити система.

У зарубіжній і зокрема вітчизняній практиці зустрічаються такі типи документів вимог:

– концепція програми (Vision);

– специфікаця програмного забезпечення (англ. Software Requirements Specification, SRS).

Специфікацію програмного забезпечення нерідко помилково називають технічним завданням. Специфікація вимог є частиною технічного завдання у випадку створення автоматизованих інформаційних систем.

За створення специфікації програмного забезпечення найчастіше в практиці відповідає системний аналітик, іноді — бізнес-аналітик.

Для графічних моделей вимог історично використовувалися діаграми або методології графічного моделювання: ER (IDEF1FX), IDEF0, IDEF3, DFD, UML, OCL, Sysml, ARIS (eepc, VAD).

2.9 Зміна вимог

У загальному випадку вимоги змінюються згодом. Після того як вимоги визначені й схвалені, зміни вимог повинні підпадати під контроль внесення змін. Для багатьох проектів вимоги змінюються до завершення створення системи. Це відбувається частково через складність програмного забезпечення й того факту, що користувачі не знають, що їм потрібно насправді. Ця особливість вимог призвела до появи процесу управління вимогами.

3 Проектування програмного забезпечення

Процес визначення архітектури, компонентів, інтерфейсів та інших характеристик системи або її компонентів називається проектуванням. Результат процесу проектування – дизайн. Розглянуте як процес, проектування є інженерною діяльністю в рамках життєвого циклу (у даному контексті – програмного забезпечення), у якій належним чином аналізуються вимоги для створення опису внутрішньої структури ПЗ, що є основою для

конструювання програмного забезпечення як такого. Програмний дизайн (як результат діяльності з проектування) повинен описувати архітектуру програмного забезпечення, тобто являти собою декомпозицію програмної системи у вигляді організованої структури компонент і інтерфейсів між компонентами. Найважливішою характеристикою готовності дизайну є той рівень деталізації компонентів, який дозволяє зайнятися їхнім конструюванням. Терміни дизайн і архітектура можуть використовуватися взаємозамінно, але частіше говорять про дизайн як про цілісний погляд на архітектуру системи.

Проектування відіграє важливу роль у процесах життєвого циклу створення програмного забезпечення (Software Development Life Cycle), наприклад IEEE[1] і ISO/IEC (ГОСТ Р ИСО.МЭК) 12207. Проектування програмних систем можна розглядати як діяльність, результат якої складається з двох складових частин:

– архітектурний або високорівневий дизайн (software architectural design, top-level design) – опис високорівневої структури й організації компонентів системи;

– деталізована архітектура (software detailed design), що описує кожний компонент у тому обсязі, який необхідний для конструювання.

У результаті консенсусу, прийнятого творцями SWEBOK[2] дана галузь знань не описує всі сутності або поняття, що мають у своїй назві слово “дизайн” або “архітектура”. У 1999 році Том Демарко (Tom Demarco), один з відомих фахівців у програмній інженерії, запропонував термінологічний поділ різних видів дизайну:

– D-Дизайн (D-Design, Decomposition Design) – декомпозиція структури програмного забезпечення у вигляді набору фрагментів або компонент;

---

[1] IEEE (англ. Institute of Electrical and Electronics Engineers) Інститут інженерів з електротехніки і електроніки.

[2] SWEBOK (Software Engineering Body of Knowledge - основи знань побудови програмного забезпечення) — документ, що готується комітетом Software Engineering Coordinating Committee, у який залучено співтовариство IEEE Computer Society. Призначення SWEBOK — в об'єднанні знань з «інженерії програмного забезпечення» (розробл[ення] програмного забезпечення).

– Fp-Дизайн (Fp-Design, Family pattern Design) – сімейство архітектурних вистав, що базуються на шаблонах;

– I-Дизайн (I-Design, Invention Design) – створення високорівневої концепції, бачення того, що собою буде являти програмна система; даний вид дизайну є результатом процесу аналізу вимог і їхньої трансформації в підходи до реалізації.

Якщо обговорювати дану галузь знань у термінах Демарко, проектування програмного забезпечення в розумінні програмної інженерії має на увазі D- і Fp-Дизайн. I-Дизайн більшою мірою стосується роботи з програмними вимогами.

Відповідно дана галузь знань тісно пов'язана з такими сферами програмної інженерії:

– Software Requirements;
– Software Construction;
– Software Maintenance;
– Software Engineering Management;
– Software Quality.

Саму ж галузь знань з проектування програмного забезпечення представлено у вигляді 6 секцій, структурованих за темами.

3.1 Основи проектування

Введемо концепції, поняття й термінологію як основу для розуміння ролі й змісту проектування (як діяльності) і дизайну (архітектури, як результату) програмного забезпечення. Питання, пов’язані з проектуванням ПЗ у вигляді схеми подано на рисунку 3.1.

До них належать мета архітектури, її обмеження, можливі альтернативи, використовувані представлення й рішення.

Наприклад, архітектурний фреймворк[1] – TOGAF [TOGAF, 2003], що розроблений й розвивається консорціумом The Open Group (міжнародна організація зі стандартизації розроблення програмного забезпечення), пропонує такі можливі цілі (goals):

– поліпшення й підвищення продуктивності бізнес-процесів;

[1] Фреймворк (англ. *framework* — каркас, структура) — структура програмної системи; програмне забезпечення, що полегшує розроблення і об'єднання різних компонентів великого програмного проекту.

– зменшення витрат;
– поліпшення операційної бізнес-діяльності;
– підвищення ефективності управління;
– зменшення ризиків;
– підвищення ефективності Іт-організації.

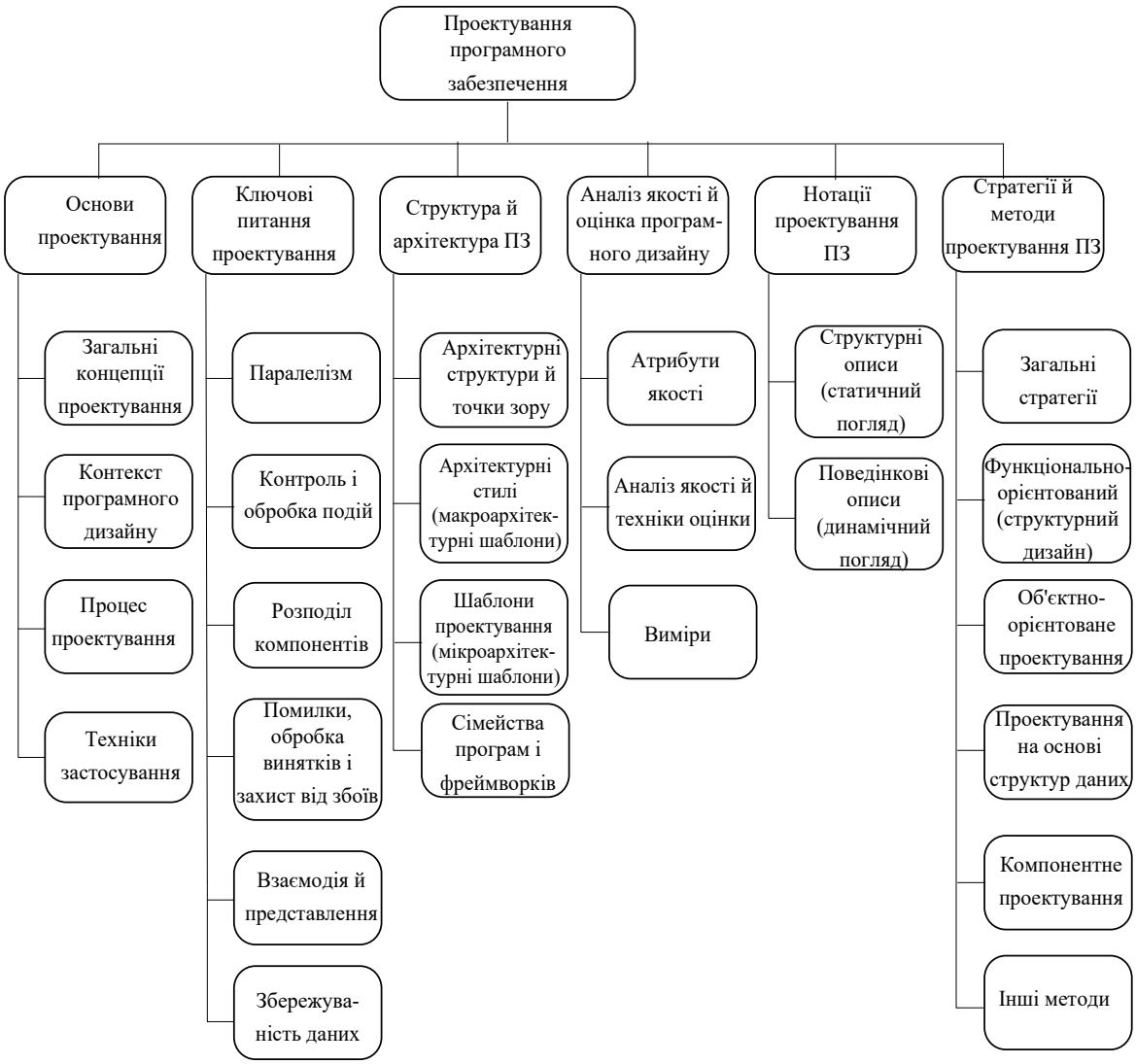

Рисунок 3.1 – Проектування програмного забезпечення

3.1.1 Загальні концепції проектування

Загальні концепції проектування – це:

– підвищення продуктивності роботи користувачів;

– підвищення інтероперабельності (можливості й прозорості взаємодії);

– зменшення вартості підтримки життєвого циклу;

– поліпшення характеристик безпеки;

– підвищення керованості.

3.1.2 Контекст проектування

Для розуміння ролі проектування програмного забезпечення важливо розуміти контекст, у якому здійснюється проектування й використовуються його результати. У якості такого контексту виступає життєвий цикл програмної інженерії, а проектування прямо пов'язане з результатами аналізу вимог, конструюванням програмних систем і їх тестуванням. Стандарти життєвого циклу, наприклад IEEE і ISO/IEC (ГОСТ Р) 12207, приділяють спеціальної увагу питанням проектування й деталізують їх, описуючи контекст проектування – від вимог до тестів.

3.1.3 Процес проектування

Проектування в основному розглядається як двокроковий процес:

- архітектурне проектування – декомпозиція структури (статичної) і організації (динамічної) компонент;
- деталізація архітектури – опис специфічної поведінки й характеристики окремих компонентів.

Виходом цього процесу є набір моделей і артефактів, що містять результати рішень, прийнятих по способах реалізації вимог у програмному коді.

3.1.4 Техніки застосування

Принципи проектування, також називані техніками застосування, є ключовими ідеями й концепціями, розглянутими на фундаментальному рівні в різних методах і підходах до проектування програмного забезпечення.

1 Абстракція.

У контексті проектування програмних систем існує два механізми абстракції – параметризація й специфікування (може інтерпретуватися як деталізація). При цьому абстракція через специфікування буває трьох видів: процедурна абстракція (динамічна, тобто відносно поведінки), абстракція даних (статична, тобто відносно інформації) і абстракція контролю (тобто управління системою й оброблюваною нею інформацією).

Звичайно під абстракцією, як результатом процесу абстракції, розуміють модель, що спрощує поставлену проблему до рамок, значущих для заданого контексту.

2 Зв'язність і міцність.

Зв'язність визначає силу зв'язку (часто взаємного впливу) між модулями. Міцність визначає, як той або інший елемент забезпечує зв'язок усередині модуля, внутрішній зв'язок.

Значення оригінальних термінів є дуже близьким і залежно від контексту “зв'язність” і “міцність ” можуть розглядатися як ступінь самодостатності або її відсутності.

Треба особливо підкреслити значущість цих понять, тому що з розвитком сервісно-орієнтованої архітектури (Service-Oriented Architecture, SOA), слабо пов’язаної за своєю природою (тобто зі слабкою “зв'язністю”, “слабкою силою зв'язку” між модулями) порівняно, наприклад з OMG CORBA (Common Object Request Broker Architecture), усе частіше доводиться порівнювати різні підходи й рішення, обумовлені способом і ступенем зв'язності різних модулів, компонент і самих програмних систем.

3 Декомпозиція й розбиття на модулі.

Декомпозиція й розбиття на модулі складних програмних систем проводиться з метою одержання більш дрібних і відносно незалежних програмних компонентів, кожний з яких несе різну функціональність (логічно зв'язані групи функціональності).

4 Інкапсуляція/приховання інформації.

Дана концепція припускає угруповання й упакування (з погляду підготовки до розгортання й експлуатації) елементів і внутрішніх деталей абстракції (тобто моделі) відносно реалізації для того, щоб ці деталі (як малозначущі для використання компонента або з інших причин) були недоступними користувачам елементів (компонент). При цьому в якості “користувача” одного компонента може виступати інший компонент. Більш того, при використанні об'єктно-орієнтованого підходу спадкоємці компонентів можуть не мати доступу до внутрішніх деталей реалізації компонента, який є їхнім предком (залежить від об'єктно-орієнтованої моделі конкретної мови програмування або платформи).

5 Поділ інтерфейсу й реалізації.

Дана техніка припускає визначення компонента через специфікування інтерфейсу, відомого (описаного) і доступного клієнтам (або іншим компонентам), від безпосередніх деталей реалізації.

6 Достатність, повнота й простота.

Цей підхід має на увазі, що створювані програмні компоненти володіють усіма необхідними характеристиками, визначеними абстракцією (моделлю), але не більш того. Тобто не включають функціональність, відсутню в моделі.

Даний принцип особливо яскраво виділений і представлений у вигляді рекомендованих практик (best practices) методологій гнучкого моделювання й екстремального програмування, де “усе, що треба, але ні грамом більше” лежить в основі самої концепції “прагматичного” підходу (і на стадії моделювання, і відносно реалізації в коді). В оригіналі цей принцип звучить як YAGNI – “You Aren’t Going to Need It”, тобто “не роби цього, поки не знадобиться”.

3.2 Ключові питання проектування

Якоюсь мірою даний розділ слід би назвати “ключові проблеми”. Як проводити декомпозицію? Як організувати й об'єднати компоненти в єдину систему? Як забезпечити необхідну продуктивність? Нарешті, як забезпечити прийнятну якість системи? Усе це – фундаментальні питання й проблеми проектування, незалежно від використовуваних при проектуванні підходів.

1 Паралелізм.

Ця тема охоплює питання, підходи й методи організації процесів, задач і потоків для забезпечення ефективності, атомарності, синхронізації й розподілу (за часом) обробки інформації.

2 Контроль і обробка подій.

У самій назві цієї теми закладений комплекс обговорюваних питань. Зокрема дана тема стосується й неявних методів обробки подій, часто реалізованих у вигляді функції зворотного виклику (call-back), як однієї з фундаментальних концепцій обробки подій.

3 Розподіл компонентів.

Розподіл (і виконання) по різних вузлах обробки в термінах апаратного забезпечення, мережної інфраструктури й т. п. Одне з найважливіших питань даної теми – використання єднального програмного забезпечення (middleware[1]).

4 Обробка помилок, виняткових ситуацій і забезпечення відмовостійкості.

Питання теми, як не дивно, формулюється досить просто – як запобігти збоїв або, якщо збій все-таки відбувся, забезпечити подальше функціонування системи.

5 Взаємодія й подання.

Тема стосується питань подання інформації користувачам і взаємодії користувачів із системою з погляду реакції системи на дії користувачів. У цій темі йдеться про реакцію системи у відповідь на дії користувачів і організації, її відгук з погляду внутрішньої організації взаємодії, наприклад у рамках популярної концепції Model-View-Controller.

У жодному разі не треба плутати дану тему з питаннями організації користувацького інтерфейсу, що є частиною “Ергономіки програмного забезпечення” – Software Ergonomics.

6 Збережуваність даних.

[1] Часто middleware переводять як “проміжне програмне забезпечення”. Такий варіант перекладу, на жаль, розглядає єднальне ПЗ в другорядній – “проміжній” – ролі. Безумовно, можна не погодитися з таким трактуванням, однак багаторічна практика в обговоренні архітектурних питань із різними фахівцями демонструє саме такий погляд користувачів, не знайомих або тих, що не мають успішного досвіду розроблення й експлуатації розподілених систем.

Саме збережуваність, а не схоронність, тому що тема стосується не доступу до баз даних, як такого, а також не гарантій схоронності інформації. Суть питання – як повинні оброблятися дані, що “довго живуть”.

3.3 Структура й архітектура програмного забезпечення

У строгому значенні архітектура програмного забезпечення (software architecture) – опис підсистем і компонент програмної системи, а також зв'язків між ними. Архітектура намагається визначити внутрішню структуру одержуваної системи, задаючи спосіб, яким система організована або конструюється.

У середині 1990-х рр. на хвилі поширення клієнт-серверного підходу й початку його трансформації в “багатоланковий клієнт-сервер”, покликаний забезпечити централізоване розгортання й управління загальною (для клієнтських додатків) бізнес-логікою, питання організації архітектури програмного забезпечення стали складатися в самостійну й досить велику дисципліну. У результаті сформувалася точка зору на архітектуру не тільки як на додаток до конкретної програмної системи, але й розвинувся погляд на архітектуру, як на додаток загальних (generic) принципів організації програмних компонентів. У підсумку уже на сьогоднішній день, на фоні такого розвитку розуміння архітектури, накопичено цілий комплекс підходів і створені (і продовжують створюватися й розвиватися!) різні архітектурні “фреймворки”, тобто систематизовані комплекси методів, практик і інструментів, покликані тією чи іншою мірою формалізувати наявний в індустрії досвід (як позитивний, так і негативний).

Приклади такої систематизації у формі фреймворків:

– TOGAF [TOGAF81, 2003] – The Open Group Architecture Framework (у версії 8.1, уперше опублікованій у грудні 2003 р.; у 2009 р. вийшла версія TOGAF 9);

– модель Захмана – Zachman Framework [Zachman];

– посібник з архітектури електронного уряду E-Gov Enterprise Architecture Guidance [E-Gov, 2002].

3.3.1 Архітектурні структури й точки зору

Будь-яка система може розглядатися з різних точок зору: наприклад, поведінкової (динамічної), структурної (статичної), логічної (задоволення функціональних вимог), фізичної (розосередженість), реалізації (як деталі архітектури представляються в коді) і т. п. У результаті ми одержуємо різні архітектурні представлення. Архітектурне представлення може бути визначено як часткові аспекти програмної архітектури, що розглядають специфічні властивості програмної системи. У свою чергу, дизайн системи – комплекс архітектурних представлень, достатній для реалізації системи й задоволення вимог, пропонованих до системи.

SWEBOK не дає явного визначення, що таке “архітектурна структура”. У той же час це поняття досить важливе. Автори хотіли б запропонувати його тлумачення як застосування архітектурної точки зору й представлення до конкретної системи й опису тих деталей, які необхідні для реалізації системи, але відсутні (через досить загальний погляд) у використовуваному представленні. Таким чином, представлення, концентруючись на заданій підмножині властивостей, є складовою частиною й/або результатом точки зору, а архітектурна структура – подальшою деталізацією відносно проектованої системи.

Модель Захмана [Zachman] є класичним джерелом комплексу архітектурних точок зору й представлень, побудованих у системі координат “питання-рівень деталізації”. Кожне архітектурне представлення є результатом відповіді на питання (як? що? де? і т. п.) у контексті необхідного рівня абстракції (зміст, тобто концепція; бізнес-модель, тобто функціональність і т. д.). Наприклад, фізична модель даних (Physical Data Model) є відповіддю на запитання “що?” у контексті технологічної моделі, а логічна модель даних, відповідаючи на те саме питання, перебуває на один рівень абстракції вище – у контексті системної або логічної моделі.

3.3.2 Архітектурні стилі

Архітектурний стиль, по своїй суті, є мета-моделлю або шаблоном проектування макроархітектури – на рівні модулів, "великоблокового" погляду. Наприклад, архітектура розподіленої сервісно-орієнтованої системи може будуватися в стилі обміну

повідомленнями через відповідні черги повідомлень, може проектуватися на основі ідеї взаємодії між компонентами й додатками через загальну об'єктну шину, а може використовувати концепцію брокера як єдиного вузла пересилання запитів. У той же час на більш концептуальному рівні ми можемо говорити про вибір клієнт-серверного стилю або розподіленого стилю архітектури системи. Таким чином, архітектурний стиль – набір обмежень, що визначають сімейство архітектур, які задовольняють ці обмеження.

3.3.3 Шаблони проектування

Найбільш коротке формулювання того, що таке шаблон проектування, може звучати так: “загальне рішення загальної проблеми в заданому контексті”. Що це означає в реальному житті? Якщо ми прагнемо організувати системи таким чином, щоб існував один і тільки один екземпляр заданого її компонента в процесі роботи з даною системою, ми можемо використовувати шаблон проектування “Singleton”, що описує таку загальну поведінку.

У той час, як архітектурний стиль визначає макроархітектуру системи, шаблони проектування задають мікроархітектуру, тобто визначають окремі аспекти деталей архітектури.

Найчастіше говорять про такі групи шаблонів проектування:

– шаблони створення (Creational patterns) - builder, factory, prototype, singleton;

– структурні шаблони (Structural patterns) - adapter, bridge, composite, decorator, facade, flyweight, proxy;

– шаблони поведінки (Behavioral patterns) - command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor.

3.3.4 Сімейства програм і фреймворків

Один з можливих підходів до повторного використання архітектурних рішень і компонент полягає у формуванні ліній продуктів (product lines) на основі загального дизайну. В об'єктно-орієнтованому програмуванні аналогічне значеннєве

навантаження несуть “фреймворки”, що забезпечують рішення тих самих задач, наприклад, внутрішньої організації компонентів користувацького інтерфейсу або загальної логіки роботи розподілених систем.

3.4 Аналіз якості й оцінка програмного дизайну

3.4.1 Атрибути якості

Існує цілий спектр різних атрибутів, що допомагають оцінити й добитися якісного дизайну. Ці атрибути можуть описувати багато характеристик системи й елементів дизайну як такого: “тестованість”, “переносність”, “модифікованість”, “продуктивність”, “безпека” і т.п.

Важливо розуміти, що обговорювані атрибути стосуються тільки дизайну (як результату), але не проектування (як процесу). У принципі, всі ці атрибути можна розбити на кілька груп:

– застосовні до run-time, тобто до часу виконання системи; наприклад, середній час відгуку системи, що дозволяє оцінити якість дизайну з погляду продуктивності;

– орієнтовані на design-time, тобто такі, що дозволяють оцінювати якість одержуваного дизайну ще на етапі проектування або в загальному випадку аж до тестування включно; наприклад, середня навантаженість класів бізнес-методами (припустимо, бізнес-методів у кожному класі в середньому 30 – цікаво, наскільки легко можна підтримувати, модифікувати й розбудовувати систему з такою внутрішньою структурою?);

– атрибути якості архітектурного дизайну як такого, наприклад концептуальна цілісність дизайну, несуперечність, повнота, завершеність; наприклад, будь-який певний бізнес-метод є викликуваним, тобто створеним не просто тому що може знадобитися в майбутньому, а визначеним відповідно до вимог або необхідним для реалізації дизайну в обраному архітектурному стилі.

Необхідно розуміти, що існують атрибути, які складно виміряти. Наприклад, портованість або безпека. Не варто плутати атрибути якості дизайну з атрибутами якості, що фігурують у ряді вимог, пропонованих до системи. Частина з них може

відображатися одна на одну й нести еквівалентне значеннєве навантаження, деякі можуть бути пов'язані, велика частина атрибутів є специфічною саме для дизайну й не пов'язана з вимогами. Наприклад, якщо ми використовуємо платформу J2EE (Java 2 Enterprise Edition) і орієнтуємося на використання компонентом моделі EJB (Enterprise Javabeans), існують ознаки гарного дизайну, специфічні для даної платформи й компонентної моделі, але абсолютно ніяк не пов'язані з якими-небудь вимогами до створюваної на цій платформі програмної системи. Якщо повернутися до вимірюваних атрибутів якості, вони описуються певними метриками. Наведений вище приклад з кількістю бізнес-методів на клас є метрикою “Виміру” (п. 3.4.3). Ця сама метрика дозволяє оцінити атрибути якості “модифікованість” і “складність” системи.

3.4.2 Аналіз якості й техніки оцінки

В індустрії поширено багато інструментів, техніки й практики, що допомагають добитися якісного дизайну:

– огляд дизайну (software design review); наприклад, неформальний огляд архітектури членами проектної команди;

– статичний аналіз (static analysis); наприклад, трасування з вимогами;

– симуляція й прототипування (simulation and prototyping) – динамічні техніки перевірки дизайну в цілому або окремих його атрибутів якості; наприклад, для оцінки продуктивності використовуваних архітектурних рішень при симуляції навантаження, близького до прогнозованих пікових.

3.4.3 Виміри

Також відомі як метрики. Можуть бути використані для кількісної оцінки очікувань відносно різних аспектів конкретного дизайну, наприклад розміру проекту, структури (її складності) або якості (наприклад, у контексті вимог, пропонованих до продуктивності). Найчастіше усі метрики поділяють на дві категорії:

– функціонально-орієнтовані;

– об'єктно-орієнтовані.

3.5 Нотації проектування

Нотація є угодою про представлення. Часто під нотацією розуміють візуальне (графічне) представлення. Нотація може задаватися:

– стандартом; наприклад, OMG UML – Unified Modeling Language, що розвивається консорціумом OMG (Object Management Group);

– загальноприйнятою практикою; наприклад, в extreme Programming часто використовуються картки функціональної відповідальності й зв'язків класу - Class Responsibility Collaborator або CRC Card (CRC по своїй природі є текстовою, тобто невізуальною нотацією);

– внутрішнім методом проектної команди (“будемо малювати й позначати так...”).

Певні нотації використовуються на стадії концептуального проектування, ряд нотацій орієнтований на створення детального дизайну, більшість можуть використовуватися на обох стадіях. Крім того, нотації частіше використовують у контексті (вибір нотації може бути обумовлений таким контекстом) застосовуваної методології або підходу. Нижче ми будемо розглядати нотації, виходячи з опису структурного (статичного) або поведінкового (динамічного) представлення.

3.5.1 Структурні описи, статичний погляд

Наступні нотації, в основному (але не завжди) є графічними, що описують і представляють структурні аспекти програмного дизайну. Найчастіше вони стосуються основних компонентів і зв'язків між ними (статичних зв'язків, наприклад, таких як відношення “один-до-багатьох”):

– ***мови опису архітектури (Architecture description language, ADL)***: текстові мови, часто – формальні, використовувані для опису програмної архітектури в термінах компонентів і конекторів (спеціалізованих компонентів, що не реалізують функціональність, а забезпечують взаємозв'язок функціональних компонентів між собою і з “зовнішнім світом”);

– ***діаграми класів і об'єктів (Class and object diagrams)***: використовуються для представлення набору класів і статичних зв'язків між ними (наприклад, спадкування);

– ***діаграми компонентів або компонентні діаграми (Component diagrams)***: деякою мірою аналогічні діаграмам класів, однак за специфікою концепції або поняття компонента[1], звичайно, представляються в іншій візуальній формі;

– ***картки функціональної відповідальності й зв'язків класу (Class responsibility collaborator card, CRC)***: використовуються для позначення імені класу, його відповідальності (тобто те, що він повинен робити) і інших сутностей (класів, компонентів, акторів/ролей і т. п.), з якими він пов'язаний; часто їх називають картками “клас-обов'язок-кооперація”;

– ***діаграми розгортання (Deployment diagrams)***: використовуються для представлення (фізичних) вузлів, зв'язків між ними й моделювання інших фізичних аспектів системи;

– ***діаграми сутність-зв'язок (Entity-relationship diagram, ERD або ER)***: використовуються для представлення концептуальної моделі даних, що зберігаються в процесі роботи інформаційної системи;

– ***мови опису/визначення інтерфейсу (Interface Description Languages, IDL)***: мови, подібні мовам програмування опису, що не включають можливостей, логіки системи й призначені для визначення інтерфейсів програмних компонентів (імен і типів експортованих або публікованих операцій);

– ***структурні діаграми Джексона (Jackson structure diagrams)***: використовуються для опису структур даних у термінах послідовності, вибору й ітерацій (повторень);

– ***структурні схеми (Structure charts)***: описують структуру викликів у програмах (який модуль викликає, ким і як викликається).

3.5.2 Поведінкові описи, динамічний погляд

[1] Необхідно відзначити відмінність у поняттях класу (або об'єкта) і компонента: компонент розглядається як фізично реалізований елемент програмного забезпечення, що несе деякою мірою самодостатню логіку й реалізований як конгломерат інтерфейсу і його реалізації (часто у вигляді комплексу класів).

Наступні нотації й мови (частина з яких – графічні, частина – текстові) використовуються для опису динамічної поведінки програмних систем і їх компонентів. Більшість із цих нотацій успішно використовуються для проектування деталей дизайну, але не тільки для цього:

– ***діаграми діяльності або операцій (Activity diagrams)***: використовуються для опису потоків робіт і управління;

– ***діаграми співробітництва (Collaboration diagrams)***: показують динамічну взаємодію, що відбувається в групі об'єктів, і приділяють особливу увагу об'єктам, зв'язкам між ними й повідомленням, якими обмінюються об'єкти за допомогою цих зв'язків;

– ***діаграми потоків даних (Data flow diagrams, DFD)***: описують потоки даних усередині набору процесів (не в термінах процесів операційного середовища, але в розумінні обміну інформацією в бізнес-контексті);

– ***таблиці й діаграми прийняття рішень (Decision tables and diagrams)***: використовуються для представлення складних комбінацій умов і дій (операцій);

– ***блок-схеми й структуровані блок-схеми (Flowcharts and structured flowcharts)***: застосовуються для представлення потоків управління (контролю) і пов'язаних операцій;

– ***діаграми послідовності (Sequence diagrams)***: використовуються для показу взаємодій усередині групи об'єктів з акцентом на часовій послідовності повідомлень/викликів;

– ***діаграми переходу й карти станів (State transition and statechart diagrams)***: застосовуються для опису потоків управління переходами між станами;

– ***формальні мови специфікації (Formal specification languages)***: текстові мови, що використовують основні поняття з математики (наприклад, множини) для строгого й абстрактного визначення інтерфейсів і поведінки програмних компонентів, часто в термінах перед- і постумов;

– ***псевдокод і програмні мови проектування (Pseudocode and program design languages, PDL)***: мови, використовувані для опису поведінки процедур і методів в основному на стадії детального проектування; подібні структурним мовам програмування.

3.6 Стратегії й методи проектування програмного забезпечення

Існують різні загальні стратегії, що допомагають у проведенні робіт із проектування. На відміну від загальних стратегій, методи проектування більш специфічні і в основному пропонують і надають нотації (або набори нотацій) для використання разом із цими методами, а також процеси, яким необхідно слідувати в рамках використовуваного методу.

Таким чином, методи в даному контексті – це не просто якісь "слабоформалізовані" або просто часткові практичні підходи або техніки. Методи тут є більш загальними поняттями, це – методології, сконцентровані на процесі (зокрема, проектування), і такі, що припускають слідування певним правилам і угодам, у тому числі використовуваним засобам відображення. Такі методи корисні як інструмент систематизації (іноді формалізації) і передачі знань у вигляді загального фреймворку (тобто комплексного набору понять, підходів, технік і інструментів) не тільки для окремих фахівців, але для команд і проектних груп програмних проектів.

3.6.1 Загальні стратегії

Це загальноприйняті стратегії, що звичайно часто згадуються:

– "розділяй-і-пануй" і покрокове уточнення;
– проектування "зверху-униз" і "знизу-нагору";
– абстракція даних і приховання інформації;
– ітеративний та інкрементальний підхід;
– та інші.

3.6.2 Функціонально-орієнтоване або структурне проектування

Це один із класичних методів проектування, у якому декомпозиція сфокусована на ідентифікації основних програмних функцій і потім детальному розробленні й уточненні цих функцій "зверху-униз". Структурне проектування зазвичай використовується після проведення структурного аналізу з

застосуванням діаграм потоків даних і зв'язаним описом процесів. Пропонуються різні стратегії й метафори або підходи для трансформації DFD у програмну архітектуру, що подається у формі структурних схем, наприклад порівнюючи управління й поведінку з одержуваним ефектом.

3.6.3 Об'єктно-орієнтоване проектування

Являє собою множину методів проектування, що базуються на концепції об'єктів. Дана область активно еволюціонує з середини 1980-х рр., ґрунтуючись на поняттях об'єкта (сутності), методу (дії) і атрибута (характеристики). Тут головну роль відіграють поліморфізм і інкапсуляція, у той час як у компонентно-орієнтованому підході більше значення надається меті-інформації, наприклад із застосуванням технології відображення. Хоча коріння об'єктно-орієнтованого проектування лежать в абстракції даних (до яких додані поведінкові характеристики), так званий responsibility-driven design або проектування на основі функціональної відповідальності щодо SWEBOK[1] може розглядатися як альтернатива об'єктно-орієнтованому проектуванню.

3.6.4 Проектування на основі структур даних

У даному підході фокус сконцентрований більшою мірою на структурах даних, якими управляє система, ніж на функціях системи. Інженери з програмного забезпечення часто спочатку описують структури даних входів і виходів, а, потім розробляють структуру управління цими даними (або, наприклад, їхні трансформації).

3.6.5 Компонентне проектування

Програмні компоненти є незалежними одиницями, які мають однозначно визначені (well-defined) інтерфейси й залежності (зв'язки) і можуть збиратися й розгортатися незалежно один від одного. Даний підхід повинен розв'язати задачі використання, розроблення та інтеграції таких компонентів з

[1] Таке протиставлення досить спірне, тому що функціональна відповідальність настільки ж є близькою принципам сучасного об'єктно-орієнтованого проектування, як і абстракція даних. Це питання еволюціонування поглядів і ступеня їх консерватизму.

метою підвищення повторного використання активів (як архітектурних, так і у формі коду).

Компонентно-орієнтоване проектування є однією з концепцій проектування, що найбільш динамічно розвиваються, і може розглядатися як провісник і основа сервісно-орієнтованого підходу (Service-Oriented Architecture, SOA) у проектуванні, не розглянутого, на жаль, у SWEBOK, але все більш активно використовується в індустрії і зміщує акценти з аспектів організації зв'язку інтерфейс-реалізація до обміну інформацією на рівні інтерфейс-інтерфейс (тобто міжкомпонентна взаємодія). Мабуть, настане або вже настав той момент, коли необхідно вводити окрему тему, присвячену сервісно-орієнтованому підходу в проектуванні й сервісно-орієнтованим архітектурам, як моделям. Зокрема нотація UML 2.0 уже дозволяє вирішувати низку питань, пов'язаних з візуальним представленням відповідних архітектурних рішень, де сервіси (служби) можуть розглядатися як функціональність, що публікується, одиночних компонентів і груп компонентів, об'єднаних у більш “великі” блоки, що забезпечують надання відповідної до сервісної функціональності.

3.6.6 Інші методи

Інші цікаві, але менш розповсюджені підходи, в основному являють собою формальні й точні (строгі) методи, а також методи трансформації.

4 Конструювання програмного забезпечення

Термін конструювання програмного забезпечення (software construction) описує детальне створення робочої програмної системи за допомогою комбінації кодування, верифікації (перевірки), модульного тестування (unit testing), інтеграційного тестування й налагодження.

Дана галузь знань пов'язана з іншими галузями. Найбільш сильний зв'язок існує з проектуванням (Software Design) і тестуванням (Software Testing). Причиною цього є те, що сам по собі процес конструювання програмного забезпечення стосується

важливих аспектів діяльності з проектування й тестування. Крім того, конструювання відштовхується від результатів проектування, а тестування (у будь-якій своїй формі) передбачає роботу з результатами конструювання. Досить складно визначити границі між проектуванням, конструюванням і тестуванням, тому що всі вони поєднані в єдиний комплекс процесів життєвого циклу й залежно від обраної моделі життєвого циклу й застосовуваних методів (методології) такий розподіл може виглядати по-різному.

Хоча ряд операцій з проектування детального дизайну може відбуватися до стадії конструювання, великий обсяг такого роду проектних робіт відбувається паралельно з конструюванням або як його частина. Це є суттю зв'язку з галуззю знань “Проектування програмного забезпечення”.

У свою чергу, протягом усієї діяльності з конструювання інженери використовують модульне й інтеграційне тестування. У такий спосіб поєднана дана галузь знань із “Тестуванням програмного забезпечення”.

У процесі конструювання звичайно створюється більша частина активів програмного проекту — конфігураційних елементів (configuration items). Тому в реальних проектах просто неможливо розглядати діяльність з конструювання у відриві від галузі знань “Конфігураційного управління” (Software Configuration Management).

Оскільки конструювання неможливе без використання відповідного інструментарію й, імовірно, дана діяльність є найбільш інструментально насиченою, важливу роль у конструюванні відіграє галузь знань “Інструменти й методи програмної інженерії” (Software Engineering Tools and Methods).

Безумовно, питання забезпечення якості значущі для всіх галузей знань і етапів життєвого циклу. У той же час код є основним результуючим елементом програмного проекту. Таким чином, явно напрошується і є присутнім зв'язок обговорюваних питань із галуззю знань “Якість програмного забезпечення” (Software Quality).

Із пов'язаних дисциплін програмної інженерії (Related Disciplines of Software Engineering) найбільш тісний і природний зв'язок даної галузі знань існує з комп'ютерними науками

(computer scince). Саме в них звичайно розглядаються питання побудови й використання алгоритмів і практик кодування. Нарешті, конструювання стосується й управління проектами (project management), причому такою мірою, наскільки діяльність з управління конструюванням важлива для досягнення результатів конструювання. Фази та етапи конструювання, інструментарій і методики, що залучаються при цьому, схематично зображені на рисунку 4.1.

Рисунок 4.1 – Галузь знань “Конструювання програмного забезпечення”

4.1 Основи конструювання

Фундаментальні основи конструювання програмного забезпечення включають:

– мінімізацію складності;
– очікування змін;
– конструювання з можливістю перевірки;
– стандарти в конструюванні.

Перші три концепції застосовуються не тільки до конструювання, але й проектування, і лежать в основі сучасних методологій управління життєвим циклом програмних систем.

4.1.1 Мінімізація складності

Основною причиною того, чому люди використовують комп'ютери в бізнес-цілях, є обмежені можливості людей в обробці й зберіганні складних структур і великих обсягів інформації, зокрема протягом тривалого періоду часу. Це міркування є однією з основних рушійних сил у конструюванні програмного забезпечення: мінімізація складності.

Потреба в зменшенні складності впливає на всі аспекти конструювання й є особливо критичною для процесів верифікації (перевірки) і тестування результатів конструювання, тобто самих програмних систем.

Зменшення складності в конструюванні програмного забезпечення досягається при приділенні особливої уваги створенню простого коду і такого, що легко читається, нехай навіть утративши прагнення зробити його ідеальним (наприклад, з погляду гнучкості або слідування тим або іншим представленням про красу, витонченість коду, спритності тих або інших прийомів, що дозволяють його скоротити, і т. п.). Це не означає, що слід обмежувати застосування тих або інших розвинених мовних можливостей використовуваних засобів програмування. Це має на увазі **лише** надання більшої значущості прочитуваності коду, простоті тестування, прийнятному рівню продуктивності й задоволенню заданих критеріїв, замість постійного вдосконалювання коду, не звертаючи уваги на строки, функціональність і інші характеристики й обмеження проекту.

Мінімізація складності досягається, зокрема, відповідністю стандартам (обговорюється в п. 4.1.4), використанням ряду специфічних технік (висвітлюються в п. 4.3.3) і підтримкою

практик, спрямованих на забезпечення якості в конструюванні (п. 4.3.6).

4.1.2 Очікування змін

Більшість програмних систем змінюються з часом. Причин цьому безліч. Очікування змін є однією з рушійних сил конструювання програмного забезпечення. Програмне забезпечення не є ізольованим від зовнішнього оточення (як системного, так і з погляду сфери діяльності, для автоматизації задач і проблем якої воно застосовується). Більш того, програмні системи є частиною змінного середовища й повинні змінюватися разом з нею, а іноді і бути джерелом змін самого середовища.

Очікування змін підтримується різними техніками, представленими у п. 4.3.3.

4.1.3 Конструювання з можливістю перевірки

“Конструювання для перевірки” припускає, що побудова програмних систем повинна вестися таким чином, щоб сама програмна система допомагала вести пошук причин збоїв, будучи прозорою для застосування різних методів перевірки (і, до речі, внесення необхідних змін), як на стадії незалежного тестування (наприклад, інженерами-тестувальниками), так і в процесі операційної діяльності – експлуатації, коли особливо важлива можливість швидкого виявлення й виправлення виникаючих помилок.

Серед технік, спрямованих на досягнення такого результату конструювання:

– огляд, оцінка коду (code review);

– модульне тестування (unit-testing);

– структурування коду для й разом із застосуванням автоматизованих засобів тестування (automated testing);

– обмежене застосування складних або важких для розуміння мовних структур.

4.1.4 Стандарти в конструюванні

Стандарти, які прямо застосовуються при конструюванні, включають:

– комунікаційні методи – наприклад, стандарти форматів документів і оформлення змісту;

– мови програмування й відповідні стилі кодування – наприклад, Java Language Specification, що є частиною стандартної документації JDK (Java Development Kit) і Java Style Guide, що пропонує загальний стиль кодування для мови програмування Java;

– платформи – наприклад, стандарти програмних інтерфейсів для викликів функцій операційного середовища, такі як прикладні програмні інтерфейси платформи Windows - Win32 API (Application Programming Interface) або .NET Framework SDK (Software Development Kit);

– інструменти не в термінах середовищ розроблення, але можливих засобів конструювання – наприклад, UML як один зі стандартів для визначення нотацій для діаграм, що представляють структуру коду і його елементів або деяких аспектів поведінки коду.

Використання зовнішніх стандартів. Конструювання залежить від зовнішніх стандартів, пов'язаних з мовами програмування, використовуваним інструментальним забезпеченням, технічними інтерфейсами й взаємним впливом конструювання програмного забезпечення й інших галузей знань програмної інженерії (у тому числі пов'язаних дисциплін, наприклад керування проектами). Стандарти створюються різними джерелами, наприклад консорціумом OMG – Object Management Group (зокрема стандарти CORBA, UML, MDA, …), міжнародними організаціями зі стандартизації, такими як ISO/IEC, IEEE, TMF, …, виробниками платформ, операційних середовищ і т. д. (наприклад, Microsoft, Sun Microsystems, CISCO, NOKIA, …), виробниками інструментів, систем керування базами даних і т. ін. (Borland, IBM, Microsoft, Sun, Oracle, ...). Розуміння цього факту дозволяє визначити достатній і повний набір стандартів, застосовуваних у проектній команді або організації в цілому.

Використання внутрішніх стандартів. Певні стандарти, угоди й процедури можуть бути також створені усередині організації або навіть проектної команди. Ці стандарти підтримують координацію між певними видами діяльності,

групами операцій, мінімізують складність (у тому числі при взаємодії членів проектної групи й за її межами), можуть бути пов'язані з питаннями очікування й обробки змін, ризиків і питаннями конструювання для перевірки й подальшого тестування. У комбінації з зовнішніми стандартами внутрішні стандарти покликані визначити загальні правила гри для всіх членів проектної команди, домовившись про терміни, процедури й інші значущі угоди, незалежно від ступеня формалізації процесів конструювання зокрема і для процесів життєвого циклу в загальному випадку.

4.2 Управління конструюванням

4.2.1 Моделі конструювання

Моделі конструювання визначають комплекс операцій, що включають послідовність, результати (наприклад, вихідний код і відповідні unit-тести) і інші аспекти, пов'язані з загальним життєвим циклом розроблення. У більшості випадків моделі конструювання визначаються використовуваним стандартом життєвого циклу, застосовуваними методологіями й практиками. Деякі стандарти життєвого циклу по своїй природі орієнтовані на конструювання, наприклад екстремальне програмування (XP - eXtreme Programming). Деякі розглядають конструювання в нерозривному зв'язку з проектуванням (у частині моделювання), наприклад RUP (Rational Unified Process).

Створена безліч моделей розроблення програмного забезпечення. Ряд з них більшою мірою зосереджений на конструюванні програмного забезпечення, як такому.

Деякі моделі є більш лінійними з погляду конструювання ПЗ. До них належать, наприклад, водоспадна (waterfall) і поетапна (staged-delivery) моделі життєвого циклу. Ці моделі розглядають конструювання як діяльність, яка починає проводитися тільки після завершення певних обов'язкових до виконання (prerequisite) робіт, що включають детальне визначення вимог, докладний дизайн і детальне планування. Більш лінійні підходи намагаються підкреслити дії, що випереджають конструювання (тобто вимоги й дизайн) і створити більш чітке розділення між

такими різними типами діяльності. У таких моделях основним змістом конструювання може бути кодування.

Інші моделі більш ітеративні, до них належать ***еволюційне прототипування, екстремальне програмування й Scrum.*** Ці підходи зводяться до розгляду конструювання як діяльності, яка ведеться одночасно з іншими видами робіт зі створення програмного забезпечення й перетинається з ними (очевидно, тут мається на увазі взаємозалежність і вплив одної на одну), включаючи визначення вимог, проектування й планування. Ці підходи змішують проектування, кодування й тестування, часто розглядаючи їх комбінацію як конструювання.

Відповідно, що саме розуміється під "конструюванням", залежить деякою мірою від використовуваної моделі життєвого циклу.

4.2.2 Планування конструювання

Вибір методу (методології) конструювання є ключовим аспектом для планування конструкторської діяльності. Такий вибір є значущим для всієї конструкторської діяльності, а також необхідних умов її здійснення, визначаючи порядок відповідних операцій і рівень виконання заданих умов перед тим, як почнеться конструювання або складові його дії. Наприклад, модульне тестування в ряді методів є частиною робіт після написання відповідного функціонального коду, у той час як ряд гнучких (agile) практик, наприклад XP (що, до речі, першими почали використовувати такі методи верифікації коду), вимагають написання Unit-Тестів до того, як пишеться відповідний код, що вимагає тестування.

Використовуваний підхід до конструювання впливає на можливість зменшення (в ідеалі — мінімізації) складності, готовності до змін і конструювання з можливістю перевірки.

Планування конструкторської діяльності визначає порядок, у якому створюються компоненти й інші активи даної галузі знань (фази діяльності), проводяться роботи з забезпечення якості одержуваного програмного забезпечення, розподіляються[1]

[1] Зверніть увагу – не розподіляють, а розподіляються, припускаючи процес, що призводить до забезпечення явного зв'язку між задачею і ресурсами. У нечітко регламентованих (це в жодному разі не лайка, це визначення – адже існує ж поняття нечітка логіка, неструктуровані бази даних, наприклад відносно нереляційних систем і

задачі й відповідні ресурси, у тому числі визначаються призначення/відображення робіт конкретним інженерам-програмістам, членам проектної групи. Усе це звичайно відбувається з дотриманням правил, обумовлених використовуваним методом (методологією, практиками й т. п.).

4.2.3 Виміри в конструюванні

Більша частина результатів, та й самої діяльності з конструювання програмного забезпечення, може бути обмірювана, у тому числі — кількісно. Це й вихідний розроблений код, і модифікований об'єм коду, і ступінь повторного використання, і багато інших характеристик. Ці виміри, або як їх ще прийнято називати – результати аудиту коду й метрики коду, несуть більшу користь для оцінки ризиків і якості (що призводять до відповідних операцій зі зниження ризиків і підвищення якості), а також для управління конструюванням і програмними проектами в цілому. Про яке планування може йтися, якщо ми не здатні передбачити (наприклад, на основі оцінки результатів попередніх проектів) ні тривалість робіт, ні вартість окремих задач, ні ймовірність виникнення дефектів проти заданих параметрів прийнятної якості?

Код є одним з найбільш чітко детермінованих активів проекту (поступово такими стають і моделі, що будуються на основі структур метаданих і тісно пов'язані з кодом, наприклад UML). Код є й самим носієм необхідної функціональності. Відповідно застосування вимірів відносно коду стає тим інструментом, який впливає й на управління, і на сам код.

Останнім часом велику увагу багато професійних розробників, тобто інженерів-конструкторів програмного забезпечення, приділяють рефакторингу коду, як методу його реструктурування, покликаному без зміни змісту (тобто функціональності й функціональної цілісності) забезпечити вирішення завдань мінімізації складності, готовності до змін (гнучкості), прозорості документування й багатьох інших актуальних аспектів конструювання. Але, на жаль, більшість із

т. п.) і неформальних методах, таких як XP, члени проектної групи самі беруть на себе відповідальність щодо вирішення певних завдань, а “володіння” кодом є спільним на основі співробітництва, як одного з ключових принципів роботи проектної команди.

них забувають про необхідність мотивованості змін, навіть на рівні рефакторингу. Застосування вимірів, зокрема метрик, дозволяє визначити необхідність внесення таких змін, проведення рефакторингу. І не тому, що “так, напевно, буде все-таки краще, красивіше...”, а тому, що в ієрархії спадкування з 10 поколінь класів 9 є абстрактними (“з любові до справи”), а на 10-му (через перевищенння строків проекту, після того як довго й “у кайф” створювали архітектурно-гарний код) “повішено” 20 (!) бізнес-методів. От це – дійсно обґрунтована причина для рефакторингу, який навіть із застосуванням найдосконаліших інструментальних засобів, разом з обмірковуванням необхідності рефакторингу, а потім іноді і боротьбою з його наслідками через неузгодженість членів команди (а це вже життя, навіть у найідеальнішому колективі, де люди розуміють один одного з півслова) часто є просто витратою часу. Якщо застосовується рефакторинг, але не застосовуються метрики, у переважній більшості випадків це негативно впливає на проект. І таких прикладів робіт, що вимагають застосування вимірів, але, на жаль їх ігнорують, можна навести досить багато.

Чому ця тема виявилася настільки емоційною? Практика показує, що в погоні за “красою” розробники занадто часто забувають про мету їх роботи й сприймають діяльність з управління проектами з боку менеджерів і, тим більше будь-який аудит, як однозначну перешкоду “польоту думки”. Даремно. Якщо все саме так виглядає у вашому випадку, проект, з великою ймовірністю, а іноді й просто, “якщо не трапиться чудо”, можна вважати нехай і не провальним (“фуух... не закрили....”), то, напевно, таким, що виходить за рамки тих або інших обмежень, будь то строки, гроші, якість або, нарешті, час, який розробник міг провести з родиною, а не сидіти вночі, дописуючи функціональність або відловлюючи чергову помилку за кілька днів до передачі проекту в експлуатацію. Усі ці питання переслідують одну єдину мету – передбачуваність робіт і результату. І виміри – важливий інструмент досягнення цієї мети.

4.3 Практичні міркування

Конструювання – діяльність, у рамках якої програмне забезпечення зводиться до угоди з довільними (іноді —

хаотичними) обмеженнями реального світу, які вимагають від програмного забезпечення точного дотримання їх. Наближаючись до обмежень реального світу, конструювання (більшою мірою, ніж будь-яка інша галузь знань) ведеться на основі практичних міркувань і технік.

4.3.1 Проектування в конструюванні

Деякі проекти припускають більший обсяг робіт з проектування саме на стадії конструювання; інші проекти явно виділяють проектну діяльність у формі фази дизайну. Незалежно від чіткості виділення діяльності з проектування як такої, практично завжди на стадії конструювання доводиться займатися й питаннями детального дизайну системи. Такі проектні роботи мають прагнення до дотримання стійким обмеженням, що нав'язуються конкретними проблемами, вирішення яких повинно бути забезпечено використанням програмної системи, що конструюється.

Деталі діяльності з проектування на стадії конструювання в основному ті самі, що й описані в галузі знань “Проектування програмного забезпечення”. Відмінність полягає в більшій увазі деталям.

4.3.2 Мови конструювання

Мови конструювання включають усі форми комунікацій, за допомогою яких людина може задати вирішення проблеми, виконуване на комп'ютері.

Найпростіший тип мов конструювання – конфігураційна мова (configuration language), що дозволяє задавати параметри виконання програмної системи.

Інструментальна мова (toolkit language) – мова конструювання з повторно використовуваних елементів; звичайно будується як сценарна мова (script) застосовувана у відповідному середовищі.

Мова програмування (programming language) – найбільш гнучкий тип мов конструювання. Містить мінімальний обсяг інформації про конкретні області додатку й процесу розроблення, вимагаючи найбільших (порівнянно з іншими типами мов

конструювання) зусиль на вивчення й набування досвіду для ефективного застосування при розв’язанні конкретних задач.

Існує три основні види нотацій, використовуваних при визначенні мов програмування:

– лінгвістична;

– формальна;

– візуальна.

Лінгвістичні нотації характеризуються, зокрема, використанням рядків тексту, що містять спеціалізовані “слова”, які являють собою складні програмні конструкції, і комбінуються в шаблони, що нагадують речення, побудовані відповідно до певного синтаксису. У випадку коректного використання таких нотацій кожний одержуваний рядок має суворе значеннєве навантаження (семантику), що забезпечує інтуїтивне розуміння того, що буде відбуватися, коли буде виконуватися програмне забезпечення, побудоване з використанням такої мови конструювання.

Формальні нотації є менш інтуїтивними, ніж лінгвістичні, і часто базуються на точних формальних (математичних) визначеннях. Формальні нотації конструкцій і формальні методи є ядром практично всіх форм системного програмування, точніше поводження систем у часі. Такі нотації забезпечують найбільшу готовність одержуваного коду до тестування, що набагато важливіше, ніж просто можливість відображення звичайною людською мовою. Формальні конструкції також використовують точний метод визначення комбінацій застосовуваних символів, що дозволяє уникнути неоднозначностей, властивих конструкціям природних мов.

Візуальні нотації найменш пов'язані з текстово-орієнтованими підходами, припускаючи безпосередню інтерпретацію візуальних конструкцій у процесі виконання описуваної логіки. При цьому логіка у візуальних нотаціях задається розташуванням відповідних візуальних сутностей, відповідальних за ті або інші операції й структури даних.

Використання візуальних конструкцій обмежене складністю візуального представлення складних виражень і тверджень тільки за рахунок переміщення візуальних сутностей на діаграмі (візуальному представленні). Однак візуальна нотація може

відігравати роль досить потужного інструменту, коли застосовується в тих задачах програмування, де необхідна побудова користувацького інтерфейсу для програм, чию логіку й деталізоване поводження визначено раніше.

Сьогоднішні роботи (і їхній стан) у сфері архітектур і додатків, керованих моделями, у першу чергу OMG MDA (Model-Driven Architecture)/UML (Unified Modeling Language), Microsoft DSL (Domain-Specific Language), спрямовані на те, щоб використовувати ту або іншу візуальну нотацію, що базується на мета-моделях, у якості інструмента, застосовуваного й для визначення функціональної логіки системи. Чи можна в чистому вигляді вважати ці нотації саме візуальними — питання спірне. Історично ці нотації визначалися споконвічно як нотації візуального представлення функціональності і вже надалі ці візуальні представлення були відображені на рівні відповідних мета-моделей (хоча це більшою мірою правильно для UML, ніж DSL, але DSL можна розглядати і як аналог UML, що припускає більшу свободу застосувань і інтегрованість із конкретною платформою — Microsoft). Інша сфера стандартів, направлених на застосування візуальних нотацій для опису функціональності, – Business Process Management Notation (BPMN) і пов'язана з нею мова Business Process Execution Language, побудована на базі XML. Таким чином, сфера обґрунтованого застосування візуальних нотацій для конструювання програмних систем якісно розширяється й, не виключено, ми станемо свідками de-facto формування нової категорії нотацій, угод і змішаних типів мовних засобів, призначених для ***конструювання програмного забезпечення як природного продовження проектування.***

4.3.3 Кодування

Практика конструювання програмного забезпечення показує активне застосування таких міркувань і технік:

– техніки створення вихідного коду, що легко розуміються, на основі використання угод про іменування, форматування й структурування коду;

– використання класів, типів, що перелічуються, змінних, іменованих констант і інших виразних сутностей;

– організація вихідного тексту (у вирази, шаблони, класи, пакети/модулі та інші структури);

– використання структур управління;

– обробка помилкових умов і виняткових ситуацій;

– запобігання можливих проломів у безпеці (наприклад, переповнення буфера або вихід за межі індексу в масиві);

– використання ресурсів на основі застосування механізмів виключення (з розгляду) і порядку доступу до паралельно використовуваних ресурсів (наприклад, на основі блокування даних, використання потоків і їх синхронізації й т. п.);

– документування коду;

– тонке “настроювання” коду;

– рефакторинг.

4.3.4 Тестування в конструюванні

При конструюванні використовуються дві форми тестування, проводжуваного інженерами, що безпосередньо створюють вихідний код:

– модульне тестування (unit testing);

– інтеграційне тестування (integration testing).

Головна мета тестування в конструюванні – зменшити часовий розрив між моментом прояву помилок, наявних у коді, і моментом їх виявлення. У багатьох випадках тестування в конструюванні проводиться після того, як код написаний. У ряді випадків тести (що зазначалося раніше, на прикладі XP) пишуться до того, як створюється код.

Тестування в конструюванні звичайно включає підмножину видів тестування, описаних у галузі “Тестування програмного забезпечення” (Software Testing). Наприклад, тестування в конструюванні звичайно не включає системного тестування, навантажувального тестування, usability-тестування (оцінки прозорості використання) і інших видів тестової діяльності.

IEEE опублікував два стандарти, присвячені тестуванню:

– IEEE Std 829-1998: “IEEE Standard for Software Test Documentation”;

– IEEE Std 1008-1987: “IEEE Standard for Software Unit Testing”.

4.3.5 Повторне використання

У введенні в стандарт IEEE Std. 1517-99 “IEEE Standard for Information Technology – Software Lifecycle Process – Reuse Processes” дається таке розуміння повторного використання в програмному забезпеченні: “Реалізація повторного використання програмного забезпечення має на увазі й призводить до чогось більшого, ніж просте створення й використання бібліотек активів. Воно вимагає формалізації практики повторного використання на основі інтеграції процесів і діяльності з повторного використання в сам життєвий цикл програмного забезпечення.” У той же час повторне використання досить важливо й безпосередньо при конструюванні програмних систем, що підкреслюється включенням цієї теми в обговорювану галузь знань конструювання ПЗ.

Завдання, пов'язані з повторним використанням у процесі конструювання й тестування, включають:

– вибір одиниць (units), баз даних тестових процедур, даних, отриманих у результаті тестів, і самих тестів, що підлягають повторному використанню;

– оцінку потенціалу повторного використання коду й тестів;

– відстеження інформації й створення звітності з повторного використання в новому коді, тестових процедурах і даних, отриманих у результаті тестів.

4.3.6 Якість конструювання

Існує ряд технік, призначених для забезпечення якості коду, виконуваних у міру його конструювання. Основні техніки забезпечення якості, використовувані в процесі конструювання, включають:

– модульне (unit) і інтеграційне (integration) тестування;

– розроблення з первинністю тестів (test-first development — тести пишуться до конструювання коду);

– покрокове кодування (діяльність з конструювання коду розбивається на дрібні кроки, тільки після тестування результатів яких проводиться перехід до наступного кроку кодування; відомий також як ітеративне кодування з тестуванням);

– використання процедур тверджень (assertion);

– налагодження (у звичному розумінні — debugging);

– технічні огляди й оцінки (review);

– статичний аналіз.

Вибір і використання конкретних технік часто диктується стандартами (внутрішніми й зовнішніми), використовуваними проектною командою, а також залежать від досвіду й підготовленості фахівців, що займаються конструюванням коду.

Діяльність з забезпечення якості в конструюванні відрізняється від інших операцій з забезпечення якості. Основна відмінність полягає у фокусуванні на програмному (вихідному) коді й інших артефактах (активах), тісно пов'язаних з кодом, зокрема детальних моделях.

4.3.7 Інтеграція

Одна з ключових діяльностей, здійснюваних у процесі конструювання, — інтеграція окремо сконструйованих операцій (процедур), класів, компонентів і підсистем (модулів). На додачу до цього, деякі програмні системи потребують спеціальної інтеграції з іншим програмним і апаратним забезпеченням.

Крім згаданих аспектів інтеграції, до обговорюваних інтеграційних питань конструювання належать:

– планування послідовності, у якій інтегруються компоненти;

– забезпечення підтримки створення проміжних версій програмного забезпечення;

– задавання “глибини” тестування (зокрема на основі критеріїв “прийнятної” якості) і інших робіт із забезпечення якості інтегрованих надалі компонент;

– нарешті, визначення етапних точок проекту, коли будуть тестуватися проміжні версії програмної системи, що конструюється.

5 Тестування програмного забезпечення

Тестування (software testing) – діяльність, виконувана для оцінки й поліпшення якості програмного забезпечення. Ця

діяльність у загальному випадку базується на виявленні дефектів і проблем у програмних системах.

Тестування програмних систем складається з **динамічної** верифікації поведінки програм на **кінцевому** (обмеженому) наборі тестів (set of test cases), **обраних** відповідним чином зі звичайно виконуваних дій прикладної галузі, що й забезпечують перевірку відповідності **очікуваній поведінці** системи.

У даному визначенні тестування виділено слова, що визначають основні питання, яким адресується дана галузь:

– ***динамічність (dynamic)***: цей термін має на увазі, що тестування завжди припускає виконання програми, що тестується з заданими вхідними даними. При цьому величини, що задаються на вхід програмному забезпеченню, що тестується, не завжди достатні для визначення тесту. Складність і недетермінованість систем призводить до того, що система може різним чином реагувати на ті самі вхідні параметри, залежно від стану системи. Термін "вхід" (input) буде використовуватися в рамках угоди про те, що вхід може також специфікувати стан системи у тих випадках, коли це необхідно. Крім динамічних технік перевірки якості, тобто тестування, існують також і статичні техніки, розглянуті в галузі "Software Quality";

– ***кінцевість (обмеженість, finite):*** навіть для простих програм теоретично можлива настільки велика кількість тестових сценаріїв, що вичерпне тестування може зайняти багато місяців і навіть років. Саме тому, із практичної точки зору, всебічне тестування вважається нескінченним. Тестування завжди припускає компроміс між обмеженими ресурсами й заданими строками, з одного боку, і практично необмеженими вимогами з тестування - з іншого. Тобто ми знову говоримо про визначення характеристик "прийнятної" якості, на основі яких плануємо необхідний обсяг тестування;

– ***вибір (selection):*** багато пропонованих технік тестування відрізняються одна від одної в тому, як вибираються сценарії тестування. Інженери з програмного забезпеченню повинні мати уявлення про те, що різні критерії вибору тестів можуть давати різні результати, з погляду ефективності тестування. Визначення прийнятного набору тестів для заданих умов є дуже складною проблемою. Звичайно для вибору відповідних тестів спільно

застосовують техніки аналізу ризиків, аналіз вимог і відповідну експертизу у сфері тестування й заданої прикладної галузі;

– ***очікувана поведінка (expected behaviour):*** хоча це не завжди легко, все-таки необхідно вирішити, яка спостережувана поведінка програми буде прийнятною, а яка – ні. А якщо ні, то зусилля з тестування – марні. Спостережувана поведінка може розглядатися в контексті користувацьких очікувань (припускаючи "тестування для перевірки" – testing for validation), специфікації ("тестування для атестації" – testing for verification) або, нарешті, у контексті передбаченої поведінки на основі неявних вимог або обґрунтованих очікувань.

Загальний погляд на тестування програмного забезпечення в останні роки активно еволюціонував, стаючи все більш конструктивним, прагматичним і наближеним до реалій сучасних проектів розроблення програмних систем. Тестування більше не розглядається як діяльність, що починається тільки після завершення фази конструювання. Сьогодні тестування розглядається як діяльність, яку необхідно проводити протягом усього процесу розроблення й супроводження і є важливою частиною конструювання програмних продуктів. Дійсно, планування тестування повинне починатися на ранніх стадіях роботи з вимогами, необхідно систематично й постійно розбудовувати й уточнювати плани тестів і відповідні процедури тестування. Навіть самі по собі сценарії тестування виявляються дуже корисними для тих, хто займається проектуванням, дозволяючи виділяти ті аспекти вимог, які можуть неоднозначно інтерпретуватися або навіть бути суперечливими.

Звичайно, легше запобігти проблемі, ніж боротися з її наслідками. Тестування, нарівні з управлінням ризиками, є тим інструментом, який дозволяє діяти саме в такому ключі. Причому діяти досить ефективно. З іншого боку, необхідно усвідомлювати, що навіть якщо приймальні тести показали позитивні результати, це зовсім не означає, що отриманий продукт не містить помилок. Цим питанням адресована галузь "Супроводження програмного забезпечення" (Software Maintenance). Однак адекватна увага питанням тестування якісно знижує ризик виникнення помилок на етапі експлуатації,

забезпечуючи більш високу задоволеність користувачів, що і є, по суті, метою будь-якого проекту.

У галузі “Якість програмного забезпечення” (Software Quality) техніки управління якістю чітко розділені на статичні (без виконання коду) і динамічні (з виконанням коду). Обидві ці категорії важливі. Дана галузь – “Тестування” – стосується динамічних технік.

Як ми вже зазначали раніше, тестування тісно пов'язане з галуззю “Конструювання” (Software Construction). Більш того, модульне (unit-) і інтеграційне тестування все частіше розглядають як невід'ємний елемент діяльності з конструювання.

Основи тестування (Software Testing Fundamentals), взаємозв’язок основних понять у сфері тестування, базових термінів, ключових проблеми і їхній зв'язок з іншими сферами діяльності представлені у вигляді схеми на рисунку 5.1.

5.1 Термінологія тестування

Визначення тестування й пов'язаної термінології досить повно дається в “Глосарії термінів з програмної інженерії” – IEEE Standard 610-90 (Standard Glossary of Software Engineering Terminology).

Недоліки й збої (Faults vs. Failures)

У літературі, присвяченій програмній інженерії, зустрічається безліч термінів, що описують порушення функціонування програмних систем, – недоліки (faults), дефекти (defects), збої (failures), помилки (errors) та ін. Відповідна термінологія описана в IEEE Std. 610-90, також обговорюється в галузі знань SWEBOK “Якість програмного забезпечення” (Software Quality). Важливо чітко розділяти причину порушення роботи прикладних систем, звичайно описувану термінами недолік або дефект, і спостережуваний небажаний ефект, викликаний цими причинами, – збій. Термін помилка, залежно від контексту, може описувати і причину збою, і сам збій. Тестування дозволяє виявити дефекти, що призводять до збоїв.

Необхідно розуміти, що причина збою не завжди може бути однозначно визначена. Не існує теоретичних критеріїв, що

дозволяють гарантовано визначити, який саме дефект призводить до спостережуваного збою.

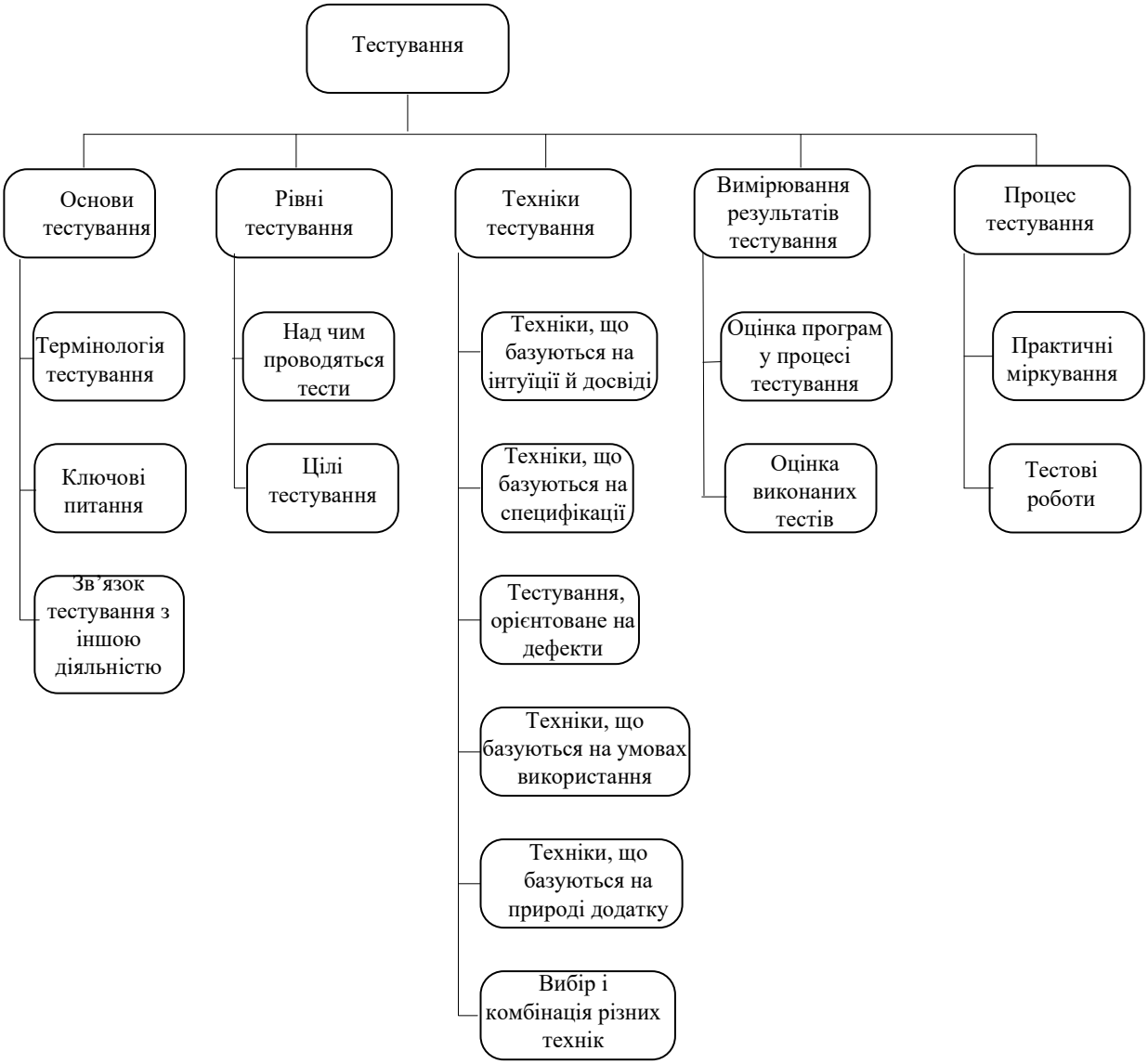

Рисунок 5.1 – Галузь знань "Тестування програмного забезпечення"

5.2 Ключові питання

5.2.1 Критерії відбору тестів, критерії адекватності тестів, правила припинення тестування

Критерії відбору тестів можуть використовуватися як для створення набору тестів, так і для перевірки, наскільки обрані тести адекватні вирішуваним завданням (тестування). При цьому,

обговорювані критерії допомагають визначити, коли можна або необхідно припинити тестування.

5.2.2 Ефективність тестування/Цілі тестування

Тестування – це спостереження за виконанням програми, запущеної з метою тестування з заданими параметрами, за заданим сценарієм або з іншими заданими початковими умовами або цілями тестування. Ефективність тесту може бути визначена тільки в контексті заданих умов.

5.2.3 Тестування для ідентифікації дефектів

Даний випадок тестування має на увазі успішність процедури тестування, якщо дефект знайдений. Це відрізняється від підходу до тестування, коли тести запускаються для демонстрації того, що програмне забезпечення задовольняє пропоновані вимоги і відповідно тест вважається успішним, якщо не знайдено дефекти.

5.2.4 Проблема оракула

“Оракул” у даному контексті – будь-який агент (людина або програма), що оцінює поведінку програми, формулюючи вердикт – тест пройдений (“pass”) чи ні (“fail”).

5.2.5 Теоретичні й практичні обмеження тестування

Теорія тестування виступає проти необґрунтованого рівня довіри до серії успішно пройдених тестів. На жаль, більшість установлених результатів теорії тестування – негативні, означаючи, за словами Дейкстри, те, що “тестування програми може використовуватися для демонстрації наявності дефектів, але ніколи не покаже їхню відсутність”. Основна причина цього в тому, що повне (всеосяжне) тестування недосяжне для реального програмного забезпечення.

5.2.6 Проблема нездійсненних шляхів

Ця найскладніша проблема автоматизованого тестування пов'язана з тим, що шляхи, по яких виконуються потоки робіт

тестованої програмної системи, не можуть бути задані вхідними параметрами.

5.2.7 Тестованість

Це поняття може мати на увазі дві різні ідеї. Перша описує ступінь легкості опису критеріїв покриття тестами для заданої програмної системи. Друга визначає імовірність, можливість статистичного виміру того, що при тестуванні виявиться збій програмної системи. Обидві інтерпретації цього поняття однаково важливі для тестування.

5.3 Зв'язок тестування з іншою діяльністю

Тестування програмного забезпечення відрізняється від статичних технік управління якістю, перевірки коректності, налагодження й програмування, але пов'язане з усіма цими роботами. Корисно розглядати тестування з погляду аналітиків і фахівців із сертифікації якості.

5.4 Рівні тестування

5.4.1 Над чим проводяться тести

Тестування звичайно проводиться протягом усього розроблення й супроводу на різних рівнях. Рівень тестування визначає “над чим” проводяться тести: над окремим модулем, групою модулів або системою в цілому. При цьому жоден з рівнів тестування не може вважатися пріоритетним. Важливі всі рівні тестування, незалежно від використовуваних моделей і методологій.

1 Модульне тестування.

Цей рівень тестування дозволяє перевірити функціонування окремо взятого елемента системи. Що вважати елементом – модулем системи, визначається контекстом. Найбільш повно даний вид тестів описаний у стандарті IEEE 1008-87 “Standard for Software Unit Testing”, що задає інтегровану концепцію систематичного й документованого підходу до модульного тестування.

2 Інтеграційне тестування.

Даний рівень тестування є процесом перевірки взаємодії між програмними компонентами/модулями.

Класичні стратегії інтеграційного тестування – “зверху-униз” і “знизу-нагору” – використовуються для традиційних, ієрархічно структурованих систем і їх складно застосовувати, наприклад, до тестування слабопов’язаних систем, побудованих у сервіс-орієнтованих архітектурах (SOA).

Сучасні стратегії більшою мірою залежать від архітектури тестованої системи й будуються на основі ідентифікації функціональних “потоків” (наприклад, потоків операцій і даних).

Інтеграційне тестування – постійно проводжувана діяльність, що припускає роботу на досить високому рівні абстракції. Найбільш успішна практика інтеграційного тестування базується на інкрементальному підході, що дозволяє уникнути проблем проведення разових тестів, пов'язаних з тестуванням результатів чергового тривалого етапу робіт, коли кількість виявлених дефектів призводить до серйозної переробки коду (традиційно негативний досвід випуску й тестування тільки великих релізів називають “big bang”).

3 Системне тестування.

Системне тестування охоплює цілком усю систему. Більшість функціональних збоїв мають бути ідентифіковані ще на рівні модульних і інтеграційних тестів. У свою чергу, системне тестування звичайно фокусується на нефункціональних вимогах – безпеці, продуктивності, точності, надійності і т. п.

На цьому рівні також тестуються інтерфейси до зовнішніх додатків, апаратного забезпечення, операційного середовища і т. д.

5.4.2 Цілі тестування

Тестування проводиться відповідно до певних цілей (можуть бути задані явно або неявно) і різним рівнем точності. Визначення цілі точним образом, що виражається кількісно, дозволяє забезпечити контроль результатів тестування.

Тестові сценарії можуть розроблятися як для перевірки функціональних вимог (відомі як функціональні тести), так і для оцінки нефункціональних вимог. При цьому існують такі тести,

коли кількісні параметри й результати тестів можуть лише опосередковано говорити про задоволення цілей тестування (наприклад, “usability” – легкість, простота використання у більшості випадків не може бути явно описана кількісними характеристиками).

Можна виділити такі найпоширеніші й обґрунтовані цілі (а відповідно, види) тестування:

1 Приймальне тестування.

Перевіряє поводження системи на предмет задоволення вимог замовника. Це можливо в тому випадку, якщо замовник бере на себе відповідальність, пов'язану з проведенням таких робіт, як сторона, “що ухвалює” програмну систему, або специфіковані типові завдання, успішна перевірка (тестування) яких дозволяє говорити про задоволення вимог замовника.

Такі тести можуть проводитися як із залученням розробників системи, так і без них.

2 Настановне тестування.

З назви випливає, що дані тести проводяться з метою перевірки процедури інсталяції системи в цільовому оточенні.

3 Альфа- і бета-тестування.

Перед тим як випускається програмне забезпечення, як мінімум, воно повинне проходити стадії альфа (внутрішнє пробне використання) і бета (пробне використання з залученням відібраних зовнішніх користувачів) версій. Звіти про помилки, що надходять від користувачів цих версій продукту, обробляються відповідно до певних процедур, що включають підтверджувальні тести (будь-якого рівня), проведені фахівцями групи розроблення.

Даний вид тестування не може бути заздалегідь спланованим.

4 Функціональні тести/тести відповідності.

Ці тести можуть називатися по-різному, однак їхня суть проста – перевірка відповідності системи пропонованим до неї

вимогам, описаним на рівні специфікації поведінкових характеристик.

5 Досягнення й оцінка надійності.

Допомагаючи ідентифікувати причини збоїв, тестування має на увазі й підвищення надійності програмних систем. Сценарії тестування, що генеруються випадково, можуть застосовуватися для статистичної оцінки надійності. Обидві цілі – підвищення й оцінка надійності – можуть досягатися при використанні моделей підвищення надійності.

6 Регресійне тестування.

Визначення успішності регресійних тестів (IEEE 610-90 “Standard Glossary of Software Engineering Terminology”) говорить: “повторне вибіркове тестування системи або компонент для перевірки зроблених модифікацій не повинне призводити до непередбачуваних ефектів”. На практиці це означає, що якщо система успішно проходила тести до внесення модифікацій, вона повинна їх проходити і після внесення таких. Основна проблема регресійного тестування полягає в пошуку компромісу між ресурсами, що є і необхідністю проведення таких тестів у міру внесення кожної зміни. Деякою мірою завдання полягає в тому, щоб визначити критерії “масштабів” змін, з досягненням яких необхідно проводити регресійні тести.

7 Тестування продуктивності.

Спеціалізовані тести перевірки задоволення специфічних вимог, пропонованих до параметрів продуктивності. Існує особливий підвид таких тестів, коли робиться спроба досягнення кількісних меж, обумовлених характеристиками самої системи і її операційного оточення.

8 Навантажувальне тестування.

Необхідно розуміти відмінності між розглянутим вище тестуванням продуктивності з метою досягнення її реальних (досяжних) можливостей продуктивності й виконанням програмної системи з підвищенням навантаження, аж до досягнення запланованих характеристик і далі, з відстеженням поведінки протягом підвищення завантаження системи.

9 Порівняльне тестування.

Одиничний набір тестів, що дозволяють порівняти дві версії системи.

10 Відновні тести.

Мета – перевірка можливостей рестарту системи у випадку непередбачуваної катастрофи (disaster), що впливає на функціонування операційного середовища, у якому виконується система.

11 Конфігураційне тестування.

У випадках, якщо програмне забезпечення створюється для використання різними користувачами (у термінах “ролей”), даний вид тестування спрямований на перевірку поведінки й працездатності системи в різних конфігураціях.

12 Тестування зручності й простоти використання.

Мета – перевірити, наскільки легко кінцевий користувач системи може її освоїти, включаючи не тільки функціональну складову – саму систему, але і її документацію; наскільки ефективно користувач може виконувати завдання, автоматизація яких здійснюється з використанням даної системи; нарешті, наскільки добре система застрахована (з погляду потенційних збоїв) від помилок користувача.

13 Розроблення, кероване тестуванням.

По суті, це не стільки техніка тестування, скільки стиль організації процесу розроблення, життєвого циклу, коли тести є невід'ємною частиною вимог (і відповідних специфікацій), замість того, щоб розглядатися незалежною діяльністю з перевірки задоволення вимог програмною системою.

Іноді говорять про такий стиль розроблення як про самостійну методологію – TDD (Test-driven development). Наскільки це правильно, залежить від того, що саме розуміти під методологією розроблення. Скоріше, це техніка, практика або стиль організації роботи, ніж самостійна методологія.

Меншою мірою це стосується FDD – Feature-Driven Development (розроблення на основі функціональних можливостей). TDD може природно розглядатися як складова

частина XP або, як мінімум, Agile-методів. У свою чергу, FDD може розглядатися як один з методів гнучкого розроблення.

У чому відмінність настільки близьких, на перший погляд, підходів (і, до речі, абревіатур, що відповідають)? Причина – проста. Тести – інструмент досягнення характеристик системи, що задовольняє задані вимоги, тобто потреби користувачів, а “можливості” (features) – практично самі (частіше – функціональні) вимоги, втілені (в ідеальному випадку) у код.

5.5 Техніки тестування

5.5.1 Техніки, що базуються на інтуїції й досвіді інженера

1 Спеціалізоване тестування.

Можливо, це техніка, що найбільш широко практикується. Тести ґрунтуються на досвіді, інтуїції й знаннях інженера, що розглядає проблему з погляду аналогій, що були раніше. Даний вид тестування може бути корисним для ідентифікації тих тестів, які не охоплюються розглянутими раніше більш формалізованими техніками.

2 Дослідницьке тестування.

Таке тестування визначається як одночасне навчання, проектування тесту і його виконання. Даний вид тестування заздалегідь не визначається в плані тестування й такі тести створюються, виконуються й модифікуються динамічно, у міру необхідності. Ефективність дослідницьких тестів прямо залежить від знань інженера, сформованих на основі поведінки продукту, що тестується в процесі проведення тестування, ступеня знайомства з додатком, платформою, типами можливих збоїв і дефектів, ризиками, асоційованими з конкретним продуктом і т. п.

5.5.2 Техніки, що базуються на специфікації.

1 Еквівалентний розподіл додатка.

Область додатка, що розглядається, поділяється на колекцію наборів або еквівалентних класів, які вважаються еквівалентними з погляду розглянутих зв'язків і характеристик специфікації.

Репрезентативний набір тестів (іноді – тільки один тест) формується з тестів еквівалентних класів (або наборів класів).

2 Аналіз граничних значень.

Тести будуються з орієнтацією на використання тих величин, які визначають граничні характеристики системи, що тестується. Розширенням цієї техніки є тести оцінки живучості (robustness testing) системи, проведені з величинами, що виходять за рамки специфікованих меж значень.

3 Таблиці прийняття рішень.

Такі таблиці представляють логічні зв'язки між умовами (можуть розглядатися в якості “входів”) і діями (можуть розглядатися як “виходи”). Набір тестів будується послідовним розглядом усіх можливих крос-зв'язків у такій таблиці.

4 Тести на основі кінцевого автомата.

Будуються як комбінація тестів для всіх станів і переходів між станами, представлених у відповідній моделі (переходів і станів додатка).

5 Тестування на основі формальної специфікації.

Для специфікацій, визначених з використанням формальної мови, можливо автоматично створювати й тести для функціональних вимог. У ряді випадків можуть будуватися на основі моделі, що є частиною специфікації і не використовує формальної мови опису.

6 Випадкове тестування.

На відміну від статистичного тестування (буде розглядатися далі), самі тести генеруються випадково за списком заданого набору специфікованих характеристик.

5.5.3 Техніки, орієнтовані на код

1 Тести, що базуються на блок-схемі.

Набір тестів будується виходячи з покриття всіх умов і рішень блок-схеми. У якійсь мірі нагадує тести на основі

кінцевого автомата. Відмінність – у джерелі набору тестів. Максимальна віддача від тестів на основі блок-схеми виходить, коли тести покривають різні шляхи блок-схеми – по суті, сценарії потоків робіт (поведінки) системи, що тестується. Адекватність таких тестів оцінюється як відсоток покриття всіх можливих шляхів блок-схеми.

2 Тести на основі потоків даних.

У даних тестах відслідковується повний життєвий цикл величин (змінних) – з моменту народження (визначення), протягом використання, аж до знищення (невизначеності). У реальній практиці використовується нестроге тестування такого виду, орієнтоване, наприклад, тільки на перевірку завдання початкових значень усіх змінних або всіх входжень змінних у код, з погляду їх використання.

3 Посилальні моделі для тестування, орієнтованого на код.

Є не стільки технікою тестування, скільки контролем структури програми, представленої у вигляді дерева викликів (наприклад, sequence-діаграми, визначеної в нотації UML і побудованої на основі аналізу коду).

5.5.4 Тестування, орієнтоване на дефекти

Як це не дивно звучить на рівні назви таких технік тестування, вони дійсно орієнтовані на помилки. Точніше – на специфічні категорії помилок.

1 Припущення помилок.

Направлені на виявлення найбільш імовірних помилок, що передбачаються, наприклад, у результаті аналізу ризиків.

2 Тестування мутацій.

Мутація – невелика зміна програми, яка тестується, що відбулася за рахунок часткових синтаксичних змін коду (зокрема, рефакторингу). Відповідні тести запускаються для оригінального й усіх “мутуючих” варіантів програми, яка тестується.

SWEBOK фокусується на можливості за допомогою тестів визначати відмінності між мутантами й вихідним варіантом коду. Якщо така відмінність установлена, мутанта “убивають”, а тест вважається успішним. Звичайно даний підхід фокусується на синтаксичних помилках, що на практиці відслідковуються сучасними середовищами розроблення й звичайно компіляторами.

5.5.5 Техніки, що базуються на умовах використання

1 Операційний профіль.

Базується на умовах використання системи.

Тестування для оцінки надійності системи має проводитися в такому тестовому оточенні, яке максимально наближено до реальних умов роботи системи. Результати таких тестів дозволяють оцінити поведінку системи в реальних умовах. Вхідні параметри тестів задаються на основі імовірнісного розподілу відповідних параметрів або їх наборів при експлуатації (вхідні дані можуть прогнозуватися виходячи з частоти можливих сценаріїв роботи користувачів).

2 Тестування, що базується на надійності інженерного процесу.

Базується на умовах розроблення системи.

Відповідні тести (позначувані також абревіатурою SRET від Software Reliability Engineered Testing) проектують у контексті використовуваного процесу розроблення й методик тестування.

5.5.6 Техніки, що базуються на природі додатка

Описані вище техніки можуть застосовуватися до будь-яких типів програмних систем. У той же час залежно від технологічної або архітектурної природи додатків можуть також застосовувати специфічні техніки, важливі саме для заданого типу додатка. Серед таких технік:

- об'єктно-орієнтоване тестування;
- компонентно-орієнтоване тестування;
- web-орієнтоване тестування;
- тестування на відповідність протоколам;

– тестування систем реального часу.

5.5.7 Вибір і комбінація різних технік

1 Функціональна й структурна.

Техніки тестування, що будуються на основі специфікацій або коду, часто називають функціональними або структурними відповідно. Обидва підходи не повинні протиставлятися, але мають доповнювати один одного.

2 Визначена або випадкова.

Звичайно тести можна розподілити по даних групах на основі використовуваної політики вибору або визначення вхідних параметрів тестів.

5.6 Вимірювання результатів тестування

Часто техніки тестування плутають із цілями тестування. Ступінь покриття тестами – не те саме, що висока якість системи, що тестується. Однак ці питання пов'язані. Чим вище ступінь покриття, тим більше ймовірність виявлення прихованих дефектів. Коли ми говоримо про результати тестування, ми повинні підходити до їхньої оцінки, як до оцінки самої системи, що тестується. Самі кількісні оцінки результатів тестування (але не самих тестів, наприклад, покриття ними можливих сценаріїв роботи системи) висвітлюються далі. У свою чергу метрики самих тестів або процесу тестування як такого – питання, розглянуті в галузях знань “Процеси програмної інженерії” (Software Engineering Process) і “Управління інженерною діяльністю” (Software Engineering Management).

Вимірювання є інструментом аналізу якості. Вимірювання результатів тестування стосується оцінки якості одержуваного продукту – програмної системи. Історія вимірювань демонструє прогрес досягнення прийнятної якості. Така історія є інструментом менеджменту якості.

5.6.1 Оцінка програм у процесі тестування

1 Вимірювання програм як частина планування й розроблення тестів.

Дані вимірювання можуть базуватися на розмірі програм (наприклад, у термінах кількості рядків коду або функціональних точок) або їхній структурі (наприклад, з погляду оцінки її складності в тих або інших архітектурних термінах). Структурні вимірювання можуть також включати частоту звернень одних модулів програми до інших.

2 Типи дефектів, їхня класифікація й статистика виникнення.

У літературі з тестування зустрічається велика кількість різних класифікацій дефектів. Ефективність тестування може бути досягнута в тому випадку, якщо ми розуміємо, які типи дефектів можуть бути знайдені в процесі тестування програмної системи і як змінюється їхня частота в часі (припускаючи історичну перспективу розвитку системи, а не її збоїв у процесі роботи). Ця інформація дозволяє прогнозувати якість системи й допомагає вдосконалювати процес розроблення у цілому.

Стандарт IEEE 1044-93 класифікує можливі програмні “аномалії”.

3 Щільність дефектів.

Програма, що тестується, може оцінюватися на основі підрахунку й класифікації знайдених дефектів. Для кожного класу дефектів можна визначити відношення між кількістю відповідних дефектів і розміром програми (у термінах обраних метрик оцінки розміру).

4 Життєвий цикл тестів, оцінка надійності.

Статистичні очікування відносно надійності програмної системи можуть використовуватися для ухвалення рішення про продовження або припинення (закінчення) тестування виходячи з заданих параметрів прийнятної якості (наприклад, щільності дефектів заданого класу).

5 Моделі збільшення надійності.

Дані моделі забезпечують можливості прогнозування надійності системи, базуючись на виявлених збоях. Моделі такого роду розбиваються на дві групи – за кількістю збоїв (failure-count) і часу між збоями (time-between-failure).

5.6.2 Оцінка виконаних тестів

1 Метрики покриття / глибини тестування.

Критерії “адекватності” тестування в ряді випадків вимагають систематичного виконання тестів для визначених наборів елементів програми, що задаються її архітектурою або специфікацією. Відповідні метрики дозволяють оцінити ступінь охоплення характеристик системи (наприклад, відсоток різних параметрів продуктивності, що тестуються) і глибину їх деталізації (наприклад, випадкове тестування параметрів продуктивності або з урахуванням граничних значень і т. п.). Такі метрики допомагають прогнозувати імовірнісне досягнення заданих параметрів якості системи.

2 Введення штучних дефектів.

“Своїми руками?! Ніколи!...” – така, звичайно, перша реакція на ідею штучного внесення дефектів, наприклад у програмний код. На практиці цей підхід допомагає класифікувати можливі помилки й наступні за ними збої, застосовуючи надалі отримані результати для моделювання (нехай, часто і інтуїтивного) можливих причин реальних збоїв, виявлених у процесі тестування.

Безумовно, дана техніка повинна використовуватися з максимальною обережністю досвідченими фахівцями, що добре уявляють загальну архітектуру програмної системи, що тестується, і таких, що розбираються в її внутрішніх зв'язках.

3 Оцінка мутацій.

Одержуване в процесі тестування мутацій відношення “убитих” до загальної кількості генерованих мутантів допомагає виміряти ефективність виконуваних тестів. У силу специфіки

такої техніки тестування кількісні оцінки мутацій мають практичне значення тільки для певних типів систем.

4 Порівняння й відносна ефективність різних технік тестування.

Різні дослідження в галузі тестування пов'язані зі спробами порівняння (з погляду досягнутої якості продукту) різних підходів до тестування. Коли ми говоримо про “ефективність” тестування, треба чітко домовитися, що саме ми маємо на увазі під ефективністю, бажано, у кількісному вираженні. Можливі варіанти інтерпретації цього поняття – кількість тестів (даної техніки), необхідних для виявлення першого дефекту; відношення кількості всіх виявлених дефектів до дефектів, знайдених із застосуванням заданого підходу, і т. п. Тільки володіючи такого роду даними, можна говорити про коректність порівняння й оцінки ефективності.

5.7 Процес тестування

Концепції, стратегії, техніки й вимірювання тестування повинні бути об’єднані в єдиний процес тестування як діяльність із забезпечення якості. Процес тестування підтримує роботи з тестування й визначає “правила гри” для членів команди тестування – від планування тестів до оцінки їхніх результатів. Хоча в більшості сучасних методів розроблення, зокрема гнучких (agile) підходів, тестування виходить на передній план і є однією з базових практик, багатобічне тестування й, тим більше, прогнозування на основі отриманих результатів часто підмінюється окремими роботами в цій галузі, що не дозволяють добитися необхідних параметрів якості (що, до речі, ясно показують уже згадувані результати досліджень Standish Group [Chaos, 2004]). Тільки в тому випадку, якщо тестування розглядати як один з важливих процесів усієї діяльності зі створення й підтримки програмного забезпечення, можна добитися оцінки вартості відповідних робіт і, зрештою, дотриматися тих обмежень, які визначені для проекту.

5.7.1 Практичні міркування

1 Програмування без персоналій.

Дуже важливим компонентом успішного тестування є спільне прагнення учасників проекту забезпечити необхідну якість продукту. Менеджери відіграють ключову роль в організації цієї діяльності і на стадії розроблення і в процесі супроводу програмних систем.

2 Посібники з тестування.

Роботи з тестування можуть керуватися різними міркуваннями й критеріями – від управління ризиками до специфікованих сценаріїв роботи програмних систем. У кожному разі бажано, виходячи з ресурсів, кількісних оцінок і інших характеристик, забезпечити використання різних технік тестування для багатобічної оцінки й поліпшення якості одержуваного продукту.

3 Управління процесом тестування.

Роботи з тестування, що ведуться на різних рівнях (див. вище “Рівні тестування”), повинні бути організовані в єдиний (що однозначно інтерпретується) процес на основі урахування 4 елементів і пов'язаних з ними факторів: людей (у тому числі в контексті організаційної структури й культури), інструментів, регламентів і кількісних оцінок (вимірювань). Стандарт життєвого циклу IEEE, ISO/IEC, ДЕРЖСТАНДАРТ Р 12207 не виділяє діяльність з тестування в якості самостійного процесу, однак розглядає відповідні принципи робіт з тестування як невід'ємну частину процесів життєвого циклу й супроводу програмних систем. В іншому розповсюдженому стандарті IEEE 1074 діяльність з тестування також об'єднана з іншими оцінними роботами як інтегральна частина повного життєвого циклу.

4 Документування тестів і робочого продукту.

Документація – складова частина формалізації процесу тестування. Існує стандарт IEEE 829-98 “Standard for Software Test Documentation”, що надає прекрасний опис тестових документів, їхніх зв'язків між собою і з процесом тестування. Серед таких документів можуть бути:

– план тестування;

– специфікація процедури тестування;
– специфікація тестів;
– лог тестів тощо.

Документування тестів у випадку його формального ведення повинно бути актуальним. А якщо ні, то, як і будь-які інші документи, документація з тестування ляже “мертвим капіталом”. У той же час діяльність з тестування у випадку відсутності відповідних регламентів і результатів (у тому числі історичних, для різних проектів) складно піддається оцінці для прогнозування й, тим більше, поліпшенню – у загальному контексті поліпшення процесів. Якщо компанія-розробник не веде відповідної документації з тестування, говорити про сертифікацію або оцінку по тих або інших моделях або стандартах (CMMI, ISO, Sixsigma і т. п.) – просто не є можливим. А це вже питання довіри замовників, що не мали досвіду роботи з конкретною компанією-розробником.

5 Внутрішні й незалежні команди тестування.

Формалізація процесу тестування може включати й організаційну формалізацію команди тестування. У неї можуть входити як члени проектної команди, зокрема ті, що розробляють код, так і зовнішні особи й групи. В ідеалі бажано мати як внутрішню команду тестування, так і зовнішню групу тестування (забезпечення якості). Відповідні організаційні рішення приймаються на основі вартісних характеристик проекту, доступних ресурсів, аналізу вартості тестування, як такого, організаційної культури й т. п.

6 Оцінка вартості й зусиль, а також інші вимірювання процесу.

Ряд метрик, пов'язаних з оцінкою ресурсів, необхідних для тестування, як і оцінка ефективності тестування на різних етапах і рівнях, ґрунтується на точці зору й практиках менеджменту проекту (підрозділу, компанії...) і використовується для оцінки й поліпшення (оптимізації) процесу тестування. Різні техніки, концепції й моделі тестування вимагають різних витрат – за часом і необхідними ресурсами. Результат – вартість тестування, як витратна складова проекту. Розуміння відповідності між

вартістю/зусиллями, необхідними для тієї або іншої форми тестування, є обов'язковою частиною сучасного управління проектами розроблення програмного забезпечення.

7 Закінчення тестування.

Дуже важливим аспектом тестування є рішення про те, у якому обсязі тестування достатнє й коли необхідно завершити процес тестування. Ретельні вимірювання, такі як досягнуте покриття коду тестами або охоплення функціональності, безумовно, дуже корисні. Однак самі по собі вони не можуть визначити критеріїв достатності тестування. Прийняття рішення про закінчення тестування також включає розгляд вартості й ризиків, пов'язаних з потенційними збоями й порушеннями надійності функціонування програмної системи, що тестується. У той же час вартість самого тестування також є одним з обмежень, на основі яких ухвалюється рішення про продовження тих або інших пов'язаних із проектом робіт (зокрема тестування) або про їхнє припинення (див. також “Критерії відбору тестів/критерії адекватності тестів, правила припинення тестування”).

8 Повторне використання й шаблони тестів.

Для доведення тестів до кінця й забезпечення супроводження програмної системи необхідно кожний фрагмент системи тестувати систематично, повторно використовуючи напрацьовані тести. Загальний репозиторій тестових активів повинен перебувати під контролем системи конфігураційного управління для того, щоб будь-які зміни у вимогах або дизайні могли бути відображені у використовуваних наборах тестів, у тому числі з погляду їх розширення новими тестами, якщо цього вимагають відповідні зміни.

Шаблони тестів конструюються на основі тестових рішень, напрацьованих для перевірки певних ситуацій або типових фрагментів програмних систем. Такі шаблони повинні бути документовані з урахуванням повторного використання, включаючи прозорі можливості їхньої адаптації під специфіку програмних рішень, до яких такі шаблони застосовуються.

5.7.2 Тестові роботи

Дана тема дає короткий огляд робіт з тестування. При цьому мається на увазі, що успішне управління тестовими роботами сильно залежить від процесів конфігураційного управління (Software Configuration Management), розглянутих пізніше як самостійна галузь знань.

1 Планування.

Так само, як і інші аспекти управління проектами, роботи з тестування мають плануватися заздалегідь. Як мінімум, на рівні організації відповідного процесу. Ключові аспекти планування тестової діяльності включають:

– координацію персоналу;

– управління обладнанням та іншими засобами, необхідними для організації тестування;

– планування обробки небажаних результатів (тобто управління певними видами ризиків).

У випадку одночасної підтримки й супроводу декількох версій програмної системи або декількох систем необхідно приділяти особливу увагу плануванню часу, зусиль і ресурсів, пов'язаних із проведенням робіт з тестування. Дана позиція перегукується з питаннями керування портфелями проектів з погляду загального управління проектами.

2 Генерація сценаріїв тестування.

Створення тестових сценаріїв ґрунтується на рівні й конкретних техніках тестування. Тести повинні перебувати під управлінням системи конфігураційного керування й описувати очікувані результати тестування.

3 Розроблення тестового оточення.

Використовуване для тестування оточення має бути сумісне з інструментами програмної інженерії (будуть розглядатися пізніше як тема самостійної галузі знань). Це оточення повинне забезпечувати розроблення і контроль тестових сценаріїв, ведення журналу тестування і можливості відновлення очікуваних і відслідковуваних результатів тестування, самих сценаріїв, а також інших активів тестування.

4 Виконання тестів.

Виконання тестів повинно містити основні принципи ведення наукового експерименту:

– повинні фіксуватися всі роботи й результати процесу тестування;

– форма документування таких робіт і їхніх результатів повинна бути такою, щоб відповідний зміст був зрозумілим, однозначно інтерпретувався і повторювався іншими особами (не тими, хто спочатку проводив тестування);

– тестування повинно проводитися відповідно до заданих і документованих процедур;

– тестування повинне проводитися над версією й конфігурацією програмної системи, які однозначно ідентифікуються.

Низка питань виконання тестів і інших робіт з тестування висвітлена у стандарті IEEE 1008-87.

5 Аналіз результатів тестування.

Для визначення успішності тестів їхні результати повинні оцінюватися і аналізуватися. У більшості випадків “успішність” тестування має на увазі, що програмне забезпечення, що тестується, функціонує так, як очікувалося, і в процесі роботи не призводить до непередбачених наслідків. Не всі такі наслідки обов'язково є збоями, вони можуть сприйматися як “перешкоди”. Однак будь-яка непередбачена поведінка може стати джерелом збоїв при зміні конфігурації або умов функціонування системи, тому вимагають уваги, як мінімум, з погляду ідентифікації причин таких перешкод. Перед усуненням виявленого збою необхідно визначити й зафіксувати ті зусилля, які необхідні для аналізу проблеми, налагодження й усунення. Це дозволить подалі забезпечити більшу глибину вимірювань, а отже, у перспективі мати можливість поліпшення самого процесу тестування. У тих випадках, коли результати тестування особливо важливі, наприклад через критичність виявленого збою, може бути сформована спеціальна група аналізу (review board).

6 Звіти про проблеми/журнал тестування.

У багатьох випадках у процесі тестової діяльності ведеться журнал тестування, що фіксує інформацію про відповідні роботи: коли проводиться тест, який тест, ким проводиться, для якої конфігурації програмної системи (у термінах параметрів і в термінах версії контексту конфігураційного управління, що ідентифікується) і т. п. Несподівані або некоректні результати тестів можуть записуватися в спеціальній підсистемі ведення звітності по збоях (problem-reporting system), забезпечуючи формування бази даних, використовуваної для налагодження, усунення проблем і подальшого тестування. Крім того, аномалії (перешкоди), які не можна ідентифікувати як збої, також можуть фіксуватися в журналі й/або системі ведення звітності по збоях. У кожному разі документування таких аномалій знижує ризики процесу тестування й допомагає вирішувати питання підвищення надійності самої системи, що тестується. Звіти по тестах можуть бути входом для процесу управління змінами й генерації запитів на зміни (change request) у рамках процесів конфігураційного управління.

7 Відстеження дефектів.

Збої, виявлені в процесі тестування, найчастіше породжуються дефектами й помилками, що присутні в програмній системі, що тестується (також вони можуть бути наслідком поведінки операційного й/або тестового оточення). Такі дефекти можуть (і, найчастіше, повинні) аналізуватися для визначення моменту й місця першої появи даного дефекту в системі, які типи помилок стали причиною цих дефектів (наприклад, погано сформульовані вимоги, некоректний дизайн, втрати пам'яті і т. д.) і коли вони могли б бути виявлені вперше. Уся ця інформація використовується для визначення того, як може бути поліпшений сам процес тестування й наскільки критична необхідність таких поліпшень.

6 Супроводження програмного забезпечення

Результат зусиль з розроблення програмного забезпечення полягає в передачі в експлуатацію програмного продукту, що

задовольняє вимоги користувачів. Відповідно у процесі експлуатації продукт буде змінюватися або еволюціонувати. Пов'язано це з виявленням при реальному використанні прихованих дефектів, змінами в операційному оточенні, необхідністю покриття нових вимог і т. п.

Фаза супроводження в життєвому циклі звичайно починається відразу після приймання/передачі продукту й діє протягом періоду гарантії або, частіше, технічної підтримки. Однак сама діяльність, пов'язана з супроводженням, починається набагато раніше.

Супроводження програмного забезпечення є складовою частиною життєвого циклу. На жаль, так склалося, що питанням супроводження приділяється суттєво менше уваги, ніж іншим фазам життєвого циклу. Історично в більшості організацій розроблення програмних систем явно у фаворі порівняно з діяльністю з супроводження. Однак така ситуація поступово починає мінятися (досить, хоча б глянути на частоту згадувань ITIL[1] – IT Infrastructure Library, що приділяє особливу увагу питанням підтримки й супроводження інфраструктури інформаційних технологій). Більшою мірою, як відзначає SWEBOK, це пов'язано зі скороченням інвестицій організацій безпосередньо в розроблення програмних систем з метою збільшення строків використання вже існуючого й застосовуваного ПЗ. Звичайно, це не єдина причина. Скоріше питання постійно мінливих бізнес-потреб, динаміка бізнесу й бажання підвищити віддачу від уже експлуатованих систем призводить до посилення ролі підтримки й супроводження програмного забезпечення й природної інтеграції такої діяльності в бізнес-процеси підрозділів інформаційних технологій.

Якщо проблема 2000 р. свого часу вплинула на зміну ставлення до супроводження на Заході, то розширення застосування продуктів Open Source в усьому світі й пов'язана з ним хвиля надій на одержання дешевого вирішення існуючих завдань призвела до того, що питання супроводження вийшли

[1] ITIL, зокрема, визначає три аспекти управління життєвим циклом додатків – визначення вимог, проектування й розроблення і, нарешті, супроводження. Усе це в контексті програмного забезпечення стосується діяльності з управління додатками – Application Management в ITIL ICT Infrastructure Management (ICT – Information and Communications Technology).

для багатьох організацій на перший план. Ситуація в багатьох IT-підрозділах показує, що такі надії виправдалися тільки частково. Використання продуктів Open Source не стало дешевою альтернативою і в ряді випадків призвело навіть до більших проектних витрат саме через недостатньо опрацьовану політику експлуатації й супроводження побудованих на їхній основі прикладних рішень. Це в жодному разі не означає, що хвиля Open Source "захлинулася". Це означає тільки, що ігнорування оцінки вартості супроводження призвело до перевищення бюджетів, нестачі ресурсів і, зрештою, частого провалу ініціатив з використання таких продуктів у корпоративному середовищі. Неготовність розглядати життєвий цикл у часі як триваючий і після передачі системи в експлуатацію неопрацьованість відповідних процедур коректування продукту після його випуску – основне лихо і в бізнес-середовищі, для якого програмне забезпечення лише інструмент, і в компаніях-інтеграторах, "що забувають" про необхідність розвитку успіху після впровадження системи в замовника, і в незалежних постачальників програмних продуктів, які, випускаючи нову версію кращого у своєму класі продукту, починають працювати над новою версією, приділяючи недостатню увагу підтримці й відновленню вже існуючих версій.

Супроводження програмного забезпечення в SWEBOK визначається як уся сукупність діяльності, необхідної для забезпечення ефективної (з погляду витрат) підтримки програмних систем. Ці роботи виконуються як перед введенням системи в експлуатацію, так і після цього. Попередні роботи включають планування діяльності з супроводження системи, а також організацію переходу до її повнофункціонального використання. Якщо нова система повинна замінити стару систему, призначену для вирішення тих же завдань просто на новому рівні ефективності, вартості використання, нових функціональних можливостей, у цьому випадку важливо забезпечити плавний перехід зі старої системи на нову, максимально природний для користувачів. Із цим пов'язано не тільки планування, наприклад перенесення інформації, збереженої у відповідних базах даних, але й навчання користувачів, підготовка, настроювання й перевірка "бойової"

конфігурації, визначення послідовності операцій, організація й навчання служби підтримки (help-desk) і т. п.

Галузь знань “Супроводження програмного забезпечення” пов'язана з іншими аспектами програмної інженерії. По суті опис цієї галузі знань безпосередньо перетинається з усіма іншими дисциплінами. На рисунку 6.1 наведена схема, що представляє аспекти супроводження програмного забезпечення.

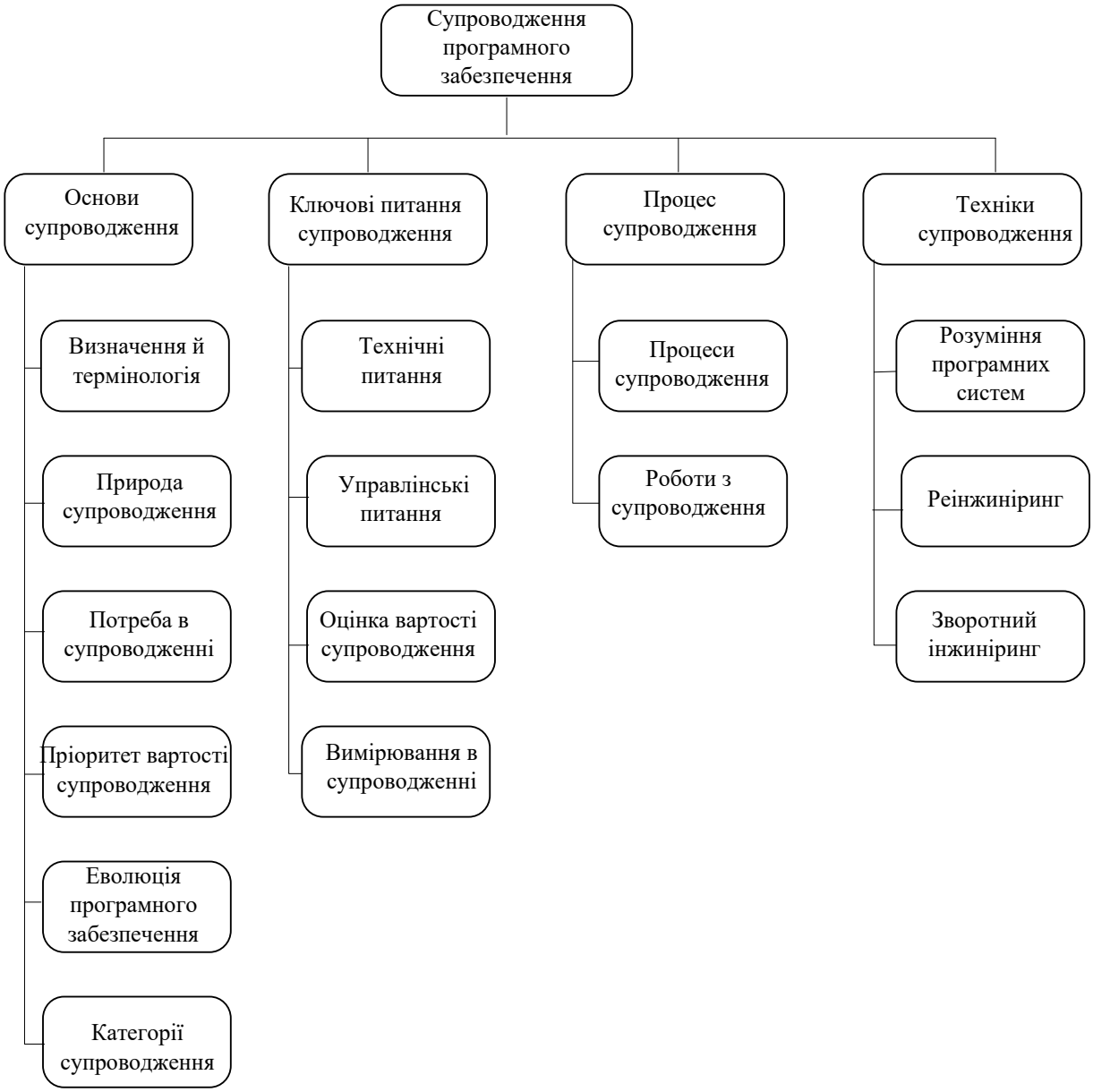

Рисунок 6.1 – Галузь знань “Супроводження програмного забезпечення”

6.1 Основи супроводження програмного забезпечення

Ця секція включає концепції й термінологію, що формують основи розуміння ролі й змісту робіт із супроводження програмних систем. Теми даної секції надають відповідні визначення й описують, чому саме існує потреба в супроводі. Категорії супроводження критично важливі для розуміння суті обговорюваних питань.

6.1.1 Визначення й термінологія

Супроводження програмного забезпечення визначається стандартом IEEE Standard for Software Maintenance (IEEE 1219) як модифікація програмного продукту після передачі в експлуатацію для усунення збоїв, поліпшення показників продуктивності й/або інших характеристик (атрибутів) продукту або адаптації продукту для використання в модифікованому оточенні. Цікаво, що даний стандарт також стосується питань підготовки супроводження до передачі системи в експлуатацію, однак структурно це зроблено на рівні відповідного інформаційного додатка, включеного в стандарт.

У свою чергу стандарт життєвого циклу 12207 (IEEE, ISO/IEC, ГОСТ Р ИСО/МЭК) позиціонує супроводження як один з головних процесів життєвого циклу. Цей стандарт описує супроводження як процес модифікації програмного продукту в частині його коду й документації для вирішення виникаючих проблем при експлуатації або реалізації потреб у поліпшеннях тих або інших характеристик продукту. Завдання полягає в модифікації продукту за умови збереження його цілісності. Міжнародний стандарт ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance) визначає супровід програмного забезпечення в тих же термінах, що й стандарт 12207, надаючи особливе значення роботам з підготовки до діяльності з супроводження до передачі системи в реальну експлуатацію, наприклад питанням планування регламентів і операцій з супроводження.

6.1.2 Природа супроводження

Супроводження підтримує функціонування програмного продукту протягом усього операційного життєвого циклу, тобто періоду його експлуатації. У процесі супроводження фіксуються

й відслідковуються запити на модифікацію (також називані “запитами на зміни” – change requests, зокрема в контексті конфігураційного управління), оцінюється вплив пропонованих змін, модифікується код і інші активи (артефакти) продукту, проводиться необхідне тестування й, нарешті, випускається оновлена версія продукту. Крім того, проводиться навчання користувачів і забезпечується їхня щоденна підтримка при роботі з поточною версією продукту. У SWEBOK відзначається, що супроводження, з погляду операцій відстеження й контролю, має більший зміст, ніж розроблення (у загальному розумінні). Обсяг і активність операцій з контролю розроблення більшою мірою залежить від сталих практик, внутрішньокорпоративних регламентів і вимог, а також застосовуваних методологій і концепції управління (зокрема проектного менеджменту). Так чи інакше відстеження й контроль – ключові елементи діяльності з супроводження програмного забезпечення (як і інших IT-активів підприємства).

Стандарт 12207 визначає поняття “maintainer” – у відповідному ДСТ він іменується як “персонал супроводження”, припускаючи організацію, що виконує роботи з супроводження. SWEBOK використовує даний термін, також і відносно осіб (individuals), що проводять певні роботи з супроводження, на відміну, наприклад, від розробників, що займаються реалізацією системи в програмному коді.

Стандарт життєвого циклу 12207 також ідентифікує основні роботи з супроводження: реалізація процесу супроводження, аналіз проблем і модифікацій (змін), реалізацій модифікацій, огляд (оцінка)/прийняття рішень з супроводження, міграція (з однієї версії програмного продукту на іншу, з одного продукту на іншій) і виведення системи з експлуатації.

Фахівці з супроводження (персонал супроводження) можуть одержувати знання про програмний продукт безпосередньо від розробників. Взаємодія з розробниками й раннє залучення цих фахівців допомагає зменшити зусилля, необхідні для адекватного супроводження програмної системи. Вірогідно, що передача знань персоналу супроводження, його навчання має починатися не пізніше початку дослідної експлуатації продукту. А якщо ні, то зусилля на одночасну підтримку прикладної системи й

навчання відповідних фахівців не тільки перевищить реально припустимі норми завантаження персоналу (як групи або служби супроводу й технічної підтримки, так і розробників системи), але й знизить ефективність підтримки користувачів на критично важливому етапі первісного використання нової системи. За емпіричними даними звичайно, залежно від складності системи, пік навантаження на службу супроводження припадає на 2 – 6 перших тижнів з моменту передачі системи в реальну експлуатацію, тим більше, при відмові від використання старої системи або її попередньої версії. SWEBOK відзначає, що у деяких випадках інженери (що створювали систему) не можуть бути притягнуті до навчання й підтримки персоналу супроводження. Особливо часто це стосується систем, що тиражуються, або "коробкових" систем. Це створює додаткові труднощі для фахівців, що забезпечують супроводження. У той же час інженери, що займаються технічною підтримкою (трохи більш вузьке коло в команді супроводження, що включає менеджерів, адміністраторів і інших фахівців), повинні (залежно від типу продукту) мати доступ до активів проекту (наприклад, опису його внутрішньої архітектури), включаючи код, документацію й т. п. Саме в такий спосіб починає формуватися інформаційна інфраструктура служби технічної підтримки й супроводження у виробників програмних продуктів.

Практика показує, що інженери з технічної підтримки виробника програмного забезпечення (не тільки "коробкового", але й створюваного, що настроюється інтеграторами, що володіють власними програмними рішеннями) повинні не просто мати доступ до всіх ключових активів проекту (код, документація, специфікації вимог, внутрішні моделі й т. п.), але до їхніх обов'язків входить створення "патчей" (patch – "латка"), виправлень помилок і, в особливих випадках, такі зміни до випуску нової версії продукту створюються з залученням безпосередньо розробників продукту (груп і підрозділів R&D – Research and Development). При цьому розробники продукту інформуються про знайдені помилки й у випадку знаходження відповідних рішень фахівцями технічної підтримки такі рішення передаються розробникам для того, щоб ті або включили такі зміни в нову версію програмного продукту (безумовно, у випадку

успішного проходження всіх необхідних тестів), або знайшли більш адекватне рішення в контексті нової функціональності або тих змін, які включені в нову версію продукту. До обов'язків інженерів служби супроводження в загальному випадку входить перевірка користувацького сценарію, що призводить до збою; ідентифікація причин збою, тобто локалізація помилки/причин її появи; надання відповідних виправлень або за неможливості створення таких на даному етапі або в заданий термін – надання обхідного шляху вирішення проблеми для досягнення необхідних бізнес-завдань (такі обхідні шляхи звичайно називають “workaround”); документування всіх робіт і операцій; розміщення опису проблеми та її рішення в базу знань служби супроводження; передача всієї інформації розробникам; своєчасне інформування користувача про статус запиту й деякі інші роботи, зміст яких може варіюватися залежно від регламентів і корпоративних стандартів у конкретній організації або параметрів контракту на супроводження і технічну підтримку, якщо така є.

6.1.3 Потреба в супроводженні

Супроводження необхідне для забезпечення того, щоб програмний продукт протягом усього періоду експлуатації задовольняв вимоги користувачів. Діяльність з супроводження застосовна для програмного забезпечення, створеного з використанням будь-якої моделі життєвого циклу (наприклад, спіральної) і методології розроблення. На перший погляд, це твердження може здатися тривіальним. Однак зверніться до свого досвіду розроблення й використання різного програмного забезпечення. Напевно, ви знайдете випадки з власної практики або практики колег, коли настільки очевидне твердження добре б донести до розробника того або іншого програмного продукту. Зміни програмної системи можуть бути обумовлені діями з коректування її поведінки або не пов'язані з необхідністю коректування (припускаючи вже не виправлення помилок, а, наприклад, підвищення продуктивності або розширення функціональності).

У загальному випадку роботи з супроводження повинні проводитися для вирішення таких завдань:

– усунення збоїв;

– поліпшення дизайну;

– реалізація розширень функціональних можливостей;

– створення інтерфейсів взаємодії з іншими (зовнішніми) системами;

– адаптація (наприклад, портування) для можливості роботи на іншій апаратній платформі (або оновленій платформі), застосування нових системних можливостей, функціонування в середовищі оновленої телекомунікаційної інфраструктури й т. п.;

– міграція успадкованого (legacy) програмного забезпечення;

– виведення програмного забезпечення з експлуатації.

Діяльність персоналу супроводження включає чотири ключові аспекти:

– підтримка контролю (керованості) програмного забезпечення протягом усього циклу експлуатації;

– підтримка модифікацій програмних систем;

– удосконалювання існуючих функцій (зважаючи на все, у цьому випадку мається на увазі функції не програмного забезпечення, а процесів супроводження й функції персоналу супроводження як такі);

– запобігання падіння продуктивності програмної системи до неприйнятного рівня.

Говорячи про запобігання деградації продуктивності, ми повинні розуміти, що це при всьому бажанні вдосконалювання системи може робитися й за рахунок відновлення потужності апаратної частини й/або відповідної телекомунікаційної інфраструктури, якщо це більш обґрунтовано, ніж модифікація самої програмної системи. Насправді це питання того, що виявиться дешевше (і менш ризикованим), тобто пов'язане з витратами/вартістю відповідних робіт, устаткування й підтримки оновленого системного оточення (що, на жаль, часто також не враховується навіть при більш-менш сталій практиці супроводження).

6.1.4 Пріоритет вартості супроводження

Роботи з супроводження споживають якщо не більшу, то значну частину фінансових ресурсів життєвого циклу програмного забезпечення. Загальне розуміння супроводження має на увазі лише усунення збоїв. Однак дослідження й опитування протягом багатьох років показують, що більше 80 % зусиль з супроводження пов'язані не стільки з усуненням збоїв, скільки з іншими роботами, не пов'язаними з виправленням дефектів. Багато менеджерів з супроводження поєднують у звітності питання розширення функціональності й виправлення помилок у підтримуваних програмних системах. Таке змішання якісно різних робіт призводить до неправильного уявлення про реальну, насправді не настільки високу вартість супроводження в частині усунення дефектів. Розуміння різних категорій робіт у рамках діяльності з супроводження допомагає зрозуміти структуру реальних витрат. Крім того, розуміння факторів, що впливають на можливості супроводження системи, допомагають не тільки зберігати необхідний рівень витрат, але й знижувати їх.

Існують як технічні, так і інші (наприклад, організаційні, що, мабуть, найбільше впливають на об'єм витрат) фактори, що мають вплив на вартість супроводження, у цілому:

– тип додатка;

– новизна програмного забезпечення;

– наявність і кваліфікація персоналу з супроводження;

– тривалість використання програмної системи;

– характеристики й специфіка апаратної частини (а також телекомунікаційної інфраструктури);

– якість дизайну (наприклад, модульність або масштабованість), коду, документації й відповідних робіт з тестування системи.

6.1.5 Еволюція програмного забезпечення

У 1969 році М.М. Леман уперше зв'язав діяльність з супроводження й питання еволюції програмного забезпечення. Результати більш ніж 20-річних досліджень на чолі з Леманом призвели до формулювання ряду важливих положень, ключове з яких стверджує, що діяльність з супроводження по суті являє собою еволюційне розроблення програмних систем. Прийняттю тих або інших рішень у процесі супроводження допомагає

розуміння того, що відбувається з програмною системою в процесі її експлуатації. Існуюче (особливо корпоративне) програмне забезпечення ніколи не буває повністю завершеним і продовжує еволюціонувати протягом усього строку експлуатації. У процесі еволюціонування програмна система стає все більш складною доти, поки не прикладають спеціальних зусиль (у тому числі в рамках спеціального проекту з модифікації) зі зменшення його складності.

У той же час, якщо можна виділити тенденції розвитку програмної системи і її поведінка достатня стабільна, її еволюціонування можна виміряти. Останніми роками робляться спроби розробити відповідні моделі оцінки зусиль з супроводження. У результаті вже створюються певні засоби (чисельні й інструментальні) управління роботами з супроводження.

6.1.6 Категорії супроводження

Багато джерел, зокрема стандарт IEEE 1216, визначають три категорії робіт із супроводження: коректування, адаптація й удосконалювання. Така класифікація була оновлена в стандарті ISO/IEC 14764 Standard for Software Engineering – Software Maintenance введенням четвертої складової. Таким чином, сьогодні говорять про чотири категорії супроводження:

– коригувальне супроводження (corrective maintenance): “реактивна” модифікація програмного продукту, виконувана вже після передачі в експлуатацію для усунення збоїв;

– супроводження, що адаптує (adaptive maintenance): модифікація програмного продукту на етапі експлуатації для забезпечення продовження його використання з заданою ефективністю (з погляду задоволення потреб користувачів), що змінився або перебуває в процесі зміни оточення; у першу чергу мається на увазі зміна бізнес-оточення, що породжує нові вимоги до системи;

– супроводження, що вдосконалює (perfective maintenance): модифікація програмного продукту на етапі експлуатації для підвищення характеристик продуктивності й зручності супроводження;

– профілактичне супроводження (preventive maintenance): модифікація програмного продукту на етапі експлуатації для ідентифікації й запобігання прихованих дефектів до того, коли вони призведуть до реальних збоїв.

ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance) класифікує адаптивне супроводження й супроводження, що удосконалює, як роботи з розширення функціональності продукту. Цей стандарт також поєднує коригувальну й профілактичну діяльність у загальну категорію робіт з коректування системи. Профілактичне супроводження (новітня категорія робіт із супроводження) найчастіше проводиться для програмних систем, пов'язаних з питанням безпеки людей.

Таблиця 6.1 – Категорії супроводження програмного забезпечення

	Коригувальні роботи	Роботи з розширення
«Проактивний» підхід	Профілактичне супроводження	Супроводження, що вдосконалює
«Реактивний» підхід	Коригувальне супроводження	Супроводження, що адаптує

6.2 Ключові питання супроводження програмного забезпечення

Для забезпечення ефективного супроводження програмних систем необхідно вирішувати цілий комплекс питань і проблем, пов'язаних з відповідними роботами. Необхідно розуміти, що процес супроводу висуває унікальні технічні й управлінські вимоги до персоналу, який займається супроводженням і в першу чергу фахівцям-інженерам. Спроба знайти дефект у продукті, що містить 500 тисяч рядків коду, написаних іншими інженерами, – яскравий приклад труднощів, з якими доводиться зустрічатися інженерам з супроводження. Інший приклад, уже організаційний, – постійна боротьба за ресурси з розробниками (це найчастіше проявляється в питаннях відволікання розробників від поточної роботи для допомоги у вирішенні проблем супроводження, а

також у конкуренції за пріоритети фінансування розроблення нової системи або супроводження існуючої). Одночасне планування перспективної версії системи, реалізація наступної версії й підготовка критичних патчей для поточної версії – ще один класичний приклад проблем, з якими доводиться зустрічатися в процесі експлуатації програмного забезпечення.

Даний підрозділ представляє деякі технічні й управлінські питання, пов'язані з супроводженням програмних систем. Ці питання й проблеми згруповані в набір тем:

– технічні питання;
– управлінські питання;
– оцінка вартості;
– вимірювання.

6.2.1 Технічні питання

1 Обмежене розуміння.

Обмежене розуміння має на увазі, як швидко інженер з супроводження може зрозуміти, де необхідно внести виправлення або зміни в код системи, яку він не розробляв. Дослідження показують, що від 40 до 60 % зусиль з супроводження витрачається на аналіз і розуміння супроводжуваного програмного забезпечення. Формування цілісного погляду про систему має велике значення для інженерів. Цей процес більш складний у випадку аналізу текстового представлення системи – її вихідного коду, особливо коли процес еволюції системи від складання до складання, від релізу до релізу в ньому ніяк не відзначений, не документований і коли розробники не можуть пояснити історію й структуру змін, що, на жаль, трапляється досить часто.

Для об'єктно-орієнтованих програм якісно спрощує задачу розуміння коду використання UML-інструментарію, здатного на основі коду відновити не тільки модель класів, але і їхню взаємодію у формі діаграм класів (class diagram), комунікацій або співробітництва (collaboration в UML1.x, перейменована в communication в UML 2.0), і особливо послідовностей (sequence diagram), що демонструє структуру взаємних викликів у часі. Якщо відповідний інструментарій надає одночасну візуалізацію

коду й діаграми і забезпечує взаємну синхронізацію їх з погляду навігації (вибір методу в кожній із представлених діаграм автоматично позиціонує відповідним чином редактор коду й навпаки), такі засоби автоматизації можуть якісно скоротити час, необхідний для формування уявлення про систему, іноді навіть не в рази, а на порядок (звичайно, при достатньому рівні знання використовуваних технологій з боку інженера з супроводження). Якщо до цього додати документованість (і доступність відповідних активів – специфікацій, моделей) архітектури й ключових технологічних рішень з боку розробників системи, обговорюване питання звичайно не стає тривіальним, однак перетворюється в цілком розв'язувану задачу. Загалом кажучи, використання відповідних засобів автоматизації побудови моделей за кодом (завдання зворотного інжинірингу – reverse engineering) є обґрунтованою практикою вивчення будь-якої системи або фреймворку. Досвід показує, що при достатній кваліфікації інженера формування загального архітектурного уявлення про систему (або фреймворк), розуміння того, які технологічні й структурні підходи й шаблони використовувалися при її побудові, дозволяє вирішувати виникаючі питання коректування коду й розширення функціональності системи, не порушуючи загальних принципів її побудови, природно забезпечуючи її еволюцію, без зашкодження її цілісності. При такому розумінні, навіть не заглядаючи в код системи або фреймворку, інженер здатний з дуже великою ймовірністю припустити можливі причини збою, а в загальному випадку і будь-яких аспектів поведінки системи.

2 Тестування.

Вартість повторення повного набору тестів для основних модулів системи може бути істотною як за часом, так і за вартістю. Для супроводження системи особливо значущим є вибіркове регресійне тестування системи або її компонент для перевірки того, що внесені зміни не призвели до ненавмисної зміни поведінки програмного забезпечення. Питання полягає в тому, що часто складно знайти час для необхідного тестування. Не меншою проблемою є й координації в проведенні тестів різними членами групи супроводження, що займаються

розв’язанням різних задач. Якщо ж система виконує критичні для бізнесу функції, тимчасове виведення системи з експлуатації (як говорять, переведення системи в offline) для виконання тестів часто виявляється просто неможливим.

Таким чином, одним із ключових питань супроводження є організація робіт з тестування модифікацій експлуатованих систем, аж до попереднього планування й розроблення регламентів, відповідно до яких, наприклад, ґрунтуючись на оцінці критичності запитів на зміни (як дефектів, так і важливих розширень – будь то нова функціональність або необхідне розширення інтеграційних можливостей) у модулях, що зачіпаються, персоналом супроводження будуть проводитися стандартні процедури. До таких процедур, нарівні з фіксацією запитів і проведених робіт, можуть і, скоріше за все, повинні належати аналіз впливу змін (impact analysis), оцінка ризиків, тестування (різними методами, у різному обсязі), випуск попередніх версій патчей/оновлень в обмежене використання (якщо це дозволяє специфікація системи), використання “клону” системи (розгортання її на ідентичному устаткуванні в ідентичній конфігурації) і т. п.

3 Аналіз впливу.

Аналіз впливу описує, як проводити (зокрема з погляду ефективності витрат) повний аналіз можливих наслідків і впливів змін, внесених в існуючу систему. Персонал супроводження повинен мати необхідні знання про специфіку системи (в ідеальному випадку мати повне уявлення про систему на рівні її розробників) – її зміст й структуру. Інженери використовують ці знання для виконання робіт з аналізу впливу, ідентифікуючи всі системи[1] і програмні продукти, на які можуть вплинути зміни, внесені в програмну систему, що обслуговується. При цьому повинні бути визначені ризики, пов'язані з внесенням обговорюваних змін.

Запити на зміни[2] (change requests – CR) інколи згадувані як запити на модифікацію (modification request – MR), так звані звіти

[1] Як ми бачимо з опису даних робіт у SWEBOK, ідеться не тільки про компоненти системи, але й про її оточення, включаючи інші системи, що функціонують у тому самому операційному/системному оточенні.

про проблеми (problem report – PR), повинні аналізуватися й трансформуватися в терміни програмної системи. Ці кроки виконуються після того, як відповідний запит на зміну починає оброблятися в рамках процесу управління змінами або, як прийнято називати, конфігураційного управління, і фіксується в системі конфігураційного управління.

Цілі аналізу впливу можуть бути сформульовані так:

– визначення сутності змін для задавання робіт із планування й реалізації;

– одержання максимально можливої оцінки ресурсів, необхідних для проведення відповідних робіт;

– аналіз вартості й вигоди від внесення запитаних змін (звичайно стосується побажань, запитів на розширення системи);

– обговорення складності питань, пов'язаних із внесенням відповідних змін.

Складність вирішення питання, поставленого відповідним запитом на зміни, часто є основним фактором визначення того, коли і як буде вирішена проблема. Інженери ідентифікують компоненти, у які необхідно внести зміни. Звичайно розглядається кілька варіантів вирішення проблеми й виробляється (також обов'язково фіксуються у відповідній системі обробки запитів на зміни) найбільш оптимальний шлях її вирішення.

При цьому оптимальність шляху не завжди означає найбільш ”гарне” технологічне рішення. Іноді це може бути тимчасове рішення, може бути таке, що навіть порушує архітектурні шаблони системи, однак обґрунтоване з погляду строків і вартості його реалізації. У той же час результати аналізу направляються розробникам системи, що звичайно працюють над наступною версією, для включення відповідної зміни вже в рамках прийнятого стилю кодування, угод, архітектурних шаблонів і т. п. Безумовно, такий шлях багатьом може здатися просто неетичним, з погляду “сьогодення” інженерного підходу. Однак якщо розробники готують наступну версію системи, зачіпаючи модуль, що модифікується службою супроводження, з

[2] Звичайно запити на зміни поділяють на дві категорії – “побажання” (suggestions), що належать до розширення системи, і “звіти про помилки” (defect або bug report), що направляються користувачами в службу супроводження або інженерами з тестування розробникам.

погляду бізнес-рішень, “некрасивий”, але швидкий шлях досягнення необхідної поведінки системи в більшості випадків буде виглядати більш обґрунтованим, ніж прийняття на себе персоналом супроводження функцій розробників системи. Іноді, якщо необхідна зміна не настільки критична, щоб рішення було надано “учора” (хоча користувачі практично завжди саме так характеризують свої запити в термінах пріоритету), логічним виглядає відкладання проведення відповідних модифікацій і передача цих робіт безпосередньо розробникам. Як це часто можна почути – “буде доступно в наступному релізі”. Нічого не нагадує? Але економічно це часто буває більш ніж виправданим.

Якщо програмне забезпечення споконвічно розроблялося з урахуванням подальшої підтримки, це може істотно полегшити аналіз впливів, як однієї із ключових робіт із супроводження.

4 Можливість супроводження.

Можливість супроводження, або супроводжуваність програмної системи, визначається, наприклад, глосарієм IEEE (стандарт 610.12-90 Standard Glossary for Software Engineering Terminology, відновлення 2002 р.) як легкість супроводження, розширення, адаптації й коректування для задоволення заданих вимог. Стандарт ISO/IEC 9126-01 (Software Engineering – Product Quality – Part 1: Quality Model, 2001 р.) визначає можливість супроводження як одну з характеристик якості.

Для зменшення вартості подальшого супроводження протягом усього процесу розроблення необхідно специфікувати, оцінювати й контролювати характеристики, що впливають на можливість супроводження. Якщо такі роботи проводяться регулярно, це полегшує подальше супроводження, підвищуючи його супроводжуваність (зокрема як характеристику якості). Часто цього складно добитися, тому що, на жаль, такого роду характеристики ігноруються при розробленні. Розробники зайняті іншими запланованими роботами й також часто нехтують вимогами, пропонованими для супроводжуваності системи.

Однією з ключових проблем супроводження є відсутність системної документації, що заважає формуванню розуміння системи й, як наслідок, неможливості адекватного аналізу впливу. Ця й інші проблеми можуть бути вирішені при

використанні систематичного підходу до побудови зрілих процесів, застосуванні відповідних технік і автоматизації необхідних задач з підтримки життєвого циклу за допомогою спеціалізованих інструментальних засобів.

6.2.2 Управлінські питання

1 Узгодження з організаційними цілями.

Організаційні цілі описують, як продемонструвати повернення інвестицій від діяльності з супроводження програмного забезпечення. Звичайно розроблення ведеться на проектній основі, з певними часовими й бюджетними обмеженнями. Головний акцент при цьому робиться на випуску системи, що відповідає потребам користувачів, у заданий термін і в рамках бюджету. На відміну від цього, супроводження системи переслідує цілі максимального продовження строку експлуатації програмного забезпечення. Такий підхід може ґрунтуватися на необхідності оновлення й розширення програмного забезпечення, як відгуку на мінливі потреби користувачів. При цьому оцінка повернення інвестицій стає більш складною й призводить до формування точки зору старшого менеджменту, що діяльність з супроводження споживає значну частину ресурсів без явно вираженої й кількісно обумовленої віддачі для організації.

2 Проблеми кадрового забезпечення[1].

Дана тема стосується питань залучення й утримання кваліфікованого персоналу з супроводження. Часто робота з супроводження не виглядає привабливою, інженери з підтримки сприймаються як фахівці “другого класу” (досить часто використовується стійке вираження “second-class citizens”), що призводить до безумовного падіння духу колективу, відповідального за підтримку систем.

Це серйозний виклик для менеджерів, відповідальних за питання супроводження й, загалом кажучи, є класичним завданням загального менеджменту. Вирішення цього завдання у першу чергу, перебуває в руках вищого менеджменту, що формує

[1] Такий переклад, замість просто “кадрового забезпечення”, більшою мірою відповідає прийнятому використанню терміну staffing (англ. вкомплектовування персоналом). Часто staffing має на увазі й високу плинність кадрів.

відповідний стиль відносин між функціональними й допоміжними підрозділами. На більш високому рівні для організацій і бізнес-споживачів інформаційних технологій це завдання пов'язане з внутрішньокорпоративними департаментами автоматизації в цілому, які дуже часто сприймаються тільки як центри витрат, а інформаційні технології не розглядаються як актив. У результаті така позиція призводить до зниження ефективності роботи підрозділів автоматизації, а отже, і падіння якості інформаційного забезпечення бізнесу, що позначається, у переважній більшості випадків, і на бізнесі як такому.

3 Процес.

Процес (у загальному випадку життєвий цикл) є набором робіт (activities), методів, практик і свого роду трансформацій, які використовуються людьми для розроблення й супроводження програмних систем і асоційованих з ними продуктів. На рівні процесу діяльність з супроводження програмного забезпечення має дуже багато спільного з розробленням, наприклад, у частині конфігураційного управління, що є критично важливою складовою обох видів діяльності. У той же час супроводження включає роботи, не представлені в процесі розроблення. Ця діяльність вимагає від менеджменту спеціальної уваги.

4 Організаційні аспекти супроводження.

У першу чергу організаційні питання мають на увазі, яка організація буде відповідати і/або які функції необхідно виконувати для забезпечення діяльності з супроводження. Команда, що розробляла програмний продукт, далеко не завжди відповідає за його супроводження. Це не тільки стандартне управлінське рішення незалежних постачальників програмного забезпечення, але також таке рішення, що часто зустрічається в організаціях, які використовують програмні продукти з метою автоматизації своїх бізнес-функцій.

При вирішенні питання, де (і ким) будуть здійснюватися функції з супроводження, може бути ухвалене рішення залишити їх безпосередньо тим, хто розробляв систему (як у термінах організації/компанії, так і припускаючи безпосередньо колектив розробників), або передати іншій команді чи стороні. Часто вибір супровідної організації здійснюється виходячи з тих міркувань,

які виглядають обґрунтованими для забезпечення адекватної підтримки системи й можливості її еволюціонування для задоволення мінливих потреб користувачів. На жаль (чого у принципі і варто очікувати), універсальних підходів у вирішенні даного питання, ким буде супроводжуватися система, немає. Відповідні рішення ухвалюються в кожному конкретному випадку з урахуванням його специфіки. Але, що дійсно важливо відзначити, делегування або призначення повноважень і відповідальності щодо супроводження має бути зроблене відносно тільки одної організації або особи (менеджера відповідної команди підтримки). Усе так чи інакше залежить від організаційної структури організації/компанії, що експлуатує програмне забезпечення.

5 Аутсоурсинг.

Запозичений термін “аутсоурсинг” (Outsourcing) уже прижився не тільки в середовищі IT-менеджерів, він став частиною сучасного бізнесу й управлінських практик. Суть його полягає в передачі робіт, у першу чергу допоміжних (непрофільних для організації), “на сторону”. Великі корпорації передають в управління іншим організаціям цілі портфелі програмних систем, а іноді і цілком усю IT-інфраструктуру. У той же час значно частіше супровід передається іншим організаціям тільки для “другорядних” програмних систем (або, як мінімум, не критичних для виконання бізнес-функцій), тому що власники таких систем не бажають втрачати контроль над асоційованими з цими системами даними й/або функціональністю. Відзначається, що деякі передають роботи з супроводження “в аутсоурсинг” тільки в тих випадках, якщо переконані в стратегічному контролі над супроводженням.

Часто можна спостерігати, коли для вирішення питань супроводження (при збереженні “стратегічного контролю”) компанії, для яких інформаційні технології не є профільними, але сприймаються в якості активу, формують спеціалізовані дочірні бізнес-структури, яким і передаються функції супроводження, а також і безпосередньо розроблення програмних систем і, більш того, підтримки й розвитку всієї IT-інфраструктури. Це робиться для того, щоб функціонуючи в якості самостійної бізнес-сутності,

підрозділи автоматизації, що колись були внутрішньокорпоративними, могли забезпечити більшу прозорість фінансових потоків, витрат, пов'язаних з інформаційним технологіями. Але ця тема вже стосується загальних питань управління й, безумовно, вимагає самостійного обговорення в контексті, знов-таки, конкретної організації або бізнес-структури. Однак неможна було не позначити важливість обговорюваного питання в даному контексті, адже саме діяльність з супроводження часто спонукає організації-споживачі IT до прийняття настільки серйозних організаційних і бізнес-рішень.

При цьому контроль складно виміряти. У свою чергу перед аутсоурсером (організацією, що бере на себе відповідальність щодо супроводження) стоїть серйозна проблема з визначення змісту відповідних робіт, у тому числі для опису змісту відповідного контракту. Відзначається, що близько 50 % сервісів, надаваних аутсоурсером, проводяться без відповідної детальної угоди, що однозначно інтерпретується (service level agreement, SLA). Компанії, що займаються аутсоурсингом, звичайно витрачають кілька місяців на оцінку програмного забезпечення перш, ніж беруть відповідний контракт. Ще одне питання, що вимагає спеціальної уваги, полягає в необхідності визначення процесу й процедур передачі програмного забезпечення на зовнішнє супроводження.

6.2.3 Оцінка вартості супроводження

Інженери повинні розуміти різницю в різних категоріях супроводження. Це більшою мірою необхідно для оцінки відповідних витрат. З погляду планування, як складової частини проектної й управлінської діяльності, оцінка вартості є важливим аспектом діяльності з супроводження програмного забезпечення.

1 Оцінка вартості.

Під час обговорення аналізу впливу говорилося про те, що такий аналіз допомагає в оцінці вартості робіт із супроводження. На ці витрати впливає безліч технічних і інших факторів. ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance) визначає, що “існує два найбільш популярні методи оцінки вартості супроводження – параметрична модель і використання

досвіду”. Найчастіше обидва ці підходи комбінуються для підвищення точності оцінки.

2 Параметричні моделі.

Існує ряд джерел, що докладно розглядають питання оцінки вартості супроводження й, зокрема, параметричні моделі. Для використання таких моделей використовуються дані попередніх проектів. Нарівні з історичними даними використовується метод функціональних точок (див. стандарт IEEE 14143.1-00).

3 Досвід.

Серед підходів, які дозволяють підвищити точність оцінок, отриманих при використанні параметричних моделей, – застосування досвіду (у формі експертної думки, наприклад, при використанні техніки оцінки “Delphi”), аналогій, а також структури декомпозиції робіт. Найкращі результати утворюються у випадку комбінації емпіричних методів з наявним досвідом. Одержувані дані використовуються як результат програми вимірювання аспектів супроводження.

6.2.4 Вимірювання в супроводженні програмного забезпечення

Форми й дані вимірів у процесі супроводження можуть поєднуватися в єдину корпоративну програму кількісних оцінок, проведених відносно програмного забезпечення. Багато організацій використовують популярний і практичний підхід для вимірювань, що базується на оцінці кількості проблем і статусу їх рішень (issue-driven measurement). Ідеї цього підходу систематизовані в проекті Practical Software and Systems Measurement (PSM). Існують загальні для всього життєвого циклу метрики й, відповідно, їхні категорії, зокрема обумовлені Інститутом Програмної Інженерії університету Карнегі-Меллон (Software Engineering Institute, Carnegie-Mellon University – SEI CMU): розмір, зусилля, розклад і якість. Застосування цих метрик є гарною відправною точкою для оцінки робіт з боку організації, відповідальної за супроводження.

Більш детальне обговорення питань вимірювань відносно продуктів і процесів презентовано в галузі знань “Процес

програмної інженерії” (Software Engineering Process). У свою чергу питання організації програми вимірювань належать до галузі знань “Управління програмною інженерією” (Software Engineering Management).

1 Спеціалізовані метрики.

Існують різні методи внутрішньої оцінки продуктивності (benchmarking) персоналу супроводження для порівняння роботи різних груп супроводження. Організація, що веде супроводження, повинна обумовити метрики, за якими будуть оцінюватися відповідні роботи. Стандарти IEEE 1219 (Standard for Software Maintenance) і ISO/IEC 9126-01 (Software Engineering – Product Quality – Part 1: Quality Model, 2001 р.) пропонують спеціалізовані метрики, орієнтовані саме на питання супроводження й відповідні програми.

Нижче подано типові метрики оцінки робіт із супроводження, що відповідають розповсюдженій класифікації експлуатаційних характеристик програмного забезпечення:

– аналізованість (Analyzability): оцінка (у першу чергу додаткових) зусиль або ресурсів, необхідних для діагностики вад або причин збоїв, а також для ідентифікації тих фрагментів програмної системи, які повинні бути модифіковані;

– змінюваність (Changeability): оцінка зусиль, необхідних для проведення заданих модифікацій;

– стабільність (Stability): оцінка випадків непередбачуваної поведінки системи, включаючи ситуації, виявлені в процесі тестування;

– тестованість (Testability): оцінка зусиль персоналу супроводження й користувачів з тестування модифікованого програмного забезпечення.

6.3 Процес супроводження

Питання організації процесу супроводження прямо пов'язані з відповідними стандартами і de facto практиками реалізації такого процесу. Тема “Роботи з супроводження” (Maintenance Activities) розрізняє питання супроводження й розроблення й

показує взаємозв'язок з іншими аспектами діяльності програмної інженерії.

Типові й розповсюджені потреби в процесах програмної інженерії докладно описані й документовані в різних джерелах. Одна з найбільш детально опрацьованих і розповсюджених (на рівні стандарту de facto) процесних моделей, спочатку створених з орієнтацією на програмне забезпечення – CMMI (Capability Maturity Model Integration – інтегрована модель зрілості), розроблена в Інституті програмної інженерії університету Карнегі-Меллон (SEI CMU). CMMI, зокрема, приділяє спеціальну увагу процесам супроводження.

6.3.1 Процеси супроводження

Процеси супроводження описують необхідні роботи й детальні входи/виходи цих робіт. Ці процеси розглядаються в стандартах IEEE 1219 (Standard for Software Maintenance) і ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance).

Процес супроводження починається за стандартом IEEE 1219 з моменту передачі програмної системи в експлуатацію (post-delivery stage) і стосується таких питань, як планування діяльності з супроводження (див. рисунок 6.2).

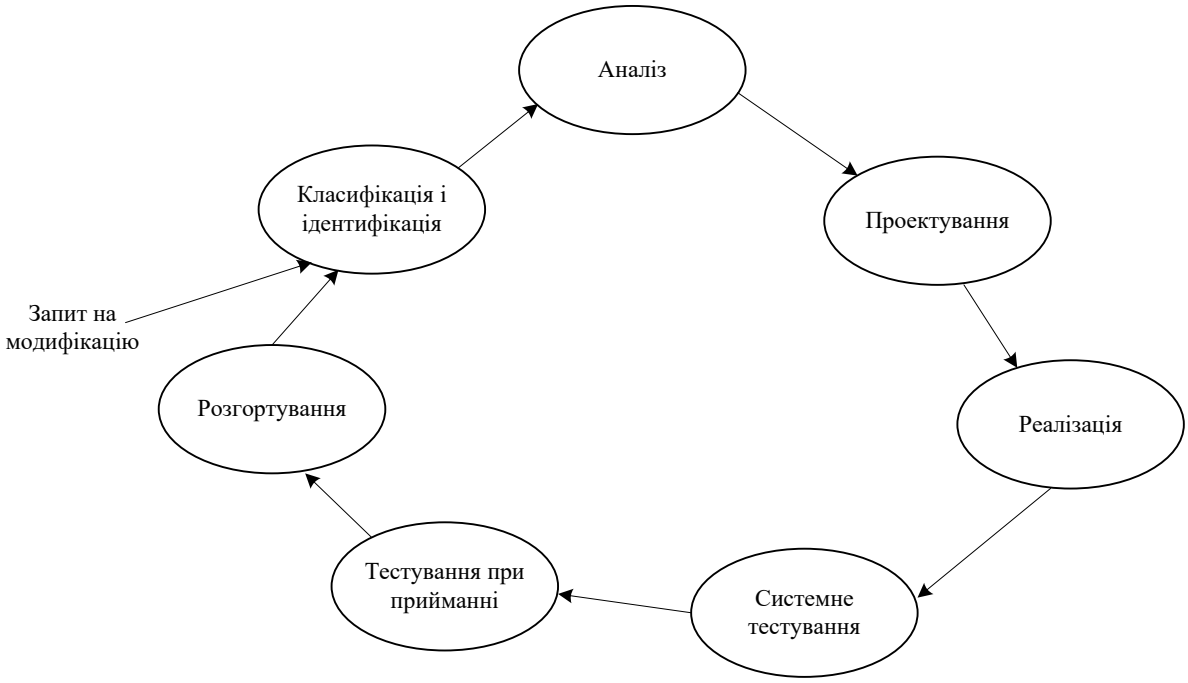

Рисунок 6.2 – Роботи в процесі супроводження

за стандартом IEEE 1219

Стандарт ISO/IEC 14764 уточнює положення, пов'язані з процесом супроводження, стандарту життєвого циклу 12207. Роботи з супроводження, описані в цьому стандарті, аналогічні роботам в IEEE 1219, за винятком того, що згруповані трохи інакше (див. рисунок 6.3).

Роботи з супроводження в стандарті 14764 розбиті на завдання:

– реалізація процесу;

– аналіз проблем і необхідних модифікацій;

– проведення модифікацій (реалізація змін);

– оцінка й прийняття проведених робіт при супроводженні;

– міграція (на модифіковану або нову версію програмного забезпечення);

– виведення з експлуатації (припинення експлуатації програмного забезпечення).

У представлених у SWEBOK джерелах можна знайти опис історії еволюції відповідних процесних моделей, що згадуються у стандартах ISO/IEC і IEEE. Крім того, існує й загальна (узагальнена) модель процесів супроводження. Agile-методології, що активно розвиваються в останні роки, пропонують “полегшені” (light або lightweight) процеси, у тому числі і для організації діяльності з супроводження, наприклад, Extreme maintenance (англ. “екстремальна підтримка”).

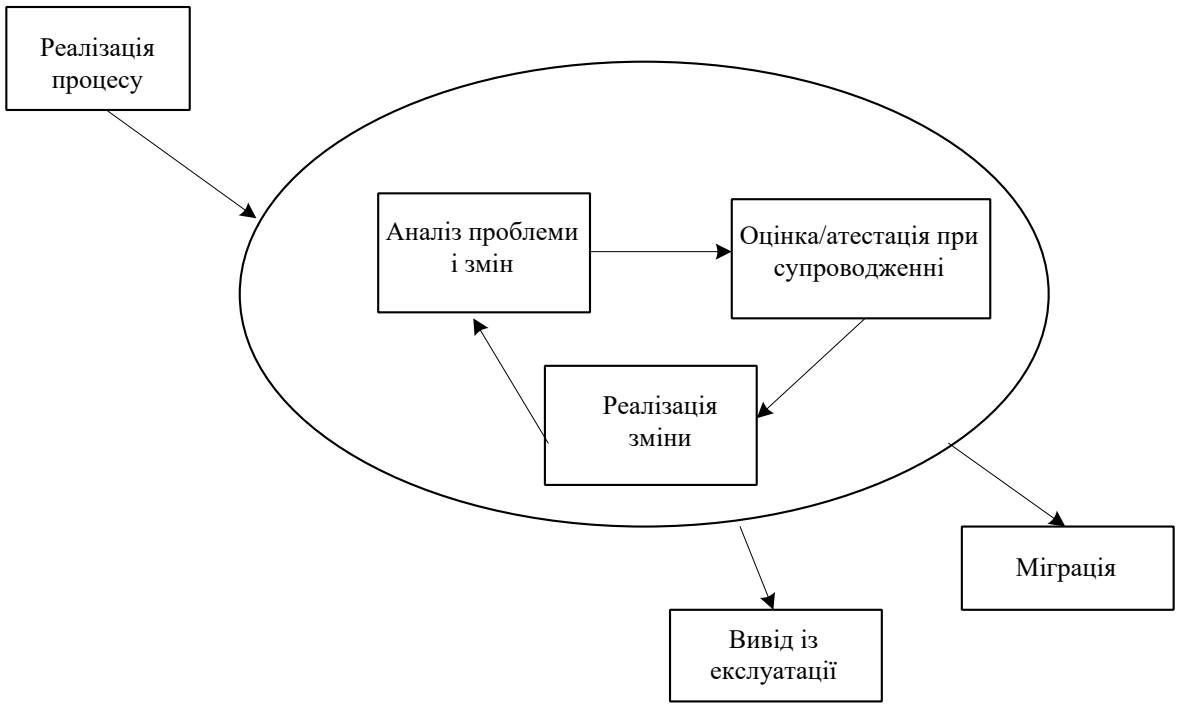

Рисунок 6.3 – Процес супроводження за стандартом ISO/IEC 14764

6.3.2 Роботи з супроводження

Як вже зазначено, багато робіт з супроводження схожі на аспекти діяльності з розроблення. В обох випадках необхідно проводити аналіз, проектування, кодування, тестування й документування. Фахівці з супроводження повинні відслідковувати вимоги так само, як і інженери-розробники, і обновляти документацію з розробленням й/або випуском оновлених або нових релізів продукту. Стандарт ISO/IEC 14764 рекомендує, щоб персонал або організації, відповідальні за супроводження, у випадку використання елементів процесів розроблення у своїй діяльності адаптували ці процеси цілком у відповідності зі своїми потребами. У той же час діяльність з супроводження має й певні унікальні риси, що призводить до необхідності використання спеціалізованих процесів.

1 Унікальні роботи.

Існує ряд процесів, робіт і практик, унікальних для діяльності з супроводження. Наведемо приклади такого роду унікальних характеристик:

– передача: контрольована й координована діяльність з передачі програмного забезпечення від розробників групі, службі або організації, відповідальної за подальшу підтримку;

– прийняття/відхилення запитів на модифікацію: запити на зміни можуть як ухвалюватися й передаватися в роботу, так і відхилятися з різних обґрунтованих причин: обсяг й/або складність необхідних змін, а також необхідні для цього зусилля. Відповідні рішення можуть також ухвалюватися на основі пріоритетності, оцінки обґрунтованості, відсутності ресурсів (у тому числі відсутності можливості залучення розробників до розв’язання задач з модифікації, при реальній наявності такої потреби), затвердженої запланованості до реалізації в наступному релізі й т. п.;

– засоби повідомлення персоналу супроводження й відстеження статусу запитів на модифікацію й звітів про помилки: функція підтримки кінцевих користувачів, що ініціює роботи з оцінки (припускаючи в тому числі оцінку необхідності), аналізу пріоритетності й вартості модифікацій, пов'язаних із запитом або повідомленою проблемою;

– аналіз впливу: аналіз можливих наслідків змін, внесених в існуючу систему;

– підтримка програмного забезпечення: роботи з консультування користувачів, проведені у відповідь на їхні інформаційні запити, наприклад, що стосуються відповідних бізнес-правил, перевірки змісту даних і специфічних питань користувачів і їхніх повідомлень про проблеми (помилки, збої, непередбачувана поведінка, нерозуміння аспектів роботи з системою й т. п.);

– контракти й зобов'язання: до них належать класична угода про рівень надаваного сервісу, а також інші договірні аспекти, на підставі яких група/служба/організація з супроводження виконує відповідні роботи.

На практиці складно провести грань між поділюваними в SWEBOK функціями Help Desk і Software Support (англ. допомога організації і програмна підтримка) – це функції, звичайно, сполучені з процесної точки зору.

2 Додаткові роботи, що підтримують процес супроводження.

Зміст цих робіт – це роботи персоналу супроводження, що не включають явної взаємодії з користувачами, але необхідні для здійснення відповідної діяльності. До таких робіт належать: планування супроводження, конфігураційне управління, перевірка й атестація, оцінка якості програмного забезпечення, різні аспекти огляду, аналізу й оцінки, аудит і навчання користувачів.

Також до таких спеціальних (внутрішніх) робіт належить навчання персоналу супроводження.

У силу особливої значущості ряду згаданих робіт їм присвячені наступні підтеми з супроводження програмного забезпечення.

3 Роботи з планування супроводження.

Планування є більш ніж необхідним для адекватного проведення робіт із супроводження й повинно стосуватися пов'язаних із цим питань із декількох точок зору:

– бізнес-планування (організаційний рівень);

– планування безпосередніх робіт із супроводження (рівень передачі програмного забезпечення);

– планування релізів/версій (рівень програмного забезпечення);

– планування обробки конкретних запитів на зміну (рівень запиту).

На рівні індивідуального запиту роботи з планування проводяться разом із проведенням аналізу впливу. У свою чергу планування релізів/версій вимагає від персоналу супроводження виконання завдань:

– одержання й збору інформації про дати розміщення індивідуальних запитів і звітів;

– досягнення угоди з користувачами про зміст (функціональність, поведінка й т. п.) наступних релізів/версій програмного забезпечення;

– ідентифікації потенційних конфліктів і можливих альтернатив реалізації необхідних запитів;

– оцінки ризиків для функціонування поточного релізу й розроблення плану “відкату” на немодифікований (поточний, до внесення змін, що стосуються розглянутого запиту) варіант системи у випадку виникнення проблем, пов'язаних з модифікацією;

– інформування всіх зацікавлених осіб.

Незважаючи на те, що розроблення програмних системи звичайно займає від кількох місяців (що більш типово) до декількох років, супроводження (як діяльність з підтримки використання) і активна експлуатація систем займає кілька років, а то й більше (5-10-...). Проведення оцінки ресурсів – невід'ємна частина планування. Ресурси, необхідні для супроводження, повинні бути оцінені й закладені в бюджет ще при розробленні системи. Планування робіт із супроводження має починатися одночасно з ухваленням рішення про створення системи й узгоджуватися з цілями забезпечення якості (відмічається в IEEE 1061-98 Standard for a Software Quality Metrics Methodology).

Спочатку необхідно визначити концепцію супроводження. Такий документ, наприклад за стандартом ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance), повинен стосуватися таких питань:

– зміст діяльності з супроводження;

– адаптація процесу супроводження;

– ідентифікація організації, яка буде займатися супроводженням;

– оцінка вартості супроводження.

Після розроблення концепції діяльності з супроводження повинен бути сформований відповідний план супроводження. Цей план повинен підготовлятися одночасно з розробленням програмної системи. План повинен визначати, як користувачі будуть розміщати свої запити на модифікацію (зміни) або повідомляти про помилки, збої й проблеми. Питанням планування приділяють спеціальну увагу вже згадувані стандарти IEEE 1219 (Standard for Software Maintenance) і ISO/IEC 14764 (Standard for Software Engineering – Software Maintenance). Стандарт ISO/IEC 14764 надає спеціальні рекомендації з організації плану супроводження.

Нарешті, на рівні бізнес-питань структура, відповідальна за супроводження, повинна проводити загальну діяльність з бізнес-планування, бюджетування, фінансового менеджменту й управління людськими ресурсами і т. ін.

4 Конфігураційне управління.

Стандарт IEEE 1219, присвячений організації супроводження програмного забезпечення, визначає конфігураційне управління як критично важливий елемент процесу супроводження. Процедури конфігураційного управління повинні забезпечувати перевірку, атестацію й аудит на всіх кроках, необхідних для ідентифікації, авторизації, реалізації й випуску програмного продукту.

Більш того, недостатньо просто відслідковувати запити на зміни й повідомлення про проблеми. Повинні бути контрольовані й сам програмний продукт і будь-які зміни (не тільки в коді, але в документації, специфікаціях і т. п., тобто будь-яких активах продукту й проекту). Такий контроль установлюється реалізацією й суворим слідуванням затвердженим процесам конфігураційного управління (software configuration management, SCM). Галузь “Конфігураційне управління” докладно описує й обговорює процеси, відповідно до яких розміщаються, оцінюються й затверджуються запити на зміни. У ряді окремих аспектів і характеристик конфігураційне управління при супроводженні й розробленні трохи відрізняється, що має контролюватися вже на операційному рівні. Реалізація SCM-процесу забезпечується розробленням і проходженням плану конфігураційного управління й відповідними процедурами. Організація, підрозділ або група супроводження в особі представників бере участь у роботі часто формованого органу управління конфігурацією (Configuration Control Board), відповідального за розгляд і прийняття в роботу запитів на зміни. Основною метою такої участі є визначення змісту наступних релізів/версій.

5 Якість програмного забезпечення.

Недостатньо лише сподіватися, що в процесі й результаті супроводження якість програмного забезпечення буде

підвищуватися. Для підтримки процесу супроводження повинні плануватися й реалізовуватися відповідні процедури й процеси, спрямовані на підвищення якості. Роботи й техніки з забезпечення якості, перевірки й атестації, огляду, аналізу й оцінки, а також аудиту повинні відбиратися в контексті взаємодії й узгодження з усіма іншими процесами, спрямованими на досягнення бажаного рівня якості. Ґрунтуючись на стандарті ISO/ IEC 14764 (Standard for Software Engineering – Software Maintenance), рекомендується адаптувати відповідні процеси, техніки й активи, що належать до розроблення програмного забезпечення. До них, наприклад, належать документація з тестування й результати тестів. Додаткові подробиці можна знайти у відповідній галузі “Якість програмного забезпечення”.

6.4 Техніки супроводження

Даний підрозділ вводить деякі загальноприйняті техніки, використовувані в процесі супроводження програмних систем.

6.4.1 Розуміння програмних систем

Для реалізації змін програмісти витрачають значну частину часу на читання й формування розуміння програмного продукту. Засоби роботи з кодом є ключовим інструментом для вирішення цього завдання. Чітка, однозначна й лаконічна документація забезпечує адекватне розуміння програмних систем.

6.4.2 Реінжиніринг

Реінжиніринг визначається як детальна оцінка і перебудова програмного забезпечення для формування розуміння, відтворення (на рівні моделі й у ряді випадків вимог) і подальшої реалізації його функцій у новій формі (наприклад, з використанням нових технологій і платформ, при збереженні існуючої, розширенням функціональності і полегшенням можливостей додавання нової функціональності). Відзначається, що в індустрії існують різні позиції відносно реінжинірингу: одні вважають, що реінжиніринг є найбільш радикальною й витратною формою змін програмних систем, інші – що такий

підхід може застосовуватися й для не настільки кардинальних змін (наприклад, як зміна платформи або архітектури). Реінжиніринг, звичайно, проводиться не стільки для поліпшення можливостей супроводження (maintainability – ремонтопридатність), скільки для заміни застарілого програмного забезпечення. У принципі реінжиніринг можна розглядати як самостійний проект, що включає в себе формування концепції, застосування відповідних інструментів і технік, аналіз і прикладення досвіду проведення реінжинірингу, а також оцінку ризиків і переваг, пов'язаних з такими роботами.

Слід зазначити, що реалізація продукту в новій якості (формі) при збереженні основної функціональності оригінального продукту є невід'ємною й визначальною частиною реінжинірингу, на відміну від зворотного інжинірингу, що буде розглянутий далі і є важливою складовою частиною реінжинірингу.

6.4.3 Зворотний інжиніринг

“Зворотний” інжиніринг (що часто плутається з реінжинірингом), або процес аналізу програмного забезпечення з метою ідентифікації програмних компонентів і зв'язків між ними, а також формування уявлення про програмне забезпечення, з подальшою перебудовою в новій формі (уже, у процесі реінжинірингу) є пасивним, таким що припускає відсутність діяльності зі зміни або створення нового програмного забезпечення. Звичайно в результаті зусиль зі зворотного інжинірингу створюються моделі викликів і потоків управління на основі вихідного коду системи. Один з типів зворотного інжинірингу – створення нової документації на існуючу систему. Інший з розповсюджених типів – відновлення дизайну системи.

До питань зворотного інжинірингу, як і до питань реінжинірингу, також належать роботи з рефакторингу. Рефакторинг – трансформація програмного забезпечення, у процесі якої програмна система реорганізується (не переписуючись) з метою поліпшення структури без зміни поведінки. Збереження “форми” (платформи, архітектурних і технологічних рішень) існуючої програмної системи дозволяє

розглядати рефакторинг як один з варіантів зворотного інжинірингу.

Для обох тем – 6.4.2 і 6.4.3 - можливе застосування слова реконструкція, залежно від контексту, що означає як повну перебудову або відтворення чого-небудь, ідентичного по певних характеристиках оригінальному зразку, так і відновлення якої-небудь сутності за збереженими і/або доступними зовнішніми ознаками, де відновлення може мати на увазі знов-таки створення нового зразка або уявлення про оригінальну сутність.

Список літератури

1 Степанов, А. Начала программирования [Текст]/ А. Степанов, П. Мак-Джонс. — М.: «Вильямс», 2011. — 272 с.

2 Бейзер, Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем [Текст] / Б. Бейзер. — СПб.: Питер, 2004. — 320 с.

3 Страуструп, Б. Программирование: принципы и практика использования C++ [Текст]/ Б. Страуструп. — М.: «Вильямс», 2011. — 1248 с.

4 Дейкстра, Э. Дисциплина программирования [Текст] / Э. Дейкстра. — М.: Мир, 1978. — 275 с.

5 Гринфилд, Д. Фабрики разработки программ (Software Factories): потоковая сборка типовых приложений, моделирование, структуры и инструменты [Текст] / Д. Гринфилд, К. Шорт, С. Кук, С. Кент, Д. Крупи. — М.: «Диалектика», 2006. —592 с.

6 Кнут, Д. Искусство программирования. Т. 1. Основные алгоритмы [Текст] / Д. Кнут. — 3-е изд. — М.: «Вильямс», 2006. — 720 с.

7 Грэхем, И. Объектно-ориентированные методы. Принципы и практика [Текст] / И. Грэхем; пер. с англ. — 3-е изд. — М.: Вильямс, 2004. — 880 с.

8 Соммервилл, И. Инженерия программного обеспечения [Текст]/ И. Соммервилл; пер. с англ. — 6-е изд. — М.: Вильямс, 2002. — 624 с.

9 Калбертсон, Р. Быстрое тестирование [Текст]/ Р. Калбертсон, К. Браун, Г. Кобб. — М.: «Вильямс», 2002. — 374 с.

10 Канер, К. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений [Текст] / К. Канер, Д. Фолк, Е. К. Нгуен. — К.: ДиаСофт, 2001. — 544 с.

11 Криспин, Л. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд [Текст] / Л. Криспин, Д. Грегори. — М.: «Вильямс», 2010. — 464 с.

12 Себеста, Р.В. Основные концепции языков программирования [Текст] / Р.В. Себеста; пер. с англ. — 5-е изд. — М.: Вильямс, 2001. — 672 с.

13 Синицын, С.В. Верификация программного обеспечения [Текст] / С.В. Синицын, Н.Ю. Налютин. — М.: БИНОМ, 2008. — 368 с.

14 Кормен, Т.Х. Алгоритмы: построение и анализ [Текст]/ Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К. Штайн. — 2-е изд. — М.: «Вильямс», 2006. —1296 с.