МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ПОЛІТЕХНІКА»
ІНСТИТУТ ШТУЧНОГО ІНТЕЛЕКТУ ТА РОБОТОТЕХНІКИ
КАФЕДРА ПРОГРАМНИХ ТА КОМП'ЮТЕРНО-ІНТЕГРОВАНИХ ТЕХНОЛОГІЙ

Методичні вказівки з дисципліні

МЕТОДИ ТЕСТУВАННЯ ПРОГРАММНОГО ЗАБЕЗПЕЧЕННЯ

(Теоретична частина)

Для студентів інституту штучного інтелекту та робототехніки

Інший (магістерський) рівень вищої освіти

Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології

Освітньо-професійна програма: Комп'ютерні технології автоматизації;

Кількість рік/кредитів ЄКТС за навчальним планом: 135/4,5

Схвалено на засіданні кафедри ПКІТ
протокол №7 від 26.01.2022р.

Одеса 2022

Методичні вказівки з дисципліни «Методи тестування програмного забезпечення» (Теоретична частина) для студ. спец. 151 – «Автоматизація та комп'ютерно-інтегровані технології» ден. та заоч. форм навч./уклад.: В.О. Давидов - Одеса: НУ "Одеська політехніка", 2022. - 482 с.

Укладачі: **В.О. Давидів**, Канд. техн. наук

ЗМІСТ

Лекція №1-2 Місце верифікації серед процесів розробки програмного забезпечення 4

Лекція №2 Тестування програмного коду, Повторюваність тестування. 43

ЛЕКЦІЯ №3 Документація, що супроводжує процес верифікації та тестування (тест-плани)..... 134

Лекція №4 Формальні інспекції 186

Лекція №5 Модульне тестування..... 211

Лекція 6 Інтеграційне тестування 255

Лекція 7 Системне тестування 292

Лекція 8: Особливості індустріального тестування.....320

Лекція 9: Регресійне тестування: цілі та завдання, умови застосування, класифікація тестів і методів відбору..... 347

Лекція 10 Регресійне тестування: методики, не пов'язані з відбором тестів і методики породження тестів 378

Лекція 11: Підтримка процесу тестування при промисловій розробці програмного забезпечення..... 460

Лекція №1-2 Місце верифікації серед процесів розробки програмного забезпечення

Лекція присвячена розгляду різних видів життєвого циклу розробки програмного забезпечення та сучасних технологій розробки. Показано місце процесу верифікації в життєвому циклі, визначено його мету і завдання. Розглядаються різні типи процесів верифікації, визначається різниця між тестуванням, перевірки та затвердження.

Мета даної лекції: дати уявлення про процес верифікації як про чітко певному виді діяльності в рамках життєвого циклу розробки програмної системи, визначити сучасні підходи до верифікації

Поняття верифікації

Верифікація - це процес визначення, чи виконують програмні засоби та їх компоненти вимоги, накладені на них в послідовних етапах життєвого циклу розробляється програмної системи.

Основна мета верифікації полягає в підтвердженні того, що програмне забезпечення відповідає вимогам. Додатковою метою є виявлення та реєстрація дефектів і помилок, які внесені під час розробки або модифікації програми.

Верифікація є невід'ємною частиною робіт при колективної розробки програмних систем. При цьому в задачі верифікації включається контроль результатів одних розробників при передачі їх у якості вихідних даних іншим розробникам.

Заздалегідь розмежуємо поняття верифікації та налагодження. Обидва ці процеси спрямовані на зменшення помилок в кінцевому програмному продукті, однак налагодження - процес, спрямований на локалізацію та усунення помилок в системі, а верифікація - процес, спрямований на демонстрацію наявності помилок та умов їх виникнення. Життєвий цикл розробки програмного **забезпечення.**

Колективна розробка, на відміну від індивідуальної, вимагає чіткого планування робіт та їх розподілу під час створення програмної системи. Один із способів організації робіт полягає в розбитті процесу розробки на окремі послідовні стадії, після повного проходження яких виходить кінцевий продукт або його частину. Такі стадії називають життєвим циклом розробки програмної системи. Як правило, життєвий цикл починається з формування загального уявлення про розроблюваної системи та його формалізації у вигляді вимог верхнього рівня. Завершується життєвий цикл розробки введенням системи в експлуатацію. Проте, потрібно розуміти, що розробка - тільки один із процесів, пов'язаних з програмною системою, яка також має свій життєвий цикл. На відміну від життєвого циклу розробки системи, життєвий цикл самої системи закінчується виведенням її з експлуатації та припиненням її використання.

Життєвий цикл програмного забезпечення - сукупність ітераційних процедур, пов'язаних з послідовною зміною стану програмного забезпечення від формування вихідних вимог до нього до закінчення його експлуатації кінцевим користувачем.

Моделі життєвого циклу

Будь-який етап життєвого циклу має чітко визначені критерії початку і закінчення. Склад етапів життєвого циклу, а також критерії, в кінцевому підсумку визначають послідовність етапів життєвого циклу, визначається колективом розробників та / або замовником. В даний час існує декілька основних моделей життєвого циклу, які можуть бути адаптовані під реальну розробку.

Каскадний життєвий цикл

Каскадний життєвий цикл (іноді званий водоспадні) заснований на поступовому збільшенні ступеня деталізації опису всієї системи, що розробляється. Кожне підвищення ступеня деталізації визначає перехід до наступного станом розробки (Рис. 1.1).

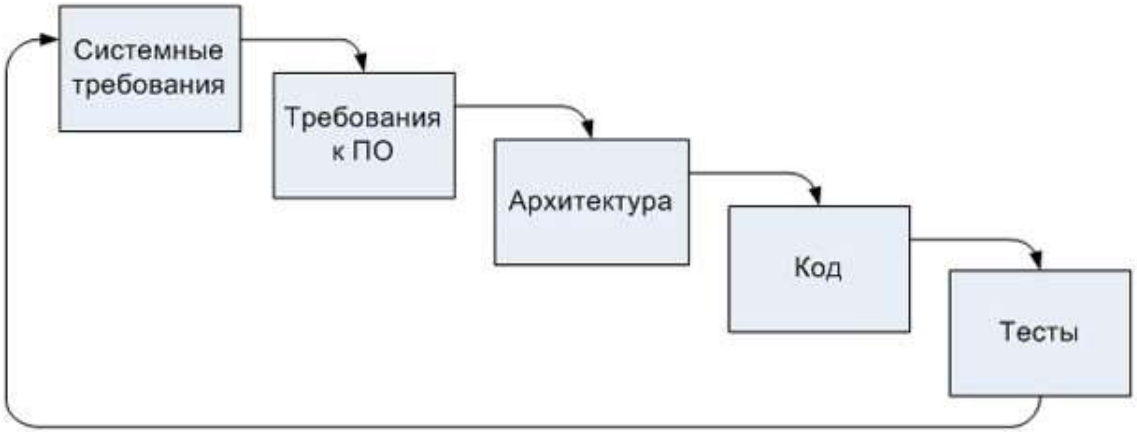


Рис. 1.1. Каскадна модель життєвого циклу

На першому етапі складається концептуальна структура системи, описуються загальні принципи її побудови, правила взаємодії з навколишнім світом,

- визначаються системні вимоги. **На другому етапі** по системним вимогам складаються вимоги до програмного забезпечення - тут основна увага приділяється функціональності програмної компоненти, програмним інтерфейсам. Природно, всі програмні комплекси виконуються на який-небудь апаратній платформі. Якщо в ході проекту потрібно також розробка апаратної компоненти, паралельно з вимогами до програмного забезпечення йде підготовка вимог до апаратного забезпечення. **На третьому етапі** на основі вимог до програмного забезпечення складається детальна специфікація архітектури системи - описуються розбиття системи по

конкретних модулям, інтерфейси між ними, заголовки окремих функцій і т.п. **На четвертому етапі** пишеться програмний код, відповідний детальної специфікації, **на п'ятому етапі** виконується тестування - перевірка відповідності програмного коду вимогам, визначеним на попередніх етапах. **Особливість каскадного життєвого циклу полягає в тому, що перехід до наступного етапу відбувається тільки тоді, коли повністю завершені всі роботи попереднього етапу.** Тобто спочатку повністю готуються всі вимоги до системи, потім по ним повністю готуються всі вимоги до програмного забезпечення, повністю розробляється архітектура системи і так далі до тестування. Природно, що в разі достатньо великих систем такий підхід себе не виправдовує. Робота на кожному етапі займає значний час, а внесення змін у первинні документи або неможливо, або викликає лавиноподібні зміни на всіх інших етапах.

V-подібний життєвий цикл

В якості своєрідної "роботи над помилками" класичної каскадної моделі стала застосовуватися модель життєвого циклу, що містить процеси двох видів - основні процеси розробки, аналогічні процесам каскадної моделі, і процеси верифікації, що представляють собою ланцюг зворотного зв'язку по відношенню до основних процесів (Рис. 1.2) .

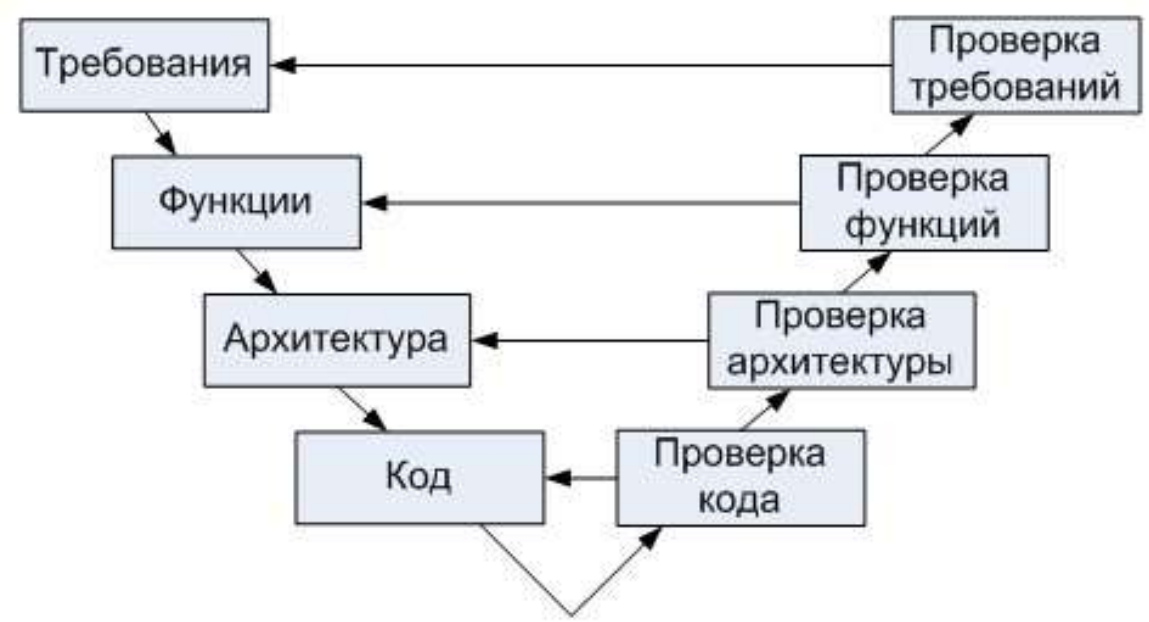


Рис. 1.2. V-подібний життєвий цикл

Таким чином, в кінці кожного етапу життєвого циклу розробки, а часто і в процесі виконання етапу, здійснюється перевірка взаємної коректності вимог різних рівнів. Дана модель дозволяє більш оперативно перевіряти коректність розробки, проте, як і в каскадної моделі, передбачається, що на кожному етапі розробляються документи, що описують поведінку всієї системи в цілому.

Спіральний життєвий цикл

Обидва розглянутих типу життєвих циклів припускають, що заздалегідь відомі всі вимоги користувачів або, користувачі системи настільки кваліфіковані, що можуть висловлювати свої вимоги до майбутньої системи, не бачити її перед очима.

Природно, така картина досить утопічна, тому поступово з'явилося рішення, що виправляє основний недолік V-образного життєвого циклу - припущення про те, що на кожному етапі розробляється чергове повний опис системи. Цим рішенням стала спіральна модель життєвого циклу (Рис. 1.3).

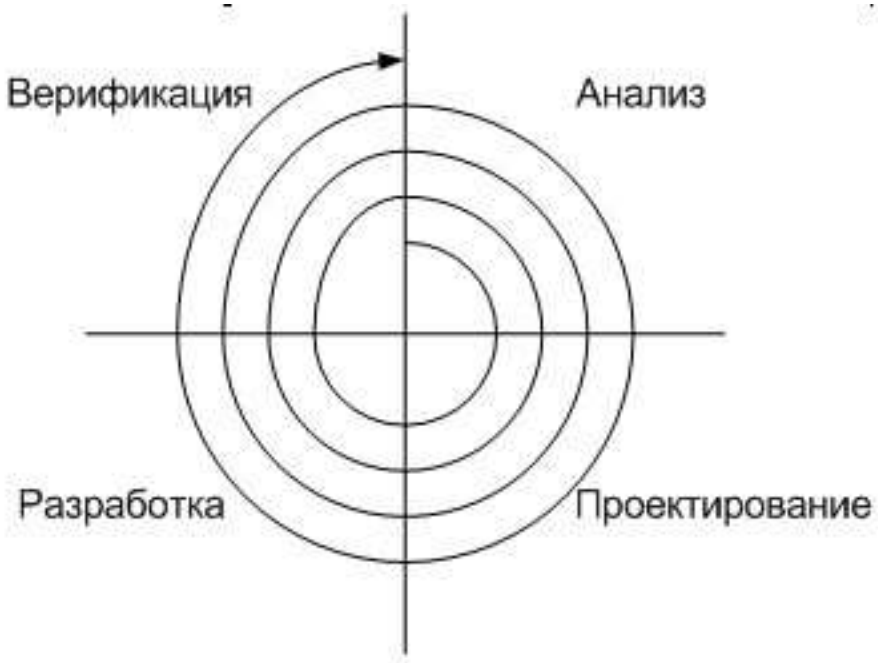


Рис. 1.3. Спіральний життєвий цикл

В спіральної моделі розробка системи відбувається повторюваними етапами - витками спіралі. Кожен виток спіралі - один каскадний або V-подібний життєвий цикл. В кінці кожного витка виходить закінчена версія системи, що реалізує певний набір функцій. Потім вона пред'являється користувачеві, на наступний виток переноситься вся документація, розроблена на попередньому витку, і процес повторюється.

Таким чином, система розробляється поступово, проходячи постійні узгодження з замовником. На кожному витку спіралі функціональність системи розширюється, поступово дорости до повної.

Екстремальне програмування

Реалії останніх років показали, що для систем, вимоги до яких змінюються досить часто, необхідно ще більше зменшити тривалість витка спірального життєвого циклу. У зв'язку з цим зараз стали дуже популярними швидкі життєві цикли розробки, наприклад, життєвий цикл в методології eXtreme Programming (XP).

Основна ідея життєвого циклу екстремального підходу - максимальне укорочення тривалості одного етапу життєвого циклу і тісна взаємодія із замовником. По суті, на кожному етапі відбувається реалізація і тестування однієї функції системи, після завершення яких система відразу передається замовнику на перевірку або експлуатацію.

Основна проблема даного підходу - інтерфейси між модулями, що реалізують цю функцію. Якщо в усіх попередніх типах життєвого циклу інтерфейси досить чітко визначаються на самому початку розробки, оскільки заздалегідь відомі всі модулі, то при екстремальному підході інтерфейси проектуються "на льоту", разом з розробляються модулями.

Порівняння різних типів життєвого циклу і допоміжні процеси

Особливості розглянутих вище типів життєвого циклу зведені в таблицю 1.1 . З неї можна бачити, що різні типи життєвих циклів застосовуються залежно від планованої частоти внесення змін до системи, термінів розробки та її складності. Життєві цикли з більш короткими фазами більше підходять для розробки систем, вимоги до яких ще не устоялися і виробляються у взаємодії із замовником системи під час її розробки.

Таблиця 1.1. Порівняння різних типів життєвого циклу

Тип життєвого циклу	Довжина циклу	Верифікація та внесення змін	Інтеграція окремих компонент системи
Каскадний	Всі етапи розробки системи. Довгий	В кінці розробки всієї системи. Зміни вносяться рідко	Чітко визначені до початку кодування інтерфейси
V-подібний	Всі етапи розробки системи. Довгий	В кінці повної розробки кожного з етапів системи. Зміни вносяться з середньою частотою	Рідко змінювані інтерфейси

Спіральний	Розробка однією версією системи. Середній	В кінці розробки кожного з етапів версії системи. Зміни вносяться з середньою частотою	Періодично змінювані інтерфейси, рідко змінювані в межах версії
XP	Розробка однієї історії. Короткий	В кінці розробки кожної історії. Зміни вносяться дуже часто	Часто змінювані інтерфейси

У наведеному вище описі різних моделей життєвого циклу по суті порушувалося тільки один процес - процес розробки системи.

Насправді в будь-якої моделі життєвого циклу можна побачити чотири види процесів:

- Основний процес розробки
- Процес верифікації
- Процес управління розробкою
- Допоміжні (підтримуючі) процеси

Процес верифікації - процес, спрямований на перевірку коректності розроблювальної системи та визначення ступеня її відповідності вимогам замовника. Докладного розгляду цього процесу і присвячений даний навчальний курс.

Процес управління розробкою - окрема дисципліна. На управління дуже сильно впливає тип життєвого циклу основного процесу розробки. По суті, чим коротше один етап життєвого циклу, тим активніше управління і тим більше завдань стоїть перед менеджером проекту. При класичних схемах досить просто побудувати ієрархічну піраміду підлеглості, в якій кожен нижчий менеджер відповідає за розробку певної частини системи. В XP-підході немає жорсткого поділу системи на частини, і менеджер повинен охоплювати всі історії. При цьому процес управління активний протягом усього життєвого циклу основного процесу розробки.

Допоміжні (підтримуючі) процеси забезпечують своєчасне створення всього, що може знадобитися розробнику або кінцевому користувачеві. Сюди входить підготовка користувача документації, підготовка приймально-здавальних тестів, управління

конфігураціями та змінами, взаємодія із замовником і т.д. Взагалі кажучи, допоміжні процеси можуть існувати протягом всього життєвого циклу розробки, а можуть бути своєрідному стику ланками між процесом розробки та процесом експлуатації.

У реальній практиці зараз найбільш широко застосовується стандарт ISO 12207, у вітчизняних держструктурах використовуються стандарти серії ГОСТ 34.

Стандарти комплексу ГОСТ 34 на створення і розвиток АС - узагальнені, але сприймаються як дуже жорсткі по структурі ЖЦ та проектної документації.

Міжнародний стандарт ISO / IEC 12207 на організацію життєвого циклу продуктів програмного забезпечення (ПО) містить загальні рекомендації по організації життєвого циклу, не постулюючи при цьому його жорсткої структури.

Microsoft Solutions Framework

Microsoft Solutions Framework (MSF) - це методологія ведення проектів і розробки рішень, що базується на принципах роботи над продуктами самої фірми Microsoft і призначена для використання в організаціях, які потребують концептуальної схеми для побудови сучасних рішень.

Microsoft Solutions Framework є схемою для прийняття рішень з планування та реалізації нових технологій в організаціях. MSF включає навчання, інформацію, рекомендації та інструменти для ідентифікації та структуризації інформаційних потоків бізнес-процесів і всієї інформаційної інфраструктури нових технологій.

Microsoft Solutions Framework представляє собою добре збалансований набір методик організації процесу розробки, який може бути адаптований під потреби практично будь-якого колективу розробників. MSF містить не тільки рекомендації загального характеру, а й пропонує адаптується модель колективу розробників, визначальну взаємини всередині колективу, гнучку модель проектного планування, заснованого на управлінні проектними групами, а також набір методик для оцінки ризиків.

MSF складається з двох моделей:

- модель проектної групи;
- модель процесів,

і трьох дисциплін:

- управління проектами;
- управління ризиками;
- управління підготовкою.

В MSF немає ролі "менеджер проекту" та ієрархії керівництва, управління розробкою розподілено між керівниками окремих проектних груп всередині колективу, які виконують такі завдання:

- Управління програмою
- Розробка
- Тестування
- Управління випуском
- Задоволення споживача
- Управління продуктом

Життєвий цикл процесів в MSF поєднує водоспадні і спіральну моделі розробки: проект реалізується поетапно, з наявністю відповідних контрольних точок, а сама послідовність етапів може повторюватися по спіралі (Рис. 1.4).

Рис. 1.4. Життєвий цикл в MSF

При такому підході від Водоспадної моделі береться простота планування, від класичної спіральної - легкість модифікацій. Завдяки проміжним контрольним точкам і зворотного спіралі верифікації полегшується взаємодія із замовником.

При управлінні проектом чітко ставиться мета, яку необхідно досягти в результаті, і враховуються обмеження, що накладаються на проект. Всі види обмежень можуть бути віднесені до одного з трьох видів: обмеження ресурсів, обмеження часу і обмеження можливостей. Ці три види обмежень і пріоритетність завдань щодо їх подолання утворюють трикутник пріоритетів в MSF (Рис. 1.5).

Рис. 1.5. Трикутник пріоритетів в MSF

Трикутник пріоритетів є основою для матриці компромісів - заздалегідь затверджених уявлень про те, які аспекти процесу розробки будуть чітко задані, а які будуть узгоджуватися або прийматися як є.

Microsoft випустила середовище розробки, повною мірою підтримує основні ідеї MSF - Microsoft Visual Studio 2010 Team Edition.

Rational Unified Process

Rational Unified Process - це методологія створення програмного забезпечення, оформлена у вигляді розміщується на Web бази знань, яка забезпечена пошуковою системою. Продукт Rational Unified Process (RUP) розроблений і підтримується Rational Software. Він регулярно оновлюється з метою врахування передового досвіду та покращується за рахунок перевірених на практиці результатів.

RUP забезпечує строгий підхід до розподілу завдань і відповідальності всередині організації-розробника. Його призначення полягає в тому, щоб гарантувати створення точно в строк і в рамках встановленого бюджету якісного ПО, що відповідає потребам кінцевих користувачів. RUP сприяє підвищенню продуктивності колективної розробки та надає найкраще з накопиченого досвіду по створенню ПО, за допомогою посібників, шаблонів і настанов з користування інструментальними засобами для всіх критично важливих робіт, протягом життєвого циклу створення та супроводження програмного забезпечення.

Забезпечуючи кожному члену групи доступ до тієї ж самої базі знань, незалежно від того, чи розробляє він вимоги, проектує, виконує тестування або управляє проектом - RUP гарантує, що всі члени групи використовують спільну мову моделювання і процес, мають узгоджене бачення того, як створювати ПО.

В якості мови моделювання в загальній базі знань використовується Unified Modeling Language (UML), що є міжнародним стандартом.

RUP - це конфігурується процес, оскільки цілком зрозуміло, що неможливо створити єдиного керівництва на всі випадки розробки ПЗ. RUP придатний як для маленьких груп розробників, так і для великих організацій, що займаються створенням ПЗ. В основі RUP лежить проста і зрозуміла архітектура процесу, яка забезпечує спільність для цілого сімейства процесів. Більш того, RUP може конфігуруватися для обліку різних ситуацій. До його складу входить Development Kit, який забезпечує підтримку процесу конфігурування під потреби конкретних організацій.

RUP описує, як ефективно застосовувати комерційно обгрунтовані і практично випробувані підходи до розробки ПО для колективів розробників, де кожен з членів отримує переваги від використання передового досвіду в:

- ітераційної розробки ПЗ;
- управлінні вимогами;
- використанні компонентної архітектури;
- візуальному моделюванні;
- тестуванні якості ПЗ;
- контролі за змінами в ПО.

RUP організовує роботу над проектом в термінах послідовності дій (workflows), продуктів діяльності, виконавців та інших статичних аспектів процесу, з одного боку, і в термінах циклів, фаз, ітерацій і тимчасових відміток завершення певних етапів у створенні ПЗ (milestones), в термінах динамічних аспектів процесу - з іншого.

eXtreme Programming

Екстремальне програмування - порівняно молода методологія розробки програмних систем, заснована на поступовому поліпшенні системи та розробки її дуже короткими ітераціями. За своєю суттю екстремальне програмування (XP) - це одна з так званих "гнучких" методологій розробки ПЗ, яка являє собою невеликий набір конкретних правил, що дозволяють максимально ефективно виконувати вимоги сучасної теорії управління програмними проектами.

XP орієнтована на:

- командну роботу з тісними зв'язками всередині команди і з замовником;
- розробку найбільш простих працюють рішень;
- гнучке адаптивне планування;
- оперативний зворотний зв'язок (шляхом модульного та функціонального тестування).

Основними принципами XP є розробка невеликими ітераціями на підставі порції вимог замовника (т.зв. користувальницьких історій), написання функціональних тестів до написання програмного коду, постійне спілкування і постійний рефакторинг коду.

Основними практиками XP є:

- Планування процесу
- Часті релізи
- Метафора системи
- Проста архітектура
- Тестування
- Рефакторинг
- Парне програмування
- Колективне володіння кодом
- Часта інтеграція
- 40-годинний робочий тиждень
- Стандарти кодування
- Тісна взаємодія з замовником

Порівняння технологій MSF, RUP і XP

Основні особливості MSF, RUP і XP зведені в таблицю 1.2 . По ній можна судити, що Rational Unified Process є добре збалансованим рішенням для середніх за розмірами колективів розробників, що працюють із застосуванням продуктів і технологій компанії Rational. Супровід розробки системи і самої системи регламентується методологією RUP, однак дана технологія досить сильно орієнтована на внутрішньофірмові інструментальні засоби.

Extreme Programming добре підходить для проектних груп малого розміру і для невеликих систем з часто змінними вимогами. Основна проблема XP - сопровождаемость. У разі плинності кадрів в колективі розробників значна частина проектної інформації може бути загублена через практично відсутньої документації.

Таблиця 1.2. Технології MSF, RUP і XP

Технологія	Оптимальна команда	Відповідність стандартам	Допустимі технології та інструменти	Зручність модифікації і супроводу

Rational Unified Process	10 - 40 чол.	стандарти Rational	UML і продукти Rational	Зручно (RUP)
Microsoft Solutions Framework	3 - 20 чол.	адаптируема	будь-які	Зручно (MSF + MOF)
XP	2 - 10 чол.	стандарти відсутні	будь-які	Складно (залежність від конкретних учасників колективу)

Рольовий склад колективу розробників, взаємодія між ролями в різних технологічних процесах

Коли проектна команда включає більше двох осіб неминуче постає питання про розподіл ролей, прав і відповідальності в команді. Конкретний набір ролей визначається багатьма факторами - кількістю учасників розробки та їх особистими уподобаннями, прийнятої методологією розробки, особливостями проекту та іншими факторами. Практично в будь-якому колективі розробників можна виділити перераховані нижче ролі. Деякі з них можуть зовсім відсутні, при цьому окремі люди можуть виконувати відразу кілька ролей, проте загальний склад змінюється мало.

Замовник (заявник). Ця роль належить представнику організації, яка замовила розроблювану систему. Зазвичай заявник обмежений у своїй взаємодії і спілкується тільки з менеджерами проекту і фахівцем з сертифікації або впровадження. Зазвичай замовник має право змінювати вимоги до продукту (тільки у взаємодії з менеджерами), читати проектну і сертифікаційну документацію, що стосується нетехнічних особливості розроблюваної системи.

Менеджер проекту. Ця роль забезпечує комунікаційний канал між замовником та проектною групою. Менеджер продукту управляє очікуваннями замовника, розробляє і підтримує бізнес-контекст проекту. Його робота не пов'язана прямо з продажем, він сфокусований на продукті, його завдання - визначити та забезпечити вимоги замовника. Менеджер проекту має право змінювати вимоги до продукту і фінальну документацію на продукт.

Менеджер програми. Ця роль управляє комунікаціями і взаєминами в проектній групі, є до певної міри координатором, розробляє функціональні специфікації і керує ними, веде графік проекту і звітує за станом проекту, ініціює прийняття критичних для ходу проекту рішень.
Менеджер програми має право змінювати функціональні специфікації верхнього рівня, план-графік проекту, розподіл ресурсів по задачам. Часто на практиці роль менеджера проекту та менеджера програми виконує одна людина.

Розробник. Розробник приймає технічні рішення, які можуть бути реалізовані та використані, створює продукт, що задовольняє специфікаціям і очікуванням замовника, консультує інші ролі в ході проекту. Він бере участь в оглядах, реалізує можливості продукту, бере участь у створенні функціональних специфікацій, відстежує і виправляє помилки за прийнятний час. В

контексті конкретного проекту роль розробника може мати на увазі, наприклад, інсталяцію програмного забезпечення, налаштування продукту або послуги. Розробник має доступ до всієї проектної документації, включаючи документацію з тестування, має право на зміну програмного коду системи в рамках своїх службових обов'язків.

Фахівець з тестування. Фахівець з тестування визначає стратегію тестування, тест- вимоги і тест-плани для кожної з фаз проекту, виконує тестування системи, збирає та аналізує звіти про проходження тестування. Тест-вимоги повинні покривати системні вимоги, функціональні специфікації, вимоги до надійності і здатності навантаження, призначені для користувача інтерфейси і власне програмний код. В реальності роль фахівця з тестування часто розбивається на дві - розробника тестів і тестувальника. Тестувальник виконує всі роботи з виконання тестів і збору інформації, розробник тестів - всю решту робіт.

Фахівець з контролю якості. Ця роль належить члену проектної групи, який здійснює взаємодію з розробником, менеджером програми і фахівцями з безпеки і сертифікації з метою відстеження цілісної картини якості продукту, його відповідності стандартам і специфікаціям, передбачених проектною документацією. Слід розрізняти фахівця з тестування та спеціаліста

з контролю якості. Останній не є членом технічного персоналу проекту, відповідальним за деталі і техніку роботи. Контроль якості на увазі в першу чергу контроль самих процесів розробки і перевірку їх відповідності визначеним в стандартах якості критеріям.

Спеціаліст з сертифікації. При розробці систем, до надійності яких пред'являються підвищені вимоги, перед введенням системи в експлуатацію потрібне підтвердження з боку уповноваженого органу (зазвичай державного) відповідності її експлуатаційних характеристик заданим критеріям. Така відповідність визначається в ході сертифікації системи. Спеціаліст по сертифікації може або бути представником сертифікуючих органів, включеним до складу колективу розробників, або навпаки - представляти інтереси розробників в сертифікує органі. Спеціаліст по сертифікації призводить документацію на програмну систему у відповідність вимогам сертифікує органу або бере участь в процесі створення документації з урахуванням цих вимог. Також фахівець з сертифікації відповідальний за все взаємодію між колективом розробників і сертифікуючий органом. Важливою особливістю ролі є незалежність фахівця від проектної групи на всіх етапах створення продукту. Взаємодія фахівця з членами проектної групи обмежується менеджерами по проекту і по програмі.

Спеціаліст з впровадження та супроводу. Бере участь в аналізі особливостей майданчика замовника, на якій планується проводити впровадження розроблюваної системи, виконує весь спектр робіт з встановлення та налаштування системи, проводить навчання користувачів.

Фахівець з безпеки. Даний фахівець відповідальний за весь спектр питань безпеки створюваного продукту. Його робота починається з участі в написанні вимог до продукту і закінчується фінальною стадією сертифікації продукту.

Інструктор. Ця роль відповідає за зниження витрат на подальший супровід продукту, забезпечення максимальної ефективності роботи користувача. Важливо, що мова йде про продуктивність користувача, а не системи. Для забезпечення оптимальної продуктивності інструктор збирає статистику по продуктивності користувачів і створює рішення для підвищення продуктивності, в тому числі з використанням різних аудіовізуальних засобів. Інструктор бере участь у всіх обговореннях користувальницького інтерфейсу і архітектури продукту.

Технічний письменник. Особа, що здійснює цю роль, несе обов'язки з підготовки документації до розробленого продукту, фінального опису функціональних можливостей. Також він бере участь в написанні супровідних документів (системи допомоги, керівництво користувача).

Завдання та цілі процесу верифікації

Спочатку розглянемо цілі верифікації. Основна мета процесу - доказ того, що результат розробки відповідає пред'явленим до нього вимогам. Зазвичай процес верифікації проводиться зверху вниз, починаючи від загальних вимог, заданих в технічному завданні та / або специфікації на всю інформаційну систему, і закінчуючи детальними вимогами до програмних модулів та їх взаємодії. До складу завдань процесу входить послідовна перевірка того, що в програмній системі:

- загальні вимоги до інформаційної системи, призначені для програмної реалізації, коректно перероблені в специфікацію вимог високого рівня до комплексу програм, що задовольняють вихідним системним вимогам;
- вимоги високого рівня правильно перероблені в архітектуру ПЗ та в специфікації вимог до функціональних компонентів низького рівня, які задовольняють вимогам високого рівня;
- специфікації вимог до функціональних компонентів ПЗ, розташованим між компонентами високого і низького рівня, задовольняють вимогам більш високого рівня;
- архітектура ПЗ та вимоги до компонентів низького рівня коректно перероблені в задовольняють їм вихідні тексти програмних та інформаційних модулів;
- вихідні тексти програм і відповідний їм виконуваний код не містять помилок.

Цілі верифікації ПЗ досягаються за допомогою послідовного виконання комбінації з інспекцій проектної документації та аналізу їх результатів, розробки тестових планів тестування та тест-вимог, тестових сценаріїв і процедур та подальшого виконання цих процедур. Тестові сценарії призначені для перевірки внутрішньої несуперечності та повноти реалізації вимог. Виконання тестових процедур має забезпечувати демонстрацію відповідності випробовуваних програм вихідним вимогам.

На вибір ефективних методів верифікації та послідовність їх застосування в найбільшою мірою впливають основні характеристики тестованих об'єктів:

- клас комплексу програм, що визначається глибиною зв'язку його функціонування з реальним часом і випадковими впливами із зовнішнього середовища, а також вимоги до якості обробки інформації та надійності функціонування;
- складність або масштаб (об'єм, розміри) комплексу програм і його функціональних компонентів, які є кінцевими результатами розробки;
- переважаючі елементи в програмах: здійснюють обчислення складних виразів і перетворення вимірюваних величин або обробні логічні і символьні дані для підготовки та відображення рішень.
- Визначимо деякі поняття та визначення, пов'язані з процесом тестування як складової

частини верифікації. Майерс дає такі визначення основних термінів.

Тестування - процес виконання програми з метою виявлення помилки.

Тестові дані - входи, які використовуються для перевірки системи.

Тестова ситуація (test case) - входи для перевірки системи і передбачувані виходи в залежності від входів, якщо система працює відповідно до специфікації вимог.

Хороша тестова ситуація - та ситуація, яка володіє великою імовірністю виявлення поки що невиявленою помилки.

Вдалий тест - тест, який виявляє поки що невиявлені помилки.

Помилка - дія програміста на етапі розробки, що приводить до того, що в програмному забезпеченні міститься внутрішній дефект, який в процесі роботи програми може привести до неправильного результату.

Відмова - непередбачувана поведінка системи, що приводить до несподіваного результату, яке могло бути викликане дефектами, що містяться в ній.

Таким чином, в процесі тестування програмного забезпечення, як правило, перевіряють наступне:

- програмне забезпечення відповідає вимогам на нього;
- в ситуаціях, не відображених у вимогах, програмне забезпечення поводиться адекватно, тобто не відбувається відмова системи;
- наявність типових помилок, які роблять програмісти.

Документація, що створюється на різних етапах життєвого циклу Синхронізація всіх етапів розробки відбувається за допомогою документів, які створюються на кожному з етапів. Документація при цьому створюється і на прямому відрізку життєвого циклу - при розробці програмної системи, і на зворотному - при її верифікації. Спробуємо на прикладі V-образного життєвого циклу простежити, які типи документів створюються на кожному з відрізків і які взаємозв'язки між ними існують (Рис 1.6).

Рис. 1.6. Процеси і документи при розробці програмних систем

Типи процесів тестування та верифікації та їх місце в різних моделях життєвого циклу

Модульне тестування

Модульне тестування призначено для невеликих модулів (процедур, класів і т.п.). В ході тестування одного модуля, розмір якого рідко перевищує 1000 рядків, можливо перевірити більшу частину логічних гілок алгоритму, типові граничні умови і т.п. Як критерій повноти тестування використовується повнота покриття тестами ключових елементів модуля (покриті всі вимоги, всі оператори, всі гілки логічних умов, всі компоненти логічних умов і т.п.) . Модульне тестування зазвичай виконується для кожного незалежного програмного модуля і є, мабуть, найбільш поширеним видом тестування, особливо для систем малих і середніх розмірів.

Інтеграційне тестування

При перевірці кожного модуля системи окремо неможливо дати гарантії того, що ці модулі будуть працювати разом. Як правило, можуть виникати (і виникають) проблеми, пов'язані з інтеграцією модулів, з їх взаємодією. Для виявлення таких проблем на ранніх

етапах розробки застосовують інтеграційне тестування, тобто тестування модулів,

об'єднаних у спільно працюють комплекси. Інтеграція модулів і інтеграційне тестування, як правило, проводиться протягом усього життєвого циклу розробки. Це дозволяє полегшити процес локалізації проблем і дефектів. При відкладанні інтеграції на останні етапи життєвого циклу локалізувати дефекти практично неможливо.

Системне тестування

Логічним завершенням інтеграційного тестування є системне тестування. На цьому етапі всі модулі системи об'єднані і працюють разом. Системне тестування призначене не для виявлення проблем окремих модулів - всі вони повинні були бути усунені раніше, а для виявлення проблем системи в цілому, проблем використання системи в реальному оточенні. Системні тести враховують такі аспекти системи, як стійкість в роботі, продуктивність, відповідність системи очікуванням користувача і т.п. Для визначення повноти системного тестування також використовуються інші способи - оцінюється повнота виконання всіх можливих сценаріїв роботи (як штатних, так і позаштатних), повнота різних методів взаємодії системи із зовнішнім світом і т.п.

Навантажувальне тестування

З системного тестування часто виділяють як самостійну процедуру тестування навантаження. В результаті навантажувального тестування можна оцінити, як буде змінюватися продуктивність системи під різним навантаженням. Дана інформація дозволяє приймати

рішення про область застосування системи, її масштабованості. В результаті навантажувального тестування найчастіше переглядається архітектура системи (якщо вона не забезпечує достатнього рівня продуктивності при заданій навантаженні) або окремі архітектурні рішення. З точки зору замовника системи тестування навантаження є одним із способів перевірки роботи системи в умовах, наближених до реальних.

Формальні інспекції

Формальна інспекція є одним із способів верифікації документів і програмного коду, що створюються в процесі розробки програмного забезпечення. В ході формальної інспекції групою фахівців здійснюється незалежна перевірка відповідності інспектованих документів вихідним документам. Незалежність перевірки забезпечується тим, що вона здійснюється інспекторами, не брали участь в розробці інспектується документа.

Верифікація сертифікується програмного забезпечення

Дамо кілька визначень, які визначають загальну структуру процесу сертифікації програмного забезпечення.

Сертифікація ПО - процес встановлення та офіційного визнання того, що розробка ПО проводилася відповідно до певних вимог.

Заявник - організація, що подає заявку у відповідний *сертифікуючий орган* на отримання сертифікату (відповідності, якості, придатності тощо) виробу.

Сертифікуючий орган - організація, яка розглядає заявку *Заявника* про проведення *Сертифікації ПО* і або самостійно, або шляхом формування спеціальної комісії виробляє набір процедур спрямованих на проведення процесу *Сертифікації ПО Заявника.*

Наглядова орган - комісія фахівців, що спостерігають за процесами розробки *Заявником* сертифікуєтмої інформаційної системи і дають висновок про відповідність даного процесу певним вимогам, яке передається на розгляд до *сертифікуючий орган.*

Тестування сертифікується програмного забезпечення має дві взаємодоповнюючі цілі.

- Перша - продемонструвати, що програмне забезпечення відповідає вимогам щодо нього.
- Друга - продемонструвати з високим рівнем довіри, що помилки, які можуть призвести до неприйнятних відмовним ситуацій, як вони визначені процесом, оцінки відмово безпечність системи, виявлено в процесі тестування.

Наприклад, відповідно до вимог стандарту DO-178B, для того, щоб задовольнити цілям тестування програмного забезпечення, необхідне наступне:

- тести, в першу чергу, повинні ґрунтуватися на вимогах до програмного забезпечення;
- тести повинні розроблятися для перевірки правильності функціонування та створення умов для виявлення потенційних помилок.
- аналіз повноти тестів, заснованих на вимогах на програмне забезпечення, повинен визначити, які вимоги не протестовані.
- аналіз повноти тестів, заснованих на структурі програмного коду, повинен визначити, які структури не виконувалися при тестуванні.

Лекція №2 **Тестування програмного коду, Повторюваність тестування.**

Тестування програмного коду (методи + оточення)

Анотація: Лекція присвячена процесу тестування програмного коду. Визначаються його завдання і цілі, перераховуються основні методи і підходи до тестування програмного коду. Вводиться поняття тестового оточення, розглядаються його компоненти та різні види оточення. Мета даної лекції: дати уявлення про процес тестування програмного коду, його видах. Визначити методи побудови тестового оточення, необхідного для виконання тестування

Ключові слова: определение , протоколирование , минимизация , цикла , очередь , модуль , модульное тестування , збірка ,ін тегращійне тестування , системне тестування , вихідні дані , відсоток , запуск , вхідні дані , висновок , доказ , unit testing

3.1. Завдання і цілі тестування програмного коду

Тестування програмного коду - процес виконання програмного коду, спрямований на виявлення існуючих в ньому дефектів. Під дефектом тут розуміється ділянку програмного коду, виконання якого за певних умов призводить до несподіваного поведінки системи (тобто поведінки, що не відповідає вимогам). Несподіване поведінку системи може призводити до збоїв у її роботі і відмов, в цьому випадку говорять про істотні дефектах програмного коду. Деякі дефекти викликають незначні проблеми, що не порушують процес функціонування системи, але кілька утрудняють роботу з нею. У цьому випадку говорять про середні або малозначних дефектах.

Завдання тестування при такому підході - *визначення* умов, при яких виявляються дефекти системи, і *протоколювання* цих умов. У завдання тестування зазвичай не входить виявлення конкретних дефектних ділянок програмного коду і ніколи не входить виправлення дефектів - це завдання налагодження, яка виконується за результатами тестування системи.

Мета застосування процедури тестування програмного коду - *мінімізація* кількості дефектів (особливо істотних) в кінцевому продукті.Тестування саме по собі не може гарантувати повної відсутності дефектів у програмному коді системи. Проте, у поєднанні з процесами верифікації та валідації, спрямованими на усунення суперечливості і неповноти проектної документації (зокрема - вимог на систему), грамотно організоване тестування дає гарантію того, що система задовольняє вимогам і веде себе відповідно з ними у всіх передбачених ситуаціях.

При розробці систем підвищеної надійності, наприклад, авіаційних, гарантії надійності досягаються за допомогою чіткої організації процесу тестування, визначення його зв'язку з іншими процесами життєвого *циклу,* введення кількісних характеристик, що дозволяють оцінювати успішність тестування. При цьому чим вище вимоги до надійності системи (її рівень критичності), тим більш жорсткі вимоги пред'являються.

Таким чином, в першу *чергу* ми розглядаємо не конкретні результати тестування конкретної системи, а загальну організацію процесу тестування, використовуючи підхід "добре організований процес дає якісний результат". Такий підхід є загальним для багатьох міжнародних та галузевих стандартів якості, про які більш докладно буде розказано в кінці даного курсу. Якість розроблюваної системи при такому підході є наслідком організованого процесу розробки і тестування, а не самостійним некерованим результатом.

Оскільки сучасні програмні системи мають вельми значні розміри, при тестуванні їх програмного коду використовується метод функціональної декомпозиції. Система розбивається на окремі модулі (класи, простору імен і т.п.), що мають певну вимогами функціональність і інтерфейси. Після цього окремо тестується кожен *модуль* - виконується *модульне тестування.* Потім відбувається *збірка* окремих модулів у більші конфігурації - виконується *інтеграційне тестування,* і нарешті, тестується система в цілому - виконується *системне тестування.*

З точки зору програмного коду, модульне, інтеграційне і *системне тестування* мають багато спільного, тому поки основна увага буде приділена модульним тестування, особливості інтеграційного і системного тестування будуть розглянуті пізніше.

У ході модульного тестування кожен *модуль* тестується як на відповідність вимогам, так і на відсутність проблемних ділянок програмного коду, які можуть викликати відмови і збої в роботі системи. Як правило, модулі не працюють поза системою - вони приймають дані від інших модулів, переробляють їх і передають далі. Для того, щоб з одного боку, ізолювати *модуль* від системи і виключити вплив потенційних помилок системи, а з іншого боку - забезпечити *модуль* всіма необхідними даними, використовується тестове оточення.

Завдання тестового оточення - створити середовище виконання для модуля, емулювати всі зовнішні інтерфейси, до яких звертається *модуль.* Про особливості організації тестового оточення піде мова далі в даній лекції.

Типова процедура тестування полягає у підготовці та виконанні тестових прикладів (також званих просто тестами). Кожен тестовий приклад перевіряє одну "ситуацію" в поведінці модуля і складається зі списку значень, переданих на вхід модуля, описи запуску і виконання переробки даних - тестового сценарію і списку значень, які очікуються на виході модуля в разі його коректної поведінки.Тестові сценарії складаються таким чином, щоб виключити звернення до внутрішніх даних модуля, вся взаємодія має відбуватися тільки через його зовнішні інтерфейси.

Виконання тестового прикладу підтримується тестовим оточенням, яке включає в себе програмну реалізацію тестового сценарію.Виконання починається з передачі модулю вхідних даних і запуску сценарію. Реальні *вихідні дані,* отримані від модуля в результаті виконання сценарію, зберігаються і порівнюються з очікуваними. У разі їх збігу тест вважається

пройденим, в іншому випадку - НЕ пройденим. Кожний не пройдений тест вказує на дефект або в тестируемом модулі, або в тестовому оточенні, або в описі тесту.

Сукупність описів тестових прикладів становить тест-план - основний документ, який визначає процедуру тестування програмного модуля. Тест-план задає не тільки самі тестові приклади, а й порядок їх слідування, який також може бути важливий. Структура та особливості тест-планів, а також проблеми, пов'язані з порядком проходження тестових прикладів, будуть розглянуті в наступних лекціях.

При тестуванні часто буває необхідно враховувати не тільки вимоги до системи, але і структуру програмного коду модуля, що тестується. У цьому випадку тести складаються таким чином, щоб детектувати типові помилки програмістів, викликані невірною інтерпретацією вимог. Застосовуються перевірки граничних умов, перевірки класів еквівалентності. Відсутність у системі можливостей, не заданих вимогами, гарантовано різними оцінками покриття програмного коду тестами, тобто оцінками того, який*відсоток* тих чи інших мовних конструкцій виконаний в результаті виконання всіх тестових прикладів. Про все це піде мова на завершення розгляду процесу тестування програмного коду.

3.2. Методи тестування

3.2.1. Чорний ящик

Основна ідея в тестуванні системи як чорного ящика полягає в тому, що всі матеріали, які доступні тестувальника, - вимоги на систему, що описують її поведінку, і сама система, працювати з якою він може, тільки подаючи на її входи деякі зовнішні впливи і спостерігаючи на виходах деякий результат. Всі внутрішні особливості реалізації системи приховані від

тестувальника, - таким чином, система являє собою "чорний ящик", правильність поведінки якого по відношенню до вимог і належить перевірити.

З точки зору програмного коду чорний ящик може представляти з собою набір класів (або модулів) з відомими зовнішніми інтерфейсами, але недоступними вихідними текстами.

Основне завдання тестувальника для даного методу тестування полягає в послідовній перевірці відповідності поведінки системи вимогам. Крім того, тестувальник повинен перевірити роботу системи в критичних ситуаціях - що відбувається у разі подання невірних вхідних значень. В ідеальній ситуації всі варіанти критичних ситуацій повинні бути описані у вимогах на систему і тестувальник залишається тільки придумувати конкретні перевірки цих вимог. Проте в реальності в результаті тестування зазвичай виявляється два типи проблем системи.

1. Невідповідність поведінки системи вимогам
2. Неадекватна поведінка системи в ситуаціях, не передбачених вимогами.

Звіти про обох типах проблем документуються і передаються розробникам. При цьому проблеми першого типу зазвичай викликають зміну програмного коду, набагато рідше - зміна вимог. Зміна вимог в даному випадку може знадобитися через їх суперечливості (кілька різних вимог описують різні моделі поведінки системи в одній і тій же самій ситуації) або некоректності (вимоги не відповідають дійсності).

Проблеми другого типу однозначно вимагають зміни вимог зважаючи на їх неповноти - у вимогах явно пропущена ситуація, яка веде до неадекватної поведінки системи. При цьому під неадекватною поведінкою може розумітися як повний крах системи, так і взагалі будь-яка поведінка, що не описане в вимогах.

Тестування чорної скриньки називають також тестуванням за вимогами, тому що це єдине джерело інформації для побудови тест-плану.

3.2.2. Скляний (білий) ящик

При тестуванні системи як скляного ящика тестувальник має доступ не тільки до вимог до системи, її входів і виходів, а й до її внутрішній структурі - бачить її програмний код.

Доступність програмного коду розширює можливості тестувальника тим, що він може бачити відповідність вимог ділянкам програмного коду і визначати тим самим, на весь чи програмний код існують вимоги. Програмний код, для якого відсутні вимоги, називають кодом, не вкритих вимогами. Такий код є потенційним джерелом неадекватної поведінки системи. Крім того, прозорість системи дозволяє поглибити аналіз її ділянок, що викликають проблеми - часто одна проблема нейтралізує іншу, і вони ніколи не виникають одночасно.

3.2.3. Тестування моделей

Тестування моделей знаходиться трохи осторонь від класичних методів верифікації програмного забезпечення. Причина насамперед у тому, що об'єкт тестування - не саме система, а її модель, спроектована формальними засобами. Якщо залишити осторонь питання перевірки коректності та застосовності самої моделі (вважається, що її коректність і відповідність вихідної системі можуть бути доведені формальними засобами), то тестувальник отримує в своє розпорядження досить потужний інструмент аналізу загальної цілісності системи. На моделі можна створити такі ситуації, які неможливо створити в тестовій лабораторії для реальної системи. Працюючи з моделлю програмного коду системи, можна аналізувати його властивості і такі параметри системи, як оптимальність алгоритмів або її стійкість.

Проте тестування моделей не набуло широкого поширення саме через труднощі, що виникають при розробці формального опису поведінки системи. Одне з небагатьох виключень - системи зв'язку, алгоритмічний і математичний апарат яких досить добре опрацьований.

3.2.4. Аналіз програмного коду (інспекції)

У багатьох ситуаціях тестування поведінки системи в цілому неможливо - окремі ділянки програмного коду можуть ніколи не виконуватися, при цьому вони будуть покриті вимогами. Прикладом таких ділянок коду можуть служити обробники виняткових ситуацій. Якщо, наприклад, два модулі передають один одному числові значення і функції перевірки коректності значень працюють в обох модулях, то функція перевірки модуля-приймача ніколи не буде активізована, тому що всі помилкові значення будуть відсічені ще в передавачі.

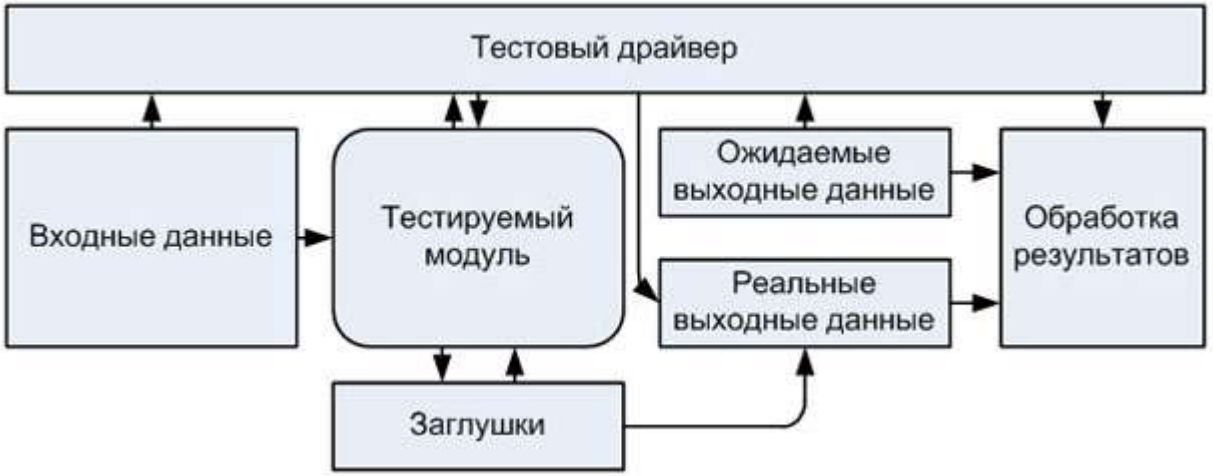
У цьому випадку виконується ручний аналіз програмного коду на коректність, званий також прогляданнями або інспекціями коду.Якщо в результаті інспекції виявляються проблемні ділянки, то інформація про це передається розробникам для виправлення нарівні з результатами звичайних тестів.

3.3. Тестове оточення

Основний обсяг тестування практично будь-якої складної системи зазвичай виконується в автоматичному режимі. Крім того, тестуєма система зазвичай розбивається на окремі модулі, кожен з яких тестується спочатку окремо від інших, потім в комплексі.

Це означає, що для виконання тестування необхідно створити деяку середу, яка забезпечить *запуск* і виконання модуля, що тестується, передасть йому *вхідні дані,* збере реальні *вихідні дані,* отримані в результаті роботи системи на заданих вхідних

даних.Після цього середовище має порівняти реальні *вихідні дані* з очікуваними і на підставі даного порівняння зробити *висновок* про відповідність поведінки модуля заданому (Рис 3.1).



збільшити зображення

Рис. 3.1. Узагальнена схема середовища тестування

Тестове оточення також може використовуватися для відчуження окремих модулів системи від всієї системи. Поділ модулів системи на ранніх етапах тестування дозволяє більш точно локалізувати проблеми, що виникають в їх програмному коді. Для підтримки роботи модуля у відриві від системи тестове оточення має моделювати поведінку всіх модулів, до функцій або даними яких звертається тестований *модуль.*

Оскільки тестове оточення саме є програмою (причому часто реалізованої неправильною мовою програмування, на якому написана система), воно саме повинно бути протестовано. Метою тестування тестового оточення є *доказ* того, що тестове оточення ніяким чином не спотворює виконання модуля, що тестується і адекватно моделює поведінку системи.

3.3.1. Драйвери і заглушки

Тестове оточення для програмного коду на структурних мовах програмування складається з двох компонентів - драйвера, який забезпечує запуск і виконання модуля, що тестується, і заглушок, які моделюють функції, що викликаються з даного модуля.Розробка тестового драйвера являє собою окрему задачу тестування, сам драйвер повинен бути протестований, щоб виключити невірне тестування. Драйвер і заглушки можуть мати різні рівні складності, необхідний рівень складності вибирається залежно від складності модуля, що тестується і рівня тестування. Так, драйвер може виконувати наступні функції:

1. Виклик модуля, що тестується
2. 1 + передача в модуль, що тестується вхідних значень і прийом результатів
3. 2 + висновок вихідних значень
4. 3 + протоколювання процесу тестування і ключових точок програми

Заглушки можуть виконувати такі функції:

1. Не проводити ніяких дій (такі заглушки потрібні для коректної збірки модуля, що тестується)
2. Виводити повідомлення про те, що заглушка була викликана

3. 1 + виводити повідомлення зі значеннями параметрів, переданих у функцію
4. 2 + повертати значення, заздалегідь задане у вхідних параметрах тесту
5. 3 + виводити значення, заздалегідь задане у вхідних параметрах тесту
6. 3 + приймати від тестованого ПО значення і передавати їх в драйвер [10].

Для тестування програмного коду, написаного на процедурному мовою програмування, використовуються драйвери, що представляють собою програму з точкою входу (наприклад, функцією main ()), функціями запуску модуля, що тестується і функціями збору результатів. Зазвичай драйвер має як мінімум одну функцію - точку входу, якій передається керування при його виклику.

Функції-заглушки можуть поміщатися в той же файл вихідного коду, що й основний текст драйвера. Імена та параметри заглушок повинні збігатися з іменами і параметрами "заглушає" функцій реальної системи. Ця вимога важливо не стільки з точки зору коректної складання системи (при складанні тестового драйвера і тестованого ПО може використовуватися приведення типів), скільки для того, щоб максимально точно моделювати поведінку реальної системи з передачі даних. Так, наприклад, якщо в реальній системі є функція обчислення квадратного кореня

```
double sqrt (double value);
```

то, з точки зору складання системи, замість типу double може використовуватися і float, але зниження точності може викликати непередбачувані результати в тестируемом модулі.

Як приклад драйвера і заглушок розглянемо реалізацію стека на мові С, причому значення, що поміщаються в стек, зберігаються не в оперативній пам'яті, а поміщаються в ППЗУ за допомогою окремого модуля, що містить дві функції - записи даних в ППЗУ за адресою і читання даних за адресою .

Формат цих функцій наступний:

```
void NV_Read (char * destination, long length, long offset);

void NV_Write (char * source, long length, long offset);
```

Тут destination - адреса області пам'яті, в яку записується значення, лічену з ППЗУ, source - адреса області пам'яті, з якої записується значення в ППЗУ, length - довжина записуваної області пам'яті, offset - зміщення відносно початкової адреси ППЗУ.

Реалізація стека з використанням цих функцій виглядає наступним чином:

```
long currentOffset;


void initStack ()
{
  currentOffset = 0;
}
```

```
void push (int value)
{
  NV_Write ((int *) & value, sizeof (int), currentOffset);
  currentOffset + = sizeof (int);
}

int pop ()
{
  int value;
  if (currentOffset> 0)
  {
    NV_Read ((int *) & value, sizeof (int), currentOffset;
    currentOffset-= sizeof (int);
  }
```

```
}
```

При виконанні цього коду на реальній системі відбувається запис в ППЗУ, однак, якщо ми хочемо протестувати тільки реалізацію стека, ізолювавши її від реалізації модуля роботи з ППЗУ, необхідно використовувати заглушки замість реальних функцій. Для імітації роботи ППЗУ можна виділити достатньо велику ділянку оперативної пам'яті, в якій і проводитиметься запис даних, одержуваних заглушкою.

Заглушки для функцій можуть виглядати наступним чином:

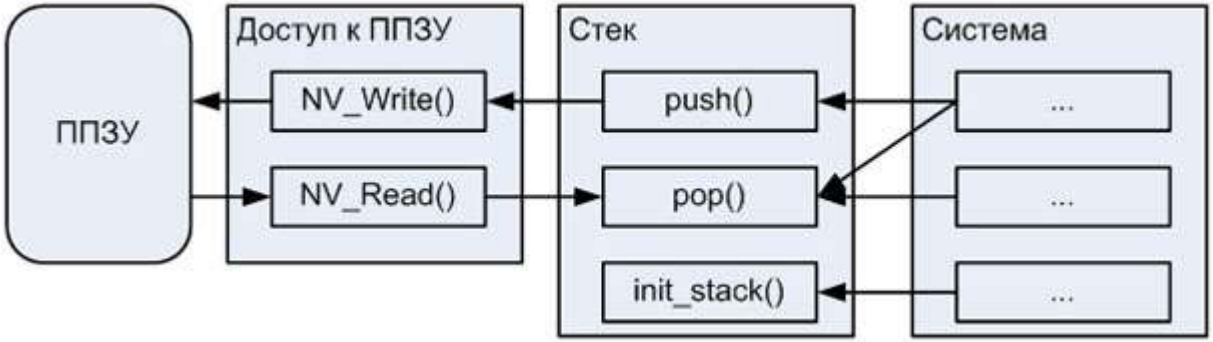
```
 char nvrom [1024];


void NV_Read (char * destination, long length, long offset)
{
  printf ("NV_Read called \ n");
  memcpy (destination, nvrom + offset, length);
}
void NV_Write (char * source, long length, long offset);
{
  printf ("NV_Write called \ n");
```

```
  memcpy (nvrom + offset, source, length);
}
```

Кожна з заглушок виводить трассіровочние повідомлення і переміщує передане значення в пам'ять, емулює ППЗУ (функціяNV_Write), або повертає за посиланням значення, яке зберігається в пам'яті, що емулює ППЗУ (функція NV_Read).

Схема взаємодії тестованого ПО (функцій роботи зі стеком) з реальним оточенням (основною частиною системи і модулем роботи з ППЗУ) і тестовим оточенням (драйвером і заглушками функцій роботи з ППЗУ) показана на Рис 3.2 і Рис 3.3 .



збільшити зображення

Рис. 3.2. Схема взаємодії частин реальної системи

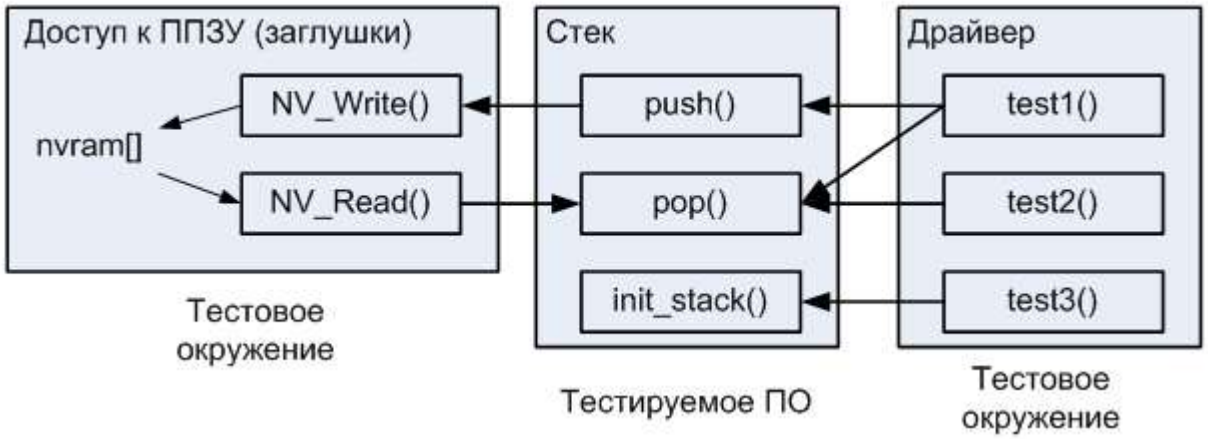


Рис. 3.3. Схема взаємодії тестового оточення і тестованого ПО

3.3.2. Тестові класи

Тестове оточення для об'єктно-орієнтованого ПЗ виконує ті ж самі функції, що і для структурних програм (на процедурних мовах).Однак, воно має деякі особливості, пов'язані із застосуванням успадкування та інкапсуляції.

Якщо при тестуванні структурних програм мінімальним тестируемим об'єктом є функція, то в об'єктно-орієнтованим ПО мінімальним об'єктом є клас. При застосуванні принципу інкапсуляції всі внутрішні дані класу і деяка частина його методів недоступна ззовні. У цьому випадку тестувальник позбавлений можливості звертатися до своїх тестах до даних класу і довільним чином викликати методи; єдине, що йому доступно - викликати методи зовнішнього інтерфейсу класу.

Існує кілька підходів до тестування класів, кожен з них накладає свої обмеження на структуру драйвера і заглушок.

1. Драйвер створює один або більше об'єктів тестованого класу, всі звернення до об'єктів відбуваються тільки з використанням їх зовнішнього інтерфейсу. Текст драйвера в цьому випадку представляє собою т.зв. тестуючий клас, який містить по одному методу для кожного тестового прикладу. Процес тестування полягає в послідовному виклику цих методів. Замість заглушок до складу тестового оточення входить програмний код реальної системи, відповідно, відсутня ізоляція тестованого класу. Однак, саме такий підхід до тестування прийнятий зараз у більшості методологій і середовищ розробки. Його класичне назва - *unit testing* (тестування модулів), більш докладно він буде розглядатися пізніше.
2. Аналогічно попередньому підходу, але для всіх класів, які використовує тестований клас, створюються заглушки
3. Програмний код тестованого класу модифікується таким чином, щоб відкрити доступ до всіх його властивостей і методів.Будова тестового оточення в цьому випадку повністю аналогічно оточенню для тестування структурних програм.
4. Використовуються спеціальні засоби доступу до закритих даних і методам класу на рівні об'єктного або виконуваного коду - скрипти відладчика або accessors в Visual Studio.

Основна перевага перших двох методів: при їх використанні клас працює точно таким же чином, як в реальній системі. Однак у цьому випадку не можна гарантувати, що в процесі тестування буде виконано весь програмний код класу і не залишиться непротестованих методів.

Основний недолік 3-го методу: після зміни вихідних текстів модуля, що тестується не можна дати гарантії того, що клас буде вести себе таким же чином, як і вихідний. Зокрема це пов'язано з тим, що зміна захисту даних класу впливає на спадкування даних і методів іншими класами.

Тестування спадкування - окрема складна задача в об'єктно-орієнтованих системах. Після того, як протестований базовий клас, необхідно тестувати класи-нащадки. Однак, для базового класу не можна створювати заглушки, т.к. в цьому випадку можна припустити можливі проблеми поліморфізму. Якщо клас-нащадок використовує методи базового класу для обробки власних даних, необхідно переконатися в тому, що ці методи працюють.

Таким чином, ієрархія класів може тестуватися зверху вниз, починаючи від базового класу. Тестове оточення при цьому може змінюватися для кожної тестованої конфігурації класів.

3.3.3. Генератори сигналів (подієво-керований код)

Значна частина складних програм в даний час використовує ту чи іншу форму взаємодії між процесами. Це обумовлено природною еволюцією підходів до проектування програмних систем, яка послідовно пройшла наступні етапи [11].

1. Монолітні програми, що містять у своєму коді всі необхідні для своєї роботи інструкції. Обмін даними всередині таких програм проводиться за допомогою передачі параметрів функцій і використання глобальних змінних. При запуску таких програм утворюється один процес, який виконує всю необхідну роботу.
2. Модульні програми, які складаються з окремих програмних модулів з чітко визначеними інтерфейсами викликів. Об'єднання модулів у програму може відбуватися або на етапі складання виконуваного файлу (статична збірка або static linking), або на етапі виконання програми (динамічна збірка або dynamic linking). Перевага модульних програм

полягає в досягненні деякого рівня універсальності - один модуль може бути замінений іншим. Однак, модульна програма все одно являє собою один процес, а дані, необхідні для вирішення завдання, передаються всередині процесу як параметри функцій.

3. Програми, що використовують межпроцессное взаємодію. Такі програми утворюють програмний комплекс, призначений для вирішення загальної задачі. Кожна запущена програма утворює один або більше процесів. Кожен з процесів або використовує для вирішення завдання свої власні дані і обмінюється з іншими процесами тільки результатом своєї роботи, або працює із загальною областю даних, поділюваних між різними процесами. Для вирішення особливо складних завдань процеси можуть бути запущені на різних фізичних комп'ютерах і взаємодіяти через мережу. Перевага використання між процесами взаємодії полягає в ще більшій універсальності - взаємодіючі процеси можуть бути замінені незалежно один від одного при збереженні інтерфейсу взаємодії. Інша перевага полягає в тому, що обчислювальна навантаження розподіляється між процесами. Це дозволяє операційній системі управляти пріоритетами виконання окремих частин програмного комплексу, виділяючи більшу або меншу кількість ресурсів ресурсоємним процесам.

При виконанні багатьох процесів, вирішальних загальну задачу, використовуються кілька типових механізмів взаємодії між ними, спрямованих на вирішення наступних завдань:

- передача даних від одного процесу до іншого;
- спільне використання одних і тих же даних кількома процесами;
- повідомлення про зміну стану процесів.

У всіх цих випадках типова структура кожного процесу являє собою кінцевий автомат з набором станів і переходів між ними.Перебуваючи в певному стані, процес виконує обробку даних, при переході між станами - пересилає дані іншим процесам або приймає дані від них [12].

Для моделювання кінцевих автоматів використовуються stateflow [13] або SDL-діаграми [13], акцент в яких робиться відповідно на умовах переходу між станами і пересилає даними.

Так, на Рис 3.4 показана схема процесу прийому / передачі даних. Закругленими прямокутниками вказані стану процесу, тонкими стрілками - переходи між станами, великими стрілками - надсилаються дані. Перебуваючи в стані "Старт", процес посилає в зовнішній світ (або процесу, з яким він обмінюється даними) повідомлення про свою готовність до початку сеансу передачі даних.Після отримання від другого процесу підтвердження про готовність починається сеанс обміну даними. У разі надходження повідомлення про кінець даних відбувається завершення сеансу і перехід в стартове стан. У разі надходження невірних даних (наприклад, неправильного формату або з невірною контрольною сумою) процес переходить у стан "Помилка", вийти з якого можливо тільки завершенням і перезапуском процесу.

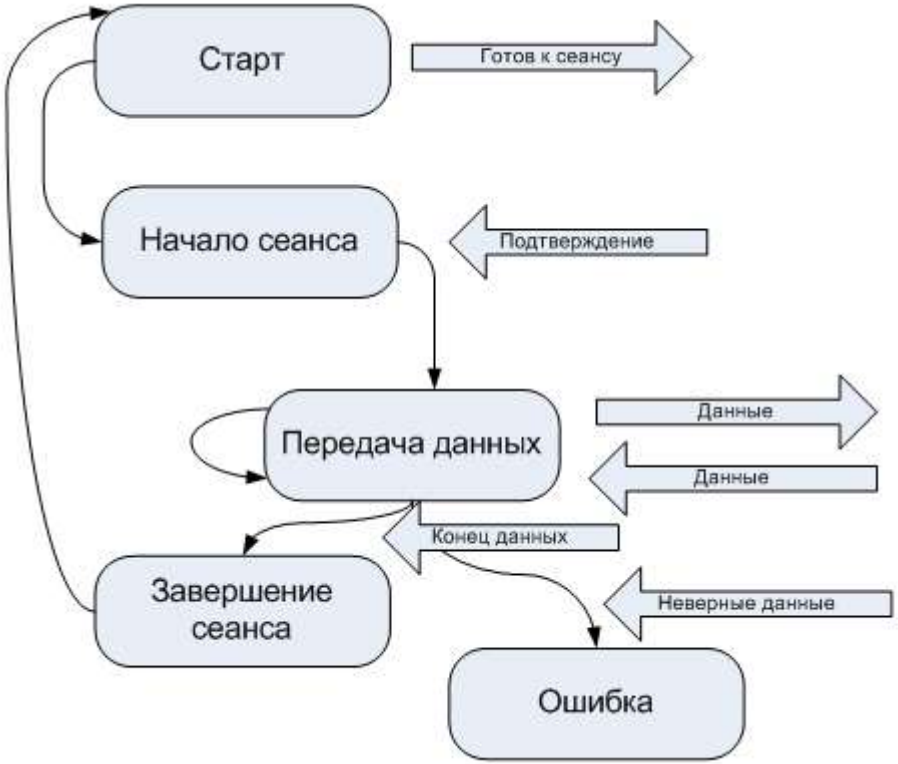


Рис. 3.4. Приклад кінцевого автомата процесу прийому-передачі даних

Тестове оточення для такого процесу також повинно мати структуру кінцевого автомата і пересилати дані в тому ж форматі, що і тестований процес. Метою тестування в даному випадку буде показати, що процес обробляє прийняті дані відповідно до вимог, формати переданих даних коректні, а також що процес під час своєї роботи дійсно проходить всі стану кінцевого автомата, моделює його поведінку.

Тестування програмного коду (тестові приклади)

Анотація: Лекція продовжує тематику тестування програмного коду, розпочату в лекції 2. Лекція присвячена розробці тестових прикладів: визначаються методи їх створення, проводиться класифікація типів тестових прикладів, розглядається тестування робастности, визначаються поняття класів еквівалентності, розглядаються приклади тестування компараторів. Мета даної лекції: дати знання, необхідні для самостійного аналізу, розробки та модифікації тестових прикладів на основі тест-вимог

4.1. Тестові приклади

Тестове оточення, розглянуте в попередній лекції, забезпечує процес тестування необхідною інфраструктурою і підтримує її. Безпосередньо для тестування крім тестового оточення необхідно визначити перевірочні завдання, які виконуватиме система або її частину. Такі перевірочні завдання називають тестовими прикладами.

Як вже було сказано вище, кожен тестовий приклад складається з вхідних значень для системи, опису сценарію роботи прикладу і очікуваних вихідних значень. Мета виконання будь-якого тестового прикладу - або продемонструвати наявність в системі дефекту, або довести його відсутність.

4.1.1. Тест-вимоги як основне джерело інформації для створення тестових прикладів

Основним джерелом інформації для створення тестових прикладів є різного роду документація на систему, наприклад, функціональні вимоги і вимоги до інтерфейсу.

Функціональні вимоги описують поведінку системи як "чорного ящика", тобто виключно з позицій того, що повинна робити система в різних ситуаціях. Іншими словами, функціональні вимоги визначають реакцію системи на різні вхідні дії.

Наприклад, функціональні вимоги на програмний модуль, який розраховує і перевіряючий контрольну суму для запису, можуть виглядати наступним чином.

Функціональні вимоги на модуль розрахунку і перевірки контрольної суми

Зовнішній інтерфейс модуля

1. Структура record_type

```
struct record_type
{
  bool A;
  int B [20];
  signed char C [5];
  unsigned int CRC;
  double D [1];
```

}

2. Мінлива Empty

bool Empty;

3. Функція підрахунку контрольної суми запису Set_CRC

void Set_CRC (record_type record);

Вхід:

Запис record, с невизначеним значенням поля CRC.

Вихід:

Запис record, з обчисленим за заданими правилами значення поля CRC.

Мінлива Empty.

4. Функція перевірки контрольної суми запису Check_CRC

bool Check_CRC (record_type record);

Вхід:

Запис Rec_Mess с певним значенням поля CRC.

Вихід:

Значення, що повертається true або false. Мінлива Empty.

Функціональні вимоги

1. Ініціалізація модуля

При ініціалізації модуля мінлива Empty повинна бути встановлена в значення TRUE.

2. Підрахунок контрольної суми запису

   a. Розрахунок контрольної суми

Процедура Set_CRC повинна робити підрахунок контрольної суми запису Rec_Mess за алгоритмом CRC32. При підрахунку контрольної суми значення поля CRC не повинно брати участь в підсумовуванні. На підставі проведених розрахунків має бути обчислено і визначено значення поля CRC таким чином, щоб при підрахунку контрольної суми разом із встановленим значенням цього поля контрольна сума дорівнювала нулю.

b. Установка значення змінної Empty

Якщо всі байти полів запису (крім можливо CRC поля) мають нульове значення (код 00000000B), то значення змінної Empty має бути встановлено в TRUE.

Їли хоча б один байт запису (виключаючи байти поля CRC) не нульовий, то значення змінної Empty має бути встановлено в FALSE.

3. Перевірка контрольної суми запису

a. Перевірка контрольної суми

Процедура повинна обчислювати за заданим алгоритмом CRC32 контрольну суму записи Rec_Mess. Що повертається процедурою значення має дорівнювати TRUE, якщо підрахована значення дорівнює нулю. При ненульове значення підрахованої контрольної суми повинно повертатися значення FALSE.

b. Установка значення змінної Empty

Якщо всі байти полів запису, включаючи значення CRC поля, мають нульове значення (код 00000000B),

то значення змінної Empty має бути встановлено в TRUE.

Їли хоча б один байт запису не нульовий, то значення змінної Empty має бути встановлено в FALSE.

Приклад 4.1.

Початковий етап роботи тестувальника полягає у формуванні тест-вимог, відповідних функціональним вимогам. Основна мета тест-вимог - визначити, яка функціональність системи повинна бути протестована. У самому простому випадку одному функціональному вимогу відповідає одне тест-вимогу.Однак найчастіше тест-вимоги деталізують формулювання функціональних вимог.

Тест-вимоги визначають, що повинно бути протестовано, але не визначають, як це повинно бути зроблено. Наприклад, для перерахованих вище функціональних вимог можна сформулювати такі тест-вимоги.

Тест-вимоги

1. Перевірка ініціалізація модуля

   Перевірити, що початкове значення змінної Empty встановлено TRUE.

2. Перевірка підрахунку контрольної суми

a. Перевірити, що в процедурі Set_CRC обчислення контрольної суми проводиться за правилами алгоритму CRC32, як визначено в секції 2a функціональних вимог.

b. Перевірити, що обчислене значення контрольної суми не залежить від початкового значення поля CRC.

c. Перевірити, що обчислене значення контрольної суми не залежить від значень байт вирівнювання полів запису.

d. Перевірити, що значення змінної Empty встановлюється при кожному виклику функції Set_CRC залежно від значень полів запису, як визначено в секції 2b функціональних вимог.

3. Перевірка процедури Check_CRC

a. Перевірити, що при зверненні до процедури Check_CRC обчислення контрольної суми проводиться за правилами алгоритму CRC32, як визначено в секції 3a функціональних вимог.

b. Перевірити, що повертається значення одно TRUE, якщо конітрольная сума перевіряється записи правильна, і FALSE - у противному випадку.

c. Перевірити, що перевірка правильності значення контрольної суми не залежить від значень байт вирівнювання

полів запису.

d. Перевірити, що значення змінної Empty встановлюється при кожному виклику функції Check_CRC залежно від значень полів запису, як визначено в секції 3b функціональних вимог.

Приклад 4.2.

Особливості реалізації тестового оточення і конкретні значення, що подаються на вхід системи та очікувані на її виході, визначаються тестовими прикладами. Одному тест-вимозі відповідає як мінімум один тестовий приклад.

4.1.2. Типи тестових прикладів

Розглянемо різні класи тестових прикладів, спрямовані на виявлення різних дефектів в роботі програмної системи.

- **Допустимі дані**

Найчастіше дефекти в програмних системах проявляються при обробці нестандартних даних, не передбачених вимогами - при введенні невірних символів, порожніх рядків або при дуже великій швидкості введення інформації. Однак, перед пошуком таких дефектів необхідно упевнитися в тому, що програма коректно обробляє вірні дані, передбачені специфікацією, тобтоперевірити роботу основних алгоритмів. Так, для функції обчислення контрольної суми допустимими вхідними даними буде довільна запис, що містить дані у всіх полях, крім поля контрольної суми CRC.

```
record_type test_value1;
int i;
```

```
test_value1.A = false;
for (i = 0; i <20; i + +)
   test_value1.B [i] = i;
for (i = 0; i <5; i + +)
   test_value1.C [i] = i +5;
test_value1.D [0] = i +8;
test_value1.CRC = 0;

Set_CRC (test_value1);

printf ("% d \ n", test_value1.CRC);
```

Сценарієм буде виклик функції Set_CRC, а очікуваним вихідним значенням - коректне значення поля CRC, розраховане за алгоритмом CRC32.

Зазвичай для перевірки допустимих даних достатньо одного тестового прикладу. Але функціональні вимоги можуть визначати різні групи допустимих даних, які можуть об'єднуватися в класи еквівалентності. У цьому випадку необхідно визначати як мінімум один тестовий приклад для одного класу еквівалентності. Більш докладно мова про класах еквівалентності піде далі.

- **Граничні дані**

Окремий вид допустимих даних, передача яких в систему може розкрити дефект - граничні дані, тобто наприклад, числа, значення яких є граничними для їх типу, рядки граничної або нульовий довжини і т.п. Зазвичай за допомогою тестування граничних умов виявляються проблеми з арифметичним порівнянням чисел або з ітераторами циклів.

Для тестування функції Set_CRC на граничних умовах можна визначити два тестових прикладу з мінімальними і максимальними значеннями полів в записі.

```
 record_type test_value2;

record_type test_value3;

int i;


test_value2.A = false;

for (i = 0; i <20; i + +)

   test_value2.B [i] = 0;
```

```
for (i = 0; i <5; i + +)
   test_value2.C [i] = 0;
test_value2.D [0] = 0;
test_value2.CRC = 0;

Set_CRC (test_value2);

printf ("% d \ n", test_value2.CRC);

test_value3.A = true;
for (i = 0; i <20; i + +)
   test_value3.B [i] = pow (2, sizeof (test_value3.B [i]) * 8) -1;
for (i = 0; i <5; i + +)
   test_value3.C [i] = pow (2, sizeof (test_value3.C [i]) * 8) -1;
test_value3.D [0] = pow (2, sizeof (test_value3.D [0]) * 8) -1;
```

```
test_value3.CRC = pow (2, sizeof (test_value3.CRC) * 8) -1;


Set_CRC (test_value3);


printf ("% d \ n", test_value3.CRC);
```

- **Відсутність даних**

Дефекти можуть проявитися і у випадку, якщо системі не передається ніяких даних або передаються дані нульового розміру.Для тестування функції Set_CRC при відсутності даних можна викликати її, передавши як параметр неініціалізованих структуру.Однак такий тест не є точним прикладом відсутності даних, скоріше, це приклад випадкових даних (можливо - невірних).

```
record_type test_value4;


Set_CRC (test_value4);


printf ("% d \ n", test_value4.CRC);
```

- **Повторне введення даних**

У разі повторної передачі на вхід системи тих же самих даних можуть виходити відмінності у вихідних даних, не передбачені у вимогах. Як правило, дефекти такого типу проявляються в результаті того, що система не встановлює внутрішні змінні в початковий стан або в результаті помилок округлення.

```
 record_type test_value5;
int i;


test_value5.A = false;
for (i = 0; i <20; i + +)
   test_value5.B [i] = i;
for (i = 0; i <5; i + +)
   test_value5.C [i] = i +5;
test_value5.D [0] = i +8;
test_value5.CRC = 0;


Set_CRC (test_value5);
```

```
printf ("% d \ n", test_value5.CRC);


Set_CRC (test_value5);
printf ("% d \ n", test_value5.CRC);
```

- **Невірні дані**

При перевірці поведінки системи необхідно не забувати перевіряти її поведінка при передачі їй даних, не передбачених вимогами - занадто довгих або занадто коротких рядків, невірних символів, чисел за межами вичислімого діапазону і т.п. Невірні дані, як і допустимі, також можна розділяти на різні класи еквівалентності. Прикладом невірних даних для функції Set_CRCможе служити запис з іншою структурою, передана у функцію через приведення типів. Якщо розрахунок контрольної суми використовує імена полів запису, то контрольна сума може виявитися обчисленої невірно або може відбутися перезапис областей пам'яті, не призначених для зберігання даних.

```
 struct record_type2
{
  int F;
  int G [45];
  int H [8];
```

```
  unsigned int CRC;
  int K [2];
}
record_type2 test_value6;

Set_CRC ((record_type) test_value6);

printf ("% d \ n", test_value6.CRC);
```

- **Реініціалізація системи**

Механізми повторної ініціалізації системи під час її роботи також можуть містити дефекти. У першу чергу, ці дефекти можуть проявлятися в тому, що не всі внутрішні дані системи після реініціалізації прийдуть в початковий стан. У результаті може відбутися збій в роботі системи.

Прикладом реініціалізації модуля обчислення CRC може служити примусове обнулення змінної empty.

```
 record_type test_value7;
int i;
```

```
test_value7.A = false;
for (i = 0; i <20; i + +)
   test_value7.B [i] = i;
for (i = 0; i <5; i + +)
   test_value7.C [i] = i +5;
test_value7.D [0] = i +8;
test_value7.CRC = 0;

Set_CRC (test_value7);

printf ("% d \ n", test_value1.CRC);

empty = true;
```

```
Set_CRC (test_value7);


printf ("% d \ n", test_value1.CRC);
```

- **Стійкість системи**

Під стійкістю системи можна розуміти її здатність витримувати нештатну навантаження, явно не передбачену вимогами.Наприклад, стоїть питання, чи збереже система працездатність після 10 тисяч викликів. Для функції Set_CRC можна реалізувати наступний тестовий приклад:

```
 record_type test_value8;
int i;


test_value8.A = false;
for (i = 0; i <20; i + +)
   test_value8.B [i] = i;
for (i = 0; i <5; i + +)
   test_value8.C [i] = i +5;
```

```
test_value8.D [0] = i +8;

test_value8.CRC = 0;


for (i = 0; i <10000; i + +)

   Set_CRC (test_value8);


printf ("% d \ n", test_value1.CRC);
```

Аналогічний аналіз може бути зроблений шляхом перегляду тексту програми (якщо він доступний при тестуванні) на підставі відсутності "історії" (збережених даних) в реалізації програми, тобто даних, значення яких може змінюватися в залежності від кількості запусків програми. Таким чином, в ряді випадків тестування може бути замінено аналізом програмного коду.

- **Позаштатні стану середовища виконання**

Позаштатні стану середовища виконання (наприклад, вичерпання пам'яті, дискового простору або тривала нестача процесорного часу) можуть ускладнювати роботу системи або робити її неможливою. Основне завдання системи в такій ситуації - коректно завершити або призупинити свою роботу.

Прикладом тестового прикладу, що створює нештатное стан середовища, для функції Set_CRC може служити виділення всієї вільної пам'яті перед викликом функції. Якщо Set_CRC використовує динамічну пам'ять, то в ній повинні бути присутніми перевірки на можливість виділити пам'ять, в іншому випадку виконання функції викличе її аварійне завершення:

```
 record_type test_value9;
int i;
int * heap;

heap = malloc (_MAXMEM);

test_value9.A = false;
for (i = 0; i <20; i + +)
   test_value9.B [i] = i;
for (i = 0; i <5; i + +)
   test_value9.C [i] = i +5;
test_value9.D [0] = i +8;
```

```
test_value9.CRC = 0;


Set_CRC (test_value9);


free (heap);


printf ("% d \ n", test_value9.CRC);
```

4.1.2.1. Граничні умови

У тестових прикладах, прямо відповідних тест-вимогам, зазвичай використовуються вхідні значення, що знаходяться завідомо всередині допустимого діапазону. Один із способів перевірки стійкості системи на значеннях, близьких до граничних, - створювати для кожного входу як мінімум три тестових прикладу:

- Значення всередині діапазону
- Мінімальне значення
- Максимальне значення

Для ще більшої впевненості в працездатності системи використовують п'ять тестових прикладів:

- Значення всередині діапазону
- Мінімальне значення
- Мінімальне значення + 1
- Максимальне значення
- Максимальне значення - 1

Такий спосіб перевірки називається перевіркою на граничних значеннях. Така перевірка дозволяє виявляти проблеми, пов'язані з виходом за межі діапазону. Наприклад, якщо у функцію

```
 char sum (char a, char b)
{
   return a + b;
}
```

обчислює суму чисел a і b, будуть передані значення 255 і 255, то в разі відсутності спеціальної обробки ситуації переповнення сума буде обчислена невірно.

Інша область, при тестуванні якої корисно користуватися перевіркою на граничних значеннях, - індекси масивів. Наприклад, функція

```
 void abs_array (char array [], char size)
```

```
{
   for (int i = 1; i <= size; i + +)
   {
        array [i] = abs (array [i]);
   }
   return;
}
```

замінює значення на значення по модулю у кожного елемента переданого їй масиву, містить помилку в циклі for, яка може бути легко виявлена при передачі у функцію масиву одиничного розміру.

4.1.3. Перевірка робастности (виходу за межі діапазону)

Робастність системи - це ступінь її чутливості до факторів, не врахованих на етапах її проектування, наприклад, до неточності основного алгоритму, що приводить до помилок округлення при обчисленнях, збоїв у зовнішньому середовищі або до даних, значення яких знаходяться поза допустимого діапазону. Найчастіше під робастного програмних систем розуміють саме стійкість до некоректних даних. Система повинна бути здатна коректно обробляти такі дані шляхом видачі відповідних повідомлень про помилки, збої і відмови системи на подібних даних неприпустимі.

Для тестування робастности до тестових прикладів, розглянутим у попередньому розділі, додаються ще два тестових прикладу:

- Мінімальне значення - 1
- Максимальне значення + 1,

перевіряючі поведінку системи за кордоном допустимого діапазону, а також у разі тестування операцій порівняння додатково дають гарантію того, що в них не допущена помилка.

Таким чином, якщо зобразити допустимий інтервал як на Рис 4.1 , то можна бачити, що для тестування інтервальних значень достатньо 7 тестових прикладів - п'яти допустимих і двох на робастність.

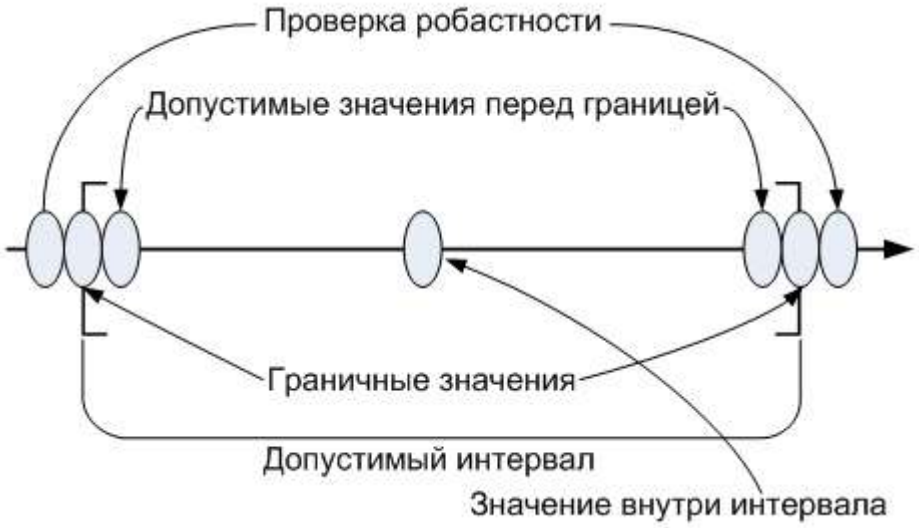


Рис. 4.1. Рекомендовані перевірочні значення

У літературі часто зустрічається твердження, що значення усередині інтервалу є надлишковим і його тестування не потрібно.Однак, перевірка внутрішнього значення є корисною як мінімум з психологічної точки зору, а також у разі, якщо інтервал обмежений складними граничними умовами. Також рекомендується окремо перевіряти значення 0 (навіть якщо воно знаходиться усередині інтервалу), тому що часто це значення обробляється некоректно (наприклад, у разі поділу на 0).

4.1.4. Класи еквівалентності

При розробці тестових прикладів може виникнути така ситуація, в якій різні вхідні значення призводять до одних і тих же реакцій системи. Якщо при цьому такі вхідні значення мають щось спільне, то можливе об'єднання таких значень в класи еквівалентності, тобто виконання еквівалентного розбиття множини допустимих вхідних значень.

Розбиття на класи еквівалентності - це, в першу чергу, спосіб зменшення необхідного числа тестових прикладів. Зазвичай, якщо в тест-вимогах спеціально не обумовлено інше, при тестуванні досить виконати тільки один тестовий приклад для кожного класу еквівалентності. Розбиття на класи еквівалентності особливо корисно, коли на вхід системи може бути подано велику кількість різних значень; тестування кожного можливого значення призвело б до занадто великого обсягу тестування.

Розглянуті вище граничні умови можуть служити прикладом класів еквівалентності:

1. Значення з середини інтервалу.
2. Граничні значення.
3. Неприпустимі значення за межами інтервалу.

Таким чином, тестування граничних умов і робастности є окремим випадком тестування з використанням класів еквівалентності - замість того, щоб тестувати всі неприпустимі значення, вибираються тільки сусідні з граничними.

При визначенні класів еквівалентності слід керуватися такими правилами:

- Завжди буде, щонайменше, два класи: коректний і некоректний
- Якщо вхідна умова визначає діапазон значень, то, як правило, буває три класи: менше ніж діапазон, всередині діапазону і більше ніж діапазон. (Значення на кінцях діапазону можуть трактуватися як граничні значення.)
- Якщо елементи діапазону обробляються по-різному, то кожному варіанту обробки будуть відповідати різні вимоги.

Іншим прикладом розбиття на класи еквівалентності може служити тестування відкриття файлу по його імені. В результаті тестування необхідно визначити, чи всі варіанти імен обробляються системою згідно з такими тест-вимогам.

1. Перевірити, що в разі присутності в імені файлу символів, які не є буквами латинського алфавіту і цифрами, система виводить повідомлення про помилку.
2. Перевірити, що в тому випадку, коли довжина імені файлу перевищує 11 символів, система видає повідомлення про помилку
3. Перевірити, що система не розрізняє регістр символів імені при відкритті файлу.
4. Перевірити, що при відкритті файлів з іменами, що не суперечать вимогам 1-3, система відкриває файл.

Вхідними значеннями тестового прикладу є різні імена файлів, вихідними - реакція системи (помилка або успішне відкриття).

Можна виділити наступні класи еквівалентності:

По довжині імені:

1. Довжина імені менше 11 символів
2. Довжина імені дорівнює 11 символам
3. Довжина імені більше 11 символів

За символам:

1. Ім'я, що складається з цифр і букв змішаного регістра
2. Ім'я, що складається з цифр і букв нижнього регістру
3. Ім'я, що складається з цифр і букв верхнього регістру
4. Ім'я, що складається тільки з цифр
5. Ім'я, що складається тільки з букв
6. Ім'я, що включає знаки пунктуації (Не буквено-цифрові символи)
7. Ім'я, що включає керуючі символи (Не буквено-цифрові символи)

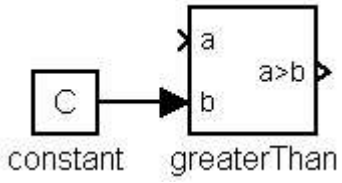
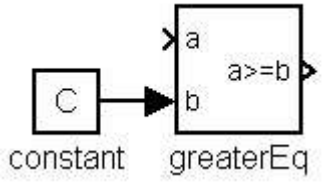
Ці класи еквівалентності ілюструють, що перевірки на кордонах інтервалів застосовні не тільки для тестування арифметичних операцій і операцій порівняння. Практично для будь-яких даних, навіть текстових, можна визначити "мінімальні" і "максимальні" допустимі значення.

4.1.5. Тестування операцій порівняння чисел

Розбиття на класи еквівалентності широко використовується при тестуванні коректності реалізації арифметичних операцій і операцій порівняння. Кожну операцію можна розглядати як блок з входами - значенням і виходом - результатом операції. Для її тестування виконується розбиття діапазону зміни змінних на входах блоку на класи еквівалентності і методом аналізу граничних значень цих змінних.

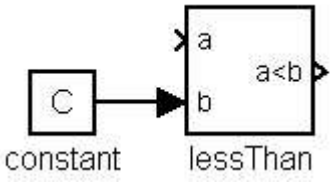
У таблиці 4.1 наведені тестові набори для блоків, що реалізують операції порівняння, у разі, коли на один з входів блоку подається константа.

Таблиця 4.1. Блоки порівняння і визначені для них тестові набори

greaterThan блок. Реалізує операцію порівняння **a> b (b** - константа, на вході **a** може бути змінна числового типу)	greaterEq блок. Реалізує операцію порівняння **a> = b (b** - константа, на вході **a** може бути змінна числового типу)
	

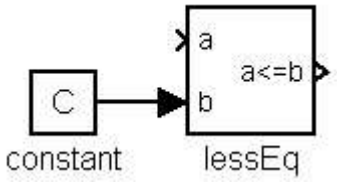
№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	b - d	b + d	b	min	max
Вихід	F	T	F	F	T

lessThan блок. Реалізує операцію порівняння **a <b (b** - константа, на вході **a** може бути змінна числового типу)



№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	b - d	b + d	b	min	max
Вихід	F	T	T	F	T

lessEq блок. Реалізує операцію порівняння **a <= b (b** - константа, на вході **a** може бути змінна числового типу)

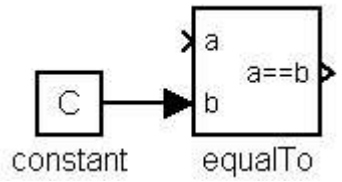


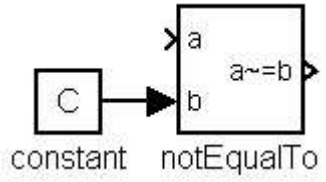
№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	b - d	b + d	b	min	max
Вихід	T	F	F	T	F

equalTo блок. Реалізує операцію порівняння **a = b (b** - константа, на вході **a** може бути змінна числового типу)

№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	b - d	b + d	b	min	max
Вихід	T	F	T	T	F

notEqualTo блок. Реалізує операцію порівняння **a** $\neq$ **b (b** - константа, на вході **a** може бути змінна числового типу)





№ набору	**1**	**2**	**3**	**4**	**№ набору**	**1**	**2**	**3**	**4**
Вхід a	$\neq$ b	b	min	max	**Вхід a**	$\neq$ b	b	min	max
Вихід	F	T	F	F	**Вихід**	T	F	T	T

* Тестовий набір реалізуємо, тільки якщо змінна на вході a - змінна цілого типу

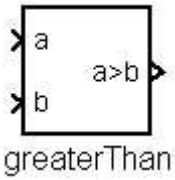
У наведених тестових наборах використовуються наступні позначення:

- d - крок зміни (resolution) змінної на вході a. Якщо змінна на вході a - змінна цілого типу, то d одно 1;
- min - мінімальне значення змінної на вході a;
- max - максимальне значення змінної на вході a.

У таблиці 4.2 наведені тестові набори для блоків, що реалізують операції порівняння, у разі, коли на обидва входи блоку подаються змінні.

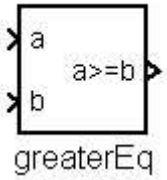
Таблиця 4.2. Блоки порівняння і визначені для них тестові набори (продовження)

greaterThan блок. Реалізує операцію порівняння **a> b (a, b** - змінні числового типу)



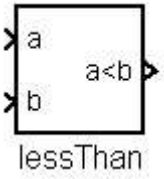
№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	val	val	val	min	max
Вхід b	val + d2	val - d2	val	max	min
Вихід	F	T	F	F	T

greaterEq блок. Реалізує операцію порівняння **a> = b (a, b** - змінні числового типу)

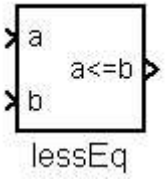


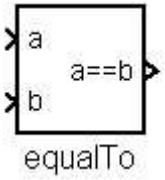
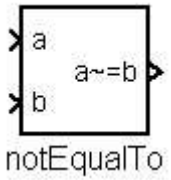
№ набору	**1**	**2**	**3 ***	**4**	**5**
Вхід a	val	val	val	min	max
Вхід b	val + d2	val - d2	val	max	min
Вихід	F	T	T	F	T

lessThan блок. Реалізує операцію порівняння **a <b (a, b** - змінні числового типу)



lessEq блок. Реалізує операцію порівняння **a <= b (a, b** - змінні числового типу)



№ набору	**1**	**2**	**3 ***	**4**	**5**	**№ набору**	**1**	**2**	**3 ***	**4**	**5**
Вхід a	val	val	val	min	max	**Вхід a**	val	val	val	min	max
Вхід b	val + d2	val - d2	val	max	min	**Вхід b**	val + d2	val - d2	val	max	min
Вихід	T	F	F	T	F	**Вихід**	T	F	T	T	F
equalTo блок. Реалізує операцію порівняння **a = b (a, b** - змінні будь-якого типу)  						notEqualTo блок. Реалізує операцію порівняння **a ≠ b (a, b** - змінні будь-якого типу)  					
№ набору	**1**	**2**	**3**	**4**		**№ набору**	**1**	**2**	**3**	**4**	
Вхід a	val1	val	min	max		**Вхід a**	val1	val	min	max	
Вхід b	val2	val	max	min		**Вхід b**	val2	val	max	min	

Вихід	F	T	F	F	**Вихід**	T	F	T	T

* Тестовий набір реалізуємо, тільки якщо змінні на входах блоку - змінні цілого типу

У наведених тестових наборах використовуються наступні позначення:

- d2 - крок зміни (resolution) змінної на вході b. Якщо змінна на вході b - мінлива цілого типу, то d2 одно 1;
- val, val1, val2 - значення, які взяті з середини діапазону, отриманого при перетині діапазонів змінних на входах a і b;
- min - мінімальне значення змінної на вході блоку;
- max - максимальне значення змінної на вході блоку.

Тестування програмного коду (покриття)

Анотація: Лекція завершує тему "тестування програмного коду". У лекції розглянуті питання об'єднання тестових прикладів у тест-плани; визначаються методи оцінки результатів виконання тестів. Значна частина лекції присвячена покриттю програмного коду тестами, різним типам покриття, методам його аналізу. Мета даної лекції: дати знання, необхідні для складання тест-планів та проведення аналізу покриття програмного коду тестами

Ключові слова: статистика , файл , branch coverage , coverage , decision coverage , NAND

6.1. Тест-плани

Тестові приклади, що розглядаються в попередніх розділах, не існують самі по собі - кожен тестовий приклад перевіряє одну ситуацію в роботі системи, але вся сукупність тестових прикладів повинна повністю перевіряти всю функціональність системи. У зв'язку з цим опису тестових прикладів об'єднують в документи, званими тест-планами.

Тест-план являє собою документ, в якому перераховані або всі тестові приклади, необхідні для тестування системи, або частина тестових прикладів, об'єднаних за певною ознакою.

Тест-план може бути написаний на природному або формальній мові; в останньому випадку можлива передача тест-плану на вхід тестового оточення для автоматичного виконання визначених у тест-плані тестових прикладів.

Існує кілька причин для об'єднання описів тестових прикладів в єдиний документ або декілька документів.

- **Єдина схема ідентифікації та трасування тестових прикладів**

Оскільки тестові приклади пишуться на підставі функціональних чи тест-вимог, при тестуванні необхідно упевнитися, що для кожного вимоги існує хоча б один тестовий приклад. Це досягається введенням єдиної схеми ідентифікації тестових прикладів (наприклад - наскрізної нумерації) і введенням посилань на вимоги, на основі яких тестовий приклад написаний.

- **Об'єднання тестових прикладів в смислові групи**

Тестові приклади, призначені для перевірки одних і тих же модулів системи, раціонально поєднувати в смислові групи. Причина в тому, що у таких прикладів, як правило, дуже схожі вхідні дані і сценарії, а групування дозволяє виявляти помилки і помилки в тестах.

- **Внесення змін до тестові приклади**

При зміні тестованої системи в ході її життєвого циклу неминуче доводиться змінювати тестові приклади. Загальні огляди тест-вимог і тест-планів дозволяють виявити, які тести повинні бути змінені або видалені, а в яких смислових групах необхідно створення нових тестових прикладів, перевіряючих нову функціональність.

- **Визначення послідовності тестування**

Одне з важливих властивостей тестового прикладу - його незалежність. Це означає, що результат виконання тестового прикладу не повинен змінюватися в залежності від того, які тести виконувалися до нього. Як правило, незалежність тестових прикладів досягається повної реініціалізації тестового оточення перед виконанням кожного нового тестового прикладу. Однак, часто виникають ситуації, в яких, для економії часу виконання, тести об'єднуються в послідовності, де кожен наступний тестовий приклад використовує стан тестового оточення або тестованої системи, досягнуте під час попереднього тесту. Такі пов'язані тестові приклади повинні бути окремо помічені для того, щоб зберегти коректний порядок їх проходження.

6.1.1. Типова структура тест-плану

Розглянемо типову структуру тест-плану, написаного на природній мові і містить тестові приклади для перевірки роботи модуля розрахунку контрольних сум.

Кожен тестовий приклад у цьому тест-плані має унікальний номер і посилання на тест-вимогу, на основі якого він написаний.

Загальний опис тесту допомагає при супроводі тест-планів - внесення змін при зміні системи, інспекціях тест-планів, які виявлятимуть неузгодженість і т.п.

Також у кожному тестовому прикладі обов'язково перераховані всі вхідні значення та очікувані вихідні значення, а також сценарій, що описує послідовність дій, які необхідно виконати тестового оточенню для виконання тестового прикладу.

Тест-план

Тестовий приклад 1

Номер тест-вимоги: 2a, 2b

Опис тесту: У даному тесті перевіряється правильність

обчислення значення контрольної суми (поля CRC)

при Непорожнє значенні поля CRC і нульових значеннях елементів запису.

Вхідні дані: CRC = 12345, A = 0, B = 0, C = 0, D = 0

Очікувані вихідні дані: CRC = 0, A = 0, B = 0, C = 0, D = 0, Empty = TRUE

Сценарій тіста:

1. Установка значення поля CRC в 12 345
2. Установка значень полів AF в 0
3. Виклик функції Set_CRC
4. Перевірка значень CRC на 0 і Empty на TRUE

Тестовий приклад 2

Номер тест-вимоги: 2a

Опис тесту: У даному тесті перевіряється відповідність

алгоритму обчислення поля CRC, заданому в специфікації вимог.

Вхідні дані: CRC = 0, AD заповнені байтами 01010101b

Очікувані вихідні дані: CRC = 0111100b, Empty = FALSE

Сценарій тіста:

1. Установка значення поля CRC в 0
2. Заповнення байт полів AD байтами 01010101b
3. Виклик функції Set_CRC
4. Перевірка значень CRC на 0111100b і Empty на FALSE

Тестовий приклад 3

Номер тест-вимоги: 2a

Опис тесту: У даному тесті перевіряється незмінність

полів AF запису при обчисленні поля CRC (підрахунку контрольної суми).

Вхідні дані: CRC = 0, AD заповнені байтами 01010101b

Очікувані вихідні дані: AD заповнені байтами 01010101b,

Сценарій тіста:

1. Установка значення поля CRC в 0
2. Заповнення байт полів AD байтами 01010101b
3. Виклик функції Set_CRC
4. Перевірка значень байт полів AD на 01010101b

Така структура тест-плану дозволяє описувати тестові приклади з абсолютно різними наборами вхідних і вихідних даних і сценаріями, однак при великій кількості тестових прикладів ця схема стане дуже громіздкою. Пізніше будуть розглянуті табличні форми подання тест-планів, що дозволяють записувати їх більш компактно.

6.2. Оцінка якості тестованого коду - статистика виконання тестів

В результаті виконання кожного тестового прикладу тестове оточення порівнює очікувані і реальні вихідні значення. У разі, якщо ці значення збігаються, тест вважається пройденим, т.к. система видала саме ті вихідні значення, які очікувалися; в іншому випадку тест вважається не пройденим.

Кожен непройденний тест потенційно вказує на потенційний дефект в тестованої системі, а загальна їх кількість дозволяє оцінювати якість тестованого програмного коду і обсяг змін, які необхідно в нього внести для усунення дефектів.

Для побудови такої інтегральної оцінки після виконання всіх тестових прикладів тестовим оточенням збирається *статистика*виконання, яка, як правило, записується в *файл* звіту про виконання тестів. Існує кілька ступенів подробиці статистики виконання тестів:

- Висновок кількості пройдених і кількості не пройдених тестових прикладів, а також їх загальної кількості.

Наприклад,

180 test cases passed

20 test cases failed

200 test cases total

- 1 + висновок ідентифікаторів не пройшли тестових прикладів. Дозволяє локалізувати тестові приклади, потенційно котрі виявили дефект.

Наприклад,

Invoking test case 1 ... Passed

Invoking test case 2 ... Failed

Invoking test case 3 ... Failed

<...>

Invoking test case 200 ... Passed

Final stats:

180 test cases passed

20 test cases failed

200 test cases total

- 2 + висновок не збіглися очікуваних і реальних вихідних даних. Дозволяє проводити більш глибокий аналіз причин неуспішного проходження тестового прикладу.

Наприклад,

Invoking test case 1 ... Passed

Invoking test case 2 ... Failed

Expected values: Actual values:

A = 200 A = 0

B = 450 B = 0

Message = "Submenu 1" Message = ""

Invoking test case 3 ... Failed

Expected values: Actual values:

A = 0 A = 200

B = 0 B = 300

Message = "" Message = "Main Menu"

<...>

Invoking test case 200 ... Passed

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

- 2 + виведення всіх очікуваних і реальних вихідних даних. Варіант попереднього пункту.

Наприклад,

Invoking test case 1 ... Passed

Invoking test case 2 ... Failed

Expected values: Actual values:

A = 200 A = 0 FAIL

B = 450 B = 0 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = "Submenu 1" Message = "" FAIL

Invoking test case 3 ... Failed

Expected values: Actual values:

A = 0 A = 200 FAIL

B = 0 B = 300 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = "" Message = "Main Menu" FAIL

<...>

Invoking test case 200 ... Passed

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

- Повне виведення очікуваних і реальних вихідних даних з відмітками про збіг інеспівпаданні та відмітками про успішне / неуспішному завершенні для кожного тестового прикладу.

Наприклад,

Invoking test case 1 ... Passed

A = 0 A = 0 P

B = 0 B = 0 P

C = 500 C = 500 P

D = 600 D = 600 P

Message = "" Message = "" P

Invoking test case 2 ... Failed

Expected values: Actual values:

A = 200 A = 0 FAIL

B = 450 B = 0 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = "Submenu 1" Message = "" FAIL

Invoking test case 3 ... Failed

Expected values: Actual values:

A = 0 A = 200 FAIL

B = 0 B = 300 FAIL

C = 500 C = 500 P

D = 600 D = 600 P

Message = "" Message = "Main Menu" FAIL

<...>

Invoking test case 200 ... Passed

Message = "Submenu 1" Message = "Submenu 1 P

Prompt = ">" Prompt = ">" P

Final Stats

180 test cases passed

20 test cases failed

200 test cases total

Більш детально різні формати звітів про тестування будуть розглянуті пізніше, а поки зупинимося більш детально на важливому критерії оцінки якості системи тестів і ступеня повноти тестування системи - рівні покриття програмного коду тестами.

6.3. Покриття програмного коду

6.3.1. Поняття покриття

Одна з оцінок якості системи тестів - це її повнота, тобто величина тієї частини функціональності системи, яка перевіряється тестовими прикладами. Зазвичай за міру повноти беруть відношення обсягу перевіреної частини системи до її об'єму в цілому. Повна система тестів дозволяє стверджувати, що система реалізує всю функціональність, зазначену у вимогах, і, що ще більш важливо, - не реалізує жодної іншої функціональності.

Один з часто використовуваних методів визначення повноти системи тестів є визначення відношення кількості тест-вимог, для яких існують тестові приклади, до загальної кількості тест-вимог. Тобто в даному випадку мова йде про покриття тестовими прикладами тест-вимог. В якості одиниці вимірювання ступеня покриття тут виступає відсоток тест-вимог, для яких існують тестові приклади, званий відсотком покритих тест-вимог.

Покриття вимог дозволяє оцінити ступінь повноти системи тестів по відношенню до функціональності системи, але не дозволяє оцінити повноту стосовно її програмної реалізації. Одна і та ж функція може бути реалізована за допомогою зовсім різних алгоритмів, що вимагають різного підходу до організації тестування.

Для більш детальної оцінки повноти системи тестів при тестуванні скляного ящика аналізується *покриття програмного коду,* зване також *структурним покриттям.*

Під час роботи кожного тестового прикладу виконується деякий ділянку програмного коду системи; при виконанні всієї системи тестів виконуються всі ділянки програмного коду, які задіє ця система тестів. У разі, якщо існують ділянки програмного коду, що не виконані при виконанні системи тестів, система тестів потенційно неповна (тобто не перевіряє всю функціональність системи), або система містить ділянки захисного коду або не використовуваний код (наприклад, "закладки" або заділ на майбутнє використання системи). Таким чином, відсутність покриття яких ділянок коду є сигналом до переробки тестів або коду (а іноді - і вимог).

До аналізу покриття програмного коду можна приступати тільки після повного покриття вимог. Повне покриття програмного коду не гарантує того, що тести перевіряють всі вимоги до системи. Одна з типових помилок починаючого тестувальника - починати з покриття коду, забуваючи про покриття вимог.

6.3.2. Рівні покриття

6.3.3. За рядками програмного коду (Statement Coverage)

Для забезпечення повного покриття програмного коду на даному рівні необхідно, щоб в результаті виконання тестів кожен оператор був виконаний хоча б один раз.

Особливість даного рівня покриття полягає в тому, що на ньому утруднений аналіз покриття деяких керуючих структур.

Наприклад, для повного покриття всіх рядків наступного ділянки програмного коду мовою С достатньо одного тестового прикладу:

Вхід: condition = true; Очікуваний вихід: * p = 123.

```
int * p = NULL;
if (condition)
    p = &variable;
* P = 123;
```

Навіть якщо до складу тестів не буде входити тестовий приклад, що перевіряє роботу фрагмента при значенні condition = false, код буде покритий. Однак, у випадку condition = false виконання фрагмента викличе помилку.

Аналогічні проблеми виникають при перевірці циклів do ... while - при даному рівні покриття досить виконання циклу тільки один раз, при цьому метод абсолютно нечутливий до логічним операторам | | і &&

Іншою особливістю даного методу є залежність рівня покриття від структури програмного коду. На практиці часто не потрібно 100% покриття програмного коду, замість цього встановлюється допустимий рівень покриття, наприклад 75%. Проблеми можуть виникнути при покритті наступного фрагмента програмного коду:

```
if (condition)
```

```
   functionA ();
else
   functionB ();
```

Якщо functionA () містить 99 операторів, а functionB () - один оператор, то єдиного тестового прикладу, встановлюєcondition в true, буде достатньо для досягнення необхідного рівня покриття. При цьому аналогічний тестовий приклад, який встановлює значення condition в false, дасть занадто низький рівень покриття.

6.3.3.1. По гілках умовних операторів (Decision Coverage)

Для забезпечення повного покриття по даному методу кожна точка входу і виходу в програмі і в усіх її функціях повинна бути виконана принаймні один раз, і всі логічні вирази в програмі повинні прийняти кожне з можливих значень хоча б один раз, - таким чином, для покриття по гілках потрібно як мінімум два тестових прикладу.

Також даний метод називають: *branch coverage,* all-edges *coverage,* basis path *coverage,* DC, C2, decision-decision-path.

На відміну від попереднього рівня покриття даний метод враховує покриття умовних операторів з порожніми гілками. Так, для покриття по гілках ділянки програмного коду

```
 a = 0;
if (condition) {
  a = 1;
```

```
}
```

необхідні два тестових прикладу:

1. Вхід: condition = true; Очікуваний вихід: a = 1;

2. Вхід: condition = false; Очікуваний вихід: a = 0;

Особливість даного рівня покриття полягає в тому, що на ньому не враховуються логічні вирази, значення компонент яких виходять викликом функцій. Наприклад, на наступному фрагменті програмного коду

```
if (condition1 && (condition2 | | function1 ()))
   statement1;
else
   statement2;
```

повне покриття по гілках може бути досягнуто за допомогою двох тестових прикладів:

1. Вхід: condition1 = true, condition2 = true

2. Вхід: condition1 = false, condition2 = true / false (будь-яке значення)

В обох випадках не відбувається виклику функції function1 (), хоча покриття даної ділянки коду буде повним. Для перевірки виклику функції function1 () необхідно додати ще один тестовий приклад (який, проте, не покращує ступеня покриття по гілках):

3. Вхід: condition1 = true, condition2 = false.

6.3.3.2. По компонентах логічних умов

Для більш повного аналізу компонент умов в логічних операторах існує кілька методів, що враховують структуру компонент умов і значення, які вони приймають при виконанні тестових прикладів.

6.3.3.3. Покриття за умовами (Condition Coverage)

Для забезпечення повного покриття по даному методу кожна компонента логічного умови в результаті виконання тестових прикладів повинна приймати всі можливі значення, але при цьому не потрібно, щоб саме логічне умова брало всі можливі значення.Так, наприклад, при тестуванні наступного фрагмента:

```
if (condition1 | condition2)
 functionA ();
else
 functionB ();
```

для покриття за умовами потрібно два тестових прикладу:

(1) Вхід: condition1 = true, condition2 = false

(2) Вхід: condition1 = false, condition1 = true.

При цьому значення логічного умови прийматиме значення тільки true, таким чином, при повному покритті за умовами не досягатиме покриття по гілках.

6.3.3.4. Покриття по гілках / умовам (Condition / Decision Coverage)

Даний метод поєднує вимоги попередніх двох методів - для забезпечення повного покриття необхідно, щоб як логічне умова, так і кожна його компонента прийняла всі можливі значення.

Для покриття розглянутого вище фрагмента з умовою condition1 | condition2 потрібно 2 тестових прикладу:

1. Вхід: condition1 = true, condition2 = true
2. Вхід: condition1 = false, condition1 = false.

Однак, ці два тестових прикладу не дозволять протестувати правильність логічної функції - замість OR в програмному коді могла бути помилково записана операція AND.

6.3.3.5. Покриття за всіма умовами (Multiple Condition Coverage)

Для виявлення невірно заданих логічних функцій був запропонований метод покриття за всіма умовами. При цьому методі покриття повинні бути перевірені всі можливі набори значень компонент логічних умов. Тобто у разі n компонент потрібно 2^n тестових прикладів, кожен з яких перевіряє один набір значень, Тести, необхідні для повного покриття по даному методу, дають повну таблицю істинності для логічного виразу.

Незважаючи на очевидну повноту системи тестів, що забезпечує цей рівень покриття, даний метод рідко застосовується на практиці у зв'язку з його складністю і надмірністю.

Ще одним недоліком методу є залежність кількості тестових прикладів від структури логічного виразу. Так, для умов, що містять однакову кількість компонент і логічних операцій:

a && b && (c | | (d && e))

((A | | b) && (c | | d)) && e

потрібно різну кількість тестових прикладів. Для першого випадку для повного покриття потрібно 6 тестів, для другого - 11.

6.3.4. Метод MC / DC для зменшення кількості тестових прикладів при 3-му рівні покриття коду

Для зменшення кількості тестових прикладів при тестуванні логічних умов фірмою Boeing був розроблений модифікований метод покриття по гілках / умовам (Modified Condition / *Decision Coverage* або MC / DC) [25 , 26]. Даний метод широко використовується при верифікації бортового авіаційного програмного забезпечення згідно процесам стандарту DO-178B [7].

Для забезпечення повного покриття за цим методом необхідно виконання наступних умов:

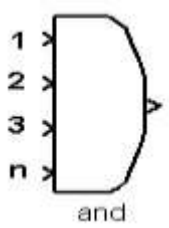
- кожне логічне умова повинна вживати всіх можливих значення;
- кожна компонента логічного умови повинна хоча б один раз приймати всі можливі значення;
- має бути показано незалежне вплив кожної з компонент на значення логічного умови, тобто вплив при фіксованих значеннях інших компонент.

Покриття з цієї метриці вимагає досить великої кількості тестів для того, щоб перевірити кожне умова, яка може вплинути на результат вираження, однак ця кількість значно менше, ніж вимагається для методу покриття за всіма умовами. У таблиці

6.1наведені приклади тестових наборів, необхідних для тестування логічних блоків по MC / DC. Так, наприклад, для блоку OR доситьn +1 тестових прикладів, де n - кількість входів логічного блоку. Перший тестовий приклад показує, що при нульових значеннях входів значення виходу також нульове. У кожному з наступних n прикладів значення кожного входу встановлюється в 1, ніж показується незалежне вплив входів на значення виходу.

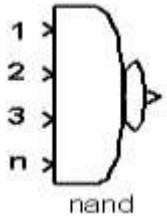
Таблиця 6.1. Логічні блоки і певні для них тестові набори

AND блок. Реалізує логічну функцію **I** для двох або більше входів



№ набору	**1**	**2**	**3**	**4**	· · ·	**n + 1**
Вхід 1	T	F	T	T	· · ·	T
Вхід 2	T	T	F	T	· · ·	T
Вхід 3	T	T	T	F	· · ·	T

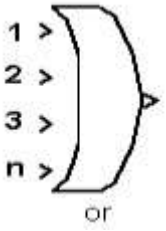
NAND блок. Реалізує логічну функцію **I-НЕ** для двох або більше входів



№ набору	**1**	**2**	**3**	**4**	· · ·	**n + 1**
Вхід 1	T	F	T	T	· · ·	T
Вхід 2	T	T	F	T	· · ·	T
Вхід 3	T	T	T	F	· · ·	T

. . .	. . .	. . .	. . .	. . .	. . .	. . .
Вхід n	T	T	T	T	· · ·	F
Вихід	T	F	F	F	· · ·	F

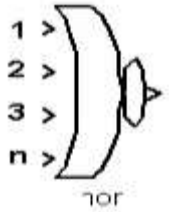
OR блок. Реалізує логічну функцію **АБО** для двох або більше входів



№ набору	**1**	**2**	**3**	**4**	· · ·	**n + 1**
Вхід 1	F	T	F	F	· · ·	F
Вхід 2	F	F	T	F	· · ·	F
Вхід 3	F	F	F	T	· · ·	F
. . .	. . .	. . .	. . .	. . .	. . .	. . .

. . .	. . .	. . .	. . .	. . .	. . .	. . .
Вхід n	T	T	T	T	· · ·	F
Вихід	F	T	T	T	· · ·	T

NOR блок. Реалізує логічну функцію **АБО - НЕ** для двох або більше входів



№ набору	**1**	**2**	**3**	**4**	· · ·	**n + 1**
Вхід 1	F	T	F	F	· · ·	F
Вхід 2	F	F	T	F	· · ·	F
Вхід 3	F	F	F	T	· · ·	F
. . .	. . .	. . .	. . .	. . .	. . .	. . .

Вхід n	F	F	F	F	· · ·	T	**Вхід n**	F	F	F	F	· · ·	T
Вихід	F	T	T	T	· · ·	T	**Вихід**	T	F	F	F	· · ·	F

6.3.5. Аналіз покриття

Метою аналізу повноти покриття коду є виявлення ділянок коду, які не виконуються при виконанні тестових прикладів. Тестові приклади, засновані на вимогах, можуть не забезпечувати повного виконання всієї структури коду. Тому для поліпшення покриття проводиться аналіз повноти покриття коду тестами, і при необхідності проводяться додаткові перевірки, спрямовані на з'ясування причини недостатнього покриття і визначення необхідних дій щодо його усунення. Зазвичай аналіз покриття виконується з урахуванням наступних угод:

- аналіз повинен підтвердити, що повнота покриття тестами структури коду відповідає необхідному виду покриття і заданому мінімально допустимому відсотку покриття;
- аналіз повноти покриття тестами структури коду може бути виконаний з використанням вихідного тексту, якщо програмне забезпечення не відноситься до рівня А. Для рівня А необхідно перевірити об'єктний код, згенерований компілятором, і з'ясувати, трасується він у вихідний текст чи ні. Якщо об'єктний коду не трасується у вихідний текст, мають бути проведені повірки об'єктного коду на предмет правильності генерації послідовності команд. Прикладом об'єктного коду, який безпосередньо не трасується у вихідний текст, але генерується компілятором, може бути перевірка виходу за задані межі масиву;
- аналіз повинен підтвердити правильність передачі даних і управління між компонентами коду.

Аналіз повноти покриття тестами може виявити частину вихідного коду, що не виконувалася в ході тестування. Для дозволу цієї обставини можуть знадобитися додаткові дії в процесі перевірки програмного забезпечення. Ця неісполняемимі частина коду може бути результатом:

- недоліків у формуванні тестових прикладів або тестових процедур, заснованих на вимогах. У цьому випадку повинен бути доповнений набір тестових прикладів або змінені тестові процедури для забезпечення покриття упущеної частини коду. При цьому може знадобитися перегляд методу (методів), що використовується для проведення аналізу повноти тестів на основі вимог;
- неадекватності у вимогах на програмне забезпечення: У цьому випадку повинні бути модифіковані вимоги на програмне забезпечення, розроблені і виконані додаткові тестові приклади і тестові процедури;
- "Мертвого" коду. Цей код повинен бути вилучений, і необхідно провести аналіз для оцінки ефекту видалення і необхідності повторної перевірки;
- дезактивируемие коду. Для дезактивируемие коду, яка не передбачається до виконання в кожній конфігурації, поєднання аналізу і тестів має продемонструвати можливості засобів, якими ненавмисне виконання такого коду запобігається, ізолюється або усувається. Для дезактивируемие коду, який виконується тільки за певних конфігураціях, повинна бути встановлена нормальна експлуатаційна конфігурація для виконання цього коду, і для неї повинні бути розроблені додаткові тестові приклади і тестові процедури, що задовольняють цілям повноти покриття тестами структури коду;

- надмірності умови. Логіка роботи такої умови має бути переглянута. Наприклад, в умові if (A && B | |! B) принципово неможливо перевірити, що частина умови A && B дорівнюватиме False у разі, коли A = True і B = False, тому що друга частина умови (! B) дорівнюватиме True, і загальний результат логічного виразу буде True;
- захисного коду. Ця частина коду використовується для запобігання виняткових ситуацій, які можуть виникнути в процесі роботи програми. Як приклад, це може бути гілка default в операторі вибору switch, причому вхідна умова оператора switchможе приймати певні значення, які він описує, і, як наслідок, гілка default ніколи не буде виконана.

Повторюваність тестування

Анотація: Лекція присвячена питанням забезпечення повторюваності тестування в промисловому оточенні. Визначаються завдання і цілі забезпечення повторюваності, розглядаються процеси настройки тестового оточення, оптимізації послідовності виконання тестових прикладів. Визначаються проблеми, що виникають при наявності залежностей між тестовими прикладами. Мета даної лекції: дати уявлення про підготовку тестів в промислових середовищах

Ключові слова: активний , циклу , регресійне тестування , вхідні дані , пам'ять , інформація , мінімум , пункт , час виконання ,значення , група , розбиття , стан системи , коректність

8.1. Завдання і цілі забезпечення повторюваності тестування при промисловій розробці програмного забезпечення

Як вже було сказано в попередніх лекціях, тестування програмної системи - не разовий захід, а постійний процес, *активний*протягом усього життєвого *циклу* розробки системи. Протягом цього процесу система неминуче змінюється - або в результаті виправлення помилок, або в результаті розширення її функціональності. Завдання тестувальника в такій

ситуації - підтвердити, що нова або виправлена функціональність не викликало нові помилки, а якщо помилки все-таки виникли - визначити причини їх виникнення.

Найпростіший, але в той же час дієвий спосіб такого підтвердження - повне виконання всіх тестових прикладів після кожного істотної зміни системи і порівняння результатів виконання тестів до і після зміни.

Якщо результати виконання тестів до внесення змін були позитивними (всі тести проходили успішно), то поява неуспішно пройдених тестів може означати, що в системі з'явилися нові дефекти, викликані виправленням старих.

У загальному випадку повторне виконання тестів може завершитися одним із трьох способів.

1. Всі тести пройдені успішно. У цьому випадку зміни не зачіпають вже протестовані функції, але може знадобитися розробка нових тестових прикладів для нових функцій системи.
2. Частина тестів, раніше виконувалися успішно, завершується з негативним результатом. Причини цього можуть бути наступні:
    - коректне зміна функціональності тестованої системи, в результаті якого тестовий приклад перестав відповідати вимогам;
    - некоректна зміна функціональності системи, в результаті якого тестовий приклад виявив розбіжність з вимогами;
    - вплив залишкових даних від попередніх тестових прикладів, раніше залишалося непоміченим.

Перші дві причини помітні тільки за допомогою аналізу змін у функціональних вимогах і тест-вимогами, а також поточного стану тест-планів та тестового оточення. За результатами цього аналізу в першому випадку тестувальник вносить зміни в

тестовий приклад (і, можливо, розробляються нові тестові приклади), у другому випадку тестувальник повідомляє розробників про наявність дефекту.

3. Виконання тестів аварійно завершується на самому початку або при виконанні певного тестового прикладу.

Дана проблема найчастіше пов'язана зі зміною зовнішнього оточення тестируемой частини системи, яке моделює тестове оточення. Через такі зміни можуть мінятися зовнішні інтерфейси, а також склад і формат вхідних і вихідних даних. В результаті тестове оточення перестає забезпечувати необхідну для виконання тестів інфраструктуру і виникає збій процесу тестування.Наприклад, такий збій може виникнути в тестовому оточенні при спробі обробити дані, що видаються системою в новому форматі.

Якщо для виконання тестів потрібно збірка програмних модулів тестового оточення і тестованої системи в єдиний виконуваний код, то при зміні інтерфейсів системи може виникнути ситуація, коли неможливо не тільки виконання тестів, а навіть збірка оточення і системи. У цьому випадку також необхідно провести аналіз змін внесених в систему і модифікувати відповідно до них тестове оточення.

У деяких випадках повторне виконання всіх тестів неможливо - наприклад, коли потрібно тривалий час для виконання всіх тестів, а час, відведений на процес тестування, обмежена. У цьому випадку часто застосовується практика вибіркового тестування окремих частин системи, порушених змінами. Повне тестування при такому підході проводиться тільки після накопичення досить великої кількості змін або на ключових стадіях проекту.

Процес, що включає в себе повторне виконання тестів, називають *регресійним тестуванням. Регресійне тестування* включає в себе наступні стадії:

1. Аналіз змін в системі
2. Вибір тестових прикладів для перевірки системи
3. Виконання тестових прикладів
4. Аналіз результатів виконання
5. Модифікація тестового оточення, тестових прикладів або повідомлення розробників про дефект системи

Таким чином, можна визначити такі основні завдання повторюваності тестування при внесенні змін:

- забезпечення можливості повного виконання всіх тестів, перевіряючих функціональність системи, або проведення аналізу, що дозволяє виявити тести, які повинні бути повторно виконані для тестування змінилася функціональності;
- розробка тестових прикладів і тестового оточення з використанням методик, що полегшують модифікацію при змінах в тестованої системі;
- розробка тестових прикладів, структура яких повністю виключає їх взаємний вплив по залишковим даними.

Наслідком повторюваності тестування є постійне забезпечення тестувальників і розробників актуальною інформацією про поточний стан системи і коректності змін, внесених в ході розробки системи.

8.2. Передумови для виконання тесту, настройка тестового оточення, оптимізація послідовностей тестових прикладів

Як вже було сказано раніше, *вхідні дані* в кожному тестовому прикладі явно задають початковий стан тестованої системи і режими її роботи при виконанні тестового сценарію.

Однак неявне вплив на виконання тесту надає і стан тестового оточення. Під станом тут розуміється набір параметрів, зміна будь-якого з яких може вплинути або на результат виконання тестового прикладу, або на можливість його коректної роботи і завершення.

Наприклад, для виконання тестового прикладу тестируемой системі може знадобитися значний обсяг дискової або оперативної пам'яті. Якщо перед виконанням тесту тестове оточення зарезервує цю *пам'ять* під свої потреби, виконання тесту виявиться неможливим. Та ж сама ситуація може виникнути і в разі, якщо оточення не звільнить *пам'ять* після виконання попереднього тестового прикладу.

Ця *інформація* зазвичай відсутня в тест-планах, проте необхідну для виконання тестів стан тестового оточення необхідно враховувати при розробці тестових прикладів.

Доброю практикою є оформлення перевірок на допустимість стану тестового оточення у вигляді передумов для виконання тесту. Це дозволяє діагностувати ситуації, що виникають при вибірковому тестуванні, які призводять до відмов тестового оточення.

Наприклад, розглянемо програмну систему, яка може стартувати двома різними способами - з налаштуваннями за замовчуванням після включення (режим FACTORY_SETTINGS) і з останніми збереженими налаштуваннями після перезавантаження (режим COLD_START). При цьому при старті в режимі FACTORY_SETTINGS значення за замовчуванням присвоюються всім налаштувань системи, а після перезавантаження (режим COLD_START) всі налаштування залишаються в значеннях, встановлених безпосередньо перед перезавантаженням.

Для перевірки таких вимог:

1. Перевірити, що після включення системи настройки встановлюються в значення за замовчуванням.

2. Перевірити, що після перезавантаження системи настройки встановлюються останнім збережене значення.

необхідні як *мінімум* три тестових прикладу з наступними сценаріями:

Тестовий приклад 1

1. Включити систему в режимі FACTORY_SETTINGS.

2. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

Тестовий приклад 2

1. Включити систему в режимі FACTORY_SETTINGS.

2. Перезавантажити систему (викликати її старт в режимі COLD_START).

3. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

Тестовий приклад 3 Розділ 1. Включити систему в режимі FACTORY_SETTINGS. 2. Змінити значення налаштувань системи (у реальному тест-плані тут повинні бути встановлені конкретні значення змінних). 3. Перезавантажити систему (викликати її старт в режимі COLD_START). 4. Перевірити, що настройки мають останні введені значення (у реальному тест-плані тут мають бути перевірки конкретних значень змінних).

Перший *пункт* сценарію у всіх трьох тестових прикладах однаковий. Якщо при цьому початковий старт системи в режимі FACTORY_SETTINGS займає значний час, то сумарне *час виконання* трьох тестових прикладів буде ще більше. Якщо загальна кількість подібних тестових прикладів досить велике (десятки і сотні), то при такому виконанні тестів буде нераціонально витрачатися час на виконання тестових прикладів - час на ініціалізацію системи в кожному тестовому прикладі перевищуватиме сумарне *час виконання* "корисних" етапів сценаріїв тестових прикладів.

Для економії часу можна ініціалізувати систему в режимі FACTORY_SETTINGS тільки в першому тестовому прикладі. Другий і третій тестовий приклади почнуть свою роботу з розрахунку, що система вже була включена в режимі FACTORY_SETTINGS і всі значення налаштувань вже встановлені в деякі значення. Сценарії тестових прикладів при цьому будуть виглядати наступним чином:

Тестовий приклад 1

1. Включити систему в режимі FACTORY_SETTINGS
2. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

Тестовий приклад 2

1. Перезавантажити систему (викликати її старт в режимі COLD_START)
2. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

Тестовий приклад 3

1. Змінити значення налаштувань системи
2. Перезавантажити систему (викликати її старт в режимі COLD_START)
3. Перевірити, що настройки мають останні введені значення

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

При такій структурі тестових прикладів важлива послідовність їх виконання. Перший тестовий приклад инициализирует тестовану систему і приводить її в необхідне початкове стан (запускає її в режимі FACTORY_SETTINGS), другий і третій приклади, вважаючи, що система вже инициализирована, перевіряють тільки її роботу при перезавантаженні.

У ході розробки системи вимоги і програмний код можуть змінитися таким чином, що при регресійному тестуванні може бути прийняте рішення про виконання тестів тільки для режиму COLD_START.

Якщо при цьому будуть виконуватися тільки тестові приклади 2 і 3, то коректне виконання сценарію стане неможливим: значення налаштувань системи не отримали значень за замовчуванням при старті системи, а сама система запускається в позаштатному режимі - перезавантажується не включена.

Щоб діагностувати такі ситуації, до складу предусловий тестових прикладів 2 і 3 необхідно включати перевірки того, що до моменту виконання тестового прикладу система знаходиться в необхідному стані. Перший тестовий приклад при цьому може виставляти деякий прапор (змінну в тестовому оточенні), встановлене *значення* якого буде сигналізувати про те, що система коректно стартувала

При наявності таких перевірок тестові приклади будуть виглядати наступним чином:

Початкові установки тестового оточення

Встановити значення прапора Флаг_Сістема_Стартовала = FALSE

Тестовий приклад 1

1. Включити систему в режимі FACTORY_SETTINGS
2. Встановити значення прапора Флаг_Сістема_Стартовала = TRUE
3. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки конкретних значень змінних).

Тестовий приклад 2

1. Перевірити, що прапор Флаг_Сістема_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення
2. Перезавантажити систему (викликати її старт в режимі COLD_START)
3. Перевірити, що настройки мають значення за замовчуванням

(У реальному тест-плані тут мають бути перевірки конкретних значень змінних).

Тестовий приклад 3

1. Перевірити, що прапор Флаг_Сістема_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення
2. Змінити значення налаштувань системи (у реальному тест-плані тут повинні бути встановлені конкретні значення змінних)
3. Перезавантажити систему (викликати її старт в режимі COLD_START)

4. Перевірити, що настройки мають останні введені значення

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

При такому підході для виконання тестових прикладів спочатку повинні бути зроблені початкові установки тестового оточення, після чого перед виконанням тестового прикладу 2 або 3 буде проведена перевірка стану тестованої системи.

Приклад може здатися дещо надуманим, проте, на практиці часто виникає ситуація в якій один за одним слід кілька десятків тестових прикладів, а при регресійному тестуванні потрібно виконати, скажімо, тестові приклади з номерами від 25 по 40. Перший тестовий приклад при цьому инициализирует систему, а решта працюють з вже стартувала системою. Якщо просто виконувати тестові приклади 25-40, то їх виконання виявиться неможливим - вони не инициализируют систему.Розумним виходом з цієї ситуації є виконання тестових прикладів 1, 25-40.

8.3. Залежність між тестовими прикладами, налаштування за замовчуванням для тестових прикладів і їх груп

Для полегшення проведення регресійного тестування (і тестування взагалі) тестові приклади часто розбивають на групи. Кожна*група* містить набір тестових прикладів, перевіряючих окрему локальну частину функціональності тестованої системи. Тестові приклади для часткового регресійного тестування можна відбирати відразу групами.

Тестові приклади з попереднього розділу можна розбити на дві групи:

Тестування старту системи: тестовий приклад 1

Тестування перезавантаження системи: тестові приклади 2-3

Розбиття тестових прикладів на групи зручно і з точки зору установки початкового стану тестового оточення для виконання тестів - так, перед виконанням групи тестів можна ініціалізувати значення змінних або *стан системи,* необхідне для виконання всієї групи.Наприклад, якщо система працює в двох режимах - нормальному та сервісному, то перед виконанням групи тестів для нормального режиму роботи системи встановлювати нормальний режим, а перед виконанням тестів для сервісного режиму - сервісний. Такі установки називаються настройками групи тестів за замовчуванням (group defaults, test group defaults).

Перед виконанням кожного тестового прикладу може знадобитися установка одних і тих же змінних в одні і ті ж значення. Для того, щоб не дублювати ці установки в описі кожного тестового прикладу, в тест-плані можна визначити настройки за замовчуванням для кожного тесту (test case defaults), наприклад, таким чином:

Початкові установки тестового оточення

Встановити значення прапора Флаг_Сістема_Стартовала = FALSE

Налаштування за замовчуванням для групи:

Встановити сервісний режим роботи системи

Налаштування за замовчуванням для тестового прикладу:

Обнулити значення вихідних змінних тестового оточення,

в якому зберігаються налаштування системи

Група 1: Тестування старту системи (режим FACTORY_SETTINGS)

Тестовий приклад 1

1. Включити систему в режимі FACTORY_SETTINGS
2. Встановити значення прапора Флаг_Сістема_Стартовала = TRUE
3. Перевірити, що настройки мають значення за замовчуванням (в реальному тест-плані тут мають бути перевірки конкретних значень змінних)

Група 2: Тестування перезавантаження системи (режим COLD_START)

Тестовий приклад 2

1. Перевірити, що прапор Флаг_Сістема_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення
2. Перезавантажити систему (викликати її старт в режимі COLD_START)
3. Перевірити, що настройки мають значення за замовчуванням (У реальному тест-плані тут мають бути перевірки конкретних значень змінних).

Тестовий приклад 3

1. Перевірити, що прапор Флаг_Сістема_Стартовала = TRUE, інакше

перервати тестування з видачею діагностичного повідомлення

2. Змінити значення налаштувань системи

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

3. Перезавантажити систему (викликати її старт в режимі COLD_START)

4. Перевірити, що настройки мають останні введені значення

(У реальному тест-плані тут мають бути перевірки

конкретних значень змінних).

Як видно з попереднього розділу, для полегшення проведення вибіркового регресійного тестування кожен тестовий приклад повинен бути повністю автономним - хід його виконання і, тим більше, результат не повинні залежати від попередніх тестових прикладів. Тим самим, при вибірковому тестуванні результат тестування не залежить від обраного набору тестових прикладів (тестового набору). Однак, на практиці створення автономних тестів часто неможливо з різних причин (як правило - через тривалого часу виконання таких тестів).

У разі, коли в наборі тестових прикладів тести не є автономними, говорять про *тестової залежності.* Тестова залежність буває двох видів - передбачена структурою тестових прикладів і паразитная.

Приклад передбаченої тестової залежності був розглянутий у попередньому розділі - *коректність* виконання тестів визначалася порядком їх виконання. Така тестова залежність вимагає документування та супроводження, як і самі описи тестових прикладів.Існує два види документування тестових залежностей:

- явне визначення допустимого порядку виконання тестових прикладів;
- визначення допустимого порядку виконання тестових прикладів за допомогою предусловий.

Перший спосіб зручний при порівняно невеликому загальній кількості тестових прикладів, а в разі розбиття на групи - при невеликому розмірі груп тестових прикладів. При другому способі *коректність* порядку виконання тестових прикладів визначається за допомогою перевірки того, що або тестована система, або тестове оточення знаходяться в необхідному стані для виконання тестового прикладу.

Паразитні тестові залежності зазвичай викликані некоректним складанням тест-плану. Виявляються вони, як і передбачені залежності, в тому, що один (або більше) тестових прикладів коректно працює тільки в тому випадку, якщо до нього були виконані інші тестові приклади, причому така залежність не є передбаченою тестувальником. Природа паразитної тестової залежності схожа з природою помилок використання неініціалізованих або залишкових даних у динамічній пам'яті при програмуванні.

ЛЕКЦІЯ №3

Документація, що супроводжує процес верифікації та тестування (тест-плани)

Анотація: Лекція продовжує тему документування процесу тестування, присвячена документації, створюваної в процесі тестування. Розглянуто тест-плани, можливі форми їх підготовки, звіти про проходження тестів і різноманітні форми їх підготовки.Мета даної лекції: визначити основні технологічні ланцюжки, в яких створюється і використовується тестова документація, дати уявлення про роль тест-планів та звітів про проходження тестів, визначити підходи до розробки й аналізу тест-планів

Ключові слова: Автоматизації тестування , Ручне тестування , data transfer , інтерпретатор таблиць , RDL , test log , incident ,summary report

11.1. Тест-плани

11.1.1. Технологічні ланцюжки і ролі учасників проекту, що використовують тест-плани. Зв'язок тест-планів з іншими типами проектної документації.

На підставі тест-вимог складаються тест-плани - програми випробувань (перевірки, тестування) програмної реалізації системи. На відміну від тест-вимог в тест-плані описуються конкретні способи перевірки функціональності системи, тобто то, **як** повинна перевірятися функціональність. Як правило, тест-план складається з окремих тестових прикладів, кожен з яких перевіряє деяку функцію або набір функцій системи. Для кожного тестового прикладу однозначно визначається критерій успішного проходження(pass / fail criteria), за допомогою якого можна судити

- чи відповідає поведінка системи заданому у вимогах чи ні (Рис 11.1).

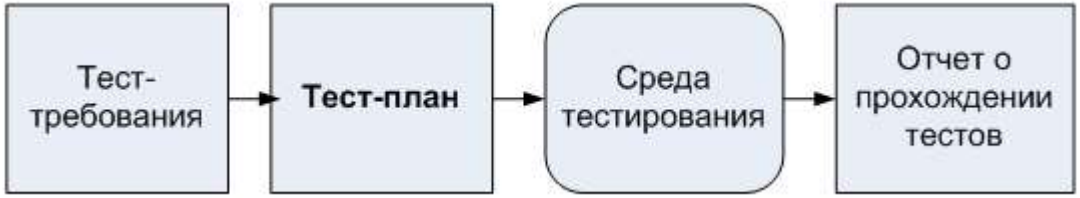


Рис. 11.1. Місце тест-планів серед проектної документації

Критерієм якості тест-плану є покриття (виконання) всіх вимог до перевірки правильності функціонування програмної реалізації.Бажаною характеристикою тест-плану є перевірка виконання всіх гілок схеми програмної реалізації.

Структура тест-плану може відповідати структурі тест-вимог або слідувати логіці зовнішньої поведінки системи. Кожен пункт тест-плану описує, **як** проводиться перевірка правильності функціонування програмної реалізації, і містить:

- посилання на вимогу (я), яке перевіряється цим пунктом;
- конкретне вхідний вплив на програму (значення вхідних даних);
- очікувану реакцію програми (тексти повідомлень, значення результатів)
- опис послідовності дій, необхідних для виконання пунктів тест-плану.

До складу тест-плану рекомендується додатково включати пункти, які служать для перевірки гілок програми, що не виконувалися при перевірці задоволення функціональних вимог. Такі пункти тест-плану можуть мати вказівку "Для повноти покриття" в полі посилання.

Тест-план може готуватися в формалізованої формі і служити вхідним документом для тестової

оснастки, за яким тести будуть виконуватися в автоматичному режимі з автоматичною фіксацією результатів. У разі, якщо тест-план готується у вигляді текстового документа, можливо тільки ручне тестування системи з даного тест-планом.

11.1.2. Можливі форми підготовки тест-планів

Форма подання тест-плану в першу чергу залежить від того, яким чином тест-план буде використовуватися в процесі тестування.При ручному тестуванні зручно уявлення тест-планів у вигляді текстових документів, в яких окремі розділи являють собою описи тестових прикладів. Кожен тестовий приклад у такому випадку включає в себе перерахування послідовності дій, які необхідно виконати тестувальника для проведення тестування - *сценарію тесту,* а також очікувані відгуки системи на ці дії. Така форма подання тест-плану незручна для *автоматизації тестування,* оскільки опису на природній мові практично не піддаються формалізації.

Для автоматизованого тестування сценарій тесту може записуватися на якому-небудь формальній мові, в цьому випадку можливе безпосереднє використання тест-планів як вхідних даних для середовища тестування.

Іншою формою представлення тест-планів є таблиця. Ця форма найбільш часто використовується при чітко і формально певних вхідних потоках даних системи. Наприклад, кожен стовпець таблиці може являти собою тестовий приклад, кожен рядок - опис вхідного потоку даних, а в комірці таблиці записується передане в даному тестовому прикладі в даний потік значення. Очікувані значення для даного тесту записуються в аналогічній таблиці, в якій в рядках перераховуються вихідні потоки даних.

І, нарешті, третьою формою представлення тестових прикладів є визначення прикладів у вигляді

кінцевого автомата. Така форма представлення використовується при тестуванні протоколів зв'язку або програмних модулів, взаємодія яких із зовнішнім світом проводиться за допомогою обміну повідомленнями по заздалегідь заданому інтерфейсу. Модуль при цьому може бути представлений як кінцевий автомат з набором станів, а тест-план складатиметься з двох частин - опису переходів між станами та їх параметрів і тестових прикладів, в яких задається маршрут переходу між станами, параметри переходів і очікувані значення. Таке уявлення тест-плану може бути придатне як для ручного, так і для автоматизованого тестування.

11.1.3. Сценарії

Подання сценаріїв, зручне для *ручного тестування* - тест-план у вигляді текстового документа, в якому кожен тестовий приклад представляє один розділ. Для кожного тестового прикладу в цей документ записується наступна інформація:

- ідентифікатор;
- опис тесту і його мета;
- посилання на тестовану частина системи;
- посилання на використовувану проектну документацію, зокрема тест-вимоги;
- перерахування дій сценарію;
- очікувана реакція системи на кожен пункт сценарію.

Мається на увазі, що дії сценарію мають бути описані таким чином, щоб їх міг відтворити людина практично з будь-яким рівнем підготовки. Опис очікуваної реакції системи має також бути записано таким чином, щоб можна було однозначно судити - відповідає реакція очікуваної чи ні.

Так, невдалою очікуваною реакцією при ручному тестуванні була б запис

Повідомлення "Завантаження" пропадає через прийнятний час

Ступінь прийнятності тут буде залежати від терплячості тестувальника, і забезпечити повторюваність тестування буде важко. Більш вдалою формою опису тієї ж самої очікуваної реакції буде

Повідомлення "Завантаження" зникає з екрану не більше,
ніж через 10 секунд після появи.

Нижче наведено приклад опису тестового прикладу у вигляді сценарію, призначеного для *ручного тестування:*

Група тестів: Робота з обліковими записами

Тестовий приклад: 1289-15

Призначення: Перевірка того, що обліковий запис користувача
перевіряється перед початком передачі даних і в разі
введення запису за замовчуванням при максимальному захисті
системи передачі не відбувається.

Тест-вимоги: 8.5.8.1, 8.5.8.2

Передумови для тіста: Система повинна бути приведена
в стан "Максимальний захист" і скинута

в налаштування за замовчуванням.

Критерій проходження тесту: Всі очікувані значення збігаються з реальними.

Сценарій тестування:

№	Крок сценарію	Очікуваний результат
1	Запустити термінальний клієнт і з'єднатися з системою за адресою 127.0.0.1	Має з'явитися запрошення терміналу TRANSFER>
2	Запустити процес передачі даних за допомогою введення команди SEND DATA	Має з'явитися запрошення *DATA TRANSFER* INITIATED і наступними двома рядками Enter your credentials ... Login:
3	Ввести ім'я облікового запису default	Повинна з'явитися рядок Password:
4	Ввести пароль default	Повинно з'явитися повідомлення Default user blocked - system set to High security і з'єднання з терміналом має бути перервано

Як можна бачити, така форма подання дійсно незручна для *автоматизації тестування* і призначена виключно для *ручного тестування.* Іноді такі тест-плани поєднують із звітами про проведення тестування, додаючи в таблицю опису сценарію третього і четверті колонки - "Реальний результат" і "Відповідає", в який заносяться реальна реакція системи і вказівка на збіг / розбіжність

результатів відповідно. Наприкінці опису кожного тестового прикладу додається графа "Пройдено не пройдений", в яку заноситься інформація про те, чи пройдено тестовий приклад в цілому. В кінці всього тест-плану, поєднаного з звітом, поміщається графа "Тестових прикладів пройдено / всього", в яку заноситься число пройдених тестових прикладів і загальне їх число.

Сценарії тестування для автоматичного тестування часто описують на тій чи іншій мові програмування. Наприклад, методи в тестуючих класах Microsoft Visual Studio Team Edition являють собою саме покрокові опису дій, які необхідно виконати тестового оточенню для проведення тестування. Можлива і більш близька до природної мови форма підготовки тестових прикладів.Наприклад, при тестуванні логічної функції з рівнем покриття MC / DC та описі тестових прикладів на одному з діалектів Visual Basic Script можливо записати сценарій тест-плану в такій формі:

```
 '---------------------------------------------- ---------------
'TEST CASES
'---------------------------------------------- ---------------
'8 testcases
'1 2 3 4 5 6 7 8
'----------------------------------------------
'Computed - 0 0 0 ---
'Good1 0 1 0 0 0 0 0 0
'Computed2 ---- 0 ---
'Good2 1 1 1 0 0 1 1 1
```

'Delay ----- 0 -

'Pack1 1 1 1 1 1 1 0 0

'Pack2 0 0 0 0 0 0 0 1

'----------

'Output_message 1 0 0 1 0 0 0 1

'---------- ----------

'Testcase # 1:

Call Test_Message_Call (-, 0, -, 1, -, 1, 0, 1)

'---------- ----------

'Testcase # 2:

Call Test_Message_Call (-, 1, -, 1, -, 1, 0, 0)

'---------- ----------

'Testcase # 2:

Call Test_Message_Call (0, 0, -, 1, -, 1, 0, 0)

'---------- ---------- 'Testcase # 4:

Call Test_Message_Call (0, 0, -, 0, -, 1, 0, 1)

'---------- ---------- 'Testcase # 5:

Call Test_Message_Call (0, 0, 0, 0, -, 1, 0, 0)

'---------- ---------- 'Testcase # 6:

```
Call Test_Message_Call (-, 0, -, 1, 0, 1, 0, 0)
'---------------------------------------------- ------------------ 'Testcase # 7:
Call Test_Message_Call (-, 0, -, 1, -, 0, 0, 0)
'---------------------------------------------- ------------------ 'Testcase # 8:
Call Test_Message_Call (-, 0, -, 1, -, 0, 1, 1)
```

Лістинг 11.1.

При такій формі подання сценарій кожного тестового прикладу складається з послідовності викликів функцій (в даному випадку функція всього одна), які передають дані в середу тестування.

11.1.4. Таблиці

Як вже говорилося вище, табличне представлення тестів зручно при чітко формалізованих вхідних і вихідних потоках даних системи. Наприклад, у попередньому фрагменті тест-плану в коментарях наведена таблиця, в якій по вертикалі вказані імена вхідних потоків даних системи, по горизонталі наведені номери тестових прикладів, а в осередках на їх перетині наведені значення.Вихідні значення наводяться в тому ж форматі нижче:

```
 '1 2 3 4 5 6 7 8
'---------------------------------------------
'Computed - 0 0 0 ---
'Good1 0 1 0 0 0 0 0 0
'Computed2 ---- 0 ---
'Good2 1 1 1 0 0 1 1 1
'Delay ----- 0 -
```

'Pack1 1 1 1 1 1 1 0 0

'Pack2 0 0 0 0 0 0 0 1

'----------

'Output_message 1 0 0 1 0 0 0 1

Табличне представлення, як правило, використовується для спрощення роботи з підготовки та супроводження великої кількості однотипних тестів. Середа тестування, що застосовує табличне опис тестових прикладів, в якості вхідних даних включає в себе*інтерпретатор таблиць,* які перетворять цей опис в послідовність команд, виконуваних середовищем для проведення тестування, тобто свого роду сценарій.

У разі, коли однотипними є не тільки вхідні і вихідні дані, але і їх значення, може використовуватися альтернативна форма представлення табличних даних. Тестові приклади в ній також нумеруються по горизонталі, а вхідні потоки даних - по вертикалі.Однак, під кожним з потоків даних перераховуються можливі вхідні значення, а факт того, що це вхідне значення повинне бути передане в даному тестовому прикладі, відзначається розташуванням спеціальної мітки (наприклад, символу X) на перетині значення і тестового прикладу в таблиці:

+ ---------- +

INPUTS: | abcdef |

---------- + ---------- +

Power_On_Mode |

COLD | XXX WARM | XXX

Configuration_Store_Id |

0xFFFD | XXXXXX

IR_Access_Mode |

1 | XXXX 0 | X

0xFFFF | X

Reset_Mode |

0 | XXXXXX

Reset_Source |

0 | XX 1 | X

2 | XXX

При інтерпретації кожного такого тестового прикладу він перетвориться в послідовність команд, які виконуються середовищем тестування. Наприклад, для тестового прикладу а:

Power_On_Mode = COLD

Configuration_Store_Id = 0xFFFD

IR_Access_Mode = 1

Reset_Mode = 0

Reset_Source = 1

Run_Test ()

Остання команда тут запускає тест на виконання до встановлених вхідними даними.

11.1.5. Кінцеві автомати

Форма подготововкі тест-планів у вигляді опису кінцевих автоматів зручна при тестуванні програмних модулів або систем, поведінка яких також може бути описано у вигляді кінцевого

автомата. У цьому випадку процес тестування являє собою обмін повідомленнями між двома кінцевими автоматами, що змінюють свій стан у процесі обміну. Критерієм повноти такого тестування буде досяжність всіх станів тестованої системи всіма можливими способами.

Опис тест-планів у вигляді кінцевого автомата зазвичай складається з двох частин: визначення самого тестирующего кінцевого автомата і визначення сценаріїв переходу між станами - тестових прикладів.

Розглянемо такий тест-план на наступному прикладі. Нехай тестований модуль являє собою простий кінцевий автомат з трьома станами - "Початкове", "Прийом даних" і "Помилка". Автомат починає свою роботу в початковому стані, з якого може бути переведений у стан "Прийом даних" після отримання повідомлення "Початок даних". Він може переходити з цього стану в нього ж з отримання кожного наступного правильного блоку даних, в стан "Помилка" по отриманні невірного блоку даних або в початковий стан з отримання повідомлення "Кінець даних". При переході в стан "Помилка" він передає повідомлення"Виникла помилка". Зі стану "Помилка" він може переходити в початковий стан після отримання повідомлення "Помилка оброблена". Структурна схема такого автомата показана на Рис 11.2 .

Рис. 11.2. Структурна схема тестованого кінцевого автомата

Тестуючий кінцевий автомат повинен вміти посилати всі сприймані тестируемим автоматом повідомлення і сприймати все посилають їм повідомлення. При цьому метою тестування буде проведення тестованого автомата по всіх станах всіма можливими способами. Один з можливих варіантів побудови тестирующего автомата полягає в побудові автомата з еквівалентними станами.Управління таким автоматом в основному буде проводитися за допомогою описів тестових прикладів, а не за допомогою повідомлень ззовні.

Так, такий тестуючий автомат буде мати три стани - "Початкове", "Передача даних" і "Обробка помилки". При переході з початкового стану в стан "Передача даних" він передає повідомлення "Початок даних", в стані "Передача даних" він передаватиме блоки даних, описані в тестовому прикладі, в т.ч., можливо, помилкові. При отриманні повідомлення "Виникла

помилка" автомат перейде в стан "Обробка помилки", з якого перейде в початковий стан, передавши повідомлення "Помилка оброблена". У початковий стан тестуючий автомат може перейти і в разі завершення послідовності блоків даних, описаних в тестовому прикладі, в цьому випадку при переході він пошле повідомлення "Кінець даних". Структурна схема такого автомата показана на Рис 11.3 .

Рис. 11.3. Структурна схема тестирующего кінцевого автомата

Далі наведено приклад визначення цього тестуючого автомата в тест-плані. Воно буде виглядати наступним чином:

STATES DEFINITION:

State1 = Початкове

State2 = Передача даних

State3 = Обробка помилки

PASS DEFINITION

Pass1 = State1-> State2 with function call BeginData (Param1)

Pass2 = State2-> State2 with function call SendData (Param1)

....

Pass5 = State2-> State3 external with function call ErrorReceived (Message)

У розділі STATES DEFINITON визначені всі стану тестирующего автомата, в розділі PASS DEFINITION - переходи між станами.Перехід зі стану M в стан N визначається виразом StateN-> StateM. При переході викликається функція тестового драйвера, ім'я якої записується після рядка with function call. Якщо у функцію мають бути передані параметри, їх імена вказуються в дужках. Якщо який-небудь перехід повинен відбуватися при отриманні зовнішнього повідомлення, це позначається ключовим словом external. При цьому викликається функція, обробна отримане повідомлення

Тестові приклади для тестування кінцевого автомата будуть виглядати наступним чином:

TESTCASE 1

Data:

begBlock = \ 027

sndBlock [0] = 'H'

sndBlock [1] = 'i'

errBlock = 0

Scenario:

Pass1 (begBlock)

Pass2 (sndBlock [0])

Pass2 (sndBlock [1])

Pass2 (errBlock)

Pass5 (message)

У цьому прикладі в секції Data визначаються дані для повідомлень, переданих автоматом, а в секції Scenario - послідовність переходів по станам з переданими даними.

При тестуванні кінцевих автоматів за допомогою звичайних тестуючих класів можна використовувати аналогічний підхід.

11.1.6. Генератори тестів

У деяких випадках для спрощення процедури тестування використовуються спеціальні інструментальні засоби, автоматично генеруючі тестові приклади. Ці системи різняться по використовуваних методів генерації тестових прикладів, а одержувані тестові приклади розрізняються по областях застосовності.

Розрізняють такі способи генерації тестових прикладів:

- по формалізованим вимогам;
- випадковим чином;
- з програмного коду.

Перший спосіб генерації тестових прикладів прийнятний для тестування системи як "чорного ящика", але вимагає, щоб тест-вимоги (або системні / функціональні вимоги) були підготовлені на

спеціальному формальній мові оформлення вимог, наприклад, *RDL*(Requirements Definition Language). Потім за вимогами будуються тестові приклади, які перевіряють функціональність системи з точки зору вимог, тобто в цьому випадку досягається основна мета верифікації - перевірити, чи веде себе система відповідно до вимог.

На жаль, цей шлях досить трудомісткий і економія часу від автоматичної генерації тестів часто зводиться нанівець необхідністю у виділенні додаткового часу на переклад всіх вимог у формальну форму. У зв'язку з цим рекомендується застосовувати даний метод тільки для тестування систем, вимоги на які можуть бути порівняно легко формалізовані з використанням тієї чи іншої мови - наприклад, системи підтримки комунікаційних протоколів.

Другий метод генерації тестових прикладів - на основі випадкових даних. У цьому випадку не може йти й мови про систематизованому тестуванні та гарантії якості системи. Такий підхід може застосовуватися тільки при необхідності перевірити поведінку системи у разі передачі в неї великої кількості невірних даних або визначити кількісні параметри поведінки системи під великим навантаженням.

Третій метод тестування заснований на аналізі вихідних текстів системи і побудови тестів, які виконують кожне логічне умова і кожен оператор системи. В результаті досягається дуже високий рівень покриття програмного коду. Однак, в цьому випадку тести перевіряють не те, що система повинна робити відповідно до вимог, а те, як вона робить вже запрограмоване. Перед тестувальником в цьому випадку стоїть завдання аналізу програмного коду системи на відповідність вимогам, що часто представляє собою завдання не менш складну, ніж ручне написання тестів для перевірки вимог. Зазвичай рекомендується спочатку написати всі тести за

вимогами, а потім, у разі потреби, скористатися генератором тестів з програмного коду. При цьому метою використання генератора буде не досягнення максимально можливого покриття будь-яку ціну, а аналіз причин непокриття при виконанні тестів вимог і, в разі необхідності, корекції вимог.

11.2. Звіти про проходження тестів

11.2.1. Технологічні ланцюжки і ролі учасників проекту, що використовують звіти про проходження тестів. Зв'язок звітів про проходження тестів з іншими типами проектної документації

Звіти про проходження тестів - основний (а іноді єдиний) джерело для висновку про відповідність протестованої системи вимогам.Після виконання всіх тестів, описаних в тест-планах, середа тестування створює звіт про те, наскільки успішно система виконала ці тести. Такий звіт як мінімум містить інформацію про кожного виконаному тестовому прикладі (його ідентифікатор) і результат його виконання - успіх або невдача.

За результатами аналізу звітів про проходження тестів можуть бути виявлені або дефекти в самій системі, або некоректно складені або суперечливі вимоги. В обох випадках результати аналізу служать основою для створення запитів на зміну вимог та / або коду системи. Після коректного виправлення дефектів при регресійному тестуванні неуспішно виконані тестові приклади повинні виконатися успішно (Рис 11.4).

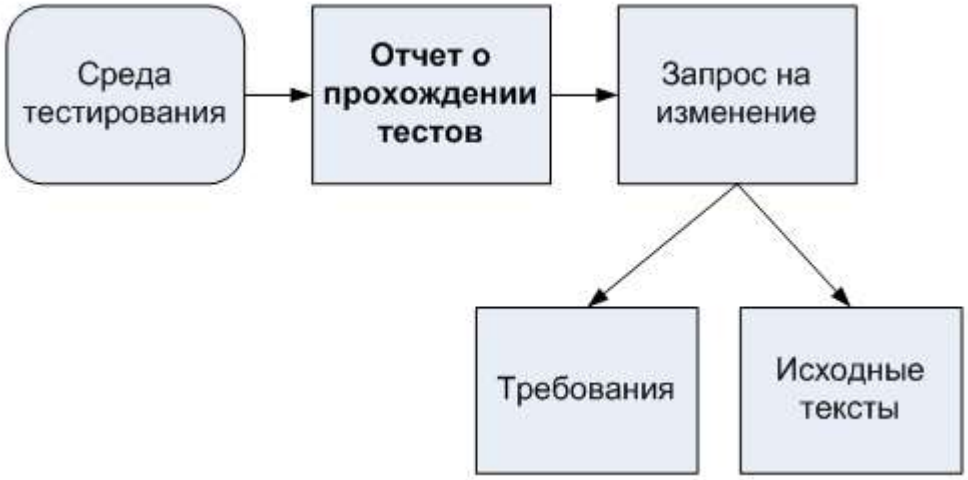


Рис. 11.4. Генерація звіту про проходження тестів і зміни за результатами його аналізу

Звіти про проходження тестів можуть служити основою для відстеження стану проекту: якщо з часом кількість виявляються дефектів (неуспішно виконаних тестових прикладів) падає за умови збереження якості тестування - це свідчить про підвищення якості розроблюваної системи. З іншого боку, при внесенні значних змін у систему кількість дефектів неминуче зростає. Таким чином, ідеальний графік залежності кількості дефектів від часу схожий на синусоїду з зменшується амплітудою на кожному напівперіод.

11.2.2. Можливі форми подання звітів про проходження тестів

У попередніх лекціях вже наводилося кілька прикладів звітів про виконання тестових прикладів, проте раніше основний ухил робився в бік загальної статистики виконання тестів.

У стандарті IEEE 829 звіт про проходження тестів розділений на три різних документа і описаний в розділах 9 (*Test log)*, 10 (Test*incident* report) і 11 (Test *summary report).* У ці розділи включені

відповідно загальний звіт про проходження тестів, звіт про проблеми, виявлених в результаті виконання тестів, і загальна статистика проходження тестів. У даному курсі звіт про проходження тестів вважається єдиним документом, розділеним на три частини:

- загальна (заголовна інформація);
- результати виконання тестових прикладів (позитивні і негативні);
- підсумкова інформація про виконання тестових прикладів (загальна статистика по виконаним тестам).

Заголовна частина звіту про проходження тестів служить для ідентифікації звіту і протоколювання того, яка частина розроблюваної системи піддавалася тестуванню, яка її версія, яка конфігурація тестового стенду використовувалася для виконання тестів.

У заголовну частину звіту про виконання тестів зазвичай включається наступна інформація:

1. Назва проекту або тестованої системи
2. Загальний ідентифікатор групи тестових прикладів, включених до звіту
3. Ідентифікатор модуля, що тестується або групи модулів і номери їх версій
4. Посилання на розділи і версії тест-вимог або функціональних вимог, на підставі чого написані тести, для яких згенерований звіт
5. Час початку виконання тесту і його тривалість
6. Конфігурацію тестового стенду, на якій виконувався тест
7. Імена та прізвища автора тестів і / або особи, виконував тести.

Нижче показані два приклади таких заголовних частин звіту, створюваних різними інструментальними засобами. Червоними цифрами в дужках позначені відповідні пункти

наведеного вище списку.


```
************************************************
** Document Test Environment
**      User's Computer:          COMPUTER_185 (6)
**      Testing Host Application: FacilityTest (6)
**      Testing Host Version:     5.12 (6)
**
*************   Server Related Data    *************
**      Server Computer:          SERVER_105 (6)
**      Server Version:           6.24.0 (Build 16) (6)
**      Configuration:            Control remote bench (6)
**      Mode:                     Realtime (6)
**      Test executed on: 7/29/06; at 10:09:40 AM (5)
**      Tester Name is [ Sidorov A. ] (7)
**      Software Version is: CNTRL 115 01 5 (1)
**      Test Station being used is: COMPUTER_185 (6)
************************************************


=====================================================================
REMOTE CONTROL FUNCTION SOFTWARE TEST REPORT
=====================================================================Pro
ject Name           : Facility Remote Control (1)
Function Name         : Infrared Transmitter Signal Handler (3)
Test Name             : IRDA_C05A_1091K (2)
Document Name         : SSRD for the Remote Control Function (4)
Paragraph Name        : Button Signals (4)
Primary Paragraph Tag : [PTAG::SSRD IR BTN SIGNALS] (4)
Template Class        : Test
Shall Tag(s)          : SSRD IR BTN SIGNALS 10 (4)
Shall(s) template     : Test
 ---------------------------------------------------------------------
MODIFICATION HISTORY:
 ---------------------------------------------------------------------
 Ver   Date         Author       Change Description         CR No.
 ---   ---------    ----------   ------------------------   --------
 01    19 Jul 06    Ivanov K. (7)Initial Development.       CR_10
=====================================================================
```

```
;  SIMULATION RESULTS FILE
;  Matrix Compiler CORE VERSION 3.00
;         TEST PLAN
;  ELEMENT: IRDA_IA.TMC     (2)
;  TITLE: Test Plan for Inrfrared source files test (1)
;  TEST DATE/TIME Wed 02.11.2005 23:12:53     (5)
;  SYS section: 2.3.5.6 Version: 24           (4)
;  SRD section: 6.3 Version: 12               (4)
;  SDD section: 12.3 Version: 33        (4)
;  SOURCE FILE(S):    IRDA.C Version: 18      (4)
;                     IRDA.H Version: 2       (4)
;
;  SIMULATOR SETUP:  (6)
;     MODE HIGH(6)
;     INC CIP.INC     (6)
```

Наступна частина звіту про проходження тестів повинна містити інформацію про результат виконання кожного тестового прикладу - чи завершився він успішно або в результаті його виконання були виявлені які-небудь невідповідності з очікуваним результатом. У деяких проектах ця частина звіту може бути представлена в одній з двох форм - повної або короткою. Повна форма містить всю інформацію про тестовому прикладі, коротка - тільки інформацію про виявлені в результаті виконання тестового прикладу невідповідностей очікуваних і реальних вихідних значень. Зазвичай кожен запис про результат проходження кожного тестового прикладу в повній формі містить наступну інформацію:

- Ідентифікатор тестового прикладу
- Короткий опис тестового прикладу
- Перерахування всіх вхідних значень тестового прикладу
- Перерахування всіх очікуваних і реальних вихідних значень тестового прикладу
- Для кожної пари "очікуване-реальне вихідне значення" - інформацію про збіг / розбіжність цих

значень

- Повідомлення про те, пройдений або не пройдений тестовий приклад

У короткій формі кожен запис зазвичай містить таку інформацію:

- Ідентифікатор тестового прикладу
- Перерахування не співпали очікуваних і реальних вихідних значень тестового прикладу
- Для кожної пари "очікуване-реальне вихідне значення" - інформацію про збіг / розбіжність цих значень
- Повідомлення про те, пройдений або не пройдений тестовий приклад

Нижче наведено два приклади інформації про проходження тестового прикладу в короткої і повної формах відповідно. Червоними цифрами в дужках відзначені відповідні пункти наведених вище списку для короткої і повної форм відповідно.

```
[Testcase 163] (1) :True: <EQ> :True: (4) ** Passed Number 163 **
[Testcase 164] :True: <EQ> :True: ** Passed Number 164 **
[Testcase 165] :True: <EQ> :True: ** Passed Number 165 **
[Testcase 166] :False: <EQ> :True: (4) ** Fail Number 1 **
      *** Inputs for Testcase 166
            DisplayTextLine2.ItemChecked  = 2 (2 expected)
            DisplayTextLine2.ItemChecked  = 2 (2 expected)
      *** Outputs for Testcase 166
            DisplayTextLine2.ItemChecked  = 2 (2 expected) (2)
            (3) --- DisplayTextLine2.ItemChecked  = 2 (1 expected)
            DisplayTextLine9.ItemChecked  = 2 (2 expected)
```

```
;  1)    Test group 1, case a. (1)
;      Test case verifies that infrared watchdog is activated by
;      startup pulse sequence (2)
;      Test requirements section 6.4.3.1.2

       CASE DEFAULTS : (3)
T_FL_Sys_Fail_Called          = 0
T_Update_Time                 = 1828ACh
T_CMT_Menu_Last_Update        = 18639Ch
T_Level_1_Status              = 180004h
T_Level_2_Status              = 180304h
T_Stop_Method                 = 0
T_Fault_Report                = 1

       INPUTS : (3)
num_iterations                = 1
entry_procedure               = 1
T_NV_Power_On_Count           = 1
T_Reset_Value                 = 0
T_Time_Since_Power_On         = 1

OUTPUTS:                       EXPECTED (4)     ACTUAL (4)  RESULT (5)
T_NV_Power_On_Count           = 1              1           PASS
T_NV_Power_On_Count_Check     = 65533          65533       PASS
T_BBRAM_Power_On_Count        = 1              1           PASS
T_Time_Since_Power_On         = 100            100         PASS
T_FH_Queue_Msg_Count          = 2              2           PASS
T_Pulse[0].Data[0]            = 0              0           PASS
T_Pulse[0].Data[1]            = 0              10          FAIL

Test case FAILED (6)
```

Завершальна частина звіту про проходження тестів повинна містити коротку підсумкову інформацію про виконання всіх тестових прикладів, за якими складався звіт.

Звичайно ця частина звіту містить наступну інформацію:

1. Загальна кількість виконаних тестових прикладів
2. Кількість успішно пройдених тестових прикладів

3. Кількість неуспішно пройдених тестових прикладів
4. Загальна кількість перевірених вихідних значень
5. Кількість вихідних значень, у яких очікуване значення не співпало з реальним

Нижче наведено приклад цієї частини звіту.

```
TEST RESULTS:

No. of Test Cases Failed     : 0 (3)
No. of Test Cases Passed     : 45 (2)
Total No. of Tests Included  : 46 (1)
Total No. of Outputs Checked : 2783 (4)
No. of failed Outputs Checks : 128 (5)
```

Часто в звіт про виконання тестів крім кількісної статистики поміщають розділ з докладним поясненням причин неуспішно пройдених тестових прикладів. Кожен пункт такого пояснення зазвичай містить таку інформацію:

1. Ідентифікатори тестових прикладів, завдяки неуспішному виконанню яких виявлена проблема
2. Посилання на розділи вимог, за якими написані тестові приклади
3. Посилання на ділянки програмного коду в якому виявлено неполадку
4. Опис суті проблеми і (опціонально) можливі шляхи її вирішення з точки зору тестувальника

Даний розділ може служити основою для створення звітів про проблеми або частково замінювати їх.

Приклад такого розділу наведено нижче:

```
TEST CASES WITH FAILURES SUMMARY
================================

testcases          | failures total | explanation in section
-------------------+----------------+-----------------------
(1) 11b-1,n-p; 12b-d |  67            | 1
-------------------+----------------+-----------------------
 total             |  67            |


1.
LOCATION:
    PR_IR_DATA.C, lines 1323, 1347; (3)
    Software requirements section 7.4.5.5; (2)
    Test requirements section 7.4.8; (2)
PROBLEM:
    Test requirements are not changed, but Software requirements are updated
     to reflect new system functionality. (4)
DEMONSTRATION:
    Test cases: 11 b-1, n-p; 12 b-d (1)
PROPOSED SOLUTION:
    Update test requirements section 7.4.8 to meet software requirements
     section 7.4.5.5. (4)
```

11.2.3. Автоматичне і ручне тестування

Деякі тестові приклади не можуть бути виконані в автоматичному режимі і тому вимагають ручної роботи тестувальника щодо їх виконання. Результати виконання ручних тестових прикладів можуть заноситися в той же самий документ, що й результати виконання автоматичних тестових прикладів. Особливо часто це робиться у випадку, якщо і автоматичні, і ручні тести перевіряють одну і ту ж функціональну частину тестованої системи. У цьому випадку при генерації звіту про проходження тестів для ручних тестів генерується форма, в яку тестувальник заносить дані про результати проведеного ним *ручного тестування*. Само ручне тестування може полягати або у виконанні тестового сценарію, заданого в тест-плані, або в експертному аналізі ділянок

програмного коду системи, які не можуть бути виконані при автоматичному тестуванні на тестовому стенді. Форма для *ручного тестування* зазвичай містить таку інформацію:

1. Ідентифікатор ручного тестового прикладу
2. Опис сценарію ручного тестового прикладу або завдання експертного аналізу
3. Ім'я особи, яка проводила ручне тестування
4. Версії вимог, на підставі яких проводилося ручне тестування
5. Посилання на ділянки програмного коду, для якого проводиться ручне тестування
6. Інформацію про відповідність програмного коду вимогам (результат *ручного тестування)* - відповідає / не відповідає
7. Інформацію про потенційно можливі проблеми всередині допустимого діапазону значень і за його межами
8. Інформацію про можливість покриття тестованого вручну програмного коду при досягненні умов, зазначених у вимогах
9. Інформацію про підсумковий результат ручного тестового прикладу - успішно / неуспішно

Нижче наведено приклад заповненої форми для *ручного тестування.* Червоними цифрами в дужках виділені відповідні пункти наведеного вище списку, зеленим виділений текст, що вводиться в форму тестувальником:

```
*******************************************************************************
Manual Analysis of Testcase 1 (1): verify that the IR scan goes at a 10Hz rate.
(2)
*******************************************************************************
1. Activity Description: Independent Code Analysis.
      Tester Name: Petrov P. (3)
Pass Criteria: Tester name is not the same as programmer name

2. Activity Description: Name and CM Version of file under review
      Module and/or Function Name: IRDA.C       CM Version: 56 (4,5)
      Document chapter: 7.8.5                   CM Version: 12 (4)
Pass criteria: All documents exists in respective versions

3. Activity Description: Requirements Implemented.
Identify which lines of code in the module under review incorporate requirements

PassCriteria:   Code implements the requirements as defined in the requirement
document
Result (Pass/Fail): Pass (6)

4. Activity Description: Code Robustness
Identify line numbers and give a brief description on the software design to
handle the following cases
      Analysis for at, Inside boundary  : The variable refreshRate is set to 10
at line 235 of IRDA.C and later used in IR_Init() function to set NVRAM values
on line 590. (5,7)
      Analysis for out-of-boundary (robustness) : N/A (7)

5. Activity Description: Structural Coverage Analysis
PassCriteria:   Software and software structures (when applicable) are
accessible under conditions specified by requirements
 Y/N:  YES (8)

6. Activity Description: Overall manual test result
Result (Pass/Fail): Pass (9)
```

Документація, що супроводжує процес верифікації та тестування (звіти)

13.1. Звіти про покриття програмного коду

13.1.1. Технологічні ланцюжки і ролі учасників проекту, що використовують звіти про покриття. Зв'язок звітів про покриття з іншими типами проектної документації

Ступінь покриття програмного коду тестами - важливий кількісний показник, що дозволяє оцінити

якість системи тестів, а в деяких випадках - і якість тестируемой програмної системи. Дані про ступінь покриття поміщаються в звіти про покриття, які генеруються при виконанні тестів інструментальними засобами, що підтримують процес тестування, тобто по суті, генеруються середовищем тестування (Рис 13.1). Формат звітів про покриття зазвичай єдиний всередині проекту або декількох проектів і часто залежить від особливостей інструментальних засобів тестування.

У звіті про покриття в стандартизованої формі вказуються ділянки програмного коду тестованої системи (або її частини), які не були виконані під час виконання тестових прикладів, тобто не були покриті тестами. Причини непокриття аналізуються тестувальниками, за результатами аналізу складаються звіти про проблеми і запити на зміну - документи, де описуються об'єкти розробки, які необхідно змінити, і причини цих змін.

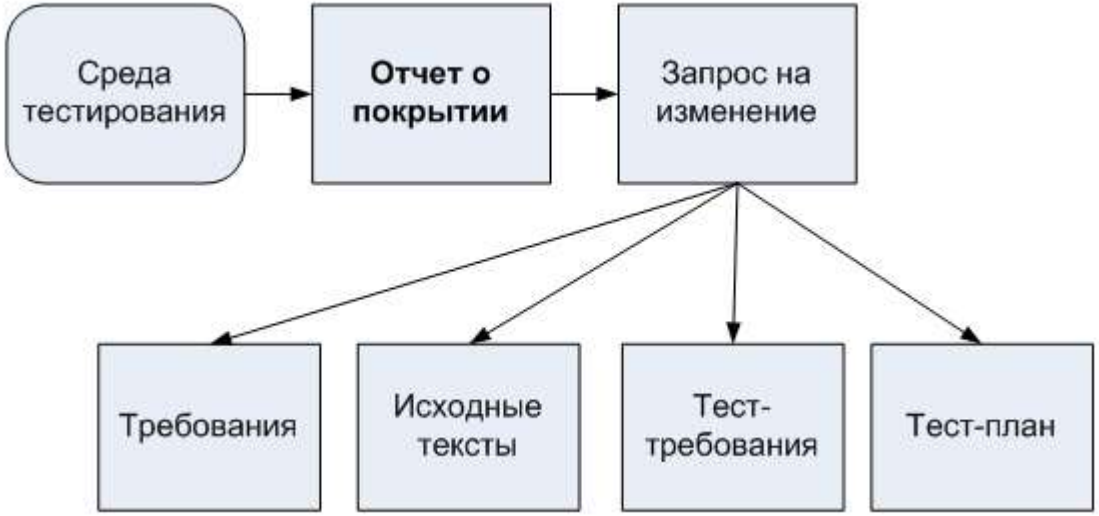


Рис. 13.1. Генерація звіту про покриття та зміни за результатами його аналізу

Недостатнє покриття може свідчити про неповноту системи тестів або тест-вимог, в цьому випадку в запиті про зміну вказується на необхідність розширення системи тестів або тест-вимог. Іншою причиною недостатнього покриття можуть бути ділянки захисного коду, які ніколи не виконаються навіть у разі нештатної роботи системи. У цьому випадку в запиті на зміни вказується на необхідність модифікації вихідних текстів або відзначається, що для цієї ділянки програмної системи не потрібно покриття. В якості третьої причини недостатнього покриття може виступати неузгодженість вимог і програмного коду системи, в результаті якого в коді можуть залишитися невикористовувані більш ділянки або, навпаки, з'явитися ділянки, розраховані на майбутнє (і реалізують функціональність, що не описану у вимогах). У цьому випадку в запиті на зміну вказується на необхідність модифікації вимог і / або коду системи для приведення їх у узгоджений стан.

13.1.2. Можливі форми звітів про покриття

Типовий звіт про покриття являє собою список структурних елементів покривається програмного коду (функцій або методів), що містить для кожного структурного елементу наступну інформацію:

- Назва функції або методу
- Тип покриття (по рядках, по гілках, MC / DC або інший)
- Кількість покриваються елементів у функції або методі (рядків, гілок, логічних умов)
- Ступінь покриття функції або методу (у відсотках або в абсолютному вираженні)
- Список непокритих елементів (у вигляді ділянок непокритого програмного коду з номерами рядків)

Крім того, звіт про покриття містить заголовну інформацію, що дозволяє ідентифікувати звіт, і загальний підсумок - загальну ступінь покриття всіх функцій, для яких збирається інформація про покриття.

Приклад такого звіту про покриття наведено нижче.

```
 Coverage Report
Generated 10/07/2006 for file Testing_Facilities.cpp
------------------------------------------------ -
1) function main_Menu ()
Coverage: Instructions
Elements: 25 structured lines of code (SLOCs)
Covered: 22 lines (88%)
Not covered:
   291 default:
   292 return -1;
   293 break;

Coverage: Branches
Elements: 5 branches
Covered: 4 branches (80%)
Not covered (starting and ending lines only):
default:
```

```
break;
```

---

2) function item_Help ()

Coverage: Instructions

Elements: 180 structured lines of code (SLOCs)

Covered: 180 lines (100%)

Coverage: Branches

Elements: 2 branches

Covered: 2 branches (80%)

Total functions: 2

Total instructions coverage: 98.5%

Total branches coverage: 86%

Звіт про покриття може створюватися або для всіх функцій програмного модуля або всього проекту, або вибірково для визначених функцій.

У разі, якщо розмір функцій, для яких генерується вибірковий звіт, невеликий, може застосовуватися інша форма звіту про покриття, в якому покритий і непокритий програмний код виділяється різними кольорами. Така форма непридатна для покриття гілок і логічних умов, але може застосовуватися для покриття по рядках.

Приклад такого звіту наведено нижче:

```
Coverage report for BaseCalculator.AnalaizerClass.Format  method.
Generated on 25/07/2006
```

```
public static string Format()
{
   string formstr = "";
   string prev = "";
   if (expression.Length <= 65536) {
        for (int i = 0; i < expression.Length; i++) {
             switch (expression[i]) {
                  case '0': {
                       if (prev == "число" || prev == "") {
                            formstr += expression[i].ToString();
                       } else {
                            formstr += " " + expression[i].ToString();
                       }
                       prev = "число";
                       break;
                  }
             }
        }
   } else {
        MessageBox.Show("Слишком длинное выражение.");
        Program.res = 7;
        return "cError 07";
   }
}
```

Зеленим кольором відмічені виконані в результаті тестування ділянки методу, червоним - не виконаних.

Конкретна форма звіту про покриття визначається інструментарієм і технологічними процесами проекту.

Розглянемо, наприклад, звіт про структурний покритті, генерований за допомогою засобу для

Збір покриття проводиться по певному рівню покриття, інформація про це входить до складу заголовка звіту про покриття. У даному випадку програмний код покривався за MC / DC.

CodeTEST Advanced Coverage Tools

Modified Condition Decision Coverage Report (MCDC), DO-178B Level A

CodeTEST Report Generator 2.2.04

Report generated on Sun Aug 21 14:28:16 2005

IDB File:

<Path_to_IDB_file>

42963330-1C0 Thu May 26 21:36:00 2005

Далі виводиться інформація про відсоток покриття по всіх модулях в сукупності (в даному випадку, це 33 модуля):

Overall Summary (MCDC): 33 Files

1405 true-false decisions

674 48.0% covered

494 35.2% partially covered

237 16.9% not covered

1928 conditions in the decisions

1068 55.4% covered

571 29.6% partially covered

289 15.0% not covered

278 case branches

154 55.4% covered

124 44.6% not covered

581 coverage events

504 86.7% covered

77 13.3% not covered

Розглянемо більш детально кожну з характеристик.

True-false decisions - кількість логічних виразів, в даному випадку їх 1405. З цього числа на обидва можливих значення (True iFalse) покрито лише 674, ще 494 покрито на якесь одне зі значень, а 237 не покритих взагалі.

Conditions in the decisions - кількість логічних умов всередині логічних виразів. Як можна здогадатися, їх кількість не може бути менше кількості логічних виразів. Непокриті логічних умов явним чином тягне за собою непокриті логічних виразів. Факт, що якщо всі умови будуть покриті, то всі вирази також будуть покриті.

Case branches - кількість гілок розгалуження операторів вибору (select). Гілка вважається покритої, якщо вираз, що стоїть в умові оператора select, приймає значення, відповідне даному варіанту вибору (case), в результаті чого гілка виконується.

Coverage events - в даному випадку, це кількість точок входу / виходу у функціях. Точка входу в функцію вважається покритої, якщо ця функція була викликана хоча б один раз. Точками виходу є точка закінчення функції (}), коли управління передається в місце виклику цієї функції, а також

оператори повернення (return), після виконання яких управління також передається в місце виклику функції.

Потім виводиться інформація для першого модуля, а саме: ім'я модуля, шлях до файлу на диску, дата останньої зміни, контрольна сума. Після цього виводиться сумарна інформація про покриття тільки для цього модуля (для всіх його функцій, кількість яких також вказується). Структура цієї інформації аналогічна структурі, описаної вище:

1. File <Module_1_name.cpp> in <Path_to_module_1>
   Last Modified: Fri Oct 15 18:34:14 2004
   Checksum: CEDBAB67

   File Summary (Extended MCDC): 6 Functions
   24 true-false decisions
   10 41.7% covered
   11 45.8% partially covered
   3 12.5% not covered
   39 conditions in the decisions
   20 51.3% covered
   15 38.5% partially covered
   4 10.3% not covered
   0 case branches
   12 coverage events

12 100.0% covered

0 0.0% not covered

Після цього відбувається почерговий висновок сумарної інформації для кожної з функцій окремо і також всіх ключових ділянок функції, що впливають на покриття, а саме: точки входу / виходу, логічні вирази, логічні умови, оператори вибору. При цьому для кожного з цих ключових ділянок виводиться інформація про ступінь покриття, де можливими значеннями можуть бути:

COVERED - ділянка повністю покритий в поняттях типу, до якого він приписаний;

PARTIALLY covered - ділянка частково покритий. Застосовується тільки до логічних виразів і умовам і вказує на те, що вираз (умова) покрито на якесь одне з двох можливих значень;

NOT covered - ділянка не покритий. Це означає, що дана ділянка не виконувався в процесі роботи скомпільованого програмного коду.

Нижче представлений типовий результат покриття для функції.

1.1 Function <Function_1>

void Function_1 (void)

Function Summary (MCDC):

3 true-false decisions

0 0.0% covered

2 66.7% partially covered

1 33.3% not covered

5 conditions in the decisions

1 20.0% covered

3 60.0% partially covered

1 20.0% not covered

0 case branches

2 coverage events

2 100.0% covered

0 0.0% not covered

coverage line 266: function entry

COVERED

decision line 267: if statement

if ((A == B) && (...

TF

1: * ttt * fxx ((A == ...) && COVERED

2: * ttt tfx ((C. ..) && PARTIALLY covered

3: * ttt ttf ((D == 0) PARTIALLY covered

decision line 271: if statement

if (E <= 0)

```
    TF
 1: t * f (E <= 0) PARTIALLY covered

decision line 276: if statement
   if (D == 0)

    TF
 1: tf (D. ..) NOT covered

coverage line 291: function end
   }
   COVERED
```

Слід звернути увагу на те, яким чином відбувається розбір логічних виразів. Припустимо, що у нас є вираз виду:

```
 if ((A == B) && (C> 0) && (D == 0))
```

і нехай результат покриття для нього такий, як наведено вище (decision line 267: if statement). Для цього виразу будується таблиця, де в кожному рядку поміщається інформація для кожного з логічних умов, що входять у це логічне вираження.Нумерація логічних умов починається з 1, в нашому випадку їх 3.

У першому стовпці вказується комбінація значень логічних умов (за участю розглянутого умови),

яку необхідно виставити, щоб логічне вираження прийняло значення True. В даному випадку, так як всі умови з'єднані логічним оператором "І" (&&),такою комбінацією може бути тільки ТТТ. Якщо зазначена комбінація була сформована в процесі тестування програми, то вона позначається символом (*).

У другому стовпці вказується комбінація значень логічних умов (за участю розглянутого умови), яку необхідно виставити, щоб логічне вираження прийняло значення False. В даному випадку достатньо, щоб хоча б одна умова прийняло значення False,при цьому значення всіх наступних умов не враховуються (це відображається символом X).

Третій стовпець - це вирізка тексту логічного умови, що дозволяє легше орієнтуватися в коді при аналізі покриття.

З вийшла таблиці видно, що друге і третє логічне умова в процесі виконання не приймали значення False. Визначення причини цього і є завданням аналізу структурного покриття.

Після того, як в рамках даного модуля будуть розглянуті всі функції, розглядається наступний модуль, і так далі. Ми бачимо, що структура звітного файлу, одержуваного в результаті роботи CodeTEST, досить проста і має вкладену структуру.

Звіти про покриттях, створювані Microsoft Visual Studio Team Edition, будуть розглянуті на семінарських заняттях.

13.1.3. Покриття на рівні вихідних текстів і на рівні машинних кодів

У деяких випадках інструментальні засоби збору покриття аналізують покриття програмного коду тестами не на рівні вихідних текстів системи, а на рівні машинних інструкцій. У цьому випадку ступінь покриття залежить і від того, який виконуваний код генерується компілятором. Звіти про

покриття в цьому випадку виглядають наступним чином (звіт про покриття сгененірован за допомогою відладчика Microtec XRAY для Motorola 680x0):

```
******* INSTRUCTION EXECUTION COVERAGE - Analyze Unit: Menu_Display
Function: DATA_PROCESSING \ Menu_Display.  Instruction (s) not executed:
  5668 switch (Key)
0004A0F6 0352 BCHG D1, (A2)
0004A102 0466 0466 SUBI.W # $ 466, - (A6)
0004A106 0466 0466 SUBI.W # $ 466, - (A6)
0004A10A 0466 0466 SUBI.W # $ 466, - (A6)
0004A10E 0466 0466 SUBI.W # $ 466, - (A6)
0004A112 0466 0466 SUBI.W # $ 466, - (A6)
0004A116 0466 0474 SUBI.W # $ 474, - (A6)
  5751 i = 1;
0004A4A6 7401 MOVEQ # $ 1, D2
   Executed Total Percent Analyze-unit
       467475 98.32 DATA_PROCESSING \ Menu_Display
            0 unit (s) excluded.

******* BRANCH EXECUTION COVERAGE - Analyze Unit: Menu_Display
Function: DATA_PROCESSING \ Menu_Display.  Branch (es) not executed:
  5612 if (First_Call == TRUE) {/ * Only execute this block during th
```

```
00049E3A 6600 0722 BNE.W $ 4A55E Branch not taken.
  5613 if (MP_Rd_Config_Straps (RAW_STRAPS, & Strap_Config) ==
00049E52 664A BNE.B $ 49E9E Branch not taken.
  5750 if (i == 17)
0004A4A4 6602 BNE.B $ 4A4A8 Fall thru not taken.
    Executed Total Percent Analyze-unit
         74 77 96.10 DATA_PROCESSING \ Menu_Display
              0 unit (s) excluded.
```

Лістинг 13.1.

Оскільки ступінь покриття може змінюватися в залежності від оптимізації при генерації коду, в деяких випадках навіть при повному виконанні всіх операторів мови високого рівня, на якому написана програмна система, не вдається досягти повного покриття на рівні виконуваного коду. Наприклад, у випадку покриття наступної конструкції на мові С:

```
 typedef enum
{
CC1 = 250,
CC2 = 251,
CC3 = 252
} E_CC;
   ...
E_CC key;
```

```
   ...
 switch (key) {
    case CC1: printf ("CC1");  break;
    case CC2: printf ("CC2");  break;
    case CC3: printf ("CC3");  break;
 }
```

компілятор Microtec C для Motorola 680x0 створить таблицю можливих значень змінної key, в якій будуть присутні всі елементи від 0 до 255. Реально в програмній системі можливе передати тільки значення констант CC1 - CC3, і в результаті покритими виявляться тільки три гілки з 255 можливих у виконуваному коді. Ніякими стандартними засобами збільшити ступінь такого покриття не можна. У цьому випадку звіт про покриття супроводжується додатковою інформацією про причини неможливості забезпечити повне покриття.

Збір інформації про покриття на рівні виконуваного коду найбільш часто застосовується в висококритічних програмних системах, де не допускається наявності "мертвого" виконуваного коду, який потенційно може привести до збою або відмови під час роботи системи. До таких систем в першу чергу можна віднести авіаційні бортові системи, медичні системи та системи забезпечення безпеки інформації.

13.2. Звіти про проблеми

13.2.1. Технологічні ланцюжки і ролі учасників проекту, що використовують звіти про проблеми.Зв'язок звітів про проблеми з іншими типами проектної документації

Кожне невідповідність до вимог, знайдене тестувальником, має бути задокументовано у вигляді

звіту про проблему. Ймовірність виявлення та виправлення помилки, що викликала цю невідповідність, залежить від того, наскільки якісно вона задокументована.Звіти про проблеми можуть надходити не тільки від тестувальників, а й від фахівців технічної підтримки або користувачів, проте їх спільна мета - вказати на наявність проблеми в системі, яка має бути усунена. Якщо звіт складений некоректно, розробник не зможе усунути проблему, тому можна вважати цей звіт одним з найважливіших документів в ланцюжку тестової документації.

Головне, що повинно бути включено в звіт про помилку, - це:

- Спосіб відтворення проблеми. Для того, щоб розробник зміг усунути проблему, він повинен розібратися в її причинах, самостійно відтворивши її (і, можливо, не один раз). Один з найважчих випадків в процесі розробки виникає при невоспроизводимость проблемі, тобто проблемі, у якої точно не відомий спосіб її викликати. Знаходження такого способу - одна з найбільш нетривіальних завдань у роботі тестувальника.
- Аналіз проблеми з коротким її описом. Найкраще приводити опис в тих же термінах, в яких складено вимоги на частину системи, в якій виявлена проблема. У цьому випадку мінімізується ймовірність непорозуміння суті проблеми.

Будь-який звіт про проблему має бути складений негайно після її виявлення. Якщо звіт буде складений через значний час, підвищується ймовірність того, що в нього не потрапить яка-небудь важлива інформація, яка допоможе усунути причину проблеми в найкоротші терміни.

13.2.2. Структура звітів про проблеми, їх трасування на програмний код і документацію

Структура звіту про проблему в цілому мало відрізняється в різних проектах, зміни зазвичай стосуються тільки порядку та імен проходження полів. Деякі поля можуть відображати специфіку

даного конкретного проекту, проте зазвичай ці поля наступні:

- Об'єкт, у якому знайдена проблема. Тут поміщається максимально повна інформація - для документації ця назва документа, розділ, автор, версія. Для вихідних текстів це ім'я модуля, ім'я функції / методу або номери рядків, версія.
- Випуск і версія системи. Визначає місце, звідки був узятий об'єкт з виявленою проблемою. Звичайно потрібно окрема ідентифікація версії системи (а не тільки версії вихідних текстів), оскільки може виникнути плутанина з повторно виявленими проблемами. У цьому випадку, якщо проблема вже була колись виявлена розробником і потім знову з'явилася через те, що в систему потрапила не сама остання версія програмного модуля, розробник може вирішити, що йому прийшов старий звіт і проблеми насправді не існує.
- Тип звіту
    - Помилка кодування - код не відповідає вимогам.
    - Помилка проектування - тестувальник не згоден з проектною документацією.
    - Пропозиція - у тестувальника виникла ідея, як можна вдосконалити код.
    - Розбіжність з документацією - поведінка ПО не відповідає керівництва користувача або проектної документації або взагалі ніде не описано. При цьому у тестувальника немає підстав оголошувати, де саме знаходиться помилка.
    - Взаємодія з апаратурою - невірна діагностика поганого стану пристрою, помилка в інтерфейсі з пристроєм.
    - Питання - тестувальник не впевнений, що це проблема, і йому потрібна додаткова інформація.

- Ступінь важливості. Суворого критерію визначення ступеня важливості не існує, і звичайно це поле кодують від 1 (незначно) до 10 (фатально). Однак способів обгрунтованої оцінки немає - дуже складно визначити, наскільки фатальною може виявитися, наприклад, помилка в один символ в керівництві користувача.
- Суть проблеми. Короткий (не більше 2 рядків) визначення проблеми. Навіть якщо дві проблеми дуже схожі, їх опису повинні різнитися.
- Чи можна відтворити проблемну ситуацію? Відповідь - Так, Ні, Не завжди. Останнє - якщо проблема носить нерегулярний характер. Потрібно описувати, коли вона проявляється, а коли - ні (наприклад, якщо не вчасно натиснути не ту клавішу).
- Детальний опис проблеми і спосіб її відтворення. При цьому потрібні подробиці - і в описі умов відтворення, і в описі причини оголошення получившейся реакції помилкою.

Звичайний вид звіту про проблему, відповідного даній структурі, наступний:

Звіт про проблему

Порядковий номер звіту:

Автор звіту: _____ Дата створення звіту: _____

Документи / розділи, пов'язані з проблемою:

.....

Ідентифікація об'єкта / процесу, де проявляється проблема:

…….

Визначення проблеми:

…….

Автор розв'язку: _____ Дата формування рішення _____

Прийняте рішення: (можливо, посилання на змінювані компоненти / запити на зміни)

…….

Результати аналізу, що визначають, на утримання яких компонент впливає рішення:

…….

План перевірок, що відновлюють поточний стан документів розробки

…….

Оцінка прийнятого автором звіту рішення про проблему: _

…….

(0 = повністю згоден

1 = не всі аспекти проблеми враховані / дозволені

2 = основна частина проблеми залишилася невирішеною

3 = рішення не адекватно проблемі - не усуває її)

Приклад 13.2.

13.3. Трассіровочние таблиці

13.3.1. Технологічні ланцюжки і ролі учасників проекту, що використовують трассіровочние таблиці.Зв'язок трасувальних таблиць з іншими типами проектної документації

На кожному етапі життєвого циклу розробки програмної системи створюється різного роду проектна документація. Як правило, документація кожного наступного етапу створюється на базі документації попереднього етапу. Для спрощення навігації по різних документів і в т.ч. спрощення верифікації документації та самої системи часто використовуються перехресні посилання між розділами документів.

Так, наприклад, часто застосовується практика полягає в тому, що кожна вимога в документах позначається унікальним ідентифікатором - якорем (anchor). Якоря у вимогах можуть мати, наприклад, такий формат:

[ANCHOR: код]

Тут код записується у вигляді AAA_RR_NNNNNN, де AAA - тип документа (SYS, ORD тощо), RR - номер розділу верхнього рівня, в якому міститься якір, і NNNNNN - номер посилання з провідними нулями.

Вимога в такому вигляді буде виглядати наступним чином:

Для кожного вичіслімого атрибута має бути визначена роль,

від якої виробляється обчислення. [ANCHOR: SYS_02_000084].

Якщо виникає необхідність послатися на вимогу з того ж самого документа або з будь-якого

іншого, то на засланні вказується код якоря для відповідної вимоги. Так, наприклад, якщо посилання записується в тексті і має наступний формат:

[REF: код]

де код має той же формат, що і для якоря, посилання на вимоги буде мати наступний вигляд:

Для доступу до назви ролі у формулі розрахунку
значення вичіслімого атрибута
повинна використовуватися мнемоніка
[RoleName]. [ANCHOR: SRD_02_000058] [REF: SYS_02_000084].

Система якорів і посилань використовує ті ж самі ідеї, що й звичайний гіпертекст, однак, часто виникає необхідність не тільки у вказівці самого факту зв'язку між вимогами або розділами документів, а й додатково вказати тип зв'язку. Наприклад, можна виділити наступні типи зв'язків:

- звичайна гіперпосилання;
- посилання вимог нижнього рівня на вимоги верхнього рівня;
- посилання на різні варіанти одного і того ж вимоги, призначеного для різних варіантів системи (наприклад, для різних платформ).

У цьому випадку можна вказувати тип посилання поряд з кодом якоря, на який вона посилається. Однак, часто застосовується й інший метод організації типізованих посилань між документами - трассіровочние таблиці.

У загальному випадку в рядках і стовпцях трасування таблиці вказані ідентифікатори якорів, на які і з яких йде посилання, а в комірці на перетині рядків і стовпців відзначається або факт наявності посилання, або її тип.

Трассіровочние таблиці можуть використовуватися для машинного аналізу посилальної цілісності проектної документації або для швидкої навігації у великих обсягах документів.

13.3.2. Можливі форми трасувальних таблиць

Як вже було сказано в попередньому розділі, одна з форм трасувальних таблиць увазі вказівку ідентифікаторів якорів в рядках і стовпцях і типу зв'язку в осередках на їх перетині. Така таблиця буде виглядати наступним чином:

	SYS_01_0001	SYS_01_0002	SYS_01_0003	SYS_01_0003
SRD_01_0001	cross-ref			variant
SRD_01_0002	based on		variant	
SRD_01_0003	based on			variant
SRD_01_0004		cross-ref	variant	

У першому стовпці перераховані якоря вимог до програмного забезпечення, в першому рядку - якоря системних вимог. На перетині вказані типи посилань: cross-ref - звичайна інформаційна перехресне посилання; based on - вимога до ПЗ, засноване на даному системному вимозі; variant - може використовуватися або одне, або інше вимога до ПЗ в залежності від варіанту системи.

Також часто використовується більш проста форма трасувальних таблиць. У першому стовпці перераховуються якоря або номери розділів документа, який посилається на інші. У рядку - назви документів, на які йде посилання, а в осередках - якоря або номери розділів, на які йде посилання. Трассіровочние таблиця в цьому випадку буде виглядати наступним чином:

РД СВТ	**Protection**	**Protection**	**Protection Profile глави 1-4**	**Коментарі**

	Profile	**Profile глава 6**		
2.2.2 КСЗ повинен вимагати від користувачів ідентифікувати себе при запитах на доступ.	5.2.2.4 FIA_UID.1.2	286, 288	82 O.USER_AUTHENTICATION, O.USER_IDENTIFICATION, 57 Identification & Authentication, 80 P.I_AND_A, 58 Non-*bypassability*	
2.2.2 КСЗ повинен піддавати перевірці справжність ідентифікації - здійснювати аутентифікацію.	5.2.2.3 FAI_UAU.1.2	277, 280		У назві трасування тега опечатка.Правильно - FIA_UAU.1.2
2.2.2 КСЗ повинен перешкоджати доступу до ресурсів, що захищаються неідентифікованих	5.1.3.1	286, 287		5.1.3.1 - необхідні дані для аутентифікації - атрибути облікового запису користувача

користувачів та користувачів, справжність ідентифікації яких при аутентифікації не підтвердилася.				

Тут РД СВТ - документ Гостехкомиссии РФ, що включає в себе вимоги по захищеності засобів обчислювальної техніки, Protection Profile - один з аналогічних документів, що використовуються в США. Таблиця являє собою трасування вимог РД СВТ на вимоги різних глав Protection Profile. У першому рядку вказані глави, а в ячейчах - конкретні розділи.

Лекція №4

Формальні інспекції

Анотація: Лекція визначає основні підходи до організацію статичного аналізу вихідних текстів програм і документації за допомогою формальних інспекцій. Мета даної лекції: визначити основні завдання та цілі проведення формальних інспекцій, визначити етапи проведення формальної інспекції

Ключові слова: корректность , непротиворечивость , автор , список , место , инициализация , объект

15.1. Завдання і цілі проведення формальних інспекцій

Не завжди можлива розробка автоматичних або хоча б чітко формалізованих ручних тестів для перевірки функціональності програмної системи. У деяких випадках виконання тестованого програмного коду неможливо в умовах, створюваних тестовим оточенням. Така ситуація можлива у вбудованих системах, якщо програмний код призначений для обробки виняткових ситуацій, створюваних тільки на реальному обладнання.

У тих випадках, коли верифицируется не програмово код, а проектна документація на систему, яку не можна "виконати" або створити для неї окремі тестові приклади, також зазвичай вдаються до методу експертних досліджень програмного коду або документації на *коректність* або *несуперечливість*.

Такі експертні дослідження зазвичай називають інспекціями або переглядами. Існує два типи інспекцій - неформальні і формальні.

При неформальній інспекції *автор* деякого документа або частини програмної системи передає його

експерту, а той, ознайомившись з документом, передає автору *список* зауважень, які той виправляє. Сам факт проведення інспекції та зауваження, як правило, ніде окремо не зберігаються, стан виправлень за зауваженнями також ніде не відстежується.

Формальна інспекція, навпаки, є чітко керованим процесом, структура якого зазвичай чітко визначається відповідним стандартом проекту. Таким чином, всі формальні інспекції мають однакову структуру та однакові вихідні документи, які потім використовуються при розробці.

Факт початку формальної інспекції чітко фіксується у загальній базі даних проекту. Також фіксуються документи, що піддаються інспекції, і списки зауважень, відстежуються внесені за зауваженнями зміни. Цим формальна інспекція схожа на автоматизоване тестування: списки зауважень мають багато спільного з звітами про виконання тестових прикладів.

У ході формальної інспекції групою фахівців здійснюється незалежна перевірка відповідності інспектованих документів вихідним документам. Незалежність перевірки забезпечується тим, що вона здійснюється інспекторами, які не брали участь у розробці інспектується документа. Входами процесу формальної інспекції є Інспектовані документи та вихідні документи, а виходами - матеріали інспекції, що включають *список* виявлених невідповідностей та рішення про зміну статусу інспектованих документів. Рис 15.1 ілюструє *місце* формальної інспекції в процесі розробки програмних систем.

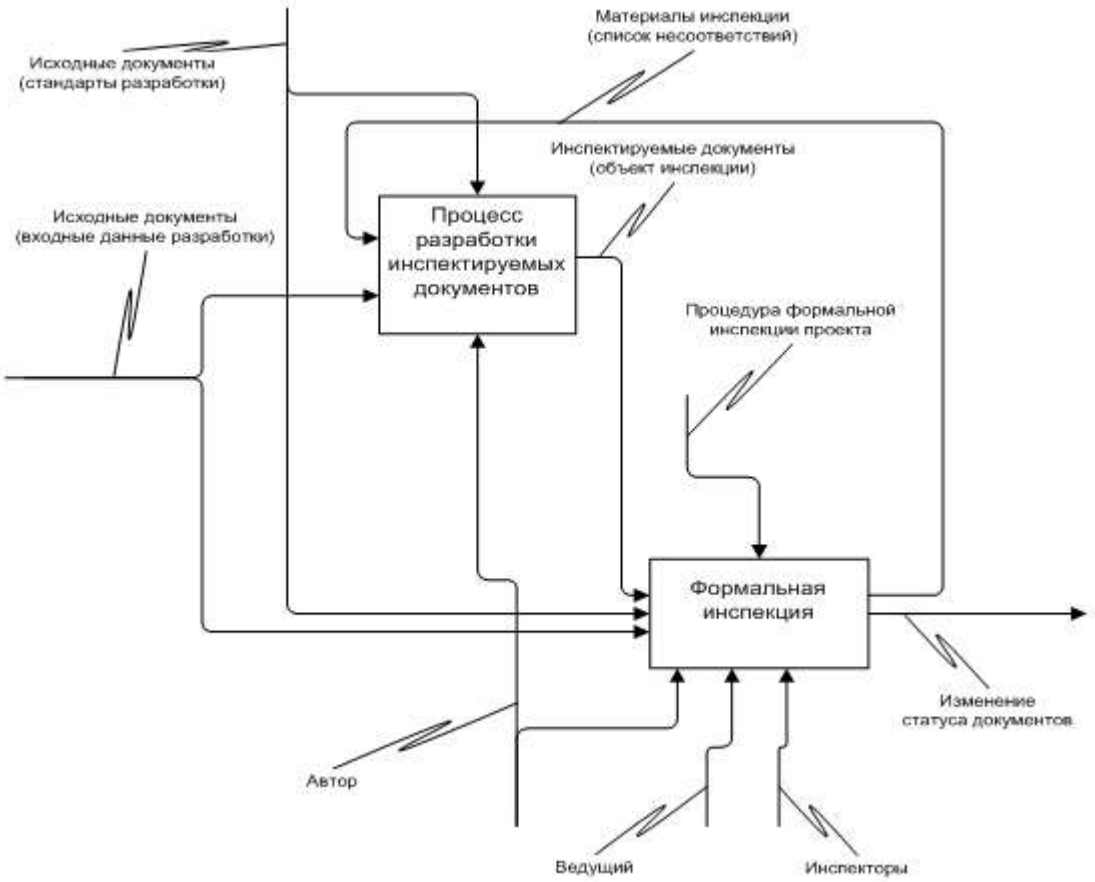


Рис. 15.1. Місце формальної інспекції у проекті з розробки програмної системи

15.2. Етапи формальної інспекції та ролі її учасників

Як правило, процес формальної інспекції складається з п'яти фаз: *ініціалізація,* планування, підготовка (експертиза), обговорення, завершення. У деяких випадках підготовку та обговорення доцільно

розглядати не як послідовні етапи, а як паралельні підпроцеси.Зокрема, така ситуація може скластися при використанні автоматизованої системи підтримки проведення формальних інспекцій.Процедура формальної інспекції проекту повинна точно описувати порядок проведення формальних інспекцій в даному проекті.

Після усунення виявлених у ході формальної інспекції невідповідностей процес формальної інспекції повторюється, можливо, в іншій формі і з іншим складом учасників. Процедура формальної інспекції повинна регламентувати можливі форми проведення повторної інспекції в залежності від обсягу та характеру змін, внесених до *об'єкт* інспекції. Як правило, допускається спрощення процесу повторної інспекції (проведення інспекції одним інспектором, відсутність фази обговорення) при внесенні в *об'єкт* інспекції незначних змін щодо раніше інспектувала версії.

15.2.1. Ініціалізація

Керівник проекту або його заступник запрошувати з бази, що зберігає всі дані проекту (наприклад, з системи управління проектно), список об'єктів, готових до інспекції, вибирає об'єкт інспекції, потім призначає учасників формальної інспекції: автора, ведучого і одного або декількох інспекторів. Ведучий також може виконувати роль інспектора; інші учасники виконують тільки одну роль. На роль ведучого або інспектора не допускається призначати співробітників, які брали участь у розробці об'єкта інспекції.

Звичайно в ролі автора виступає один з розробників об'єкта інспекції, але можливі ситуації, коли розробник недоступний - наприклад, переведений в інший проект або перебуває у відпустці. Тоді на

роль автора призначається співробітник, який буде виправляти виявлені невідповідності в інспектованих документах. При інспектуванні документів, розроблених замовником, автор може не призначатися.

Рекомендується призначати не менше двох інспекторів. Їх кількість може бути збільшена, якщо інспектуються документи, що відрізняються особливою складністю або новизною понять, а також, якщо як інспекторів залучаються співробітники з недостатнім досвідом. Однак рекомендований загальне число учасників інспекції не повинно перевищувати п'яти.

В обгрунтованих випадках процедура формальної інспекції проекту може допускати проведення інспекції єдиним інспектором, наприклад, коли об'єкт інспекції відрізняється особливою простотою і оцінювані характеристики такого об'єкта інспекції тривіальні.Прикладом такого об'єкта інспекції може служити пакет результатів збору структурного покриття, одержуваний після виконання раніше проінспектованих тестів, для якого перевіряється тільки складу пакету і узгодженість версій.

У разі, якщо проводиться повторна інспекція за скороченою формою, провідний самостійно ініціює процес повторної інспекції без участі керівника проекту. Процедура формальної інспекції проекту може дозволяти ведучому самостійно ініціювати процес повторної інспекції (у тому ж складі учасників), навіть коли вона проводиться в повній формі, якщо це диктується специфікою проекту.

15.2.2. Планування

Як тільки процес формальної інспекції був ініційований, провідний перевіряє, що Інспектовані документи розміщені в базі даних проекту, а їх статус відповідає готовності до формальної

інспекції. Якщо це не так, інспекція відкладається.

Потім ведучий повинен змінити статус інспектованих документів так, щоб відзначити факт початку інспекції та обмежити доступ до інспектується документації. Під час інспекції зміна документів неможливо, а відповідний статус зберігається до кінця інспекції. Далі будемо називати цей статус Review.

Після цього ведучий повинен скопіювати з бази даних проекту бланк інспекції та занести в нього ідентифікатори інспектованих і вихідних документів і номери їх версій, список учасників із зазначенням їх ролей і дату фактичного початку процесу інспекції, тобтотого моменту, коли Інспектовані документи були переведені в стан Review.

Ведучий повинен оцінити час, необхідний інспекторам для підготовки, і тривалість обговорення. Час, що відводиться на етап підготовки, не може бути менше однієї години. Також ведучий повинен визначити дату, час і місце обговорення, якщо воно буде проходити у формі зборів. При цьому може знадобитися узгодження з іншими учасниками інспекції. Якщо оцінка тривалості обговорення у формі зборів перевищує 2 години, то необхідно запланувати кілька зборів, кожне з яких триватиме не більше двох годин.

Процедура формальної інспекції проекту може допускати проведення повторної інспекції без зборів, якщо підсумком попередньої інспекції було рішення про проведення повторної інспекції у скороченій формі. Також допускається не проводити збори, якщо результати формальної інспекції ведуться і зберігаються в електронному вигляді. У цьому випадку процедура формальної інспекції проекту

повинна регламентувати взаємодії учасників формальної інспекції між собою. Крім того, процедура формальної інспекції проекту повинна визначати механізм підготовки, проведення обговорення та прийняття рішення.

Підготувавши бланк інспекції та визначивши час і місце зборів, ведучий повинен сповістити учасників інспекції про час і місце проведення зборів і розіслати їм підготовлений бланк інспекції.

Процедура формальної інспекції проекту може передбачати використання бланка, заповненого в ході попередньої інспекції, якщо проводиться повторна інспекція у скороченій формі і при цьому ведучий є єдиним інспектором.

15.2.3. Підготовка

Отримавши листа або призначення з прикріпленим до нього бланком інспекції, інспектори повинні винести з бази даних проекту вихідні і Інспектовані документи, використовуючи зазначені в бланку ідентифікатори та номери версій. При цьому інспектори повинні переконатися, що всі документи знаходяться у відповідному стані.

У ході підготовки інспектори детально вивчають Інспектовані документи, керуючись списком контрольних питань. Виявлені невідповідності повинні бути точно локалізовані, сформульовані і записані.

При проведенні повторної інспекції у скороченій формі дозволяється провести лише аналіз змін по відношенню до тієї версії об'єкта інспекції, яка перевірялася на попередній інспекції. Якщо при цьому виявляється, що є зміни, не пов'язані із зафіксованими зауваженнями, то процес інспекції

переривається і призначається нова інспекція в повній формі. Винятком з цього правила може бути випадок, коли такі зміни полягають у виправленні тривіальних помилок, що не зачіпають сутності інспектованих документів, таких, наприклад, як друкарські помилки в коментарях, які не впливають на зміст фрази.

Якщо повторна інспекція проводилася у скороченій формі єдиним інспектором і він вважає, що обсяг змін дуже великий або зміни занадто складні, він має право перервати процес інспекції, сповістивши керівника проекту, з тим, щоб була призначена нова інспекція в повній формі.

Автор, якщо він не є розробником об'єкта інспекції, повинен в процесі підготовки детально з ним ознайомитися, щоб бути готовим відповідати на питання інспекторів в ході обговорення, а після завершення інспекції усунути знайдені невідповідності.

15.2.4. Обговорення

Обговорення проводиться у формі одного або декількох зібрань, кожне з яких триває не більше двох годин. В один день рекомендується проводити не більше одного зібрання. Якщо обговорення не вкладається в заплановане число зібрань, то призначаються додаткові збори. Для проведення зборів необхідна присутність ведучого, хоча б одного з інспекторів і, як правило, автора. Однак, ведучий може за своїм розсудом провести збори в відсутність автора, якщо той хворий або з якоїсь іншої причини не може бути присутнім на зборах, за умови, що жоден з інспекторів не знайшов невідповідностей, або їхні зауваження очевидні і не вимагають роз'яснень з боку автора, або з автором встановлено телефонний зв'язок. Якщо збори було розпочато в відсутність автора, а надалі виникла

необхідність його присутності, ведучий повинен перервати і відкласти збори.

Збори відкладається, якщо жоден з інспекторів не готовий до обговорення. Ведучий також може на свій розсуд відкласти збори, якщо не підготувався або відсутній хоча б один з інспекторів.

У ході обговорення ведучий синхронізує роботу учасників, зачитуючи інспектується документ або послідовно називаючи розділи або абзаци тексту або елементи діаграм, або ж якимось іншим способом забезпечує синхронний перегляд документа всіма учасниками.У міру просування по документу інспектори переривають ведучого в тих місцях, до яких у них є зауваження. У разі відсутності розбіжностей провідний фіксує невідповідність і продовжує просування по документу. При інспектуванні документів невеликого обсягу ведучий, за своїм розсудом, може не синхронізувати перегляд документа всіма учасниками, а просто опитувати учасників про наявність зауважень; таке опитування може бути поєднаний із заповненням списку контрольних питань (див. нижче).

Якщо думки учасників по висловлену зауваженням розходяться, то ведучий управляє дискусією, послідовно надаючи слово всім бажаючим висловитися, причому автор користується правом позачергового надання слова. Якщо в результаті дискусії змінилося формулювання зауваження, то ведучий записує цю нову формулювання, потім зачитує її і, якщо всі учасники з нею згодні, продовжує просування по документу.

Результатом дискусії може також бути визнання відсутності проблеми. У цьому випадку ведучий переконується в тому, що всі з цим згодні, і продовжує просування по документу.

Учасники повинні прагнути позначити проблеми, але не шукати їх вирішення. Досягнення консенсусу

щодо спірних питань також не є метою дискусії. Якщо є розбіжність у думках, то повинні бути зафіксовані всі альтернативні думки. Ведучий повинен перервати дискусію, якщо оцінює її як непродуктивну.

Всі учасники зобов'язані шанобливо ставитися до опонентів, не перебивати мовця і висловлюватися тоді, коли ведучий надасть їм слово. Не допускаються паралельні обговорення вузьким складом - кожен учасник зобов'язаний адресувати свої висловлювання всього збору, а не сусідові.

Необхідно також уникати критики і оцінки кваліфікації колег. Метою інспекції є підвищення якості інспектованих документів, а не оцінка кваліфікації автора або інших учасників інспекції. Ведучий формальної інспекції не володіє перевагою перед іншими учасниками обговорення, він лише організовує цей процес і фіксує його результати в бланку інспекції.

У ході обговорення необхідно в бланку інспекції проставити відповіді на контрольні запитання та зафіксувати зауваження. Для цього ведучий послідовно зачитує контрольні питання. За відсутності у всіх інспекторів зауважень, що порушують сформульоване в питанні властивість, проти питання ставиться відмітка (галочка або хрестик) у графі "Yes" або "Так"; в іншому випадку відмітка ставиться в графі "No" або "Ні", а в графі "Зауваження" (або аналогічної) перераховуються номери відповідних зауважень, записаних в таблиці для зауважень, яка поміщена в кінці бланка інспекції.

Відмітка в графі "N / A" ("Не підходить") ставиться тільки в тому випадку, коли сформульоване у відповідному питанні властивість не може бути оцінено для даного об'єкта інспекції; в цьому випадку в графі "Зауваження" записується обгрунтування неможливості оцінити дану властивість.

Якщо при проведенні повторної інспекції використовується бланк від попередньої інспекції, в якому вже проставлені відповіді на контрольні питання, то ці відповіді не справляються, а в таблиці для зауважень проти зафіксованих раніше невідповідностей робляться відмітки про їх усунення. У разі виявлення нових невідповідностей зауваження записуються в таблицю після записаних раніше, а наступна повторна інспекція обов'язково призначається в повній формі.

Наприкінці обговорення учасники приймають рішення про можливість прийняття об'єкта інспекції в наявній версії або про необхідність внесення виправлень і проведення повторної інспекції у повній або скороченій формі. Об'єкт інспекції може бути прийнятий в наявній версії тільки за відсутності невідповідностей. Рішення про проведення повторної інспекції у скороченій формі приймається тільки в тому випадку, якщо всі учасники з цим згодні. Якщо хоча б один з учасників наполягає на повній формі повторної інспекції, то повторна інспекція повинна проводитися в повній формі. Думка провідного враховується нарівні з думками інших учасників. Прийняте рішення фіксується ведучим на бланку інспекції та завіряється підписами всіх учасників, включаючи представника служби якості, якщо він був присутній на зборах.

Теоретично можлива ситуація, коли автор не згоден ні з одним із зафіксованих зауважень, а інспектори наполягають, що невідповідності є. У такому випадку неможливо прийняти рішення про зміну статусу інспектованих документів, тому інспекція повинна бути відкладена, а вирішення проблеми винесено за рамки процесу формальної інспекції.

На бланку інспекції також фіксується тривалість зборів і час, витрачений кожним з учасників на

підготовку.

Процедура формальної інспекції проекту може допускати скасування обговорення і зборів, якщо ні у одного з інспекторів немає зауважень. Це можливо або при проведенні інспекції одним інспектором, що одночасно є провідним, або при використанні автоматизованої системи підтримки проведення формальних інспекцій, яка забезпечить ведучого інформацією про невідповідності, виявлених кожним з інспекторів, і про те, що кожен з них підтвердив завершення вивчення об'єкта інспекції. У цьому випадку заповнення бланка інспекції проводиться провідним самостійно.

15.2.5. Завершення

Наприкінці зборів, по закінченні обговорення, інспектори здають ведучому свої робочі матеріали, які включають в себе роздруківки інспектованих документів з позначками і бланки інспекції. Ведучий складає ці матеріали в прозору папку разом з примірником бланка інспекції, заповненим в ході обговорення, причому титульний лист бланка інспекції повинен лежати зверху, щоб можна було по ньому ідентифікувати папки.

Після зборів ведучий змінює статус інспектованих документів у базі даних проекту у відповідності з прийнятим рішенням - або їм присвоюється статус "Прийнятий", або "Переробити".

В останньому випадку необхідна повторна інспекція, вигляд якої уточнюється коротким коментарем.

Формальні інспекції (закінчення)

Анотація: Лекція завершує тему "Формальні інспекції" і розглядає документи, створювані в ході формальної інспекції, а також визначає особливості формальних інспекцій програмного коду та

проектної документації. Мета даної лекції: визначити основні документи, створювані в ході формальної інспекції, і вказати на основні особливості процесів формальної інспекції програмного коду та проектної документації

Ключові слова: минимум , объект , очередь , непротиворечивость , полнота , определение

16.1. Документування процесу формальної інспекції

Зазвичай, якщо підприємство веде кілька проектів з розробки програмних систем, процедура формальної інспекції регламентується у вигляді стандарту підприємства. Це дозволяє співробітникам, які беруть участь у формальних інспекціях, легко адаптуватися при переході з проекту в проект.

Однак кожен проект може мати свою специфіку. У силу цього рекомендується розробляти для кожного проекту свою процедуру формальної інспекції. Процедура формальної інспекції проекту повинна уточнювати і доповнювати цей Стандарт з урахуванням специфіки даного проекту і не повинна суперечити вимогам цього стандарту.

Процедура формальної інспекції проекту повинна точно описувати порядок проведення формальних інспекцій в даному проекті.

У процедурі формальної інспекції проекту не рекомендується дублювати загальні положення даного стандарту, за винятком окремих, особливо важливих моментів, таких, наприклад, як зміна статусу інспектованих документів.

У процедурі формальної інспекції проекту повинні бути названі всі ідентифікатори станів інспектованих документів у базі даних проекту, з якими доводиться мати справу учасникам

формальної інспекції. Як *мінімум,* повинні бути названі ідентифікатори станів, що позначають:

- готовність документа до проведення інспекції;
- проходження фаз планування, підготовки та обговорення;
- необхідність переробки документа;
- підтвердження відповідності вихідним документам.

Процедура формальної інспекції проекту повинна регламентувати мінімальну кількість інспекторів для кожного типу об'єктів інспекції, якщо в проекті інспектуються документи різного рівня складності, що вимагають участі різної кількості інспекторів.

Процедура формальної інспекції проекту повинна регламентувати можливі форми проведення повторної інспекції в залежності від обсягу та характеру змін, внесених до *об'єкт* інспекції. Як правило, допускається спрощення процесу повторної інспекції (виконання інспекції одним інспектором, відсутність фази обговорення) при внесенні в *об'єкт* інспекції незначних змін. Процедура формальної інспекції проекту може передбачати використання бланка від попередньої інспекції, якщо проводиться повторна інспекція у скороченій формі. Процедура формальної інспекції проекту може дозволяти ведучому самостійно ініціювати процес повторної інспекції (у тому ж складі учасників), навіть коли він проводиться в повній формі, якщо це диктується специфікою проекту.

16.1.1. Бланк інспекції

Бланк інспекції - основний документ, який заповнюється в ході проведення інспекцій. Зазвичай він розробляється в ході розробки стандартів проекту. Для кожного типу об'єктів інспекції у проекті має

бути розроблений свій бланк інспекції.

Бланк інспекції складається з трьох основних частин:

- титульний лист;
- список контрольних питань;
- список невідповідностей.

Крім того, рекомендується на всіх сторінках бланка, крім першої, поміщати колонтитул, що включає в себе як мінімум номер бланка інспекції.

16.1.1.1. Титульний аркуш

Титульний лист призначений для ідентифікації формальної інспекції та запису рішення і звичайно включає, як мінімум, наступні елементи:

- слова "формальна інспекція";
- ідентифікатор проекту;
- ідентифікатор типу об'єкта інспекції, наприклад, "Тест", "Стандарт проекту" тощо;
- ідентифікатор версії бланка інспекції;
- ідентифікатор конфігураційної бази даних;
- місце для запису ідентифікаторів кожного з інспектованих документів;
- місце для запису ідентифікаторів версій кожного з інспектованих документів;
- місце для запису ідентифікаторів кожного з вихідних документів;
- місце для запису ідентифікаторів версій кожного з вихідних документів;

- місце для запису дати початку інспекції;
- місце для запису фактичних дати і часу початку зборів;
- місце (таблиця) для запису прізвищ учасників інспекції із зазначенням їх ролей і місцями для підпису та запису часу, витраченого на підготовку;
- місце для запису тривалості зборів;
- місце для фіксації прийнятого рішення.

Ідентифікатор документа складається з імені файлу в базі даних проекту і повного шляху до нього. Загальні для різних документів елементи ідентифікації, такі, як шлях або ім'я бази, можуть бути винесені в окремі поля бланка.

Якщо процедурою формальної інспекції проекту передбачена можливість проведення повторної інспекції з використанням бланка від попередньої інспекції, то титульний лист також повинен включати наступні поля:

- місце для запису дати проведення повторної інспекції;
- місце для запису ідентифікаторів версій кожного з повторно інспектованих документів;
- місце для запису прізвища ведучого повторної інспекції;
- місце для запису часу, витраченого провідним на проведення повторної інспекції;
- місце для фіксації прийнятого рішення;
- місце для підпису ведучого.

Всі перераховані елементи повинні розташовуватися на одній сторінці.

16.1.1.2. Список контрольних питань

Список контрольних питань повинен бути оформлений у вигляді таблиці, що складається з наступних колонок:

- порядковий номер;
- текст питання;
- місце для позитивної відповіді ("Yes" або "Так");
- місце для негативної відповіді ("No" або "Ні");
- місце для відповіді "N / A" або "Не підходить";
- місце для посилання на невідповідність.

Контрольні питання повинні бути сформульовані таким чином, щоб позитивну відповідь означав відсутність невідповідностей.Формулювання повинні бути зрозумілими, чіткими й однозначними.

16.1.1.3. Список невідповідностей

Список невідповідностей повинен бути оформлений у вигляді незаповненою таблиці з трьома колонками:

- для порядкового номера;
- для опису невідповідності;
- для позначки про виправлення.

16.1.1.4. Колонтитул

Колонтитул повинен включати:

- ідентифікатор проекту;
- ідентифікатор версії бланка інспекції;
- місце для запису ідентифікаторів хоча б одного з інспектованих документів;
- місце для запису ідентифікаторів версій хоча б одного з інспектованих документів.

16.1.2. Життєвий цикл інспектується документа

У процесі формальної інспекції існує 2 типу документів:

- документи проекту (цільовий документ, вихідний документ, що підтримує документ);
- допоміжні документи (звіт про проведену інспекції, список контрольних питань, список виявлених проблем).

Допоміжні документи виникають в процесі інспекції і можуть змінюватися протягом процесу інспекції. Титульний аркуш створюється на стадії ініціалізації. Список виявлених проблем створюється на стадії підготовки. Список контрольних питань заповнюється на стадії обговорення. Після завершення процесу інспекції допоміжні документи поміщаються в архів і більш не підлягають зміні. Допоміжні документи зберігаються відповідно до встановлених для них термінами.

У процесі формальної інспекцію документ послідовно змінює кілька станів. У процесі розробки (до початку формальної інспекції) документ має стан Active (Активний). У цьому стані автор може звертатися до документа як для читання, так і для запису. Після того, як автор порахував, що закінчив роботу над документом, він переводить документ у стан Ready (Готовий). Це означає, що документ готовий до формальної інспекції. У стані Ready автор вже не може змінювати документ. Наступним

станом документа єReview (формальна інспекція). У цей стан документ міститься на стадії ініціалізації формальної інспекції. Переклад документа в станReview здійснює ведучий. У стані Review доступ до документа можливий тільки для читання для всіх учасників формальної інспекції. Якщо документ пройшов формальну інспекцію (не було виявлено проблем), то він переходить в стан Approved(Затверджено). Переклад документа в стан Approved здійснює ведучий. У цьому стані документ доступний тільки для читання для всіх учасників формальної інспекції, а також для інших учасників проекту. Якщо ж після формальної інспекції в цільовому документі потрібні виправлення, документ переводиться в стан Update (Переробка). У цьому стані автор має доступ до документа як для читання, так і для запису. Після переробки документа автор привласнює документу стан Ready, і процес переходу по станах повторюється доти, поки документ не буде переведений у стан Approved. Якщо в інспектується документ не вимагається вносити значних змін, то після того, як ведучий переконається в тому, що необхідні виправлення були зроблені, цільової документ може бути переведений у стан Approved.

Інспектується документ підлягає виправленню після завершення процесу інспекції. Після виправлення цільової документ може пройти повторну інспекцію. Таким чином, цільовий документ може пройти кілька послідовних інспекцій (здійснити кілька витків життєвого циклу документів у процесі формальної інспекції). Загальний життєвий цикл інспектується документа може зображений на Рис 16.1 .

Рис. 16.1. Життєвий цикл інспектується документа в процесі формальної інспекції

16.2. Формальні інспекції програмного коду

Процес формальної інспекції програмного коду підкоряється всім правилам, визначеним для абстрактної формальної інспекції, однак, має деякі особливості, пов'язані, в першу *чергу,* зі структурою інспектується програмного коду, а також з тим, що зазвичай інспектуються ділянки коду, які неможливо перевірити за допомогою автоматизованого тестування, заснованого на тестових прикладах.

16.2.1. Особливості етапу перегляду інспектується коду

Виділення ключових точок і побудова або використання таблиць трасування. Перед початком перегляду вихідного коду рекомендується відзначити пункти вимог, на відповідність яким перевіряється вихідний код, а також записати обгрунтування того, чому ці вимоги не можуть бути

перевірені в автоматичному режимі. Після цього можна переходити до перегляду власне вихідного коду. Всі позначки, які доведеться вносити в ході інспектування у вихідний код, необхідно робити не в файлі, що зберігається в базі даних проекту, а в його копії, яка потім буде підшита до матеріалів інспекції. Копія може існувати в тому ж форматі, що й вихідний файл, або може бути роздрукована на папері або виведена в формат DOC, PDF або аналогічний, що допускає коментування.

За допомогою трасувальних таблиць у вихідному коді визначаються Інспектовані функції або методи, відповідні необхідним вимогам.Ділянки коду виділяються і відзначаються міткою або номером відповідної вимоги. Якщо ділянка коду відповідає вимогам, то необхідно відзначити цей факт або кольором виділення, або відповідним текстовим приміткою. Якщо ділянка коду має проблеми, цей факт повинен бути відображений або кольором виділення, або посиланням на відповідний пункт списку зауважень в бланку інспекції.

У разі відсутності трасувальних таблиць вимог на вихідний код рекомендується робити позначки, що пояснюють, чому саме дана ділянка коду реалізує зазначені вимоги. Такі позначки допоможуть на етапі обговорення документа.

Перевірка стилю кодування. Окремим об'єктом перевірки при формальної інспекції програмного коду є стиль кодування. У більшості проектів існують стандарти, що описують правила оформлення вихідних текстів програм і файлів даних. Невірний стиль кодування не впливає на працездатність програми в цілому, але значно ускладнює супровід і підтримку змін в ході подальшого розвитку системи. Тому відхилення від стилю кодування в інспектованих ділянках коду також повинні

відзначатися в тексті і в списку зауважень. У деяких випадках проводять інспекції, цілком спрямовані на перевірку стилю кодування.

Перевірка надійності коду. Іноді рекомендується перевіряти наявність ділянок, що гарантують робастність, навіть якщо вимоги прямо не визначають необхідності обробки неприпустимих значень. У разі, якщо потенційно можлива некоректна робота програми через відсутність обробників невірних значень, рекомендується зазначити це у списку зауважень.

16.2.2. Особливості етапу проведення зборів

Розподіл ролей. У складі інспекторів бажано мати хоча б одного фахівця, що представляє собі особливості виконання інспектується коду в реальному встановленні системи. Це особливо важливо при тестуванні вбудованих систем, яке проводиться на емуляторах. Під час зборів такий фахівець може допомагати ведучому визначати послідовність розгляду зауважень у разі великої їх кількості.

Управління зборами. При проведенні зборів недоцільно зачитувати текст інспектується файлу, як це зазвичай рекомендується.Замість цього ведучому краще обмежитися перерахуванням імен функцій або методів або, у випадку, якщо в ході інспекції перевіряється відповідність вихідного коду вимогам - перерахуванням номерів або ідентифікаторів вимог. Інспектора за наявності зауважень по функції або вимозі піднімають руку і зачитують зауваження.

16.2.3. Особливості етапу завершення і повторної інспекції

Документування зборів. Для полегшення праці автора інспектується документа щодо виправлення зауважень кожне зауваження, визнане на зборах істотним, рекомендується точно трассіровать на рядки

вихідного коду і вимог.

Контроль за внесенням змін. При повторній інспекції вихідних текстів рекомендується використовувати спеціалізовані інструментальні засоби для порівняння файлів. Зміни за підсумками інспекції повинні вноситися тільки в ті ділянки, до яких були висловлені зауваження. У випадку наявності інших змін провідний вправі призначити нову інспекцію в повній формі.

16.3. Формальні інспекції проектної документації

Процес формальної інспекції проектної документації підкоряється всім правилам, визначеним для абстрактної формальної інспекції, однак, має деякі особливості, пов'язані, в першу *чергу,* з тим, що у проектній документації перевіряється її *несуперечність* і *повнота.*Точне *визначення* цих термінів у розглянутому контексті важко, тому під непротиворечивостью будемо розуміти відсутність у проектній документації вимог з протилежним змістом, згідно з якими можливо кілька абсолютно різних варіантів реалізації програмної системи. Під повнотою будемо розуміти достатність вимог для однозначної реалізації поведінки системи.

16.3.1. Особливості етапу перегляду документації

При інспектуванні вимог до системи, як правило, розглядається як "зовнішня", так і "внутрішня" інформація, що стосується даного документа. Під зовнішньою інформацією розуміється, в першу чергу, суть технічних рішень, прийнятих при розробці системи, ті принципи, які відрізняють її від інших систем. При цьому перевіряється узгодженість вимог з цими принципами. Під внутрішньою інформацією розуміється передусім внутрішня цілісність і несуперечність документа - властивості, які

дозволяють розробляти програмний код, позбавлений двозначностей і нестабільних ділянок. Головне питання, на яке повинен відповісти інспектор під час перевірки внутрішніх властивостей документа: "Чи визначені вимоги так, щоб колектив розробників зміг працювати з ним?" або, інакше, "Ці вимоги недвозначні, сповнені і авторитетно виражені?". Процес інспекції може допомогти відповісти на ці питання, а список контрольних питань, представлений нижче, може бути використаний при проведенні інспекцій.

1. Чи є кожна вимога абсолютно недвозначним? (Якщо вимога прочитати підряд кілька разів, роблячи наголос спочатку на першому слові, потім - на другому, потім - на третьому і т.д., чи буде при цьому змінюватися сенс цієї вимоги?)
2. Чи існує для кожного з встановлених вимог деякий компетентний фахівець, який зможе сказати після завершення розробки, чи виконано дана вимога чи ні? Визначено чи метод вирішення цієї проблеми в документації за вимогами?
3. Чи існує будь невстановлені або відсутні вимоги?
4. Чи існують серед заданих такі вимоги, які не є необхідними?
5. Якщо існують які-небудь конфліктуючі вимоги, чи відомо зрозуміле вирішальне правило, в яких ситуаціях якої вимоги слід віддавати перевагу?

16.3.2. Особливості етапу завершення

Вплив неузгодженості документації на процес розробки. Трасування змін на програмний код. Первинна інспекція проектної документації, як правило, проводиться тоді, коли сама програмна

система ще не написана. Однак при проведенні інспекції змін у вимогах до вже працюючій системі (наприклад, при оновленні її версії) може знадобитися комплексна одночасна інспекція документації і створеного на її основі програмного коду. При цьому може виникнути ситуація, коли зміни, які потрібно внести в документацію за результатами інспекції, вимагають відповідної зміни в програмному коді. Вирішення цієї проблеми досягається використанням трасувальних таблиць.

Дещо інша ситуація виникає у разі, коли комплексна інспекція проводиться не після зміни вимог, а після завершення всього ланцюжка змін - після зміни функціональних вимог, архітектури, низькорівневих вимог і програмного коду. У цьому випадку при виявленні суперечливих вимог необхідно виявити всі частини програмної системи, які реалізують ці вимоги. У разі, якщо розробка цих частин виконувалася різними людьми, могла відрізнятися і трактування суперечливих вимог. У цьому випадку ліквідація суперечливості може спричинити за собою "хвилю змін" у проектній документації та вихідних текстах системи. Для того, щоб уникнути "хвиль змін" по завершенню інспекцій, рекомендується проводити її поетапно до початку наступного етапу життєвого циклу або до розробки документів наступного рівня деталізації.

Лекція №5

Модульне тестування

Анотація: Лекція є першою з трьох розглядають рівні процесу верифікації. Тема даної лекції - процес модульного тестування, його завдання і цілі. Визначається поняття модуля і його кордонів, визначаються підходи до проектування тестового оточення при модульному тестуванні. Розглядаються організаційні аспекти модульного тестування. Мета даної лекції: дати уявлення про процес модульного тестування, його технічної і організаційної складових

Ключові слова: циклу , системні требования , архитектура , разбиение , модуль , ПО , драйвер , заглушки , интеграция ,протоколирование , development , механизмы , модульное тестирование , определение , компонент , единица , класс ,инкапсуляция даних , поділ , декомпозиція , приховування даних , локалізація , стан системы , сценарий , работ , анализ , группа ,IEEE , график , тестировщик , функциональные вимоги , список , testing design , інформація

18.1. Рівні процесу верифікації

Як вже було сказано раніше, процес верифікації активний протягом практично всього життєвого *циклу* системи і працює паралельно з процесом розробки. Розробка системи, як правило, йде на різних рівнях: спочатку розробляється концепція системи, *системні вимоги,* потім *архітектура* системи, її *розбиття* на модулі, потім розробляються окремі модулі. Послідовність цих рівнів залежить від типу життєвого *циклу,* але їх склад практично

завжди однаковий. Процес верифікації також розбивається на окремі рівні:

- *системне тестування,* в ході якого тестується система в цілому;
- *інтеграційне тестування,* в ході якого тестуються групи взаємодіючих модулів і компонент системи;
- *модульне тестування,* в ході якого тестуються окремі компоненти.

Технічні аспекти методів розробки і проведення тестування на кожному з трьох рівнів були розглянуті в попередніх лекціях - на кожному з рівнів розробляються тестове оточення, автоматизовані тести, проводяться формальні інспекції. Однак, кожен з цих трьох рівнів має свої організаційні особливості, розгляду яких будуть присвячені наступні три лекції, починаючи з даною.

18.2. Завдання і цілі модульного тестування

Кожна складна програмна система складається з окремих частин - модулів, що виконують ту чи іншу функцію в складі системи. Для того, щоб упевнитися в коректній роботі всієї системи, необхідно спочатку протестувати кожен *модуль* системи *окремо*. У разі виникнення проблем при тестуванні системи в цілому це дозволяє простіше виявити модулі, що викликали проблему, і усунути відповідні дефекти в них. Таке тестування модулів *окремо* отримало називання *модульного тестування (unit testing)*.

Для кожного модуля, що піддається тестуванню, розробляється тестове оточення, що включає в себе *драйвер* і *заглушки,*готуються тест-вимоги і тест-плани, описують конкретні тестові приклади.

Основна мета модульного тестування - впевнитись у відповідності вимогам кожного окремого модуля системи перед тим, як буде проведена його *інтеграція* до складу системи.

При цьому в ході модульного тестування вирішуються такі основні завдання [18]:

Пошук і документування невідповідностей вимогам

Підтримка розробки та рефакторинга низкоуровневой архітектури системи і межмодульного взаємодії

Підтримка рефакторинга модулів

Підтримка усунення дефектів і налагодження

Перше завдання - класична завдання тестування, що включає в себе не тільки розробку тестового оточення і тестових прикладів, а й виконання тестів, *протоколювання* результатів виконання, складання звітів про проблеми.

Друге завдання більше властива "легким" методологіям типу XP, де застосовується принцип тестування перед розробкою (Test-driven *development),* при якому основним джерелом вимог для програмного модуля є тест, написаний до реалізації самого модуля.Однак, навіть при класичній схемі тестування модульні тести можуть виявити проблеми в дизайні системи та нелогічні або заплутані*механізми* роботи з модулем.

Третє завдання пов'язана з підтримкою процесу зміни системи. Досить часто в ході розробки потрібно проводити рефакторінг модулів або їх груп - оптимізацію або повну переробку програмного коду з метою підвищення його сопровождаемости, швидкості роботи або надійності. Модульні тести при цьому є потужним інструментом для перевірки того, що новий

варіант програмного коду виконує ті ж функції, що і старий.

Остання, четверта, завдання пов'язана зі зворотним зв'язком, яку отримують розробники від тестувальників у вигляді звітів про проблеми. Детальні звіти про проблеми, складені на етапі модульного тестування, дозволяють локалізувати і усунути багато дефекти в програмній системі на ранніх стадіях її розробки або розробки її нової функціональності.

В силу того, що модулі, що піддаються тестуванню, звичайно невеликі *за* розміром, *модульне тестування* вважається найбільш простим (хоча і досить трудомістким) етапом тестування системи. Однак, незважаючи на зовнішню простоту, з модульним тестуванням сполучені дві проблеми.

Перша з них пов'язана з тим, що не існує єдиних принципів визначення того, що в точності є окремим модулем.

Друга полягає у відмінностях у трактуванні самого поняття модульного тестування - чи розуміється під ним відокремлений тестування модуля, робота якого підтримується тільки тестовим оточенням, чи мова йде про перевірку коректності роботи модуля в складі вже розробленої системи. Останнім часом термін "*модульне тестування*"частіше використовується у другому сенсі, хоча в цьому випадку мова скоріше йде про інтеграційний тестуванні.

Ці дві проблеми розглянуті в двох наступних розділах.

18.3. Поняття модуля і його кордонів. Тестування класів

Традиційне *визначення* модуля з точки зору його тестування: "*модуль* -

це *компонент* мінімального розміру, який може бути незалежно протестований в ході верифікації програмної системи ". У реальності часто виникають проблеми з тим, що вважати модулем. Існує кілька підходів до даного питання:

модуль - це частина програмного коду, що виконує одну функцію з точки зору функціональних вимог;

модуль - це програмний модуль, тобто мінімальний компільований елемент програмної системи;

модуль - це завдання у списку завдань проекту (з точки зору його менеджера);

модуль - це ділянка коду, який може вміститися на одному екрані або одному аркуші паперу;

модуль - це один клас або їх безліч з єдиним інтерфейсом.

Зазвичай за тестований *модуль* приймається або програмний *модуль* (*одиниця* компіляції) у разі, якщо система розробляється на процедурному мовою програмування, або *клас,* якщо система розробляється на об'єктно-орієнтованої мови.

У випадку систем, написаних на процедурних мовах, процес тестування модуля відбувається так, як це було розглянуто в попередніх лекціях: для кожного модуля розробляється тестовий *драйвер,* що викликає функції модуля і збирає результати їх роботи, і набір заглушок - вони імітують поведінку функцій, які містяться в інших модулях, що не потрапляють під тестування даного модуля. При тестуванні об'єктно-орієнтованих систем існує ряд особливостей, насамперед викликаних *инкапсуляцией даних* і методів в класах.

У разі об'єктно-орієнтованих систем більш дрібне *поділ* класів і використання окремих методів

в якості тестованих модулів недоцільно, оскільки для тестування кожного методу потрібно розробка тестового оточення, порівнянного *по* складності з вже написаним програмним кодом класу. Крім того, *декомпозиція* класу порушує принцип інкапсуляції, згідно з яким об'єкти кожного класу повинні вести себе як єдине ціле з точки зору інших об'єктів.

Процес тестування класів як модулів іноді називають компонентним тестуванням. У ході такого тестування перевіряється взаємодія методів усередині класу і правильність доступу методів до внутрішніх даних класу. Тут можливе виявлення не тільки стандартних дефектів, пов'язаних з виходами за межі діапазону або невірно реалізованими вимогами, але і специфічних дефектів об'єктно-орієнтованого програмного забезпечення:

дефектів інкапсуляції, в результаті яких, наприклад, *приховані дані* класу виявляється недоступними для відповідних публічних методів;

дефектів спадкування, за наявності яких схема успадкування блокує важливі дані або методи від класів-нащадків;

дефектів поліморфізму, при яких поліморфну поведінку класу виявляється поширеним не на всі можливі класи;

дефектів інстанціювання, при яких в новостворюваних об'єктах класу не встановлюються коректні значення за замовчуванням параметрів і внутрішніх даних класу.

Однак, згідно [10], вибір класу в якості модуля, що тестується має і ряд сполучених проблем.

- **Визначення ступеня повноти тестування класу.** У тому випадку, якщо в якості модуля, що тестується вибраний клас, не зовсім ясно, як визначати ступінь повноти його

тестування. З одного боку, можна використовувати класичний критерій повноти покриття програмного коду тестами: якщо повністю виконані всі структурні елементи всіх методів, як публічних, так і прихованих, то тести можна вважати повними.

Однак існує альтернативний підхід до тестування класу, в якому всі публічні методи повинні надавати користувачеві даного класу узгоджену схему роботи - тоді достатньо перевірити типові коректні і некоректні сценарії роботи з даним класом. Тобто, наприклад, в класі, об'єкти якого представляють записи в телефонній книжці, одним з типових сценаріїв роботи буде "Створити запис → шукати запис і знайти її → видалити запис → шукати запис вдруге і отримати повідомлення про помилку ".

Відмінності в цих двох методах нагадують відмінності між тестуванням чорного і білого ящиків, але, насправді, другий підхід відрізняється від чорного ящика тим, що функціональні вимоги на систему можуть бути складені на рівні вищому, ніж окремі класи, і встановлення адекватності тестових сценаріїв вимогам залишається на відкуп тестувальника.

- **Протоколювання станів об'єктів та їх змін.** Деякі методи класу призначені не для видачі інформації користувачеві, а для зміни внутрішніх даних об'єкта класу. Значення внутрішніх даних об'єкта визначає його стан у кожний окремий момент часу, а виклик методів, що змінюють дані, змінює і стан об'єкта. При тестуванні класів необхідно перевіряти, що клас адекватно реагує на зовнішні виклики в будь-якому з станів. Однак, найчастіше через інкапсуляції даних неможливо визначити внутрішній стан класу програмними способами всередині драйвера.

У цьому випадку може допомогти складання схеми поведінки об'єкта як кінцевого автомата з певним набором станів. Така схема може входити в низкоуровневую проектну документацію (наприклад, у складі опису архітектури системи), а може складатися тестувальником або розробником на основі функціональних вимог до системи. В останньому випадку для визначення всіх можливих станів може знадобитися ручний аналіз програмного коду і визначення його відповідності вимогам.Автоматизоване тестування в цьому випадку може лише визначити, чи за всіма виявленими станам здійснювалися переходи і чи всі можливі реакції перевірялися.

- **Тестування змін.** Як вже згадувалося вище, модульні тести - потужний інструмент перевірки коректності змін, внесених у вихідний код при рефакторінгу. Однак, в результаті рефакторинга тільки одного класу, як правило, не змінюється його зовнішній інтерфейс з іншими класами (інтерфейси змінюються при рефакторінгу відразу декількох класів). В результаті звичайних еволюційних змін системи у класу може змінюватися зовнішній інтерфейс, причому як за формальними (змінюються імена і склад методів, їх параметри), так і за функціональними (при збереженні зовнішнього інтерфейсу змінюється логіка роботи методів) ознаками. Для проведення модульного тестування класу після таких змін буде потрібно зміна драйвера і, можливо, заглушок.Але тільки модульного тестування в цьому випадку недостатньо, необхідно також проводити і інтеграційне тестування даного класу разом з усіма класами, які пов'язані з ним за даними або з управління.

Незалежно від того, на які модулі, що піддаються тестуванню, розбивається система, рекомендується викласти принципи виділення тестованих модулів в плані і стратегії тестування, а також скласти на базі структурної схеми архітектури системи нову структурну схему, на якій потрібно відзначити всі тестовані модулі. Це дозволить спрогнозувати склад і складність драйверів і заглушок, необхідних для модульного тестування системи. Така схема також може використовуватися пізніше на етапі модульного тестування для виділення укрупнених груп модулів, що піддаються інтеграції.

18.4. Підходи до проектування тестового оточення

Незалежно від того, яка мінімальна *одиниця* вихідних кодів системи прийнята за мінімальний тестований *модуль,* існує ще одна відмінність у підходах до модульного тестування.

Перший підхід до модульного тестування грунтується на припущенні, що функціональність кожного знову розробленого модуля повинна перевірятися в автономному режимі без його інтеграції з системою. При такому підході для кожного знову розроблюваного модуля створюються тестовий *драйвер* і *заглушки,* за допомогою яких виконується набір тестів. Тільки після усунення всіх дефектів в автономному режимі виробляється *інтеграція* модуля в систему і проводиться тестування на наступному рівні. Перевагою даного підходу є більш проста *локалізація* помилок в модулі, оскільки при автономному тестуванні виключається вплив інших частин системи, яке може викликати маскування дефектів (ефект парного числа помилок). Основний недолік даного методу - підвищена трудомісткість написання драйверів і заглушок, оскільки *заглушки* повинні адекватно моделювати поведінку системи в різних

ситуаціях, а *драйвер* повинен не тільки створювати тестове оточення, а й імітувати внутрішнє *стан системи,* в складі якої буде функціонувати *модуль.*

Другий підхід побудований на припущенні, що *модуль* все одно працює в складі системи і якщо модулі інтегрувати в систему *по*одному, то можна протестувати поведінку модуля в складі всієї системи. Цей підхід властивий більшості сучасних "полегшених" методологій розробки, в тому числі і XP.

В результаті застосування такого підходу різко скорочуються трудовитрати на розробку заглушок і драйверів - в ролі заглушок виступає вже відтестувати частина системи, а *драйвер* виконує тільки функції передачі і прийому даних, що не моделюючи внутрішнє *стан системи.*

Проте, при використанні даного методу зростає складність написання тестових прикладів - для приведення системи в потрібне *стан системи* заглушок, як правило, потрібно тільки встановити значення тестових змінних, а для приведення в потрібний стан частини реальної системи необхідно виконати *сценарій,* що приводить у цей стан. Кожен тестовий приклад у цьому випадку повинен містити такий *сценарій.*

Крім того, при цьому підході не завжди вдається локалізувати помилки, приховані всередині модуля, які можуть виявитися при інтеграції наступних модулів.

18.5. Організація модульного тестування

Модульне тестування, з точки зору тестувальника, - це комплекс *робіт з* виявлення дефектів у тестованих модулях. У ці роботи включається *аналіз* вимог, розробка тест-вимог і тест-планів,

розробка тестового оточення, виконання тестів, збір інформації про їх проходження.

Однак, з точки зору керівника групи тестування (або з точки зору керівника проекту, якщо в ньому не виділена окрема *група*тестування), *модульне тестування* є більш широким поняттям. Для того, щоб процес модульного тестування міг функціонувати спільно з іншими процесами розробки, він повинен включати в себе кілька фаз: планування процесу, розробку тестів, виконання тестів, збір статистики, управління звітами про виявлені дефекти.

Відповідно до стандарту *IEEE* 1008 [19] процес модульного тестування складається з трьох фаз, до складу яких входить 8 видів діяльності (етапів).

- Фаза планування тестування
    1. Етап планування основних підходів до тестування, ресурсне планування і календарне планування
    2. Етап визначення властивостей, що підлягають тестуванню
    3. Етап уточнення основного плану, сформованого на етапі (1)
- Фаза отримання набору тестів
    1. Етап розробки набору тестів
    2. Етап реалізації уточненого плану
- Фаза вимірювань модуля, що тестується
    1. Етап виконання тестових процедур
    2. Етап визначення достатності тестування
    3. Етап оцінки результатів тестування та тестової модуля.

Під час етапу планування основних підходів у якості вхідних даних використовується загальний план проекту (*модульне тестування,*як частина проектних *робіт,* має укладатися в загальний *графік)* і вимоги до системи (для оцінки трудомісткості *робіт* і будь-якого планування необхідно проводити *аналіз* складності системи на підставі вимог до неї).

Основні завдання, які вирішуються в ході етапу планування, включають в себе:

- **визначення загального підходу до тестування модулів** - визначаються ризики і на їх основі - ступінь повноти і охоплення тестування системи. Визначаються джерела вхідних і вихідних даних. Визначаються технології перевірки результатів тестування та формати запису даних про проведене тестуванні. Описується зовнішній інтерфейс тестованих модулів та їх інформаційне оточення;
- **визначення вимог до повноти тестування** - визначається необхідна ступінь покриття програмного коду різних ділянок тестованого модуля, визначається підходи до класів еквівалентності (чи потрібне тестування за кордонами діапазону);
- **визначення вимог до завершення тестування** - визначаються умови, перевірка яких дозволяє стверджувати, що тестування модуля завершено, і умови, за яких подальше тестування модуля вважається неможливим до його зміни і доопрацювання.Прикладом таких умов може служити досягнення певного рівня покриття вихідного коду тестами і неможливість компіляції модуля відповідно;
- **визначення вимог до ресурсів** - для розробки і виконання тестів, а також для аналізу результатів тестування необхідні ресурси - як технічні (комп'ютери та програмне

забезпечення), так і людські (тестувальники). При вирішенні цього завдання необхідно вказувати вимоги до програмного і апаратного забезпечення, вимоги до необхідної кваліфікації людей, а також має визначатися необхідне для проведення кількість ресурсів і час їх зайнятості;

- **визначення загального плану-графіка робіт** - на підставі загального плану проекту складається план робіт за модульним тестування. Основний критерій початку робіт з тестування - готовність модулів, тобто загальний план робіт з тестування узгоджується по датах початку робіт з датами закінчення робіт загального плану розробки.

Після завершення етапу планування починається етап визначення властивостей системи, що підлягають тестуванню.

Основні завдання, які вирішуються в ході діяльності *за* визначенням властивостей системи, що підлягають тестуванню, включають в себе:

- **вивчення функціональних вимог** - визначення тестопригодності вимог, при необхідності запитується уточнення вимог;
- **визначення додаткових вимог і пов'язаних процедур** - визначення вимог, які не потрапляють під функціональні вимоги, але можуть бути протестовані на рівні модульного тестування (наприклад, це можуть бути вимоги до продуктивності системи, що входять до складу системних вимог);
- **визначення станів тестованого модуля** - якщо тестований модуль може бути представлений у вигляді кінцевого автомата з певним набором станів, то кожне стан має

бути ідентифіковано, а також повинні бути виділені всі вимоги, що відносяться до цього стану;

- **визначення характеристик вхідних і вихідних даних** - для всіх даних, які надходять в модуль, а також виходять з нього, повинні бути визначені формати, частота надходження, допустимі значення тощо;
- **вибір елементів, що піддаються тестуванню** - у разі, коли не може застосовуватися повне тестування, необхідно вибрати елементи модуля, що тестується, які будуть піддаватися тестуванню. Основне джерело інформації тут - дані про ризики, проаналізовані на рівні структури вихідного коду модуля, що тестується. Для тестування в першу чергу повинні відбиратися елементи з максимальним ступенем ризику.

І, нарешті, на завершення фази планування проводиться уточнення основного плану - уточнюється загальний підхід до тестування, формулюються спеціальні та додаткові вимоги до ресурсів, складається детальний план- *графік робіт.*

По завершенню цих етапів фаза планування вважається закінченою і починається фаза розробки тестів. При цьому процес розробки тестів підкоряється тим планам і вимогам, які були створені на попередньому етапі. Таким чином, якщо на першому етапі основну роль виконував керівник групи тестування, то на другому етапі основну роль починає грати *тестувальник,* діючий в згоді з вказівками керівника.

Фаза розробки тестів починається з власне розробки набору тестів, який буде використаний для тестування модуля. Основні документи, які використовуються на цьому етапі: *функціональні*

вимоги до модуля, *архітектура* модуля, *список* елементів, що піддаються тестуванню, план-*графік робіт,* визначення тестових прикладів від попередньої версії модуля (якщо вони існували) і результати тестування минулій версії (якщо вони існували).

У ході цього етапу повинні бути вирішені наступні завдання:

- **розробка архітектури тестового набору** - під тестовим набором тут розуміється не набір конкретних тестових прикладів, а загальна структура системи тестів для перевірки функціональності модуля, що тестується. Організація тестів в такій системі як правило відображає структуру функціональних вимог і часто являє собою ієрархію, на кожному рівні якої визначається свій набір тестів;
- **розробка явних тестових процедур (тест-вимог)** - у випадку досить докладних функціональних вимог і чітко прописаної концепції розробки тестів явні тестові процедури можуть і не розроблятися. Однак, за наявності необхідних ресурсів, розробка тест-вимог дозволить більш чітко інтерпретувати піддаються тестуванню функціональні вимоги - тест-вимоги знижують ризик неоднозначного трактування функціональних вимог;
- **розробка тестових прикладів** - тестові приклади повинні відповідати вимогам до повноти тестування і складатися або на базі тест-вимог, або на підставі функціональних вимог. Даний вид діяльності найбільш тривалий у часі;
- **розробка тестових прикладів, заснованих на архітектурі (у разі необхідності)** - деякі тестові приклади грунтуються не на функціональних вимогах, а на особливостях

архітектури модуля, що тестується. Для розробки тестових прикладів, заснованих на архітектурі, необхідно використовувати підхід скляного ящика. Ці тестові приклади пишуться або на основі низькорівневих вимог до системи, або на основі низькорівневих тест-вимог, якщо вони розроблялися на одному з попередніх етапів;

- **складання специфікації тестових прикладів** - результатом діяльності тестувальника в ході даного етапу складається документ *Test Design* Specification (формат якого описаний в стандарті IEEE 829 [15]).

На наступному етапі проводиться реалізація тестів (наприклад, у вигляді тестового оточення і формалізованих описів тестових прикладів). У ході цього етапу формуються тестові набори даних, які використовуються в тестових прикладах, створюється тестове оточення, і також здійснюється *інтеграція* тестового оточення з тестованим модулем.

Після того, як всі тести реалізовані, вони виконуються на тестовому стенді в ручному або автоматичному режимі. Незалежно від виду тестування в ході цього етапу вирішуються два завдання: виконання тестових прикладів, та збір і *аналіз* результатів тестування.

Збору підлягає наступна *інформація:*

- результат виконання кожного тестового прикладу (пройшов / не пройшов);
- інформація про інформаційний оточенні системи у випадку, якщо тест не пройшов;
- інформація про ресурси, які потрібні були для виконання тестового прикладу.

За результатами аналізу цієї інформації складаються запити на зміну проектної документації, програмного коду модуля, що тестується або тестового оточення.

Етапи розробки (доопрацювання), реалізації та виконання тестів тривають до тих пір, поки не буде досягнутий критерій завершення модульного тестування. Прикладом такого критерію може служити відсутність не пройшли тестових прикладів при 75% покритті рядків вихідного коду.

Після припинення тестування виконуються роботи *за* оцінкою проведеного тестування, в ході яких:

- описуються відмінності реального процесу тестування від запланованого;
- відмінності поведінки тестованого модуля від описаного у вимогах (з метою подальшої корекції вимог);
- складається загальний звіт про проходження тестів, що включає в себе і інформацію про покриття.

На завершення модульного тестування необхідно перевірити, що всі створені в його ході артефакти - документи, програмний код, файли звітів і даних - поміщені в базу даних проекту, яка зберігає всі дані, використовувані і створювані в процесі розробки програмної системи.

19.2. Поняття покриття

19.2.1. Покриття програмного коду

На цьому семінарі познайомимося з однією з оцінок якості системи тестів - з її повнотою, тобто величиною тієї частини функціональності системи, яка перевіряється тестовими прикладами. Повна система тестів дозволяє стверджувати, що система реалізує всю функціональність, зазначену у вимогах, і, що ще більш важливо - не реалізує жодної іншої

функціональності. Ступінь покриття програмного коду тестами - важливий кількісний показник, що дозволяє оцінити якість системи тестів, а в деяких випадках - і якість тестируемой програмної системи.

Одним з найбільш часто використовуваних методів визначення повноти системи тестів є визначення відношення кількості тест-вимог, для яких існують тестові приклади, до загальної кількості тест-вимог, - тобто в даному випадку мова йде про покриття тестовими прикладами тест-вимог. В якості одиниці вимірювання ступеня покриття тут виступає відсоток тест-вимог, для яких існують тестові приклади, званий відсотком покритих тест-вимог. Покриття вимог дозволяє оцінити ступінь повноти системи тестів по відношенню до функціональності системи, але не дозволяє оцінити повноту стосовно її програмної реалізації. Одна і та ж функція може бути реалізована за допомогою зовсім різних алгоритмів, що вимагають різного підходу до організації тестування.

Для більш детальної оцінки повноти системи тестів при тестуванні скляного ящика аналізується покриття програмного коду, зване також структурним покриттям.

Під час роботи кожного тестового прикладу виконується деякий ділянку програмного коду системи, при виконанні всієї системи тестів виконуються всі ділянки програмного коду, які задіє ця система тестів. У разі, якщо існують ділянки програмного коду, що не виконані при виконанні системи тестів, система тестів потенційно неповна (тобто не перевіряє всю функціональність системи), або система містить ділянки захисного коду або не використовуваний код (наприклад, "закладки" або заділ на майбутнє використання

системи). Таким чином, відсутність покриття яких ділянок коду є сигналом до переробки тестів або коду (а іноді - і вимог).

До аналізу покриття програмного коду можна приступати тільки після повного покриття вимог. Повне покриття програмного коду не гарантує того, що тести перевіряють всі вимоги до системи. Одна з типових помилок починаючого тестувальника - починати з покриття коду, забуваючи про покриття вимог.

Зауваження. Необхідно пам'ятати, що розробка тестових прикладів, які забезпечують повне покриття тестованого програмного коду, відноситься до структурного тестування коду. Перед початком структурного тестування має бути повністю закінчено функціональне тестування коду як чорного ящика (чим ми і займалися на попередніх семінарах). Тільки після цього можна переходити до поліпшення покриття. В ідеальному випадку при повному покритті функціональних вимог мусить виходити 100% покриття коду. Однак на практиці таке відбувається тільки у випадку дуже простого коду. Причина недопокритія коду при повному покритті вимог - або неповнота вимог, або недостатньо повний аналіз вимог тестувальником. У першому випадку звичайно потрібна доробка вимог, у другому - тест-вимог і тест-плану.

19.2.2. Рівні покриття

19.2.2.1. За рядками програмного коду (Statement Coverage)

Для забезпечення повного покриття програмного коду на даному рівні необхідно, щоб в результаті виконання тестів кожен оператор був виконаний хоча б один раз. Перед початком тестування необхідно виділити змінні, від яких залежить виконання різних гілок умов і циклів

в коді - керуючі вхідні змінні. Зміна значень цих змінних буде впливати на те, які рядки коду будуть виконуватися в різних тестових прикладах.

Приклад. У наступному фрагменті коду вхідними змінними є prev і ShowMessage.

```
 if (prev == "оператор" | | prev == "унарний оператор")
{
   if (ShowMessage)
   {
      MessageBox.Show ("Два поспіль оператора на
        "+ I.ToString () +" символі. ");
   }
   else
   {
       Log.Write ("Два поспіль оператора на" +
              + I.ToString () + "символі.")
   }
   Program.res = 4;
   return "& Error 04 at" +
     + I.ToString ();
}
```

Для того, щоб повністю покрити даний код по рядках, тобто виконати всі рядки коду,

достатньо двох тестових прикладів:

	1	**2**
prev	оператор	оператор
ShowMessage	true	false

У першому тестовому прикладі здійснюється вхід в перший умовний оператор if, потім в першу гілку другого оператора if.Другий тестовий приклад здійснює аналогічні дії, але для другої гілки другого оператора if. У результаті всі рядки коду виявляються виконаними. Легко побачити, що, незважаючи на повне покриття по рядках, цей набір тестових прикладів не перевіряє всієї функціональності (навіть не бачачи вимог, логічно припустити, що в них має описуватися поведінку системи і для значення змінної prev = "оператор", і для значення prev = " *унарний оператор* ").

Також проблеми цього методу покриття можна побачити і на прикладах інших керуючих структур. Наприклад, проблеми виникають при перевірці циклів do ... while - при даному рівні покриття досить виконання циклу тільки один раз, при цьому метод абсолютно нечутливий до логічним операторам | | і &&.

Іншою особливістю даного методу є залежність рівня покриття від структури програмного коду. На практиці часто не потрібно 100% покриття програмного коду, замість цього встановлюється допустимий рівень покриття, наприклад 75%. Проблеми можуть виникнути при покритті наступного фрагмента програмного коду:

```
if (condition)
  MethodA ();
else
  MethodB ();
```

Якщо MethodA () містить 99 операторів, а MethodB () - один оператор, то єдиного тестового прикладу, встановлюєcondition в true, буде достатньо для досягнення необхідного рівня покриття. При цьому аналогічний тестовий приклад, який встановлює значення condition в false, дасть занадто низький рівень покриття.

19.2.2.2. По гілках умовних операторів (Decision Coverage)

Для забезпечення повного покриття по даному методу кожна точка входу і виходу в програмі і в усіх її функціях повинна бути виконана принаймні один раз і всі логічні вирази в програмі повинні прийняти кожне з можливих значень хоча б один раз; таким чином, для покриття по гілках потрібно як мінімум два тестових прикладу.

Також даний метод називають: *branch coverage*, all-edges *coverage*, basis path *coverage*, DC, C2, decision-decision-path.

На відміну від попереднього рівня покриття даний метод враховує покриття умовних операторів з порожніми гілками.

Приклад. Для покриття попереднього прикладу коду по гілках потрібно вже три тестових прикладу. Це пов'язано з тим, що перший умовний оператор if має неявну гілка - порожню гілку else. Для забезпечення покриття по гілках необхідно покривати і порожні гілки.

	1	2	3
prev	оператор	оператор	операнд
ShowMessage	true	false	true

Перші два тестових прикладу аналогічні попередньому нагоди, третій призначений для покриття неявній гілки. При цьому треба зауважити, що значення змінної ShowMessage не грає ніякої ролі для покриття - ділянка коду, що використовує цю змінну, просто не виконується.

Особливість даного рівня покриття полягає в тому, що на ньому не враховуються логічні вирази, значення компонент яких виходять викликом методів. Наприклад, на наступному фрагменті програмного коду

```
 if (condition1 && (condition2 | | Method ()))
    statement1;
 else
    statement2;
```

повне покриття по гілках може бути досягнуто за допомогою двох тестових прикладів:

	1	2
condition1	true	false
condition2	true	true / false

В обох випадках не відбувається виклику методу Method (), хоча покриття даної ділянки коду буде повним. Для перевірки виклику методу Method () необхідно додати ще один тестовий

приклад (який, проте, не покращує ступеня покриття по гілках):

	1	**2**	**3**
condition1	true	false	true
condition2	true	true / false	false

19.2.2.3. По компонентах логічних умов

Для більш повного аналізу компонент умов в логічних операторах існує кілька методів, що враховують структуру компонент умов і значення, які вони приймають при виконанні тестових прикладів.

19.2.2.4. Покриття за умовами (Condition Coverage)

Для забезпечення повного покриття по даному методу кожна компонента логічного умови в результаті виконання тестових прикладів повинна приймати всі можливі значення, але при цьому не потрібно, щоб саме логічне умова брало всі можливі значення.Так, наприклад, при тестуванні наступного фрагмента

```
 if (condition1 | | condition2)
  MethodA ();
else
  MethodB ();
```

для покриття за умовами потрібно два тестових прикладу:

1 2

condition1 true False

condition2 false True

При цьому значення логічного умови прийматиме значення тільки true; таким чином, при повному покритті за умовами не досягатиме покриття по гілках.

19.2.2.5. Покриття по гілках / умовам (Condition / Decision Coverage)

Даний метод поєднує вимоги попередніх двох методів - для забезпечення повного покриття необхідно, щоб як логічне умова, так і кожна його компонента прийняла всі можливі значення.

Для покриття розглянутого вище фрагмента з умовою

(Condition1 | | condition2)

потрібно 2 тестових прикладу:

	1	**2**
condition1	true	false
condition2	true	false

Однак, ці два тестових прикладу не дозволять протестувати правильність логічної функції - замість OR в програмному коді могла бути помилково записана операція AND.

19.2.2.6. Покриття за всіма умовами (Multiple Condition Coverage)

Для виявлення невірно заданих логічних функцій був запропонований метод покриття за всіма умовами. При цьому методі покриття повинні бути перевірені всі можливі набори значень компонент логічних умов. Тобто у разі n компонент буде потрібно 2n тестових прикладів,

кожен з яких перевіряє один набір значень, Тести, необхідні для повного покриття по даному методу, дають повну таблицю істинності для логічного виразу.

Незважаючи на очевидну повноту системи тестів, що забезпечує цей рівень покриття, даний метод рідко застосовується на практиці у зв'язку з його складністю і надмірністю.

Ще одним недоліком методу є залежність кількості тестових прикладів від структури логічного виразу. Так, для умов, що містять однакову кількість компонент і логічних операцій:

a && b && (c | | (d && e))

((A | | b) && (c | | d)) && e

потрібно різну кількість тестових прикладів. Для першого випадку для повного покриття потрібно 6 тестів, для другого - 11.

19.2.2.7. Метод MC / DC для зменшення кількості тестових прикладів при 3-му рівні покриття коду

Для зменшення кількості тестових прикладів при тестуванні логічних умов фірмою Boeing був розроблений модифікований метод покриття по гілках / умовам (Modified Condition / *Decision Coverage* або MC / DC). Даний метод широко використовується при верифікації бортового авіаційного програмного забезпечення згідно процесам стандарту DO-178B.

Для забезпечення повного покриття за цим методом необхідно виконання наступних умов:

- кожне логічне умова повинна вживати всіх можливих значення;
- кожна компонента логічного умови повинна хоча б один раз приймати всі можливі значення;

- має бути показано незалежне вплив кожної з компонент на значення логічного умови, тобто вплив при фіксованих значеннях інших компонент.

Покриття з цієї метриці вимагає досить великої кількості тестів для того, щоб перевірити кожне умова, яка може вплинути на результат вираження, однак ця кількість значно менше, ніж вимагається для методу покриття за всіма умовами.

Приклад 1. Розглянемо фрагмент коду, який ми використовували як приклад для покриття по рядках і по гілках. Для покриття даної ділянки коду за методом MC / DC введемо умовні позначення. Позначимо перевірку prev == "оператор" як A, перевіркуprev == "*унарний оператор* "- як B, а змінну ShowMessage - як C. Перші два позначення зроблені для того, щоб елементарними змінними для методу MC / DC були булеві змінні, а третє позначення - для одноманітності.

З урахуванням зроблених позначень фрагмент коду може бути записаний так:

```
 if (A | | B)
{
 if (C)
  {
   ...
  }
 else
  {
```

```
    ...
  }
 }
```

Для тестування першої умови по MC / DC треба показати незалежність результату (тобто функції A | | B) від кожного аргументу.Відповідно, для цього використовуються три тестових прикладу:

1. A = 0, B = 0, A | | B = 0 (початкове значення)
2. A = 1, B = 0, A | | B = 1 (показано вплив аргументу A)
3. A = 0, B = 1, A | | B = 1 (показано вплив аргументу B)

Для тестування гілок (входить до MC / DC) залежно від умови C необхідно, щоб в тестових прикладах C брало значення яктrue, так і false.

Підсумкова таблиця тестових прикладів для покриття по MC / DC буде виглядати наступним чином:

	1	**2**	**3**	**4**
prev	операнд (A = 0, B = 0)	оператор (A = 1, B = 0)	*унарний оператор* (A = 0, B = 1)	оператор
ShowMessage	false	false	false	true

Кількість тестових прикладів можна скоротити до 3, якщо поєднати приклади 3 і 4. Таке поєднання не вплине на покриття.

	1	2	3
prev	операнд (A = 0, B = 0)	оператор (A = 1, B = 0)	*унарний оператор* (A = 0, B = 1)
ShowMessage	false	false	true

Приклад 2. Для покриття по MC / DC більш складних виразів розглянемо наступну ділянку коду:

```
 if ((operators.Count! = 0 && operators.Peek (). ToString () ==
   == "M") | | operators.Peek (). ToString () == "p")
{
    strarr.Add (operators.Pop ());
}
```

Початкове умовний вираз в операторі if можна записати як (A & B) | | C. Даний вираз залежить від 3 змінних, тобто може бути розглянуте як булева функція з трьома аргументами. Згідно з методом MC / DC для тестування функції з трьома входами достатньо 4 тестових прикладів - один базовий і три показують незалежне вплив кожного входу на вихід.

Почнемо побудова набору тестів з самою зовнішньої операції, тобто з | |. Одним з аргументів цієї операції є вираз (A & B).Будемо поки розглядати цей вислів як єдине ціле. Для тестування операції | | по MC / DC потрібно три тестових прикладу:

1. A & B = 0, C = 0 (базовий приклад)
2. A & B = 0, C = 1 (незалежне вплив C на вихід)

3. A & B = 1, C = 0 (незалежне вплив A & B на вихід)

У третьому тестовому прикладі значення A і B можуть бути отримані відразу, тобто маємо

1) A = 1, B = 1, C = 0

Значення A = 1 і B = 1 є базовими для тестування з MC / DC операції &&. Відповідно, необхідно розглянути ще два випадки, при яких A = 0, B = 1 і A = 1, B = 0 для демонстрації незалежного впливу аргументів A і B на значення функції. При цьому необхідно, щоб аргумент З дорівнював 0, щоб виключити його вплив на вихід. Виходячи з цих міркувань, перший тестовий приклад може бути записаний як

1) A = 1, B = 0, C = 0

тобто при цьому перевіряється вплив змінної B на значення функції. У другому тестовому прикладі значення C = 1, тому він не може бути використаний для перевірки незалежності аргументів A і B. Значення A і B в цьому прикладі можуть бути будь-якими, за умови, що A & B = 0. Запишемо другий тестовий приклад як

2) A = 1, B = 0, C = 1

Для тестування незалежного впливу аргументу A необхідно додати ще один тестовий приклад, в якому A = 0, B = 1:

4) A = 0, B = 1, C = 0

Таким чином, ми побудували 4 тестових прикладу для перевірки даної ділянки коду.

1 2 3 3

A 1 (true) 1 (true) 1 (true) 0 (false)

B 0 (false) 0 (false) 1 (true) 1 (true)

C 0 (false) 1 (true) 0 (false) 0 (false)

При переході від позначень A, B, C до вихідних отримаємо наступні тестові приклади:

	1	**2**	**3**	**3**
operators.Count	10 (Не 0)	10 (Не 0)	10 (Не 0)	0 (дорівнює 0)
operators. *Peek* (). ToString ()	"K» (не m і не p)	"P» (не m, але "p")	"M" (m, але не p)	"M" (m, але не p)

19.2.3. Аналіз покриття

Метою аналізу повноти покриття коду є виявлення ділянок коду, які не виконуються при виконанні тестових прикладів. Тестові приклади, засновані на вимогах, можуть не забезпечувати повного виконання всієї структури коду. Тому для поліпшення покриття проводиться аналіз повноти покриття коду тестами і, при необхідності, додаткові перевірки, спрямовані на з'ясування причини недостатнього покриття, а також визначення необхідних дій щодо його усунення. Зазвичай аналіз покриття виконується з урахуванням наступних угод.

1. Аналіз повинен підтвердити, що повнота покриття тестами структури коду відповідає необхідному виду покриття і заданому мінімально допустимому відсотку покриття.
2. Аналіз повноти покриття тестами структури коду може бути виконаний з використанням вихідного тексту, якщо програмне забезпечення не відноситься до рівня A. Для

рівня А необхідно перевірити об'єктний код, згенерований компілятором, щоб встановити, трасується він у Оригінальний текст чи ні. Якщо Об'єктний коду не трасується в Оригінальний текст, мають бути проведені повірки об'єктного коду на предмет правильності генерації послідовності команд. Прикладом об'єктного коду, який безпосередньо не трасується в Оригінальний текст, але генерується компілятором, може бути перевірка виходу за задані межі масиву.

3. Аналіз повинен підтвердити правильність передачі даних і управління між компонентами коду.

Аналіз повноти покриття тестами може виявити частину вихідного коду, що не виконувалася в ході тестування. Для дозволу цієї обставини можуть знадобитися додаткові дії в процесі перевірки програмного забезпечення. Ця неісполняемимі частина коду може бути результатом:

1. недоліків у формуванні тестових прикладів або тестових процедур, заснованих на вимогах. У цьому випадку повинні бути доповнений набір тестових прикладів або змінені тестові процедури для забезпечення покриття упущеної частини коду. При цьому може знадобитися перегляд методу (методів), що використовується для проведення аналізу повноти тестів на основі вимог;
2. неадекватності у вимогах на програмне забезпечення. У цьому випадку повинні бути *модифіковані вимоги* на програмне забезпечення, розроблені і виконані додаткові тестові приклади і тестові процедури;
3. "Мертвого" коду. Цей код повинен бути вилучений, та проведено аналіз для оцінки

ефекту видалення і необхідності повторної перевірки;

4. дезактивируемие коду. Для дезактивируемие коду, яка не передбачається до виконання в кожній конфігурації, поєднання аналізу і тестів має продемонструвати можливості засобів, якими ненавмисне виконання такого коду запобігається, ізолюється або усувається. Для дезактивируемие коду, який виконується тільки за певних конфігураціях, повинна бути встановлена нормальна експлуатаційна конфігурація для виконання цього коду, і для неї повинні бути розроблені додаткові тестові приклади і тестові процедури, що задовольняють цілям повноти покриття тестами структури коду;
5. надмірності умови. Логіка роботи такої умови має бути переглянута. Наприклад, в умові if (A && B | |! B) принципово неможливо перевірити, що частина умови A && B дорівнюватиме False у разі, коли A = True і B = False, тому що друга частина умови (! B) дорівнюватиме True і загальний результат логічного виразу буде True;
6. захисного коду. Ця частина коду використовується для запобігання виняткових ситуацій, які можуть виникнути в процесі роботи програми. Як приклад, це може бути гілка default в операторі вибору switch, причому вхідна умова оператора switchможе приймати певні значення, які він описує, і як наслідок, гілка default ніколи не буде виконана.

19.2.4. Звіти про покриття програмного коду

19.2.4.1. Звіти про покриття та їх зв'язок з іншими типами проектної документації

Дані про ступінь покриття поміщаються в звіти про покриття, що генеруються при виконанні

тестів інструментальними засобами, що підтримують процес тестування, тобто по суті генеруються середовищем тестування. Формат звітів про покриття зазвичай єдиний всередині проекту або декількох проектів і часто залежить від особливостей інструментальних засобів тестування.

У звіті про покриття в стандартизованої формі вказуються ділянки програмного коду тестованої системи (або її частини), які не були виконані під час виконання тестових прикладів, тобто не були покриті тестами. Причини непокриття аналізуються тестувальниками, за результатами аналізу складаються звіти про проблеми і запити на зміну - документи, де описується об'єкти розробки, які необхідно змінити, і причини цих змін. (рис. 19.1)

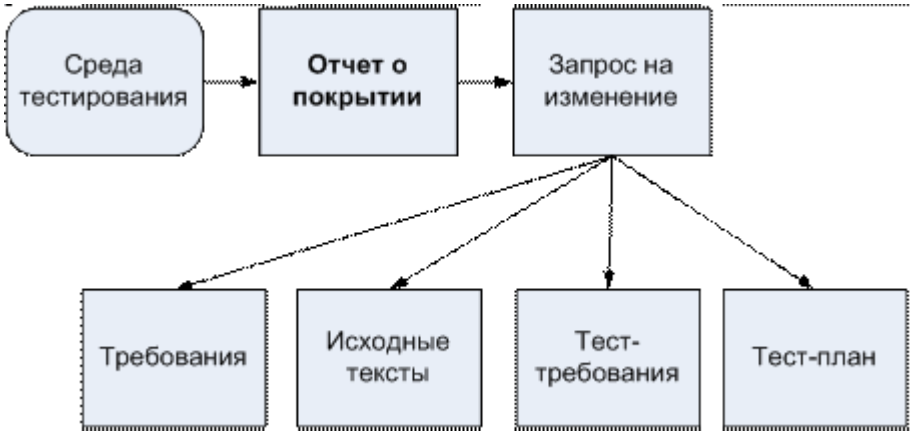


Рис. 19.1. Генерація звіту про покриття та зміни за результатами його аналізу

Недостатнє покриття може свідчити про неповноту системи тестів або тест-вимог, в цьому випадку в запиті про зміну вказується на необхідність розширення системи тестів або тест-

вимог. Іншою причиною недостатнього покриття можуть бути ділянки захисного коду, які ніколи не виконаються навіть у разі нештатної роботи системи. У цьому випадку в запиті на зміну вказується на необхідність модифікації вихідних текстів або відзначається, що для цієї ділянки програмної системи не потрібно покриття. В якості третьої причини недостатнього покриття може виступати неузгодженість вимог і програмного коду системи, в результаті якого в коді можуть залишитися невикористовувані більш ділянки або, навпаки, з'явитися ділянки, розраховані на майбутнє (і реалізують функціональність, що не описану у вимогах). У цьому випадку в запиті на зміну вказується на необхідність модифікації вимог і / або коду системи для приведення їх у узгоджений стан.

19.2.4.2. Можливі форми звітів про покриття

Типовий звіт про покриття являє собою список структурних елементів покривається програмного коду (функцій або методів), що містить для кожного структурного елементу наступну інформацію:

1. назва функції або методу;
2. тип покриття (по рядках, по гілках, MC / DC або інший);
3. кількість покриваються елементів у функції або методі (рядків, гілок, логічних умов);
4. ступінь покриття функції або методу (у відсотках або в абсолютному вираженні);
5. список непокритих елементів (у вигляді ділянок непокритого програмного коду з номерами рядків).

Крім того, звіт про покриття містить заголовну інформацію, що дозволяє ідентифікувати звіт, і

загальний підсумок - загальну ступінь покриття всіх функцій, для яких збирається інформація про покриття.

Звіт про покриття може створюватися або для всіх функцій або методів програмного модуля або всього проекту, або вибірково для визначених функцій чи методів.

У разі, якщо розмір функцій, для яких генерується вибірковий звіт, невеликий, може застосовуватися інша форма звіту про покриття, в якому покритий і непокритий програмний код виділяються різними кольорами. Така форма непридатна для покриття гілок і логічних умов, але може використовуватися для покриття по рядках.

19.2.4.3. Покриття на рівні вихідних текстів і на рівні машинних кодів

У деяких випадках інструментальні засоби збору покриття аналізують покриття програмного коду тестами не на рівні вихідних текстів системи, а на рівні машинних інструкцій. У цьому випадку ступінь покриття залежить і від того, який виконуваний код генерується компілятором.

Оскільки ступінь покриття може змінюватися в залежності від оптимізації при генерації коду, в деяких випадках навіть при повному виконанні всіх операторів мови високого рівня, на якому написана програмна система, не вдається досягти повного покриття на рівні виконуваного коду.

Збір інформації про покриття на рівні виконуваного коду найбільш часто застосовується в висококритічних програмних системах, де не допускається наявності "мертвого" виконуваного коду, який потенційно може привести до збою або відмови під час роботи системи. До таких

систем, в першу чергу, можна віднести авіаційні бортові системи, медичні системи та системи забезпечення безпеки інформації.

Верифікація програмного забезпечення

[+]

Подобається (7 користувачів)

| Поділитися |

Підтримати

| Завантажити електронну книгу

Поделиться

Семінар 20:

Покриття програмного коду

A

|

версія для друку

<Лекція 19 | | **Семінар 20:** 1 2 3 **4** | | Лекція 21>

Анотація: Семінар присвячений покриттю програмного коду. Розглянуто методи перевірки покриття, методи поліпшення покриття, покриття за методом MC / DC. Обговорюються звіти про покриття, різні їх форми, аналіз неповноти покриття, вплив залежностей тестових

прикладів на повноту покриття. Описано процес збору покриття і генерації звітів про покриття в MVSTE

Ключові слова: унарний оператор , branch coverage , coverage , decision coverage , peek , модифікуючу вимога , ПО , Visual Studio ,team , system , відсоток , меню , файл , діалогове вікно , поле , USER , computer name , DATE , TIME , пункт , статистика , шрифт ,колір тексту

19.3. Можливості MVSTE з побудови покриття коду

Зауваження. Детальніше про покриття коду можна прочитати *за* адресою: http://msdn2.microsoft.com/en-us/library/ms182496 (VS.80). aspx

Щоб побачити, яка частина коду вашого проекту фактично тестується, використовуйте такий інструмент для тестувальників в *Visual Studio Team System,* як Покриття коду. Цей інструмент показує *відсоток* коду, який був виконаний, і "розфарбовує" його, показуючи, які лінії коду були виконані, а які ні.

На минулих семінарах ми познайомилися з можливостями MVSTE *з* автоматизації модульного тестування на прикладах тестування методу Add класу CalcClass і методу RunEstimate класу AnalaizerClass. Покажемо на цих же прикладах, яку частину коду покрили створені нами модульні тести.

Для початку відкриємо BaseCalculator, з яким ми працювали на "Тестові приклади. Класи еквівалентності. Ручне тестування в MVSTE" .

Далі в *меню* **Test** вибираємо **Edit Test Run Configuration.** У підменю вибираємо **Local Test**

Run (localtestrun.testrunconfig),щоб запустити *файл* конфігурації. (Аналогічно запустити *файл* конфігурації можна, клацнувши в **Solution Explorer** під **Solution Items** на **localtestrun.testrunconfig).** З'явиться *діалогове вікно* **localtestrun.testrunconfig** (рис. 19.2).

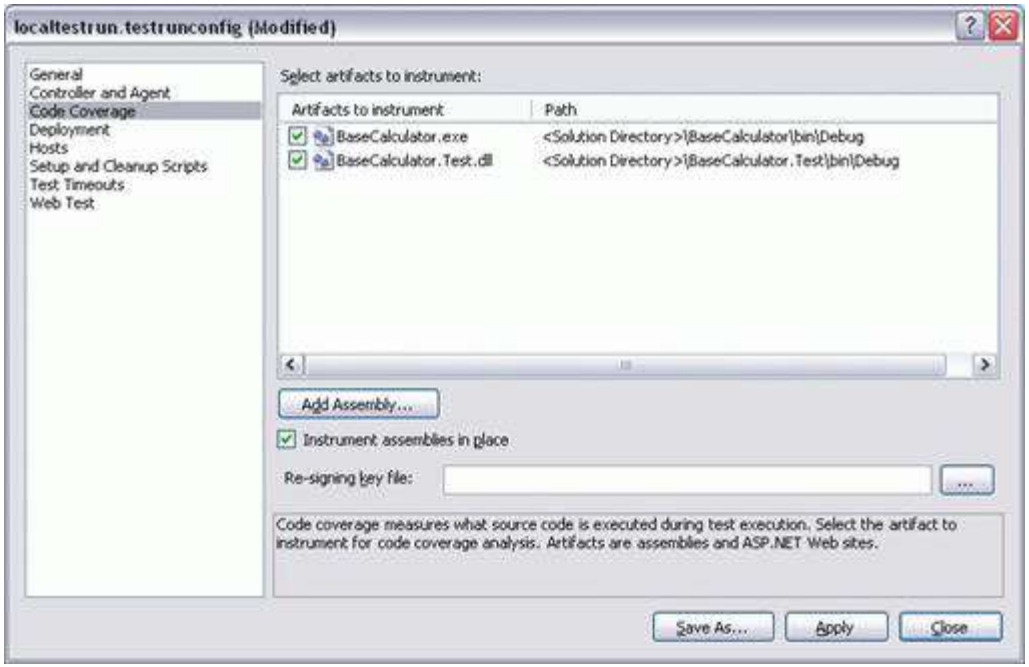


Рис. 19.2. Діалогове вікно "localtestrun.testrunconfig"

Вибираємо **Code Coverage.**

У *поле* **Select artifacts to instrument** відзначаємо пункти **BaseCalculator.exe** і **BaseCalculator.Test.dll**

Зауваження. Якщо ви вибрали **BaseCalculator.Test.dll,** то MVSTE генерує інформацію про покриття коду для методів у вашому тестовому проекті.

Натисніть **Apply,** потім закрийте *діалогове вікно.* Ми налаштували *файл* конфігурації.

Далі в *меню* **Test** виберіть **Select Active Test Run Configuration.** Підменю показує всі конфігурації запуску тесту вашого рішення.Помістимо мітку на конфігурацію запуску, яку ми тільки що редагували, **localtestrun.testrunconfig;** що зробить її активної конфігурацією запуску тесту.

У вікні **Test View** виділяємо всі тести і натискаємо кнопку **Run Selection.** Запустяться створені нами на "Тестові приклади. Класи еквівалентності. Ручне тестування в MVSTE" тести.

Після виконання всіх тестів у вікні **Test Results** в *меню* **Test** виберемо **Windows** і в розкрився підміню натиснемо на **Code Coverage Results.** Відкриється вікно **Code Coverage Results.** (рис. 19.3)

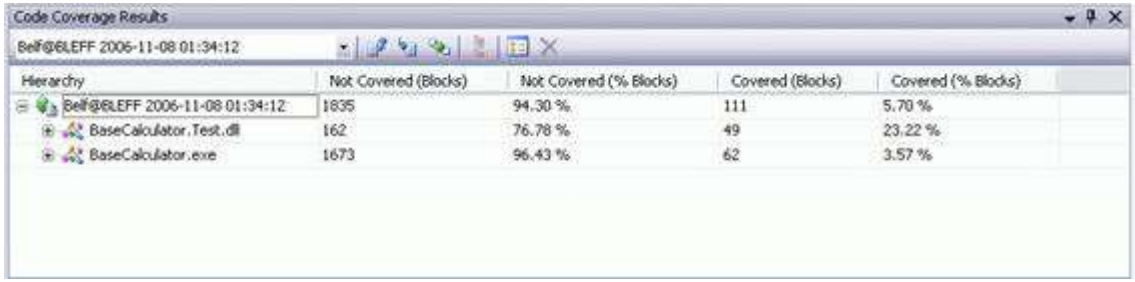


Рис. 19.3. Вікно "Code Coverage Results"

Зауваження. Колонка **Hierarchy** спочатку показує єдиний вузол (в даному випадку Belf @

BLEFF 2006-11-08 1:34:12), який містить дані про все покритті коду, досягнутому в останньому запущеному і виконаному тесті. Він названий, використовуючи наступний формат: < *user* name> @ < *computer name*> < *date*> < *time*>.

Якщо ви розкриєте цей вузол, то побачите обрані нами в файлі конфігурації проекти **BaseCalculator.exe** і **BaseCalculator.Test.dll**

Розкриємо вузол для складання **BaseCalculator.exe,** для простору імен BaseCalculator і для класу CalcClass, як показано наріс.19.4 .

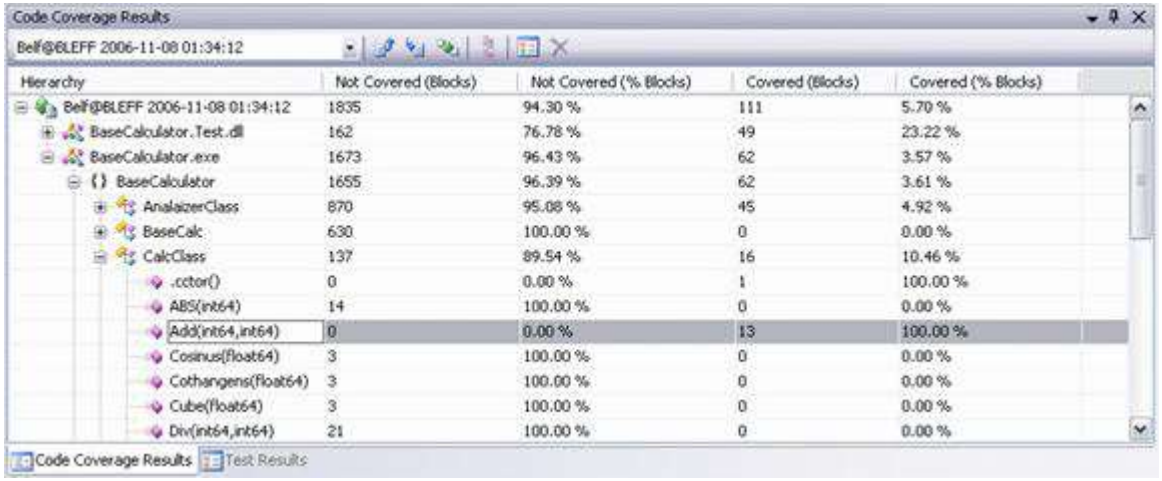


Рис. 19.4. Вікно "Code Coverage Results"

Рядки в класі CalcClass представляють його методи. Колонки у вікні **Code Coverage Results** показують статистику покриття для окремих методів, для класів, і для всього простору імен.

Зауваження. У цій таблиці можна вибирати, в якому порядку і які колонки будуть відображатися. Для цього потрібно натиснути правою кнопкою миші у вікні **Code Coverage Results** і вибрати в контекстному *меню пункт* **Add / Remove Columns.**

Зауваження. *Статистика* покриття коду показує покриття блоків і ліній коду.

Щоб подивитися, яка саме частина коду була покрита, клацнемо два рази на рядок з методом Add.

Відкриється *файл* вихідного тексту **CalcClass.cs** на методі Add. У цьому файлі ми відем "пофарбований" код. Лінії, пофарбовані в блакитний колір, були виконані в процесі виконання тестів, а лінії, пофарбовані в червоний, не були виконані. (pic.19.5)

Зауваження. Кольори фарбування ліній покриття коду можна змінити. Для цього потрібно зайти в **Tools-> Options.** У діалоговому вікні потрібно вибрати **Environment-> Fonts and Colors.** Далі в списку **Show settings for** вибираємо **Text Editor.** Далі в **Display items** потрібно вибрати область покриття коду, колір якої ви хочете змінити: це **Coverage Not Touched Area,** або **Coverage Partially Touched Area,** або **Coverage Touched Area.** Для цих областей покриття коду можна змінити *шрифт,* його розмір, жирність, *колір тексту* і його "фарбування". *По* завершенні, щоб зберегти зміни і вийти з діалогового вікна, натисніть **OK.**

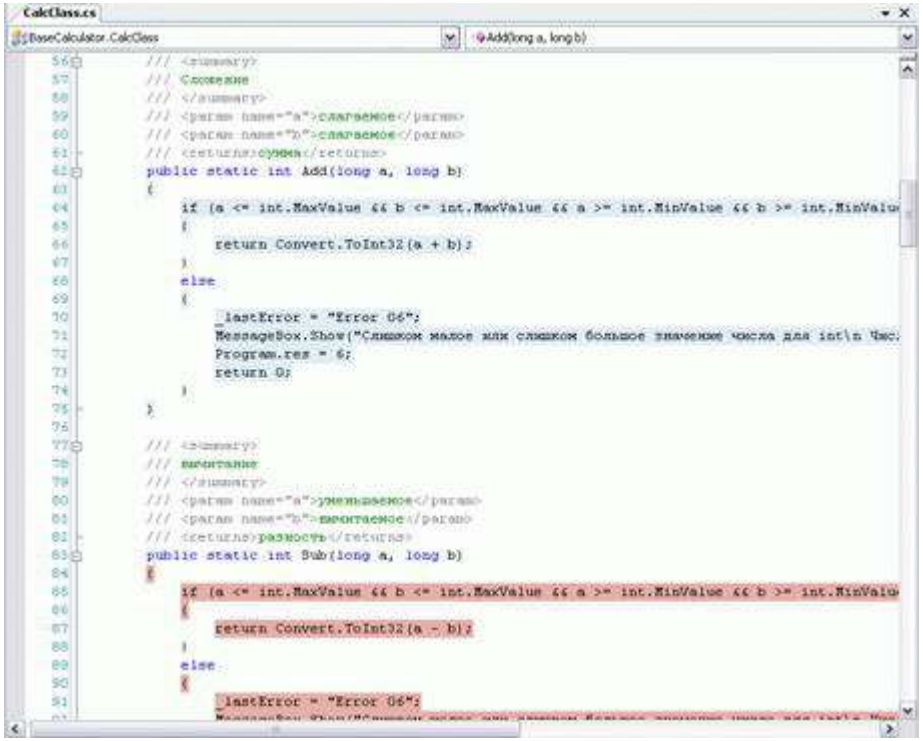


Рис. 19.5. "Фарбування" покриття коду

Прокручуючи *файл,* ви можете побачити в ньому покриття для інших методів.

Зауваження. Точно так же можна переглянути покриття коду наших модульних тестів, тобто, можна побачити, які з тестових методів були здійснені (розкривши у вікні **Code Coverage Results** збірку **BaseCalculator.Test.dll).** Застосовується та ж сама схема фарбування: блакитний показує виконаний в процесі виконання тесту код; червоний - невиконаний.

Зауваження. Результати покриття коду можна експортувати в окремий XML- *файл.* Для цього

у вікні **Code Coverage Results**потрібно натиснути на кнопку **Export Results**  , Вказати ім'я і розташування.

Лекція 6

Інтеграційне тестування

Анотація: Лекція є другою з трьох розглядають рівні процесу верифікації. Тема даної лекції - процес інтеграційного тестування, його завдання і цілі. Розглядаються організаційні аспекти інтеграційного тестування - структурна і тимчасова класифікації методів інтеграційного тестування, планування інтеграційного тестування. Мета даної лекції: дати уявлення про процес інтеграційного тестування, його технічної і організаційної складових

Ключові слова: компонент , інтеграційне тестування , модуль , стек , архітектура , висхідне тестування , монолітне тестування ,спадне тестування , висхідний метод , unit testing , фаза тестування

20.1. Завдання і цілі інтеграційного тестування

Результатом тестування та верифікації окремих модулів, що складають програмну систему, має бути висновок про те, що ці модулі є внутрішньо несуперечливими і відповідають вимогам. Однак окремі модулі рідко функціонують самі по собі, тому наступна задача після тестування окремих модулів - тестування коректності взаємодії декількох модулів, об'єднаних в єдине ціле. Таке тестування називають *інтеграційним.* Його мета - упевнитися в коректності спільної роботи *компонент* системи.

Інтеграційне тестування називають ще *тестуванням архітектури системи.* З одного боку, це назва обумовлена тим, що інтеграційні тести містять у собі перевірки всіх можливих видів взаємодій між програмними модулями і елементами, які визначаються в архітектурі системи - таким чином, інтеграційні тести перевіряють *повноту* взаємодій в тестованої реалізації системи. З іншого боку, результати виконання інтеграційних тестів - один з основних джерел інформації для процесу поліпшення та уточнення архітектури системи, міжмодульних і межкомпонентних інтерфейсів. Тобто, з цієї точки зору, інтеграційні тести перевіряють *коректність*взаємодії *компонент* системи.

Прикладом перевірки коректності взаємодії можуть служити два модулі, один з яких накопичує повідомлення протоколу про вжиті файлах, а другий виводить цей протокол на екран. У функціональних вимогах до системи записано, що повідомлення повинні виводитися в зворотному хронологічному порядку. Однак, *модуль* зберігання повідомлень зберігає їх у прямому порядку, а *модуль*виводу використовує *стек* для виведення в зворотному порядку. Модульні тести, що зачіпають кожен *модуль* окремо, не дадуть тут ніякого ефекту - цілком реальна зворотна ситуація, при якій повідомлення зберігаються в

зворотному порядку, а виводяться з використанням черги. Виявити потенційну проблему можна тільки перевіривши взаємодія модулів за допомогою інтеграційних тестів. Ключовим моментом тут є те, що у зворотному хронологічному порядку повідомлення виводить система в цілому, тобто, перевіривши *модуль* виводу і виявивши, що він виводить повідомлення в прямому порядку, ми не зможемо гарантувати, що ми виявили дефект.

У результаті проведення інтеграційного тестування та усунення всіх виявлених дефектів виходить погоджена та цілісна *архітектура*програмної системи, тобто можна вважати, що *інтеграційне тестування* - це тестування архітектури та низькорівневих функціональних вимог.

Інтеграційне тестування, як правило, являє собою ітеративний процес, при якому перевіряється функціональної все більш і більш збільшується в розмірах сукупності модулів.

20.2. Організація інтеграційного тестування

20.2.1. Структурна класифікація методів інтеграційного тестування

Як правило, інтеграційне тестування проводиться вже по завершенні модульного тестування для всіх інтегровних модулів. Однак це далеко не завжди так. Існує кілька методів проведення інтеграційного тестування:

- *висхідне тестування;*
- *монолітне тестування;*
- *спадне тестування.*

Всі ці методики грунтуються на знаннях про архітектуру системи, яка часто зображується у вигляді структурних діаграм або діаграм викликів функцій [10]. Кожен вузол на такий діаграмі являє собою програмний модуль, а стрілки між ними являють собою залежність за викликами між модулями. Основна відмінність методик інтеграційного тестування полягає в напрямку руху за цими діаграмами і в широті охоплення за одну ітерацію.

Висхідне тестування. При використанні цього методу мається на увазі, що спочатку тестуються всі програмні модулі, що входять до складу системи і лише потім вони об'єднуються для інтеграційного тестування. При такому підході значно спрощується локалізація помилок: якщо модулі протестовані окремо, то помилка при їх спільній роботі є проблема їх

інтерфейсу. При такому підході область пошуку проблем у тестувальника досить вузька, і тому набагато вища ймовірність правильно ідентифікувати дефект.

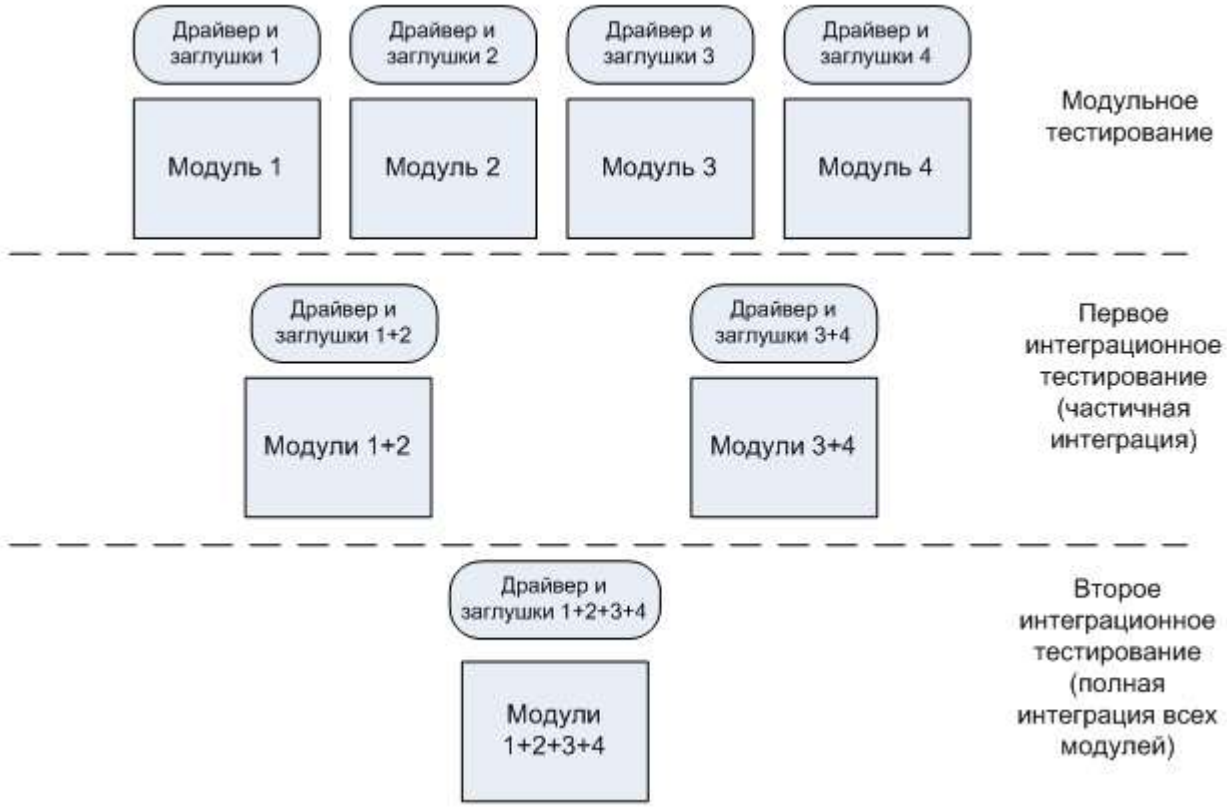


Рис. 20.1. Розробка драйверів і заглушок при висхідному інтеграційному тестуванні

Однак, у *висхідного методу* тестування є істотний недолік - необхідність у розробці драйвера і заглушок для модульного тестування перед проведенням інтеграційного тестування і необхідність у розробці драйвера і заглушок при інтеграційному тестуванні частини модулів системи (Рис 20.1)

З одного боку драйвери і заглушки - потужний інструмент тестування, з іншого - їх розробка потребує значних ресурсів, особливо при зміні складу інтегруються модулів, інакше кажучи, може знадобитися один набір драйверів для модульного тестування кожного модуля, окремий драйвер і заглушки для тестування інтеграції двох модулів з набору, окремий - для тестування інтеграції трьох модулів і т.п. У першу чергу це пов'язано з тим, що при інтеграції модулів відпадає необхідність у деяких заглушках, а також потрібна зміна драйвера, яке підтримує нові тести, що зачіпають кілька модулів.

Монолітне тестування передбачає, що окремі компоненти системи серйозного тестування не проходили. Основна перевага даного методу - відсутність необхідності в розробці тестового оточення, драйверів і заглушок. Після розробки всіх модулів виконується їх інтеграція, потім система перевіряється вся в цілому. Цей підхід не слід плутати з системним тестуванням, якому присвячена наступна лекція. Попри те, що при монолітному тестуванні проверятся робота всієї системи в цілому, основне завдання цього тестування - визначити проблеми взаємодії окремих модулів системи. Завданням же системного тестування є оцінка якісних і кількісних характеристик системи з точки зору їх прийнятності для кінцевого користувача.

Монолітне тестування має низку серйозних недоліків.

- Дуже важко виявити джерело помилки (ідентифікувати помилковий фрагмент коду). У більшості модулів слід припускати наявність помилки. Проблема зводиться до визначення того, яка з помилок у всіх залучених модулях привела до отриманого результату. При цьому можливе накладення ефектів помилок. Крім того, помилка в одному модулі може блокувати тестування іншого.
- Важко організувати виправлення помилок. В результаті тестування тестувальником фіксується знайдена проблема. Дефект у системі, що викликав цю проблему, усуватиме розробник. Оскільки, як правило, тестовані модулі написані різними людьми, виникає проблема - хто з них є відповідальним за пошук усунення дефекту? При такій "колективної безвідповідальності" швидкість усунення дефектів може різко впасти.
- Процес тестування погано автоматизується. Перевага (немає додаткового програмного забезпечення, супроводжуючого процес тестування) обертається недоліком. Кожна внесена зміна вимагає повторення всіх тестів.

Спадний тестування передбачає, що процес інтеграційного тестування рухається слідом за розробкою. Спочатку тестують тільки самий верхній керуючий рівень системи, без модулів більш низького рівня. Потім поступово з більш високорівневими

модулями інтегруються більш низькорівневі. В результаті застосування такого методу відпадає необхідність в драйверах (роль драйвера виконує більше високорівнева модуль системи), проте зберігається потреба в заглушках (Рис 20.2).

збільшити зображення

Рис. 20.2. Поступова інтеграція модулів при низхідному методі тестування

У різних фахівців в області тестування різні думки з приводу того, який з методів більш зручний при реальному тестуванні програмних систем. Йордан доводить, що *спадне тестування* найбільш прийнятно в реальних ситуаціях [27], а Майерс вважає, що кожен з підходів має свої переваги і недоліки, але в цілому висхідний метод краще [28].

У літературі часто згадується метод інтеграційного тестування об'єктно-орієнтованих програмних систем, який заснований на виділенні кластерів класів, які мають разом деяку замкнуту і закінчену функціональність [10]. За своєю суттю такий підхід не є новим типом інтеграційного тестування, просто змінюється мінімальний елемент, що отримується в результаті інтеграції. При інтеграції модулів на процедурних мовах програмування можна інтегрувати будь-яку кількість модулів за умови розробки заглушок.При інтеграції класів у кластери існує досить нестроге обмеження на закінченість функціональності кластера. Однак, навіть у разі об'єктно-орієнтованих систем можливо інтегрувати будь-яку кількість класів за допомогою класів-заглушок.

Незалежно від застосовуваного методу інтеграційного тестування, необхідно враховувати ступінь покриття інтеграційними тестами функціональності системи. В роботі [17] було запропоновано спосіб оцінки ступеня покриття, заснований на

керуючих викликах між функціями і потоках даних. При такій оцінці код всіх модулів на структурній діаграмі системи повинен бути виконаний (повинні бути покриті всі вузли), всі виклики повинні бути виконані хоча б один раз (повинні бути покриті всі зв'язки між вузлами на структурній діаграмі), все послідовності викликів повинні бути виконані хоча б один раз (всі шляхи на структурній діаграмі повинні бути покриті) [10].

20.2.2. Тимчасова класифікація методів інтеграційного тестування

На практиці найчастіше в різних частинах проекту застосовуються всі розглянуті в попередньому розділі методи в сукупності. Кожен модуль тестують по мірі готовності окремо, а потім включають у вже готову композицію. Для одних частин тестування виходить низхідним, для інших - висхідним. У зв'язку з цим видається корисним розглянути ще один тип класифікації типів інтеграційного тестування - класифікацію за часом інтеграції.

У рамках цієї класифікації виділяють:

- тестування з пізньої інтеграцією;
- тестування з постійною інтеграцією;
- тестування з регулярною або пошаровим інтеграцією.

Тестування з пізньої інтеграцією - практично повний аналог монолітного тестування. Інтеграційне тестування при такій схемі відкладається на якомога більш пізні терміни проекту. Цей підхід виправданий у тому випадку, якщо система є конгломератом слабко пов'язаних між собою модулів, які взаємодіють з якого-небудь стандартного інтерфейсу, визначеним поза проекту (наприклад, у випадку, якщо система складається з окремих Web-сервісів). Схематично тестування з пізньої інтеграцією може бути зображено у вигляді ланцюжка RCVRCVRCVIRCVRCVI, де R - розробка вимог на окремий модуль, C - розробка програмного коду, V - тестування модуля, I - інтеграційне тестування всього, що було зроблено раніше.

Тестування з постійною інтеграцією увазі, що, як тільки розробляється новий модуль системи, він відразу ж інтегрується з усією іншою системою. При цьому тести для цього модуля перевіряють як суто його внутрішню функціональність, так і його взаємодію з іншими модулями системи. Таким чином, цей підхід поєднує в собі модульне тестування та інтеграційне. Розробки заглушок при такому підході не потрібно, але може знадобитися розробка драйверів. В даний час саме цей підхід називають *unit testing,*незважаючи на те, що на відміну від класичного модульного тестування тут не перевіряється функціональність

ізольованого модуля.Локалізація помилок міжмодульних інтерфейсів при такому підході дещо ускладнена, але все ж значно нижче, ніж при монолітному тестуванні. Велика частина таких помилок виявляється досить рано саме за рахунок частоти інтеграції і за рахунок того, що за одну ітерацію тестування перевіряється порівняно невелике число міжмодульних інтерфейсів.

Схематично тестування з постійною інтеграцією може бути зображено у вигляді ланцюжка RCIRCIRCI, в якій *фаза тестування*модуля навмисно опущена і замінена на тестування інтеграції.

При **тестуванні з регулярною або пошаровим інтеграцією** інтеграційному тестуванню підлягають сильно пов'язані між собою групи модулів (шари), які потім також інтегруються між собою. Такий вид інтеграційного тестування називають також *ієрархічним інтеграційним тестуванням,* оскільки укрупнення інтегрованих частин системи, як правило, відбувається за ієрархічним принципом.Однак, на відміну від спадного або висхідного тестування, напрямок проходу по ієрархії в цьому підході не задано.

Таблиця 20.1. Основні характеристики різних видів інтеграційного тестування

Властивість	**Вид інтеграції**					
	Висхідний	**Спадний**	**Монолітне**	**Пізня інтеграція**	**Постійна інтеграція**	**Регулярна інтеграція**
Час інтеграції	пізно (після тестування модулів)	рано (паралельно з розробкою)	пізно (після розробки модулів)	пізно (після розробки всіх модулів)	рано (паралельно з розробкою)	рано (паралельно з розробкою)
Частота інтеграції	Рідко	часто	рідко	рідко	часто	часто
Чи потрібні драйвери	Так	немає	немає	немає	да	да

Чи потрібні заглушки	Так	да	немає	немає	немає	да

У табл. 20.1 наведені основні характеристики розглянутих в даній лекції видів інтеграційного тестування. Час інтеграції характеризує момент, коли проводиться перша інтеграційне тестування і всі наступні. Частота інтеграції - наскільки часто при розробці виконується інтеграція. Необхідність в драйверах і заглушках визначена в останніх двох рядках таблиці.

20.2.3. Планування інтеграційного тестування

Процес організації та планування інтеграційного тестування багато в чому схожий з процесом організації модульного тестування, розглянутому в попередній лекції. Однак інтеграційне тестування має низку організаційних особливостей, перерахованих нижче.

На етапі планування розробляється концепція і стратегія інтеграції - документ, де описаний загальний підхід до визначення послідовності, в якій повинні інтегруватися модулі. Як правило, концепція грунтується на одному з видів інтеграції, розглянутих вище (наприклад, на низхідній), але враховує особливості конкретної системи (наприклад, спочатку повинні інтегруватися компоненти роботи з базою даних, потім користувача інтерфейсу; потім інтерфейсні компоненти і компоненти роботи з БД інтегруються разом).

Складається інтеграційний тест-план, наприклад, кластерного типу [10 , 20], в якому для кожного кластера з інтегрованих модулів визначається наступне:

- кластери, від яких залежить даний кластер;
- кластери, які повинні бути протестовані до тестування даного кластеру;
- опис функціональності тестованого кластера;
- список модулів в кластері;
- опис тестових прикладів для перевірки кластера;

Планування інтеграційного тестування має бути синхронізоване з загальним планом проекту, причому виділяються для інтеграційного тестування кластери і терміни їх тестування повинні враховувати пріоритети важливості частин системи. Найчастіше розгляд пріоритетів пов'язано з тим, що системи розробляються в кілька етапів, на кожному з яких в експлуатацію вводиться тільки частина нової системи. Інтеграційне тестування в даному випадку має укладатися в загальний план-графік проекту і враховувати витрати ресурсів на тестування інтеграції з уже працюючими частинами системи.

21.1. Тест

У кожному *тестовому завданні* може бути кілька варіантів відповіді. Після проведення тесту студенти можуть спробувати обгрунтувати свої невірні відповіді.

1. Повна система тестів дозволяє стверджувати, що:
    1. система реалізує всю функціональність, зазначену у вимогах
    2. система працює коректно
    3. система не реалізує функціональність, яка не вказана у вимогах
    4. система працює правильно
    5. система реалізує функціональність, яка не вказана у вимогах
    6. система не реалізує функціональність, яка вказана у вимогах

Відповідь: 1, 3

2. Виберіть вірні твердження:
    1. Повне покриття по гілках дає повне покриття по рядках.
    2. Повне покриття по гілках не дає повного покриття по рядках.
    3. Повне покриття по рядках без розгалуження дає повне покриття коду по гілках.

4. Повне покриття по MC \ DC не дає повного покриття по рядках.

Відповідь: 1, 3

3. Які умови повинні бути виконані для забезпечення повного покриття по методу MC \ DC?
    1. має бути показано залежне вплив кожної з компонент на значення логічного умови
    2. кожне логічне умова повинна приймати всі можливі значення
    3. кожна компонента логічного умови повинна хоча б один раз приймати всі можливі значення
    4. будь-яка частина логічного умови повинна приймати хоча б раз всі можливі значення
    5. має бути показано незалежне вплив кожної з компонент на значення логічного умови

Відповідь: 2, 3, 5

4. Згідно з методом MC \ DC для тестування логічної функції з трьома входами і одним виходом достатньо:
    1. 3-х тестових прикладів
    2. 4-х тестових прикладів
    3. 5-х тестових прикладів
    4. 6-х тестових прикладів

Відповідь: 2

5. Одним з основних завдань аналізу повноти покриття коду є:
    1. виявлення ділянок коду, які виконуються при виконанні тестових прикладів
    2. виявлення ділянок коду, які містять помилки
    3. виявлення ділянок коду, які не виконуються при виконанні тестових прикладів

4. виявлення ділянок коду, які не містять помилок

Відповідь: 3

21.2. Повторюваність тестування

21.2.1. Теоретичне вступ

21.2.1.1. Завдання і цілі забезпечення повторюваності тестування при промисловій розробці програмного забезпечення

Як вже було сказано в попередніх темах, тестування програмної системи - не разовий захід, а постійний процес, активний протягом усього життєвого циклу розробки системи. Протягом цього процесу система неминуче змінюється - або в результаті виправлення помилок, або в результаті розширення її функціональності. Завдання тестувальника в такій ситуації - підтвердити, що нова або виправлена функціональність не викликало нові помилки, а якщо помилки все-таки виникли - визначити причини їх виникнення.

Найпростіший, але в той же час дієвий спосіб такого підтвердження - повне виконання всіх тестових прикладів після кожного істотної зміни системи і порівняння результатів виконання тестів до і після зміни.

Якщо результати виконання тестів до внесення змін були позитивними (всі тести проходили успішно), то поява неуспішно пройдених тестів може означати, що в системі з'явилися нові дефекти, викликані виправленням старих.

У загальному випадку повторне виконання тестів може завершитися одним із трьох способів.

1. Всі тести пройдені успішно. У цьому випадку зміни не зачіпають вже протестовані функції, але може знадобитися розробка нових тестових прикладів для нових функцій системи.
2. Частина тестів, раніше виконувалися успішно, завершується з негативним результатом. Причини цього можуть бути наступні:
    - коректне зміна функціональності тестованої системи, в результаті якого тестовий приклад перестав відповідати вимогам;
    - некоректна зміна функціональності системи, в результаті якого тестовий приклад виявив розбіжність з вимогами;

- вплив залишкових даних від попередніх тестових прикладів, раніше залишалося непоміченим.

Перші дві причини помітні тільки за допомогою аналізу змін у функціональних вимогах і тест-вимогами, а також поточного стану тест-планів та тестового оточення. За результатами цього аналізу в першому випадку тестувальник вносить зміни в тестовий приклад (і, можливо, розробляються нові тестові приклади), у другому випадку тестувальник повідомляє розробників про наявність дефекту.

3. Виконання тестів аварійно завершується на самому початку або при виконанні певного тестового прикладу.

Дана проблема найчастіше пов'язана зі зміною зовнішнього оточення тестируемой частини системи, яке моделює тестове оточення.Через такі зміни можуть мінятися зовнішні інтерфейси, а також склад і формат вхідних і вихідних даних. В результаті тестове оточення перестає забезпечувати необхідну для виконання тестів інфраструктуру і виникає збій процесу тестування. Наприклад, такий збій може виникнути в тестовому оточенні при спробі обробити дані, що видаються системою в новому форматі.

Якщо для виконання тестів потрібно збірка програмних модулів тестового оточення і тестованої системи в єдиний виконуваний код, то при зміні інтерфейсів системи може виникнути ситуація, коли неможливо не тільки виконання тестів, а навіть збірка оточення і системи. У цьому випадку також необхідно провести аналіз змін, внесених в систему, і модифікувати відповідно до них тестове оточення.

Іноді повторне виконання всіх тестів неможливо. Це може бути пов'язано з великим часом виконання всіх тестів і обмеженим часом, відведеним на процес тестування. У цьому випадку часто застосовується практика вибіркового тестування окремих частин системи, порушених змінами. Повне тестування при такому підході проводиться тільки після накопичення досить великої кількості змін або на ключових стадіях проекту.

Процес, що включає в себе повторне виконання всіх тестів, називають регресійним тестуванням. Регресійне тестування включає в себе наступні стадії:

1. Аналіз змін в системі
2. Вибір тестових прикладів для перевірки системи

3. Виконання тестових прикладів
4. Аналіз результатів виконання
5. Модифікація тестового оточення, тестових прикладів або повідомлення розробників про дефект системи.

Таким чином можна визначити такі основні завдання повторюваності тестування при внесенні змін.

- Забезпечення можливості повного виконання всіх тестів, перевіряючих функціональність системи або проведення аналізу, що дозволяє виявити тести, які повинні бути повторно виконані для тестування змінилася функціональності.
- Розробка тестових прикладів і тестового оточення з використанням методик, що полегшують модифікацію при змінах в тестованої системі.
- Розробка тестових прикладів, структура яких повністю виключає їх взаємний вплив по залишковим даними.

Метою повторюваності тестування є постійне забезпечення тестувальників і розробників актуальною інформацією про поточний стан системи і коректності змін, внесених в ході розробки системи.

21.2.1.2. Передумови для виконання тесту, настройка тестового оточення, оптимізація послідовностей тестових прикладів

Як вже було сказано раніше, вхідні дані в кожному тестовому прикладі явно задають початковий стан тестованої системи і режими її роботи при виконанні тестового сценарію.

Однак неявне вплив на виконання тесту надає і стан тестового оточення. Під станом тут розуміється набір параметрів, зміна будь-якого з яких може вплинути або на результат виконання тестового прикладу, або на можливість його коректної роботи і завершення.

Наприклад, для виконання тестового прикладу тестируемой системі може знадобитися значний обсяг дискової або оперативної пам'яті. Якщо перед виконанням тесту тестове оточення виділить цю пам'ять під свої потреби, виконання тесту виявиться неможливим. Та ж сама ситуація може виникнути і в разі, якщо оточення не звільнить пам'ять після виконання попереднього тестового прикладу.

Ця інформація зазвичай відсутня в тест-планах, проте необхідну для виконання тестів стан тестового оточення необхідно враховувати при розробці тестових прикладів.

Доброю практикою є оформлення перевірок на допустимість стану тестового оточення у вигляді передумов для виконання тесту.Це дозволяє діагностувати ситуації, що виникають при вибірковому тестуванні і призводять до відмов тестового оточення.

На практиці часто виникає ситуація в якій один за одним слід кілька десятків тестових прикладів, а при регресійному тестуванні потрібно виконати, наприклад, тестові приклади з номерами від 25 по 40. Перший тестовий приклад при цьому инициализирует систему, а решта працюють з вже стартувала системою. Якщо просто виконувати тестові приклади 25-40, то їх виконання виявиться неможливим - вони не инициализируют систему. Розумним виходом з цієї ситуації є виконання тестових прикладів 1, 25-40.

21.2.1.3. Залежність між тестовими прикладами, налаштування за замовчуванням для тестових прикладів і їх груп

Для полегшення проведення регресійного тестування (і тестування взагалі) тестові приклади часто розбивають на групи. Кожна група містить набір тестових прикладів, перевіряючих окрему замкнуту частина функціональності тестованої системи. При відборі тестових прикладів для часткового регресійного тестування їх можна відбирати відразу групами.

Розбиття тестових прикладів на групи зручно і з точки зору установки початкового стану тестового оточення для виконання тестів - так, перед виконанням групи тестів можна ініціалізувати значення змінних або стан системи, необхідне для виконання всієї групи.Наприклад, якщо система працює в двох режимах - нормальному та сервісному, то перед виконанням групи тестів для нормального режиму роботи системи потрібно встановлювати нормальний режим, а перед виконанням тестів для сервісного режиму - сервісний. Такі установки називаються настройками групи тестів за замовчуванням (group defaults, test group defaults).

Перед виконанням кожного тестового прикладу може знадобитися установка одних і тих же змінних в одні і ті ж значення. Для того, щоб не дублювати ці установки в описі кожного тестового прикладу, в тест-плані можна визначити настройки за замовчуванням для кожного тесту (*test case* defaults).

Як видно з попереднього розділу, для полегшення проведення *вибіркового регресійного тестування* кожен тестовий приклад повинен бути повністю автономним - хід його виконання і тим більше, результат не повинні залежати від попередніх тестових прикладів. Тим самим, при вибірковому тестуванні результат тестування не залежить від обраного набору тестових прикладів (тестового набору). Однак, на практиці створення автономних тестів часто неможливо з різних причин (як правило - через тривалого часу виконання таких тестів).

У разі, коли в наборі тестових прикладів тести не є автономними, говорять про тестової залежності. Тестова залежність буває двох видів - передбачена структурою тестових прикладів і паразитная.

Приклад передбаченої тестової залежності був розглянутий у попередньому розділі - коректність виконання тестів визначалася порядком їх виконання. Така тестова залежність вимагає документування та супроводження, як і самі описи тестових прикладів.Існує два види документування тестових залежностей:

- явне визначення допустимого порядку виконання тестових прикладів. Такий спосіб зручний при порівняно невеликому загальній кількості тестових прикладів, або, при розбитті на групи - при невеликому розмірі груп тестових прикладів;
- визначення допустимого порядку виконання тестових прикладів за допомогою предусловий. При такому способі коректність порядку виконання тестових прикладів визначається за допомогою перевірки того, що або тестована система, або тестове оточення знаходяться в необхідному стані для виконання тестового прикладу.

Паразитні тестові залежності зазвичай викликані некоректним складанням тест-плану. Виявляються вони, як і передбачені залежності, в тому, що один (або більше) тестових прикладів коректно працює тільки в тому випадку, якщо до нього були виконані інші тестові приклади. Причому така залежність не є передбаченою тестувальником. Природа паразитної тестової залежності схожа з природою помилок використання неініціалізованих або залишкових даних у динамічній пам'яті при програмуванні.

21.2.1.1. На прикладі "калькулятор матеріалів"

Розглянемо повторюваність тестування на прикладі нашого "калькулятор матеріалів".

Розглянемо властивість CalcClass.lastError:

```
/ / / <summary>
 / / / Останні повідомлення про помилку.
 / / / </ Summary>
 private static string _lastError = "";


 public static string lastError
 {
      get
       {


       }
 }
```

Воно зберігає останні повідомлення про помилку. При цьому "Калькулятор", обчислюючи вираз після кожної арифметичної операції, перевіряє значення змінної і, якщо воно не дорівнює порожній рядку, видає повідомлення про помилку і перериває роботу. Однак у властивості lastError немає аксессор set, і значить, ніякої зовнішній модуль не може поміняти його значення. Напрошується питання - а як же скидається це значення? Проведемо три тести поспіль на методі складання:

```
try
  {
   richTextBox1.Text = "";
   richTextBox1.Text + = "Test Case 1 \ n";
```

```
richTextBox1.Text + = "Вхідні дані: a = 78508, b = -304 \ n";
richTextBox1.Text + = "Очікуваний результат: res = 78204 &&
                error = \ "\" "+" \ n ";
int res = CalcClass.Add (78508, -304);
string error = CalcClass.lastError;
richTextBox1.Text + = "Код помилки:" + error + "\ n";
richTextBox1.Text + = "Одержаний результат:" + "res =" +
                res.ToString () + "error =" + error.ToString () + "\ n";
if (res == 78204 && error == "")
{
 richTextBox1.Text + = "Тест пройдено \ n \ n";
}
else
{
 richTextBox1.Text + = "Тест не пройдений \ n \ n";
}
}
catch (Exception ex)
{
```

```
 richTextBox1.Text + = "Перехоплено виняток:" +
                ex.ToString () + "\ nТест не пройдений. \ n";
}

try
{
 richTextBox1.Text + = "Test Case 2 \ n";
 richTextBox1.Text + = "Вхідні дані: a = -2850800078, b = 3000000000 \ n";
 richTextBox1.Text + = "Очікуваний результат: res = 0 && error =
                \ "Error 06 \" \ n ";
 int res = CalcClass.Add (-2850800078, 3000000000);
 string error = CalcClass.lastError;
 richTextBox1.Text + = "Код помилки:" + error + "\ n";
 richTextBox1.Text + = "Одержаний результат:" + "res =" +
                res.ToString () + "error =" + error.ToString () + "\ n";
 if (res == 0 && error == "Error 06")
 {
  richTextBox1.Text + = "Тест пройдено \ n \ n";
 }
```

```
 else
 {
  richTextBox1.Text + = "Тест не пройдений \ n \ n";
 }
}
catch (Exception ex)
{
 richTextBox1.Text + = "Перехоплено виняток:" +
                ex.ToString () + "\ nТест не пройдений. \ n";
}

try
{
 richTextBox1.Text + = "Test Case 3 \ n (повторний тест)";
 richTextBox1.Text + = "Вхідні дані: a = 78508, b = -304 \ n";
 richTextBox1.Text + = "Очікуваний результат: res = 78204 &&
                error = \ "\" "+" \ n ";
 int res = CalcClass.Add (78508, -304);
 string error = CalcClass.lastError;
```

```
richTextBox1.Text + = "Код помилки:" + error + "\ n";
richTextBox1.Text + = "Одержаний результат:" + "res =" +
                    res.ToString () + "error =" + error.ToString () + "\ n";
if (res == 78204 && error == "")
{
 richTextBox1.Text + = "Тест пройдено \ n \ n";
}
else
{
 richTextBox1.Text + = "Тест не пройдений \ n \ n";
}
}
catch (Exception ex)
{
richTextBox1.Text + = "Перехоплено виняток:" +
                    ex.ToString () + "\ nТест не пройдений. \ n";
}
```

21.1.

Результат:

Test Case 1

Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 78204 && error = ""

Код помилки:

Одержаний результат: res = 78204 error =

Тест пройдено

Test Case 2

Вхідні дані: a = -2850800078, b = 3000000000

Очікуваний результат: res = 0 && error = "Error 06"

Код помилки: Error 06

Одержаний результат: res = 0 error = Error 06

Тест пройдено

Test Case 3

(Повторний тест) Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 78204 && error = ""

Код помилки: Error 06

Одержаний результат: res = 78204 error = Error 06

Тест не пройдено

Як видно, незважаючи на те, що третій тест операції додавання повинен бути виконаний, він не проходить, хоча по першому тесту видно, що складання працює правильно, а значення lastError точно таке ж, що і в другому тесті. Це може свідчити, наприклад, про те, що при виклику методу Add на початку своєї роботи не очищається поле _lastError. Проведемо тестування всіх функцій:

```
try
 {
  richTextBox1.Text + = "Test Case 2 \ n";
  richTextBox1.Text + = "Вхідні дані: a = -2850800078, b = 3000000000 \ n";
  richTextBox1.Text + = "Очікуваний результат: res = 0 && error =
                \ "Error 06 \" \ n ";
  int res = CalcClass.Add (-2850800078, 3000000000);
  string error = CalcClass.lastError;
  richTextBox1.Text + = "Код помилки:" + error + "\ n";
  richTextBox1.Text + = "Одержаний результат:" + "res =" +
                res.ToString () + "error =" + error.ToString () + "\ n";
  if (res == 0 && error == "Error 06")
  {
   richTextBox1.Text + = "Тест пройдено \ n \ n";
  }
```

```
 else
 {
  richTextBox1.Text + = "Тест не пройдений \ n \ n";
 }
}
catch (Exception ex)
{
 richTextBox1.Text + = "Перехоплено виняток:" +
                ex.ToString () + "\ nТест не пройдений. \ n";
}

try
{
 richTextBox1.Text + = "Test Case 3 \ n (повторний тест)";
 richTextBox1.Text + = "Вхідні дані: a = 78508, b = -304 \ n";
 richTextBox1.Text + = "Очікуваний результат: res = 78204 &&
                error = \ "\" "+" \ n ";
 int res = CalcClass.Add (78508, -304);
 string error = CalcClass.lastError;
```

```
 richTextBox1.Text + = "Код помилки:" + error + "\ n";
 richTextBox1.Text + = "Одержаний результат:" + "res =" +
                    res.ToString () + "error =" + error.ToString () + "\ n";
 if (res == 78204 && error == "")
 {
  richTextBox1.Text + = "Тест пройдено \ n \ n";
 }
 else
 {
  richTextBox1.Text + = "Тест не пройдений \ n \ n";
 }
}
catch (Exception ex)
{
 richTextBox1.Text + = "Перехоплено виняток:" +
       ex.ToString () + "\ nТест не пройдений. \ n";
}

try
```

```
{
 richTextBox1.Text + = "Test Case 4 - перевіряємо віднімання на коректних даних";
 richTextBox1.Text + = "Вхідні дані: a = 78508, b = -304 \ n";
 richTextBox1.Text + = "Очікуваний результат: res = 78812 &&
                 error = \ "\" "+" \ n ";
 int res = CalcClass.Sub (78508, -304);
 string error = CalcClass.lastError;
 richTextBox1.Text + = "Код помилки:" + error + "\ n";
 richTextBox1.Text + = "Одержаний результат:" + "res =" +
                 res.ToString () + "error =" + error.ToString () + "\ n";
 if (res == 78508 && error == "")
 {
  richTextBox1.Text + = "Тест пройдено \ n \ n";
 }
 else
 {
  richTextBox1.Text + = "Тест не пройдений \ n \ n";
 }
}
```

```
catch (Exception ex)
{
 richTextBox1.Text + = "Перехоплено виняток:" +
       ex.ToString () + "\ nТест не пройдений. \ n";
}

try
{
 richTextBox1.Text + = "Test Case 5 - перевіряємо твір на коректних даних";
 richTextBox1.Text + = "Вхідні дані: a = 78508, b = -304 \ n";
 richTextBox1.Text + = "Очікуваний результат: res = 23866432 &&
                    error = \ "\" "+" \ n ";
 int res = CalcClass.Mult (78508, -304);
 string error = CalcClass.lastError;
 richTextBox1.Text + = "Код помилки:" + error + "\ n";
 richTextBox1.Text + = "Одержаний результат:" + "res =" +
                    res.ToString () + "error =" + error.ToString () + "\ n";
 if (res == 23866432 && error == "")
 {
```

```
 richTextBox1.Text + = "Тест пройдено \ n \ n";
}
else
{
 richTextBox1.Text + = "Тест не пройдений \ n \ n";
}
}
catch (Exception ex)
{
 richTextBox1.Text + = "Перехоплено виняток:" +
                ex.ToString () + "\ nТест не пройдений. \ n";
}
```

21.2.

Результат

Test Case 2

Вхідні дані: a = -2850800078, b = 3000000000

Очікуваний результат: res = 0 && error = "Error 06"

Код помилки: Error 06

Одержаний результат: res = 0 error = Error 06

Тест пройдено

Test Case 3

(Повторний тест) Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 78204 && error = ""

Код помилки: Error 06

Одержаний результат: res = 78204 error = Error 06

Тест не пройдено

Test Case 4 - перевіряємо віднімання на коректних даних

Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 78812 && error = ""

Код помилки: Error 06

Одержаний результат: res = 78812 error = Error 06

Тест не пройдено

Test Case 5 - перевіряємо твір на коректних даних

Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 23866432 && error = ""

Код помилки: Error 06

Одержаний результат: res = -23866432 error = Error 06

Тест не пройдено

Ми бачимо, що жоден з методів не очищає поле _lastError. Це може бути або помилкою проектування, або неправильної реалізацією властивості lastError. Можна або відправити на доопрацювання всі методи і функціональні вимоги, вказавши в них, що методи повинні перед початком роботи очищати властивість lastError, або доопрацювати властивість таким чином:

```
public static string lastError
{
  get
   {
    string temp = _lastError;
    _lastError = "";
    return temp;
   }
}
```

Таким чином, після будь-якого читання цієї змінної, її значення знову буде дорівнювати порожньому рядку.

Зауваження. Незважаючи на гадану "притягненість" цього прикладу, він дуже характерний. Можна повернутися до "Тестові приклади. Класи еквівалентності. Ручне тестування в MVSTE" і видалити рядки в тестовому модулі, очищаючі значення_lastError. При запуску тестів третій тест замість перехоплення виключення повідомить, що метод закінчив роботу з кодом помилки 3. Причина саме в тому, що після виконання тесту 2 значення _lastError не було очищено. Це ще раз свідчить

про те, що тестам треба созавать коректне тестове оточення. Далі буде наведено ще один приклад неправильного побудови тестового оточення.

Розглянутий приклад є досить простим, і помилка буде легко виявлена при тестуванні. У четвертому семінарі, при написанні тестового драйвера для методу RunEstimate (), ми підключали збірку **My.dll:**

```
System.IO.BinaryReader reader = new System.IO.BinaryReader
(New System.IO.FileStream (Application.StartupPath +
+ "\ \ My.dll", System.IO.FileMode.Open,
System.IO.FileAccess.Read));
Byte [] asmBytes = new Byte [reader.BaseStream.Length];
reader.Read (asmBytes, 0, (Int32) reader.BaseStream.Length);
reader.Close ();
reader = null;
System.Reflection.Assembly assm =
System.Reflection.Assembly.Load (asmBytes);
```

Може здатися, що ми виконуємо зайві дії, і замість всіх цих рядків коду легше застосувати методSystem.Reflection.Assembly.LoadFile, який відразу підключить необхідну збірку по шляху бібліотеки.

Зауваження. У Framework 2.0 метод LoadFile () оголошений як застарілий.

Далі ми можемо проводити скільки завгодно тестів методів класу AnalaizerClass.

Але якщо нам знадобиться поміняти заглушку класу CalcClass (наприклад, на ту, яка виводить на екран якісь додаткові відомості), то ми отримаємо помилку *access denied,* т.к. збірка вже завантажена і використовується процесом, і перекомпілювати її не вийде.

Це інша проблема регресійного тестування. Тут вже помилка не в програмі, що тестується, а в самій побудові тестів і тестового оточення. Як було сказано у вступі, потрібно або для кожного тесту заново проводити ініціалізацію тестового оточення, або об'єднувати тести в групи із загальною ініціалізацією, але стежачи за тим, щоб тести не запускалися окремо.

21.3. Впорядковані тести (ordered tests) в MVSTE

Зауваження. Детальніше про впорядкованих тестах можна почитати за адресою http://msdn2.microsoft.com/en-us/library/ms182629.aspx

Упорядкований тест містить у собі інші тести і призначається для того, щоб запускати їх в зазначеному порядку. Упорядкований тест з'являється як окремий тест у вікні **Test View,** а результати його виконання з'являються одним рядком у вікні **Test Results.**

Зауваження. Упорядкований тест може містити тести будь-якого типу крім навантажувальних тестів. Ще однією особливістю упорядкованого тесту є те, що, якщо ми запустимо його віддалено або з командного рядка, буде показано попередження про те, що, щоб запустити тест, всі ручні тести, які містяться в ньому, будуть тимчасово з нього вилучені.

Зауваження. Перш ніж Ви почнете створювати впорядковані тести, у Вас повинні бути в наявності інші тести, які можна буде включити в упорядкований тест.

Розглянемо спочатку створення упорядкованого тесту.

Для початку відкриємо BaseCalculator, з яким ми працювали на "Тестові приклади. Класи еквівалентності. Ручне тестування в MVSTE" і "Тестування програмного коду (покриття)" . Він вже містить тести.

Додамо в тестовий проект упорядкований тест. Для цього в *меню* **Test** виберемо **New Test.** У діалоговому вікні, **Add New Test**вибираємо **Ordered Test.** У *поле* **Test Name** введемо назву тесту, наприклад, OrderedTest.orderedtest, а в пункті **Add to Test Project** виберемо наш тестовий проект BaseCalculator.Test. Для додавання тесту в проект натискаємо **OK.** (pic.21.1)

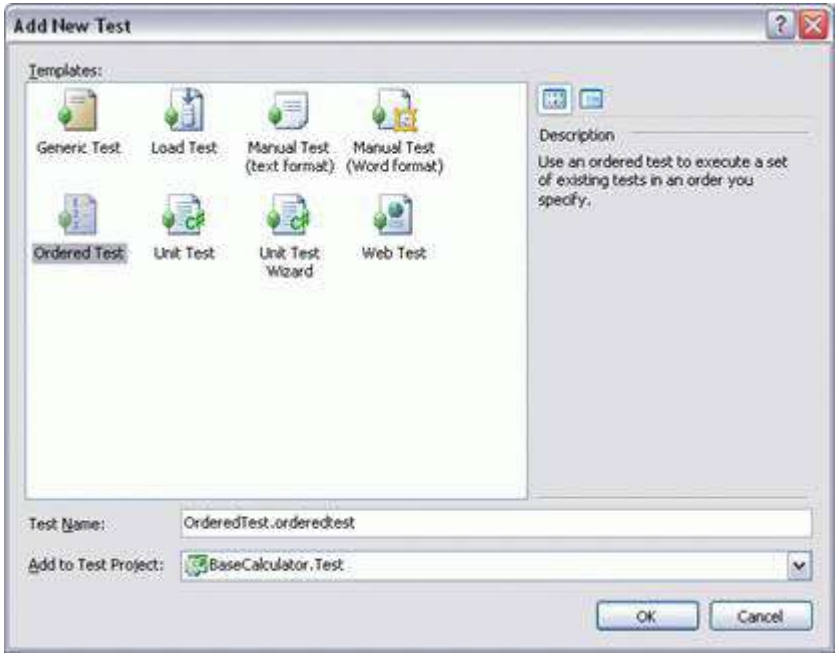


Рис. 21.1. Діалогове вікно Add New Test

У **Solution Explorer** додасться *файл* OrderedTest.orderedtest і відкриється вікно редагування упорядкованого тесту**OrderedTest.orderedtest.** Ми будемо використовувати це вікно, щоб вибирати і включати тести в наш упорядкований тест (рис. 21.2).

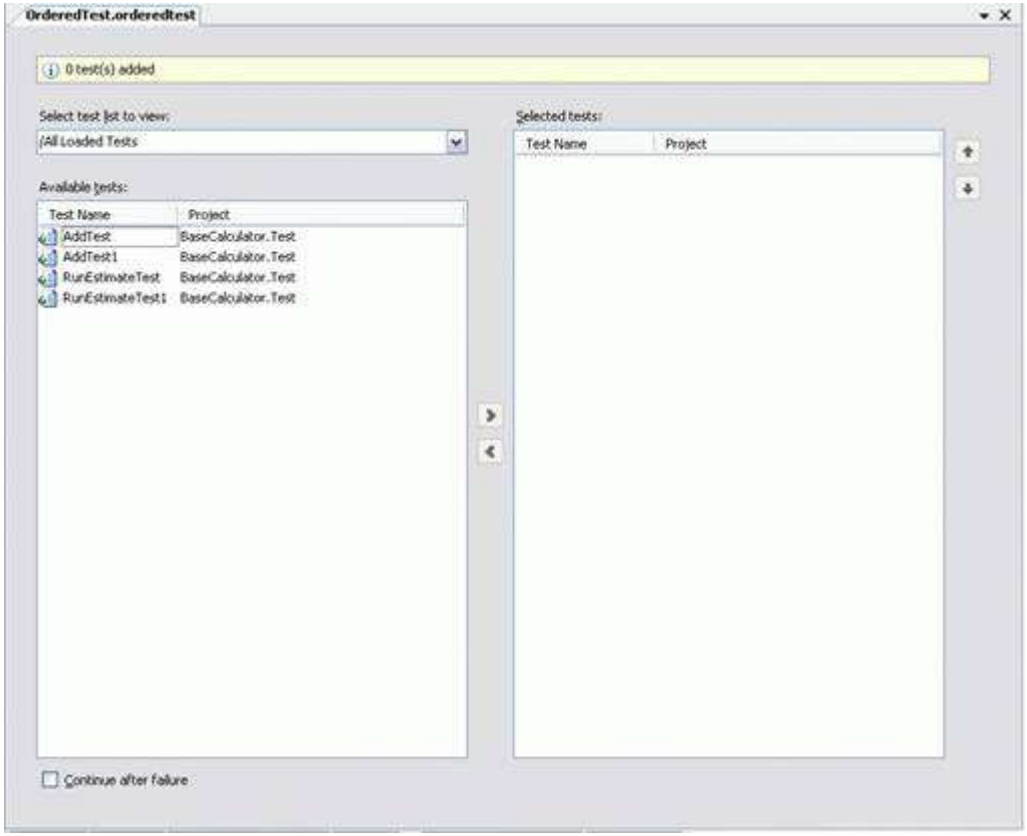


Рис. 21.2. Вікно редагування упорядкованого тесту

Розглянемо докладніше роботу з вікном редагування нашого упорядкованого тесту **OrderedTest.orderedtest.**

У спадаючому *меню* **Select Test List to View** можна вибрати *список* тестів: **Lists of Tests, Tests Not in a List, All Loaded Tests**або певний тестовий *список.* Виберемо **All Loaded Tests.** У **Available tests** відобразяться всі створені нами раніше тести.

Щоб додати тести в упорядкований тест, потрібно в **Available tests** виділити ті тести, які ви хочете додати (наприклад, використовуючи **SHIFT + click** і **CTRL + click),** і натиснути на стрілку вправо . Тести додані в упорядкований тест.

Зауваження. Можна додавати одні й ті ж тести багаторазово до одного і того ж впорядкованого тесту.

Щоб видалити тест з упорядкованого тесту, потрібно в **Selected tests** виділити ті тести, які ви хочете видалити (наприклад, використовуючи **SHIFT + click** і **CTRL + click),** і натиснути на стрілку вліво . Тести видалені з упорядкованого тесту.

Щоб змінити порядок тестів в упорядкованому тесті, потрібно в **Selected tests** виділити ті тести, порядок яких ви хочете змінити (наприклад, використовуючи **SHIFT + click** і **CTRL + click),** і натиснути на стрілку вгору або вниз . Порядок тестів в упорядкованому тесті буде змінений.

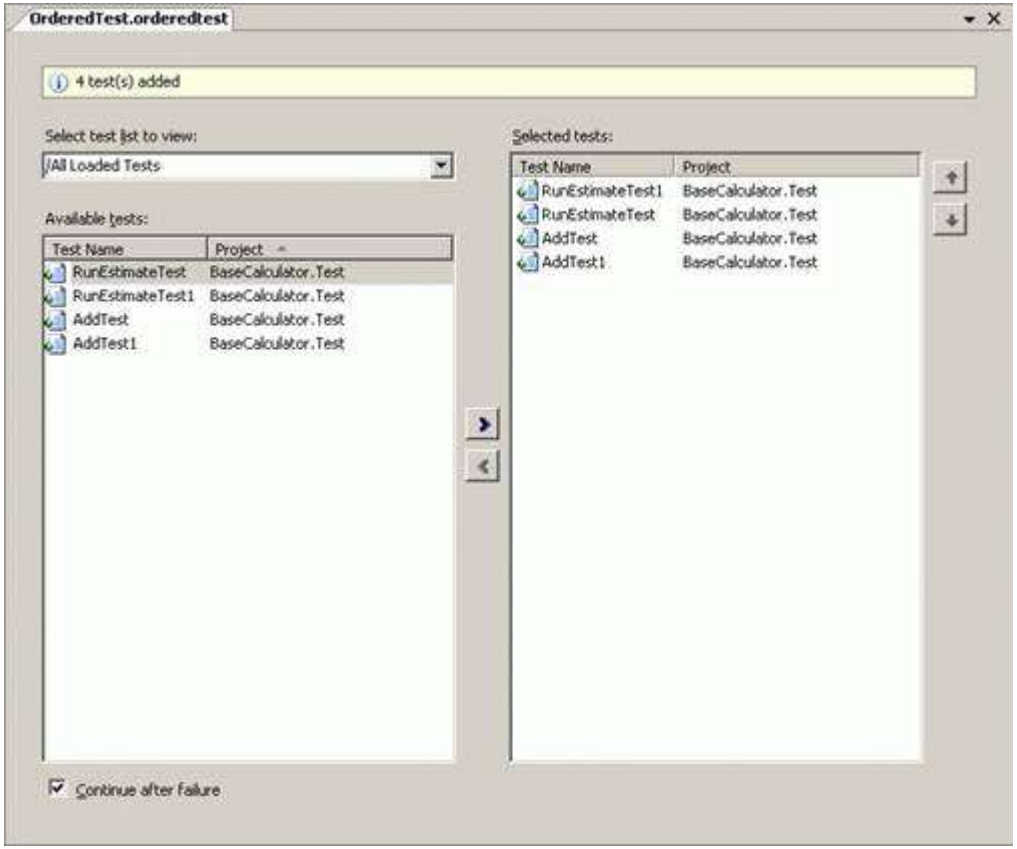


Рис. 21.3. Вікно редагування упорядкованого тесту

Зауваження. Виставлений прапор **Continue after failure** вказує на те, що упорядкований тест продовжить працювати незалежно від того, чи закінчаться один або кілька входять до його складу тестів невдачею. Якщо прапор **Continue after failure** не буде виставлений, то впорядкований тест припинить свою роботу при першому ж виникненні невдалого тесту

Упорядкований тест готовий для запуску. Наступний етап - *запуск* тесту тестувальником.

У вікні **Test View** натиснемо правою кнопкою миші по створеному нами впорядкованого тесту **(OrderedTest)** і виберемо **Run Selection** (або натиснемо у вікні **Test View** на кнопку).

Відкриється вікно **Test Results,** у якому після виконання упорядкованого тесту відобразяться результати його виконання **Passed**або **Failed** (pic.21.4).

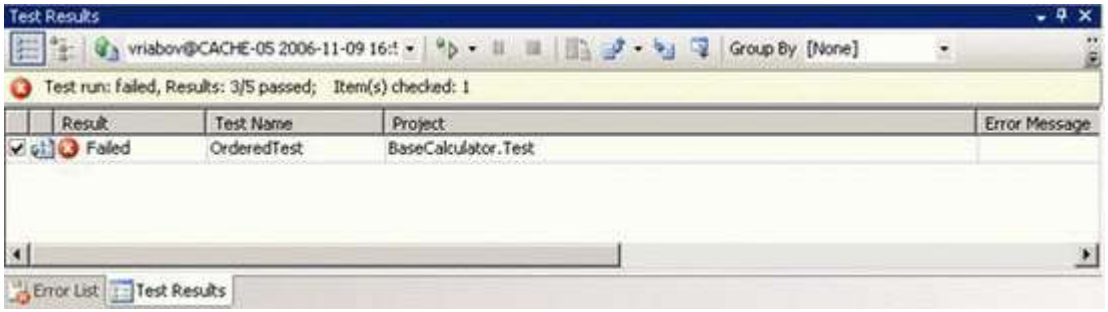


Рис. 21.4. Вікно Test Results

Щоб побачити результати виконання кожного з тестів, що входять в упорядкований тест, у вікні **Test Results** потрібно клацнути два рази на рядку для впорядкованого тесту. Ці результати з'являться у вікні **OrderedTest [Results].** (pic.21.5)

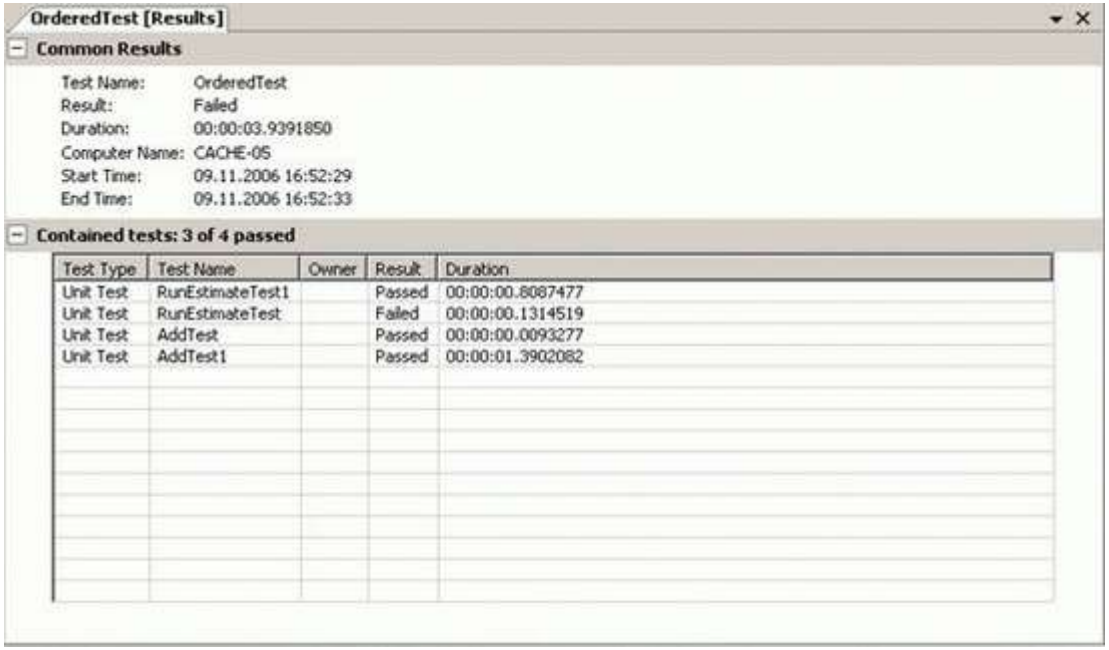


Рис. 21.5. Вікно OrderedTest [Results]

Щоб побачити детальні результати окремих тестів, потрібно клацнути два рази на них у вікні **OrderedTest [Results].**

Зауваження. Якби перед запуском тесту не був виставлений прапор **Continue after failure,** то впорядкований тест припинив би свою роботу при першому ж виникненні невдалого тесту. (pic.21.6)

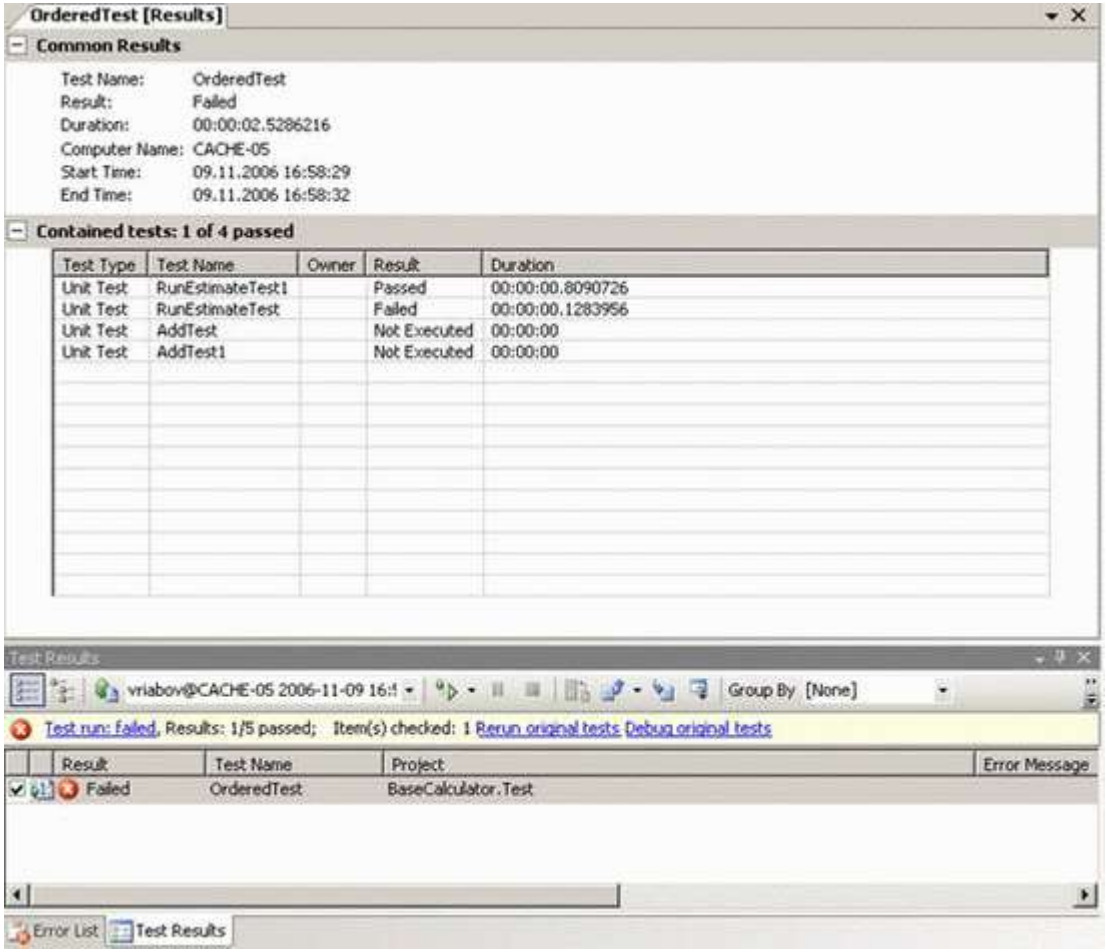


Рис. 21.6. Вікна OrderedTest [Results] і Test Result

Зауваження. Результати виконання тестів можна експортувати в окремий *файл.* Для цього у вікні **Test Result** треба вибрати**Export Test Run Results** і вказати ім'я і розташування.

Лекція 7

Системне тестування

Анотація: Лекція є останньою з трьох розглядають рівні процесу верифікації. Тема даної лекції - процес системного тестування, його завдання і цілі. Розглядаються види системного тестування, особливості системного тестування і випробувань при розробці сертифікованого програмного забезпечення. Мета даної лекції: дати уявлення про процес системного тестування, його технічної і організаційної складових

Ключові слова: ПО , системне тестування , функціональне тестування , Стресовий тестування , тестування безпеки , Тестування зручності використання , функціональні вимоги , запит , користувач , чорний ящик , повнота , визначення , тестуванняпроизводительности , производительность , загрузка , поток , устойчивость , генерат ор , интерфейс , информация , доступ ,международный стандарт , common criteria , сертифікація , цілісність з восстановлением , группа , COTS , объект , цикла ,безопасность , устойчивость до відмов , життєвий цикл програмного забезпечення , процеси життєвого циклу , базис , програмне забезпечення

22.1. Завдання і цілі системного тестування

По завершенню інтеграційного тестування всі модулі системи є узгодженими *по* інтерфейсах і функціональності. Починаючи з цього моменту можна переходити до тестування системи в цілому як єдиного об'єкта тестування - до *системного тестування.* На рівні інтеграційного тестування тестувальника цікавили в основному структурні аспекти системи, на рівні системного тестування цікавлять поведінкові аспекти системи. Як правило, для системного тестування застосовується підхід чорного ящика, при цьому в якості вхідних і вихідних даних використовуються реальні дані, з якими працює система, або дані, подібні їм.

Системне тестування - один з найскладніших видів тестування. На цьому етапі проводиться не тільки *функціональне тестування,* а й оцінка характеристик якості системи - її стійкості, надійності, безпеки і продуктивності. На цьому етапі виявляються багато проблем зовнішніх інтерфейсів системи, пов'язані з невірним взаємодією з іншими системами, апаратним забезпеченням, невірним розподілом пам'яті, відсутністю коректного звільнення ресурсів і т.п.

Після завершення системного тестування розробка переходить у фазу приймально-здавальних випробувань (для програмних систем, що розробляються на замовлення) або в фазу альфа-і бета-тестування (для програмних систем загального застосування).

Оскільки *системне тестування* - процес, що вимагають значних ресурсів, для його проведення часто виділяють окремий колектив тестувальників, а часто *системне тестування* виконується організацією, яка не пов'язана з колективом розробників і тестувальників, які виконували роботи на попередніх етапах тестування. При цьому необхідно відзначити, що при розробці деяких типів програмного забезпечення (наприклад, авіаційного бортового) вимога незалежного тестування на всіх етапах розробки є обов'язковим.

Системне тестування проводиться в кілька фаз, на кожній з яких перевіряється один з аспектів поведінки системи, тобтопроводиться один з типів системного тестування. Всі ці фази можуть протікати одночасно або послідовно. Наступний розділ присвячений розгляду особливостей кожного з типів системного тестування на кожній фазі.

22.2. Види системного тестування

Прийнято виділяти такі види системного тестування:

- функціональне тестування;
- тестування продуктивності;
- навантажувальний або *стресове тестування;*
- тестування конфігурації;
- *тестування безпеки;*
- тестування надійності і відновлення після збоїв;
- *тестування зручності використання.*

У ході системного тестування проводяться далеко не всі з перерахованих видів тестування - конкретний їх набір залежить від тестованої системи.

Вихідною інформацією для проведення перерахованих видів тестування є два класи вимог: функціональні та нефункціональні.*Функціональні вимоги* явно описують, що система повинна робити і які виконувати перетворення вхідних значень у вихідні.Нефункціональні вимоги визначають властивості системи, безпосередньо не пов'язані з її функціональністю. Прикладом таких властивостей може служити час відгуку на *запит* користувача (наприклад, не більше 2 секунд), час безперебійної роботи (наприклад, не менше 10000 годин між двома збоями), кількість помилок, які допускає початківець *користувач* за перший тиждень роботи (не більше 100), і т.п.

Розглянемо кожен вид тестування докладніше.

Функціональне тестування. Даний вид тестування призначений для доказу того, що вся система в цілому веде себе відповідно до очікувань користувача, формалізованими у вигляді системних вимог. У ході даного виду тестування перевіряються всі функції системи з точки зору її користувачів (як користувачів-людей, так і "користувачів" - інших програмних систем). Система при функціональному тестуванні розглядається як *чорний ящик,* тому в даному випадку корисно використовувати класи еквівалентності.Критерієм повноти тестування в даному випадку буде *повнота* покриття тестами системних функціональних вимог (або системних тест-вимог) і *повнота* тестування класів еквівалентності, а саме:

- всі функціональні вимоги повинні бути протестовані;
- всі класи допустимих вхідних даних повинні коректно оброблятися системою;
- всі класи неприпустимих вхідних даних повинні бути відкинуті системою, при цьому не повинна порушуватися стабільність її роботи;
- в тестових прикладах повинні генеруватися всі можливі класи вихідних даних системи;
- під час тестування система повинна побувати у всіх своїх внутрішніх станах, пройшовши при цьому по всіх можливих переходах між станами.

Результати системного тестування протоколюються і аналізуються абсолютно аналогічно тому, як це робиться для модульного та інтеграційного тестування. Основна складність тут полягає в локалізації дефектів у програмному коді системи та визначенні залежностей одних дефектів від інших (ефект "парного числа помилок").

Тестування продуктивності. Даний вид тестування спрямований на *визначення* того, що система забезпечує належний рівень продуктивності при обробці користувача запитів. *Тестування продуктивності* виконується при різних рівнях навантаження на систему, на різних конфігураціях обладнання. Виділяють три основні чинники, що впливають на *продуктивність* системи: кількість підтримуваних системою потоків (наприклад, користувача сесій), кількість вільних системних ресурсів, кількість вільних апаратних ресурсів.

Тестування продуктивності дозволяє виявляти вузькі місця в системі, які проявляються в умовах підвищеного навантаження або браку системних ресурсів. У цьому випадку *за* результатами тестування проводиться доробка системи, змінюються алгоритми виділення та розподілу ресурсів системи.

Всі вимоги, що ставляться до продуктивності системи, повинні бути чітко визначені і обов'язково повинні включати в себе числові оцінки параметрів продуктивності. Тобто, наприклад, вимога "Система повинна мати прийнятний час відгуку на *запит* користувача "є непридатним для тестування. Навпаки, вимога" Час відгуку на *запит* користувача не повинно перевищувати 2 секунди "може бути протестовано.

Те ж саме відноситься і до результатів тестування продуктивності. У звітах *по* даному виду тестування зберігають такі показники, як*завантаження* апаратного та системного програмного забезпечення (кількість циклів процесора, виділеної пам'яті, кількість вільних системних ресурсів тощо). Також важливі швидкісні характеристики тестованої системи (кількість оброблених в одиницю часу запитів, тимчасові інтервали між початком обробки кожного наступного запиту, рівномірність часу відгуку в різні моменти часу тощо).

Для проведення тестування продуктивності потрібна наявність генератора запитів, який подає на вхід системи *потік* даних, типових для сеансу роботи з нею. Тестове оточення має включати в себе крім програмної компоненти ще й апаратну, причому на такому тестовому стенді повинна існувати можливість моделювання різного рівня доступних ресурсів.

Стресове тестування. *Стресове тестування* має багато спільного з тестуванням продуктивності, проте його основне завдання - не визначити *продуктивність* системи, а оцінити *продуктивність* і *стійкість* системи у випадку, коли для своєї роботи вона виділяє максимально доступну кількість ресурсів або коли вона працює в умовах критичної недостачі. Основна мета стресового тестування - вивести систему з ладу, визначити ті умови, за яких вона не зможе далі нормально функціонувати. Для проведення стресового тестування використовуються ті ж самі інструменти, що і для тестування продуктивності. Однак,

наприклад, *генератор*навантаження при стресовому тестуванні повинен генерувати запити користувачів з максимально можливою швидкістю або генерувати дані запитів таким чином, щоб вони були максимально можливими *за* обсягом обробки.

Стресове тестування дуже важливо при тестуванні web-систем і систем з відкритим доступом, рівень навантаження на які часто дуже складно прогнозувати.

Тестування конфігурації. Більшість програмних систем масового призначення призначене для використання на самому різному обладнанні. Незважаючи на те, що в даний час особливості реалізації периферійних пристроїв ховаються драйверами операційних систем, які мають уніфікований з точки зору прикладних систем *інтерфейс,* проблеми сумісності (як програмної, так і апаратної) все одно існують.

У ході тестування конфігурації перевіряється, що програмна система коректно працює на всьому підтримуваному апаратному забезпеченні і спільно з іншими програмними системами. Необхідно також перевіряти, що система продовжує стабільно працювати при гарячій заміні будь-якого підтримуваного пристрою на аналогічне. При цьому система не повинна давати збоїв ні в момент заміни пристрою, ні після початку роботи з новим пристроєм.

Також необхідно перевіряти, що система коректно обробляє проблеми, що виникають в обладнанні, як штатні (наприклад, сигнал кінця паперу в принтері), так і позаштатні (збій харчування).

Тестування безпеки. Якщо програмна система призначена для зберігання або обробки даних, вміст яких являє собою таємницю певного роду (особисту, комерційну, державну і т.п.), то до властивостей системи, що забезпечує збереження цієї таємниці, будуть пред'являтися підвищені вимоги. Ці вимоги повинні бути перевірені при тестуванні безпеки системи. У ході цього тестування перевіряється, що *інформація* не втрачається, не пошкоджується, її неможливо підмінити, а також до неї неможливо отримати несанкціонований *доступ,* в тому числі за допомогою використання вразливостей в самій програмній системі.

У вітчизняній практиці прийнято проводити сертифікацію програмних систем, призначених для зберігання даних для службового користування, секретних, цілком таємних і цілком таємних особливої важливості. Існує ряд вітчизняних стандартів Федеральної служби *з* технічного та експортного контролю (ФСТЕК), що регламентують властивості програмних систем *щодо* забезпечення необхідного рівня безпеки [5] і *по* відсутності недокументованих можливостей ("закладок") [4],

які можуть бути використані зловмисником для несанкціонованого доступу до даних. Крім того, існує *міжнародний стандарт Common Criteria* [37], також регламентує питання захисту інформації в програмних системах.

Незважаючи на те, що *сертифікація* - процес, наступний за верифікацією, вимоги цих стандартів можуть бути використані і при тестуванні системи. Так, стандарт [5] ФСТЕК, традиційно скорочено званий РД СВТ, виділяє наступні групи властивостей програмної системи, що підлягають перевірці (деякі групи властивостей укрупнені для скорочення списку):

- розмежування і контроль доступу - запобігання доступу до "чужої" інформації;
- очищення та захист пам'яті - запобігання доступу до залишкової інформації після видалення об'єктів з пам'яті;
- маркування і захист інформації, переданої у зовнішній світ - збереження рівня секретності навіть поза системою;
- ідентифікація та аутентифікація - надання доступу тільки санкціонованим користувачам і відмова в доступі всім іншим;
- реєстрація (аудит подій) - реєстрація в спеціальному журналі всіх подій системи, пов'язаних з безпекою для подальшого аналізу;
- гарантії проектування та архітектури - система повинна бути спроектована таким чином, щоб гарантувати захищеність інформації з певним рівнем впевненості;
- тестування - всі функції щодо забезпечення безпеки повинні бути протестовані у всіх режимах;
- *цілісність і відновлення* засобів захисту - система повинна мати засоби контролю коректності всіх правил розмежування доступу та системи безпеки в цілому, а також засоби їх відновлення при збої;
- документація розробника, адміністратора і користувача - всі засоби системи із забезпечення безпеки повинні бути описані у відповідних посібниках.

При розробці та верифікації програмної системи, яка буде піддаватися подальшої сертифікації, роботи *з* сертифікації повинні включати в себе перевірку всіх перерахованих властивостей.

Тестування надійності і відновлення після збоїв. Для коректної роботи системи в будь-якій ситуації необхідно упевнитися в тому, що вона відновлює свою функціональність і продовжує коректно працювати після будь-якої проблеми, прервавшей її

роботу.При тестуванні відновлення після збоїв імітуються збої обладнання або навколишнього програмного забезпечення або збої програмної системи, викликані зовнішніми чинниками. При аналізі поведінки системи в цьому випадку необхідно звертати увагу на два фактори - мінімізацію втрат даних в результаті збою і мінімізацію часу між збоєм і продовженням нормального функціонування системи

Тестування зручності використання. Окрема *група* нефункціональних вимог - вимоги до зручності використання користувальницького інтерфейсу системи. Цей вид тестування буде розглянуто в наступній лекції.

В результаті виконання всіх розглянутих вище видів тестування робиться висновок про функціональність і властивостях системи, після чого вузькі місця системи допрацьовуються до реалізації необхідної функціональності або до досягнення системою необхідних

22.3. Системне тестування, приймально-здавальні та сертифікаційні випробування при розробці сертифицируемого програмного забезпечення

При розробці масового ("коробкового", *COTS)* програмного забезпечення після проведення системного тестування система проходить етапи альфа-і бета-тестування, під час якого роботу системи перевіряють потенційні користувачі (або спеціально виділені фокус-групи користувачів, або всі бажаючі). На цьому етапі в програмну систему вносяться останні незначні зміни, які не впливають на суть системи. Після завершення цієї стадії система надходить у продаж кінцевим користувачам.

При розробці програмного забезпечення на фазу альфа-і бета-тестування замінюють приймально-здавальні випробування. Під час цих випробувань замовник засвідчується, що система працює відповідно до його потреб (як зафіксованими в технічному завданні на систему, так і не зафіксованими). Замовник може проводити такі випробування самостійно, виконуючи заздалегідь підготовлені тести системи, або проводити їх спільно з представниками колективу розробників. У цьому випадку тестові приклади також готуються розробниками, наприклад, на основі тестових прикладів, що використовувалися на етапі системного тестування.

Завершуються приймально-здавальні випробування або підписанням акта приймання, або видачею замовником додаткових вимог до системи, які повинні бути виправлені до приймання системи. Після усунення всіх недоліків системи приймально-здавальні випробування повторюються (можливо, *за* скороченою програмою). Після успішного підписання акта система надходить в експлуатацію замовнику.

Існує спеціальний вид програмних систем, до властивостей яких пред'являються особливі вимогами. Прикладом таких систем можуть служити бортові авіаційні програмні системи, для яких особлива увага приділяється питанням безпеки, надійності та відмовостійкості. Незважаючи на те, що більша частина таких систем може бути віднесена до категорії замовленого програмного забезпечення, для отримання дозволу на установку системи на борт потрібне отримання сертифіката на льотну придатність.

Таким чином, після проведення системного тестування та приймально-здавальних випробувань проводяться сертифікаційні випробування. *Сертифікація* програмного забезпечення - процес встановлення і офіційного визнання того, що розробка *ПО*проводилася відповідно до певних вимог. У процесі сертифікації відбувається взаємодія заявника, органа, що сертифікує і наглядового органу.

Заявник - це організація, що подає заявку у відповідний сертифікуючий орган на отримання сертифіката (відповідності, якості, придатності і т.п.) вироби.

Сертифікуючий орган - організація, яка розглядає заявку заявника про проведення сертифікації *ПО* і або самостійно, або шляхом формування спеціальної комісії проводить набір процедур, спрямованих на проведення процесу сертифікації *ПО* заявника.

Наглядова орган - комісія фахівців, що спостерігають за процесами розробки заявником сертифікується інформаційної системи і дають висновок про відповідність даного процесу певним вимогам, яке передається на розгляд до сертифікуючий орган. [34]

Основний *об'єкт* перевірки в ході сертифікаційних випробувань - чи задовольняє процес розробки програмної системи регламентом і рекомендаціям стандарту, на відповідність якому проводиться *сертифікація.* Така відповідність визначається за допомогою аналізу життєвого *циклу* сертифікується системи та документів, що створюються на ключових його етапах. Весь процес аналізу й ті властивості системи, які піддаються сертифікації, описується в плані сертифікаційних випробувань, який затверджується спільно заявником та сертифікуючим органом.

У разі сертифікації бортової системи *за* стандартом DO-178B (або його аналогам КТ-178, JB-12 і т.п.) план додатково визначає рівень впливу відмови програмної системи на *безпеку* польоту (рівень отказобезопасность) *за* яким буде проводитися *сертифікація.* Будь-які питання, які виникають у сертифікує органу щодо змісту плану сертифікаційних випробувань, повинні бути дозволені до початку самих випробувань.

Згідно вимог DO-178B план сертифікаційних випробувань (план програмних аспектів сертифікації) повинен включати:

- **огляд системи.** Цей розділ описує систему, включаючи опис її функцій та їх розміщення в програмне і апаратне забезпечення, її архітектуру, використовуваний процесор (процесори), апаратно-програмний інтерфейс, і особливості отказобезопасность;
- **огляд програмного забезпечення.** Цей розділ коротко описує функції програмного забезпечення з акцентом на концепцію забезпечення отказобезопасность і поділу на відокремлені частини, наприклад, розподіл ресурсів, резервування, несиметрично резервувати програмне забезпечення, *стійкість до відмов,* стратегії таймування і діспетчерізірованія;
- **сертифікаційні міркування.** Цей розділ містить зведення сертифікаційного базису, включаючи засоби підтвердження відповідності, як це визначається програмними аспектами сертифікації. У цьому розділі також заявляється запропонований рівень (рівні) програмного забезпечення та наводяться підтвердження правильності цього рівня, отримані в процесі оцінки отказобезопасность системи, включаючи потенційний внесок програмного забезпечення в відмовні ситуації;
- **життєвий цикл програмного забезпечення.** Цей розділ визначає *життєвий цикл програмного забезпечення,* який буде використовуватися, а також включає зведення його процесів, детальна інформація про яких визначається у відповідних планах програмного забезпечення. У зведенні роз'яснюється, як будуть задовольнятися цілі кожного процесу життєвого циклу, вказуються залучаємо організації, організаційна відповідальність, а також відповідальність за процеси життєвого циклу системи та за процес підтримки контактів в ході сертифікації;
- **дані життєвого циклу програмного забезпечення.** Цей розділ визначає дані життєвого циклу, які будуть випущені і контролюватимуться в процесах життєвого циклу програмного забезпечення. Цей розділ також описує взаємозв'язок даних між собою або з іншими даними, що визначають систему, дані життєвого циклу програмного забезпечення, що подаються сертифікуючим владі, форму даних і засоби, за допомогою яких дані життєвого циклу програмного забезпечення можуть бути зроблені доступними для сертифікуючих влади;

- **план-графік.** Цей розділ описує засоби, які заявник буде використовувати для того, щоб забезпечити для сертифікуючих влади обозримость діяльності в процесах життєвого циклу програмного забезпечення і, отже, можливість планування перевірок;
- **додаткові міркування.** Цей розділ описує особливості, які можуть вплинути на процес сертифікації, наприклад, альтернативні методи підтвердження відповідності, кваліфікацію інструментальних засобів, раніше розроблене програмне забезпечення, вариантное програмне забезпечення, яке може бути вибрано за бажанням, програмне забезпечення, доступне для модифікації користувачем, готове програмне забезпечення *COTS,* використовуване без модифікацій, програмне забезпечення, завантажуване в польових умовах, несиметрично резервувати програмне забезпечення або використання історії експлуатації продукту.

У процесі самих сертифікаційних випробувань заявник надає свідчення того, що *процеси життєвого циклу* програмного забезпечення задовольняють планам програмного забезпечення. Заявник організовує *доступ* органа, що сертифікує до даних життєвого *циклу* програмного забезпечення. При цьому мінімальний перелік цих даних включає в себе:

- план сертифікаційних випробувань (план програмних аспектів сертифікації);
- індекс зміни програмного забезпечення - документ, який повинен однозначно ідентифікувати кожен компонент проекту (включаючи вимоги, вихідні коди, об'єктний і виконуваний код), середу реалізації системи, інструкції з компіляції системи, апаратне та програмне забезпечення для роботи системи, апаратне і програмне забезпечення для проведення сертифікації.
- підсумковий висновок про програмне забезпечення.

Підсумкове висновок *з* програмного забезпечення є основним документом для демонстрації відповідності програмного забезпечення Плану програмних аспектів сертифікації. Підсумкове висновок повинен включати:

- **огляд системи.** Цей розділ містить огляд системи, включаючи опис її функцій і їх розміщення в апаратному та програмному забезпеченні, архітектуру, використовуваний процесор (процесори), апаратно-програмний інтерфейс, засоби забезпечення отказобезопасность. У цьому розділі також описуються всі відмінності від опису системи, раніше поміщеного в план програмних аспектів сертифікації;

- **огляд програмного забезпечення.** Цей розділ коротко описує функції програмного забезпечення (особлива увага приділяється використовуваної концепції отказобезопасность і поділу на відокремлені частини), а також роз'яснює відмінності від огляду програмного забезпечення, раніше поміщеного в план програмних аспектів сертифікації;
- **сертифікаційні міркування.** Цей розділ повторно формулює сертифікаційні міркування, наведені в плані програмних аспектів сертифікації, а також описує будь-які відмінності від раніше наведених міркувань;
- **характеристики програмного забезпечення.** Цей розділ констатує дані про розмір виконуваного коду, запасах за часом і пам'яті, обмеженнях ресурсів, а також описує засоби для вимірювання кожної характеристики;
- **життєвий цикл програмного забезпечення.** Цей розділ підсумовує реальний життєвий цикл (цикли) програмного забезпечення і роз'яснює відмінності від життєвого циклу програмного забезпечення і процесів життєвого циклу, раніше запропонованих в плані програмних аспектів сертифікації;
- **дані життєвого циклу програмного забезпечення.** Цей розділ дає посилання на дані життєвого циклу програмного забезпечення, віднайдені в процесах розробки програмного забезпечення та процесах забезпечення цілісності. Він описує взаємозв'язок даних між собою та з іншими даними, що визначають систему, і засоби, за допомогою яких до даних життєвого циклу програмного забезпечення може бути забезпечений доступ з боку сертифікуючих влади. Цей розділ також описує будь-які відмінності від опису даних життєвого циклу, раніше поміщеного в плані програмних аспектів сертифікації;
- **додаткові міркування.** Цей розділ підсумовує питання, які можуть привернути увагу сертифікуючих влади, і дає посилання на дані, застосовні до цих питань, такі, як випущені документи або спеціальні умови;
- **ідентифікація програмного забезпечення.** Цей розділ ідентифікує конфігурацію програмного забезпечення по номенклатурному номеру або версії;
- **історія змін.** Цей розділ, якщо це доречно, включає зведення змін програмного забезпечення. Особлива увага приділяється змінам, які зроблені для виправлення помилок, що впливають на отказобезопасность, а також ідентифікацію змін у процесах життєвого циклу програмного забезпечення з часу попередньої сертифікації;

- **статус програмного забезпечення.** Цей розділ містить зведення повідомлень про проблеми, не дозволених на момент сертифікації, включаючи заяви про функціональні обмеженнях;
- **заяву про відповідність.** Цей розділ включає заяву про відповідність програмного забезпечення з цим документом, а також зведення методів, використаних для демонстрації відповідності із зазначенням критеріїв, які специфіковані в планах програмного забезпечення. У цьому розділі також зазначаються додаткові, по відношенню до планів програмного забезпечення, стандартам і цьому документу, використані правила і відхилення від планів, стандартів і цього документа.

Повний перелік даних життєвого *циклу,* які можуть знадобитися при сертифікації, включає в себе:

- план програмних аспектів сертифікації;
- план розробки програмного забезпечення;
- план верифікації програмного забезпечення;
- план управління конфігурацією програмного забезпечення;
- план гарантії якості програмного забезпечення;
- стандарти на вимоги до програмного забезпечення;
- стандарти проектування програмного забезпечення;
- стандарти на код програмного забезпечення;
- дані вимог на програмне забезпечення;
- опис проекту;
- вихідний текст;
- виконуваний об'єктний код;
- тестові приклади і тестові процедури верифікації програмного забезпечення;

- звіт за результатами верифікації програмного забезпечення;
- індекс зміни навколишнього середовища життєвого циклу програмного забезпечення;
- індекс зміни програмного забезпечення;
- повідомлення про проблеми;
- документи з управління конфігурацією програмного забезпечення;
- документи по гарантії якості програмного забезпечення;
- підсумковий висновок з програмного забезпечення.

Сертифікуючий орган встановлює т.зв. сертифікаційний *базис* для системи в ході консультацій із заявником. Сертифікаційний *базис*визначає конкретні правила разом з будь-якими спеціальними умовами, які можуть доповнювати опубліковані правила сертифікації, регламентовані стандартом.

Для програмного забезпечення установка базису проводиться *з* розгляду підсумкового висновку про програмне забезпечення та свідоцтв відповідності.

В ході сертифікації сертифікуючий орган оцінює план програмних аспектів сертифікації на повноту і узгодженість з критеріями оцінки отказобезопасность системи й іншими даними життєвого *циклу* програмного забезпечення. Якщо вірні всі дані життєвого *циклу,* що є доказом того, що в ході проекту були активні всі необхідні процеси розробки та верифікації, то сертифікуючий орган видає позитивне рішення про видачу сертифіката.

Сертифікати на *програмне забезпечення* можна віднести до двох типів: сертифікати відповідності та сертифікати якості.

- **Сертифікат якості** - свідоцтво, що засвідчує якість фактично поставленого товару та його відповідність умовам договору. У сертифікаті якості дається характеристика товару або підтверджується відповідність товару певним стандартам чи технічним умовам замовлення. Сертифікат якості видається компетентними організаціями, торговими палатами, спеціальними лабораторіями як у країні експорту, так і імпорту. Сторони договору купівлі-продажу можуть домовитися про надання сертифікатів різних контрольних і перевірочних установ.

- **Сертифікат відповідності** - результат дій третьої сторони (документ), що підтверджує впевненість у тому, що належним чином ідентифікована продукція, процес або послуга відповідають конкретному стандарту чи іншому нормативному документу.

Сертифікат на льотну придатність в розглянутому прикладі поєднує в собі властивості обох типів сертифікатів. З одного боку, він засвідчує, що розроблена система має певний рівень якості реалізації, а з іншого - що процеси *з* її розробки відповідають міжнародному авіаційному галузевому стандарту.

23.2. Інтеграційне тестування

23.2.1. Завдання і цілі інтеграційного тестування

Результатом тестування та верифікації окремих модулів, що складають програмну систему, є висновок про те, що ці модулі є внутрішньо несуперечливими і відповідають вимогам. Однак окремі модулі рідко функціонують самі по собі, тому наступна задача після тестування окремих модулів - тестування коректності взаємодії декількох модулів, об'єднаних в єдине ціле. Таке тестування називають інтеграційним. Його мета - упевнитися в коректності спільної роботи компонент системи.

Інтеграційне тестування називають ще тестуванням архітектури системи. З одного боку, це назва пояснюється тим, що, інтеграційні тести містять у собі перевірки всіх можливих видів взаємодій між програмними модулями і елементами, які визначаються в архітектурі системи, - таким чином, інтеграційні тести перевіряють повноту взаємодій в тестованої реалізації системи. З іншого боку, результати виконання інтеграційних тестів - один з основних джерел інформації для процесу поліпшення та уточнення архітектури системи, міжмодульних і межкомпонентних інтерфейсів. Тобто з цієї точки зору інтеграційні тести перевіряють коректність взаємодії компонент системи.

У результаті проведення інтеграційного тестування та усунення всіх виявлених дефектів виходить погоджена та цілісна архітектура програмної системи, тобто можна вважати, що інтеграційне тестування - це тестування архітектури та низькорівневих функціональних вимог.

Інтеграційне тестування, як правило, являє собою ітеративний процес, при якому перевіряється функціональність все більш і більш збільшується в розмірах сукупності модулів.

23.2.2. Завдання і цілі інтеграційного тестування

23.2.2.1. Структурна класифікація методів інтеграційного тестування

Як правило, інтеграційне тестування проводиться вже по завершенні модульного тестування для всіх інтегровних модулів. Однак це далеко не завжди так. Існує кілька методів проведення інтеграційного тестування:

- *висхідне тестування;*
- *монолітне тестування;*
- *спадне тестування;*

Висхідне тестування. При використанні цього методу мається на увазі, що спочатку тестуються всі програмні модулі, що входять до складу системи, і тільки потім вони об'єднуються для інтеграційного тестування. При такому підході значно спрощується локалізація помилок: якщо модулі протестовані окремо, то помилка при їх спільній роботі є проблема їх інтерфейсу. Тоді область пошуку проблем у тестувальника стає досить вузькою, а тому набагато вища ймовірність правильно ідентифікувати дефект.

Однак у *висхідного методу* тестування є істотний недолік - необхідність у розробці драйвера і заглушок для модульного тестування перед проведенням інтеграційного тестування і необхідність у розробці драйвера і заглушок при інтеграційному тестуванні частини модулів системи.

З одного боку, драйвери і заглушки - потужний інструмент тестування, з іншого - їх розробка потребує значних ресурсів, особливо при зміні складу інтегруються модулів. Тобто може знадобитися один набір драйверів для модульного тестування кожного модуля, окремий драйвер і заглушки для тестування інтеграції двох модулів з набору, окремий - для тестування інтеграції трьох модулів і т.п.У першу чергу, причина в тому, що при інтеграції модулів відпадає необхідність у деяких заглушках, а також потрібна зміна драйвера, яка підтримуватиме нові тести, що зачіпають кілька модулів.

Монолітне тестування передбачає, що окремі компоненти системи серйозного тестування не проходили. Основна перевага даного методу - відсутність необхідності в розробці тестового оточення, драйверів і заглушок. Після розробки всіх модулів виконується їх інтеграція, потім система перевіряється вся в цілому, як вона є. Цей підхід не слід плутати з системним тестуванням.Незважаючи на те, що при монолітному тестуванні перевіряється робота всієї системи в цілому, основне завдання

цього тестування - визначити проблеми взаємодії окремих модулів системи. Завданням же системного тестування є оцінка якісних і кількісних характеристик системи з точки зору їх прийнятності для кінцевого користувача.

Проте, *монолітне тестування* має низку серйозних недоліків:

- дуже важко виявити джерело помилки (ідентифікувати помилковий фрагмент коду);
- важко організувати виправлення помилок;
- процес тестування погано автоматизується.

Спадний тестування передбачає, що процес інтеграційного тестування рухається слідом за розробкою. Спочатку при низхідному підході тестують тільки самий верхній керуючий рівень системи, без модулів більш низького рівня. Потім поступово з більш високорівневими модулями інтегруються більш низькорівневі. В результаті застосування такого методу відпадає необхідність в драйверах (роль драйвера виконує більше високорівнева модуль системи), проте зберігається потреба в заглушках.

23.2.2.2. Тимчасова класифікація методів інтеграційного тестування

На практиці найчастіше в різних частинах проекту застосовуються всі розглянуті в попередньому розділі методи в сукупності. Кожен модуль тестують по мірі готовності окремо, а потім включають у вже готову композицію. Для одних частин тестування виходить низхідним, для інших - висхідним. У зв'язку з цим видається корисним розглянути ще один тип класифікації типів інтеграційного тестування - класифікацію за частотою інтеграції:

- тестування з пізньої інтеграцією;
- тестування з постійною інтеграцією;
- тестування з регулярною або пошаровим інтеграцією.

Тестування з пізньої інтеграцією - практично повний аналог монолітного тестування. Інтеграційне тестування при такій схемі відкладається на якомога більш пізні терміни проекту. Цей підхід виправдовує себе в тому випадку, якщо система являє собою

конгломерат слабко пов'язаних між собою модулів, які взаємодіють з якого-небудь стандартного інтерфейсу, визначеним поза проекту (наприклад, у випадку, якщо система складається з окремих Web-сервісів).

Схематично тестування з пізньої інтеграцією може бути зображено у вигляді ланцюжка RCVRCVRCVIRCVRCVI, де R - розробка вимог на окремий модуль, C - розробка програмного коду, V - тестування модуля, I - інтеграційне тестування всього, що було зроблено раніше.

Тестування з постійною інтеграцією увазі, що як тільки розробляється новий модуль системи, він відразу ж інтегрується з усією іншою системою. Тести для цього модуля перевіряють як суто його внутрішню функціональність, так і його взаємодію з іншими модулями системи. Таким чином, цей підхід поєднує в собі модульне тестування та інтеграційне. Розробки заглушок при такому підході не потрібно, але може знадобитися розробка драйверів. В даний час саме цей підхід називають *unit testing*, незважаючи на те, що на відміну від класичного модульного тестування тут не перевіряється функціональність ізольованого модуля. Локалізація помилок міжмодульних інтерфейсів при такому підході дещо ускладнена, але все ж значно нижче, ніж при монолітному тестуванні.Велика частина таких помилок виявляється досить рано саме за рахунок частоти інтеграції і за рахунок того, що за одну ітерацію тестування перевіряється порівняно невелике число міжмодульних інтер фейсом.

Схематично тестування з постійною інтеграцією може бути зображено у вигляді ланцюжка RCIRCIRCI, в якій *фаза тестування* модуля навмисно опущена і замінена на тестування інтеграції.

При **тестуванні з регулярною або пошаровим інтеграцією** інтеграційному тестуванню підлягають сильно пов'язані між собою групи модулів (шари), які потім також інтегруються між собою. Такий вид інтеграційного тестування називають також ієрархічним інтеграційним тестуванням, оскільки укрупнення інтегрованих частин системи, як правило, відбувається за ієрархічним принципом.Однак, на відміну від спадного або висхідного тестування, напрямок проходу по ієрархії в цьому підході не задано.

Таблиця 23.1. Основні характеристики різних видів інтеграційного тестування

	Висхідний	**Спадний**	**Монолітне**	**Пізня інтеграція**	**Постійна інтеграція**	**Регулярна інтеграція**

Час інтеграції	пізно (після тестування модулів)	рано (паралельно з розробкою)	пізно (після розробки всіх модулів)	пізно (після розробки всіх модулів)	рано (паралельно з розробкою)	рано (паралельно з розробкою)
Частота інтеграції	Рідко	часто	рідко	рідко	часто	часто
Чи потрібні драйвери	Так	немає	немає	немає	да	да
Чи потрібні заглушки	Так	да	немає	немає	немає	да

Таблиця 23.1 представляє основні характеристики розглянутих вище видів інтеграційного тестування. Час інтеграції характеризує момент часу, коли проводиться перша інтеграційне тестування і всі наступні, частота інтеграції - наскільки часто при розробці виконується інтеграція. Необхідність в драйверах і заглушках визначена в останніх двох рядках таблиці.

23.2.3. На прикладі "калькулятор матеріалів"

Як вже зазначалося, в MVSTE під unit-testing мається на увазі саме інтеграційне тестування, а конкретно - тестування з постійною інтеграцією. У"Автоматизація модульного тестування" ми вже протестували метод RunEstimate (), при інтеграції класів AnalaizerClass і CalcClass.Аналогічно, склавши вимоги до цієї підсистемі з двох класів (а це будуть вимоги до всіх методів AnalaizerClass), можна провести наступний етап тестування. Як приклад протестуємо всі методи такої підсистеми, зробивши по одному тестовому наприклад на кожен метод:

```
/ / / <summary>
    / / / A test for RunEstimate ()
    / / / Перевіряємо, що, якщо в стеку знаходиться коректний вираз, представлене зворотної польської записом, то
    / / / Метод RunEstimate правильно вважатиме це вираз
```

```
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void RunEstimateTest ()
{
    string expected = "3";
    string actual;
    TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.opz =
        new ArrayList ();
    ArrayList _opz =
        TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.opz;
    _opz.Add ("7");
    _opz.Add ("8");
    _opz.Add ("+");
    _opz.Add ("5");
    _opz.Add ("/");

    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.RunEstimate ();
```

```
    Assert.AreEqual (expected, actual,
                "BaseCalculator.AnalaizerClass.RunEstimate did not return the expected value.");
}

/ / / <summary>
/ / / A test for CheckCurrency ()
/ / / Перевіряє, що, якщо у виразі порушена Дужковий структура, то метод повертає false
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void CheckCurrencyTest ()
{
    bool expected = false;
    bool actual;

    TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
        "((5 +3) * 2 - (3 * 10)) -2) + ((5 +3)";
    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.CheckCurrency ();
```

```
    Assert.AreEqual (expected, actual,
                "BaseCalculator.AnalaizerClass.CheckCurrency did not return the expected value.");
    Assert.AreEqual (18,
                TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.erposition,
                "BaseCalculator.AnalaizerClass.CheckCurrency did not return the expected value.");
}

/ / / <summary>
/ / / A test for CreateStack ()
/ / / Перевіряє, що якщо отформатированное вираз дорівнює
/ / / "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)",
/ / / То стек містить такі елементи "5 3 + 2 * 3 10 * - 2 - 5 3 + +"
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void CreateStackTest ()
{
    string expected = "5 3 + 2 * 3 10 * - 2 - 5 3 + +";
    ArrayList actual;
```

```
        TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
                        "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)";


actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Create
Stack ();

        string actualstr = "";
        foreach (object obj in actual)
        {
            actualstr + = obj.ToString () + "";
        }
        Assert.AreEqual (expected, actualstr,
                    "BaseCalculator.AnalaizerClass.CreateStack did not return the expected value.");
    }

    / / / <summary>
    / / / A test for Estimate ()
```

```
/ / / Перевіряє, що, якщо вираз, який необхідно перевірити, одно
/ / / ((5 +3) * 2 - (3 * 10)) -2 + (5 +3), то метод поверне його значення,
/ / / Рівне -8
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void EstimateTest ()
{
    string expected = "-8";
    string actual;

   TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
          "((5 +3) * 2 - (3 * 10)) -2 + (5 +3)";
    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Estimate ();

    Assert.AreEqual (expected, actual,
           "BaseCalculator.AnalaizerClass.Estimate did not return the expected value.");

}
```

```
/ / / <summary>
/ / / A test for Format ()
/ / / Перевіряє, що якщо вираз дорівнює "
/ / / ((5 +3) * 2 - (3 * 10)) -2 + (5 +3) ", то метод отформатирует його до вигляду:
/ / / "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)"
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void FormatTest ()
{
    string expected = "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)";
    string actual;
    TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expression =
          "((5 +3) * 2 - (3 * 10)) -2 + (5 +3)";
    actual = TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Format ();

    Assert.AreEqual (expected, actual,
           "BaseCalculator.AnalaizerClass.Format did not return the expected value.");
```

}

23.1.

Запустивши тести, можна переконатися в коректній (для даних тестових прикладів) роботі методів.

Зауваження. Тестових вимог, як і функціональних, до методів немає, проте їх можна скласти з системних вимогам і опису методів.

Як вже зазначалося в "Модульне тестування" , ці методи мають багатьма недоліками, і деякі з них генерують виключення на некоректних вхідних даних. Це, в принципі, логічно, так як запуск методу RunEstimate () має на увазі, що вхідна вираз вже оброблено в методахCheckCurrency (), Format (), CreateStack (), а окремо від них цей метод викликатися не буде. Для цього необхідно зробити рівень доступу до них private і протестувати їх тільки на коректних даних. Основна ж увага варто приділити методу Estimate (), який по черзі запускає всі перераховані вище методи і має рівень доступу public. Один з тестів для нього наведений вище.

Після того, як така система з двох модулів протестована, переходимо до тестування всієї системи, приєднавши останні модулі.

Зауваження. Модуль BaseCalc, який відповідає за користувальницький інтерфейс, ми не тестуємо, так як це виходить за рамки курсу.

Тепер можна використовувати системні вимоги, для тестування програми в цілому. Проведемо тести для методу Main (string [] args),так як це єдиний метод, не рахуючи вже протестованого Estimate (), який не відноситься до графічного інтерфейсу і з яким працюють користувачі:

/ / / <summary>

/ / / A test for Main (string [])

/ / / 4.2.1.1. Для чисел, кожне з яких менше або дорівнює MAXINT і більше або дорівнює MININT,

/ / / Функція підсумовування повинна повертати правильну суму з точки зору математики

```
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void MainTest ()
{
    string [] args = new string [1];  / / TODO: Initialize to an appropriate value
    args [0] = "2 +2";

    int expected = 0;
    int actual;

    actual = TestProjectCalculator.BaseCalculator_ProgramAccessor.Main (args);
    Assert.AreEqual (expected, actual,
        "BaseCalculator.Program.Main did not return the expected value.");
}

/ / / <summary>
/ / / A test for Main (string [])
/ / / 4.2.1.2.  Для чисел, сума яких більше ніж MAXINT і менше ніж MININT,
```

```
/ / / А також у разі, якщо будь-який з доданків більше ніж MAXINT або менше ніж MININT,
/ / / Програма повинна видавати помилку Error 06 (см 2.2.3)
/ / / </ Summary>
[DeploymentItem ("BaseCalculator.exe")]
[TestMethod ()]
public void MainTest1 ()
{
    string [] args = new string [1];  / / TODO: Initialize to an appropriate value
    args [0] = "2711477380 +1000000";

    int expected = 6;
    int actual;

    actual = TestProjectCalculator.BaseCalculator_ProgramAccessor.Main (args);

    Assert.AreEqual (expected, actual,
        "BaseCalculator.Program.Main did not return the expected value.");
}
```

23.2.

І так далі. Таким чином, перевіривши методи Main () і Estimate (), а також деякі методи візуального інтерфейсу, можна переконатися у відповідності системи вимогам або, навпаки, виявити якісь помилки в межмодульного взаємодії.

Лекція 8:

Особливості індустріального тестування

Анотація: Розглядаються особливості підходу до забезпечення якості програмного продукту засобами тестування. Наводиться приклад і методика вибору критеріїв якості тестування. Визначаються фази процесу тестування і кроки тестового циклу, застосовувані в індустріальному тестуванні. Розглядається структура документа "Тестовий план". Розглядаються плановані типи тестування для різних частин продукту або для перевірки різних характеристик продукту. Описуються підходи до тестування специфікацій і сценаріїв. Наводиться річний підхід і підхід генерації тестових наборів при розробці тестів. Порівнюються методи автоматизації виконання тестів.

Ключові слова: якість
програмного продукта , представление , очередь , группа , вывод , корректность , browser ,функциональная повнота , матриця , аудит , зручність використання , модульне тестування , інтеграційне тестування , системне тестування , фаза тестування , тестовий цикл , Автоматизації тестування , тестова процедура , test log , тестовий план , тестова стратегія , автоматизована система тестування , рівень серйозності , тестування сценаріїв , функціональна специфікація ,формальна специфікація , мова скриптів

Якість програмного продукту і тестування

Якість програмного продукту можна оцінити деяким набором характеристик, що визначають, наскільки продукт "хороший" з погляду всіх потенційно зацікавлених у ньому сторін. Такими сторонами є:

- замовник продукту
- спонсор
- кінцевий користувач

- розробники продукту
- тестувальники продукту
- інженери підтримки
- відділ навчання
- відділ продажів і т.п.

Кожен з учасників може мати різне *уявлення* про продукт і по-різному судити про те, наскільки він хороший чи поганий, тобто наскільки висока якість продукту. З точки зору розробника, продукт може бути настільки хороший, наскільки хороші закладені в ньому алгоритми і технології. Користувачеві продукту, швидше за все, байдужі деталі внутрішньої реалізації, його в першу *чергу*хвилюють питання функціональності і надійності. Спонсора цікавить ціна і сумісність з майбутніми технологіями. Таким чином, завдання забезпечення якості продукту виливається у завдання визначення зацікавлених осіб, погодження їх критеріїв якості і знаходження оптимального рішення, що задовольняє цим критеріям.

У рамках такого завдання *група* тестування розглядається не просто як ще одна зацікавлена сторона, але і як сторона, здатна оцінити задоволення обраних критеріїв і зробити *висновок* про якість продукту з точки зору інших учасників. На жаль, далеко не всі критерії можуть бути оцінені групою тестування. Тому її увагу в основному зосереджено на критеріях, що визначають *якість програмного продукту* з точки зору кінцевого користувача.

Тестування як спосіб забезпечення якості. Тестування, з технічної точки зору, є процес виконання програми на деяких вхідних даних і перевірка одержуваних результатів з метою підтвердити їх *коректність* по відношенню до результату.

Тестування не позиціонується як єдиного способу забезпечення якості. Воно є частиною загальної системи забезпечення якості продукту, елементи якої вибираються за критерієм найбільшої ефективності застосування в конкретному проекті.

Розглянемо приклад. Як додаток візьмемо програму для роботи з мережею (*browser),* критерії якості якої наведені в Табл.9.1 .

Таблиця 9.1. Критерії якості програми browser

	Користувач	**Замовник**	**Інженер підтримки**
Функціональна повнота	+	-	-
Ціна розробки	-	+	-
Відсутність дефектів	+	Побічно	+
Зручність використання	+	-	-
Можливість внесення змін у майбутньому	-	Побічно	+
Легкість виправлення дефектів	-	-	+
Документація на реалізацію, в тому числі коментарі	-	-	+
Своєчасність виконання проекту	-	+	-

Матриця критеріїв якості зацікавлених в них учасників для розглянутого проекту наведена в таблиці 9.2 . Припустимо, що вид матриці критеріїв якості і перевіряючих елементів системи забезпечення якості для даного проекту буде наступним:

Таблиця 9.2. Матриця критеріїв якості та елементів системи забезпечення якості

	Тестування	**Аналіз ринку та спеціальні лабораторії** [1]	**Огляди коду**	**Аналіз дизайну**	**Аудити процесу розробки**
Повнота функціональності	+, Не завжди ефективно	+	-	-	-

Вартість розробки	-	-	-	-	+
Відсутність дефектів	+	-	+	-	-
Зручність використання	+, Не завжди ефективно	+	-	-	-
Можливість внесення змін у майбутньому	-	-	+	-	+	-
Легкість виправлення дефектів	-	-	+	+	-
Документація на реалізацію, в тому числі коментарі	-	-	+	-	+
Своєчасність виконання проекту	-	-	-	-	+

Дані (Табл. 9.1 , Табл. 9.2) показують, що з восьми елементів загальної якості продукту тестування здатне оцінити і контролювати тільки три (1, 3, 4), причому найбільш ефективно тестування контролює відсутність дефектів (3).

У кожному конкретному проекті елементи системи повинні бути вибрані так, щоб забезпечити прийнятну якість, виходячи з пріоритетів та наявних ресурсів. Вибираючи елементи для системи забезпечення якості конкретного продукту, можна застосувати комбіноване тестування, огляди коду, *аудит.* При подібному виборі деякі якості, наприклад легкість модифікації і виправлення дефектів, що не будуть оцінені і, можливо, виконані. Завданням тестування в розглянутому випадку буде виявлення дефектів і оцінка зручності використання продукту, включаючи повноту функціональності. Виходячи із завдань, поставлених перед групою тестування в конкретному проекті, вибирається

відповідна стратегія тестування. Так, в даному прикладі, зважаючи на необхідність оцінити *зручність використання* і повноту функціональності, переважний підхід до розробки тестів слід планувати на основі використання сценаріїв.

Отже, основна послідовність дій при виборі та оцінці критеріїв *якості програмного продукту* включає:

1. Визначення всіх осіб, так чи інакше зацікавлених у виконанні та результати даного проекту.
2. Визначення критеріїв, що формують уявлення про якість для кожного з учасників.
3. Пріоритезацію критеріїв, з урахуванням важливості конкретного учасника для компанії, що виконує проект, і важливості кожного з критеріїв для даного учасника.
4. Визначення набору критеріїв, які будуть відстежені і виконані в рамках проекту, виходячи з пріоритетів та можливостей проектної команди. Постановка цілей по кожному з критеріїв.
5. Визначення способів і механізмів досягнення кожного критерію.
6. Визначення стратегії тестування виходячи з набору критеріїв, що потрапляють під відповідальність групи тестування, обраних пріоритетів і цілей

Процес тестування

Як зазначалося в підрозділі 2.4, у тестуванні виділяються три основних рівня, або три фази:

1. *Модульне тестування.*
2. *Інтеграційне тестування.*
3. *Системне тестування.*

Завдання планування активності тестування полягає в оптимальному розподілі ресурсів між усіма типами тестування. У подальшому викладі ми сконцентруємося на системній *фазі тестування,* як на найбільш важливою і критичною активності для розробки якісного програмного продукту.

Фази процесу тестування

У процесі тестування виділяють наступні фази:

1. **Визначення цілей** (вимог до тестування), що включає наступну конкретизацію: які частини системи будуть тестуватися, які аспекти їхньої роботи будуть обрані для перевірки, яке бажану якість і т.п.
2. **Планування:** створення графіка (розкладу) розробки тестів для кожної тестованої підсистеми; оцінка необхідних людських, програмних і апаратних ресурсів; розробка розкладу *тестових циклів.* Важливо відзначити, що розклад тестування обов'язково має бути погоджено з розкладом розробки створюваної системи, оскільки наявність виконуваної версії розробляється**(Implementation Under Testing (IUT)** або **Application Under Testing (AUT)** - часто вживані позначення для тестованої системи) є одним з необхідних умов тестування , що створює взаємозалежність у роботі команд тестувальників і розробників.
3. **Розробка тестів,** тобто тестового коду для тестованої системи, якщо необхідно - коду системи *автоматизації тестування* і*тестових процедур* (виконуваних вручну).
4. **Виконання тестів:** реалізація *тестових циклів.*
5. **Аналіз результатів.**

Після аналізу результатів можливе повторення процесу тестування, починаючи з пунктів 3, 2 або навіть 1.

Тестовий цикл

Тестовий цикл - це цикл виконання тестів, що включає фази 4 і 5 тестового процесу. *Тестовий цикл* полягає в прогоні розроблених тестів на деякій однозначно визначається зрізі системи (стані коду розроблюваної системи). Зазвичай такий зріз системи називають **build.** *Тестовий цикл* включає наступну послідовність дій:

1. Перевірка готовності системи і тестів до проведення *тестового циклу* включає:
    - Перевірку того, що всі тести, заплановані для виконання на даному циклі, розроблені і поміщені в систему версионного контролю.
    - Перевірку того, що всі підсистеми, заплановані для тестування на даному циклі, розроблені і поміщені в систему версионного контролю.
    - Перевірку того, що розроблена і задокументована процедура визначення та створення зрізу системи, або build.
    - Перевірки деяких додаткових критеріїв.
2. **Підготовка тестової машини** відповідно до вимог, визначених на етапі планування (наприклад, повне очищення та перевстановлення системного програмного забезпечення). Конфігурація тестової машини, так само, як і зріз системи, повинні бути однозначно відтворюваними.
3. **Відтворення зрізу системи.**
4. **Прогін тестів** відповідно до задокументованими процедурами.
5. **Збереження тестових протоколів (test log).** *Test log* може містити висновок системи в STDOUT, список результатів порівняння отриманих при виконанні даних з еталонними або будь-які інші вихідні дані тестів, за допомогою яких можна перевірити правильність роботи системи.
6. **Аналіз протоколів тестування** та прийняття рішення про те пройшов або не пройшов кожен з тестів **(Pass / Fail).**

7. **Аналіз і документування** результатів циклу.

Останній перед випуском продукту *тестовий цикл* не повинен включати змін коду build або коду продукту тестованої системи. Цей цикл називається "фінальним". Таким чином забезпечується ситуація, коли фінальний цикл повністю повторюємо, а випускається продукт повністю збігається з продуктом, який пройшов тестування. Фінальний цикл необхідний для гарантії достовірності результатів тестування.

Планування тестування

Тестовий план

Тестовий план - це документ, або набір документів, що містить наступну інформацію:

1. Тестові ресурси.
2. Перелік функцій і підсистем, що підлягають тестуванню.
3. *Тестову стратегію,* що включає:
    - Аналіз функцій і підсистем з метою визначення найбільш слабких місць, тобто областей функціональності тестованої системи, де поява дефектів найбільше ймовірно.
    - Визначення стратегії вибору вхідних даних для тестування. Так як безліч можливих вхідних даних програмного продукту, як правило, практично нескінченно, вибір кінцевого підмножини, достатнього для проведення вичерпного тестування, є складним завданням. Для її рішення можуть бути застосовані такі методи, як покриття класів вхідних і вихідних даних, аналіз крайніх значень, покриття моделі використання, аналіз тимчасової лінії і тому подібні. Обрану стратегію необхідно обгрунтувати і задокументувати.
    - Визначення потреби в *автоматизованій системі тестування* і дизайн такої системи

4. Розклад *тестових циклів* (приклад наведено на Рис. 9.1).
5. Фіксацію тестовій конфігурації: складу і конкретних параметрів апаратури та програмного оточення (приклад наведено на Рис. 9.2).
6. Визначення списку тестових метрик, які на *тестовому циклі* необхідно зібрати і проаналізувати. Наприклад, метрик, що оцінюють ступінь покриття тестами набору вимог, ступінь покриття коду тестованої системи, кількість і *рівень серйозності* дефектів, обсяг тестового коду та інші характеристики.

The test cycle will be an interactive process. Specifics for each test cycle are described in the following tables

Table 1: Test Category Overview

Test Categories	Test to be Conducted	Activities	Deliverables
System test cycle	tests described in items 6.1 - 6.7 shall be conducted	- Set up test machines - Run all the test suites - Analyze the results - Log defects - Verify resolved defects - Generate Test Report chapter - Store results in vault	Updated Test Report Defect Reports
Final test cycle	tests described in items 6.1 - 6.7 shall be conducted	- Set up test machines - Run the test suites - Analyze the results - Verify resolved defects - Generate Test Report chapter - Store results in vault	Updated Test Report

Рис. 9.1. Приклад розкладу двох останніх тестових циклів

Table 2: System Test Details

Test Day	Ultra Sparc 2 station 200MHz, 256M RAM
1	Prepare the test machine and the test suite for a test cycle. Perform the installation tests and the manual tests. Perform the automated tests.
2	Perform the automated tests. Perform the performance tests. Log results and time to the Test Report. Log new defects to DDTS. Store test results to Vault-mst/test/ALM/Logs. Update the Test Report.

Table 3: Final Test Details

Test Day	Ultra Sparc 2 station 200MHz, 256M RAM
1	Prepare the test machine and the test suite for a test cycle. Perform the installation tests and the manual tests. Perform the automated tests. Perform the performance tests. Log results and time to the Test Report. Store test results to Vault-mst/test/ALM/Logs. Verify resolved defects in DDTS. Complete the Test Report.

Рис. 9.2. Приклад деталізації умов проведення системних циклів

Типи тестування

У *тестовому плані* визначаються і документуються різні типи тестів. Типи тестів можуть бути класифіковані за двома категоріями: по тому, що піддається тестуванню (по виду підсистеми) і за способом вибору вхідних даних.

Типи тестування по виду підсистеми або продукту:

1. **Тестування основної функціональності,** коли тестуванню піддається власне система, що є основним випускається продуктом
2. **Тестування інсталяції** включає *тестування сценаріїв* первинної інсталяції системи, сценаріїв повторної інсталяції (поверх вже існуючої копії), тестування деінсталяції, тестування інсталяції в умовах наявності помилок в інсталюється пакеті, в оточенні або в сценарії і т.п.
3. **Тестування для користувача документації** включає перевірку повноти і зрозумілості опису правил та особливостей використання продукту,

наявність опису всіх сценаріїв і функціональності, синтаксис і граматику мови, працездатність прикладів і т.п.

Типи тестування за способом вибору вхідних значень:

1. **Функціональне тестування,** при якому перевіряється:
    - Покриття функціональних вимог.
    - Покриття сценаріїв використання.
2. **Стресове тестування,** при якому перевіряються екстремальні режими використання продукту.
3. **Тестування граничних значень.**
4. **Тестування продуктивності.**
5. **Тестування на відповідність стандартам.**
6. **Тестування сумісності** з іншими програмно-апаратними комплексами.
7. **Тестування роботи з оточенням.**
8. **Тестування роботи на конкретній платформі**

У реальних розробках використовуються і комбінуються різні типи тестів для забезпечення спланованого якості продукту.

Підходи до розробки тестів

Розглянемо різні підходи до розробки тестів, два до вибору тестових даних і два до реалізації тестового коду.

Тестування специфікації

При розробці тестів, заснованих на *функціональної специфікації* продукту, вимоги до продукту є основним джерелом, що визначає, які тести будуть розроблені. Для кожної вимоги пишеться один або більше тестів, які в сукупності повинні перевірити виконання даної вимоги в продукті.

Приклад використання специфікації вимог для розробки тестів.

Нехай заданий наступний фрагмент набору вимог для моделі обміну транзакціями:

1. Функція DoTransaction повинна приймати адреса і дані відповідно до параметрів, створювати в черзі новий елемент, заповнювати його адресну частину і частину полів даних переданої інформацією та ініціювати транзакцію
2. Функція DoAddressTenure повинна приймати адреса відповідно до параметрів, створювати в черзі новий елемент і заповнювати його адресну частину
3. Функція DoDataTenure повинна приймати дані відповідно до параметрів, знаходити в черзі перший елемент з частково незаповненими полями даних, доповнювати його переданої інформацією та ініціювати транзакцію

Концептуальне опис набору тестів, перевіряючого специфікацію, може виглядати наступним чином:

1. Викликати DoTransaction з адресою та даними. Перевірити поява в черзі ще одного елемента. Перевірити поява на шині транзакції з правильними адресою і даними.
2. Викликати DoAddressTenure з адресою. Перевірити поява в черзі ще одного елемента. Перевірити відсутність нової транзакції на шині.
3. Викликати DoDataTenure з даними. Перевірити заповнення полів даних. Перевірити поява на шині транзакції з правильними адресою та даними

Тестування сценаріїв

Розробка тестів, заснованих на використанні сценаріїв, здійснюється за наступною методикою:

1. Визначається модель використання, що включає операційне оточення продукту і "акторів". Актором може бути користувач, інший продукт,

апаратна частина тощо, тобто все, з чим продукт обмінюється інформацією. Поділ на оточення і акторів умовно і служить для опису оптимальних способів використання продукту.

2. Розробляються сценарії використання продукту. Опис сценарію залежно від продукту і вибраного підходу може бути строго певним, параметризованим або дозволяти деяку ступінь невизначеності. Наприклад, опис сценарію на мові MSC допускає завдання параметрезованих сценаріїв з можливістю переупорядочивания подій.
3. Розробляється набір тестів, що покривають задані сценарії. З урахуванням ступеня невизначеності, закладеної в сценарії, кожен тест може покривати один сценарій, кілька сценаріїв, або, навпаки, частина сценарію.

Використання сценаріїв не вимагає наявності повної *формальної специфікації* вимог, але зате може зажадати більше часу на розробку та аналіз.

Ще одна особливість *тестування сценаріїв* полягає в тому, що цей метод направляє тестування на перевірку конкретних режимів використання продукту, що дозволяє знаходити дефекти, які метод тестування за вимогами може пропустити.

Приклад використання специфікації вимог для розробки тестів.

Так, для розглянутого вище прикладу можливе створення наступного сценарію і тестів.

1. Сценарій: користувач має дві незалежні нитки управління, одна з яких відповідає за генерацію повних транзакцій за допомогоюDoTransaction, а інша - за збір транзакцій з адресної частини і частини даних, коли ця інформація приходить з різних джерел. Таким чином, друга нитка використовує виклики до DoAddressTenure і DoDataTenure.
2. Опис тестів: Викликати DoAddressTenure с адресою A1, викликати DoTransaction з адресою A2 і

даними D2,викликати DoDataTenure з даними D1. Перевірити послідовне поява на шині двох транзакцій: {A1, D1} і {A2, D2}

При виконанні цього тесту було, зокрема, виявлено, що функція DoTransaction була реалізована через виклики доDoAddressTenure і DoDataTenure, що призводило до появи на шині транзакцій виду {A1, D2} і {A2, D1}. Подібний дефект може бути виявлений з великими труднощами, якщо розробляти тести, грунтуючись тільки на специфікації вимог.

Ручна розробка тестів

Найбільш поширеним способом розробки тестів є створення тестового коду вручну. Це найбільш гнучкий спосіб розробки тестів, однак характерна нього продуктивність праці інженерів-тестувальників у створенні тестового коду не набагато вище швидкості створення коду продукту, а обсяги тестового коду на практиці найчастіше перевищують обсяг коду продукту в 10 разів. Враховуючи цей факт, в сучасній індустрії все більше схиляються до більш інтелектуальним способам отримання тестового коду, таким як використання спеціальних тестових *мов (скриптів)* і генерації тестів.

Генерація тестів

В даний час деякі мови специфікацій, що використовуються для опису алгоритмів тестування, можуть бути використані для генерації тестового коду. Розглянемо генерацію коду з мови MSC. Тест, описаний вище, формалізований на мові MSC (Рис. 9.3).Тут кожна стрілка з позначкою DoTransaction, DoAddressTenure або DoDataTenure представлет собою виклик відповідної функції продукту з передачею параметрів. Стрілка checkTr відповідає перевірці проходження по шині транзакції з відповідними параметрами. Кожна зі стрілок діаграми генератором тестів перетвориться в виконані код, при цьому стрільцям, які представляють собою виклики функцій може відповідати досить простий і маленький ділянку коду, що викликає відповідну функцію і перевіряючий її вихідне значення на наявність помилок.

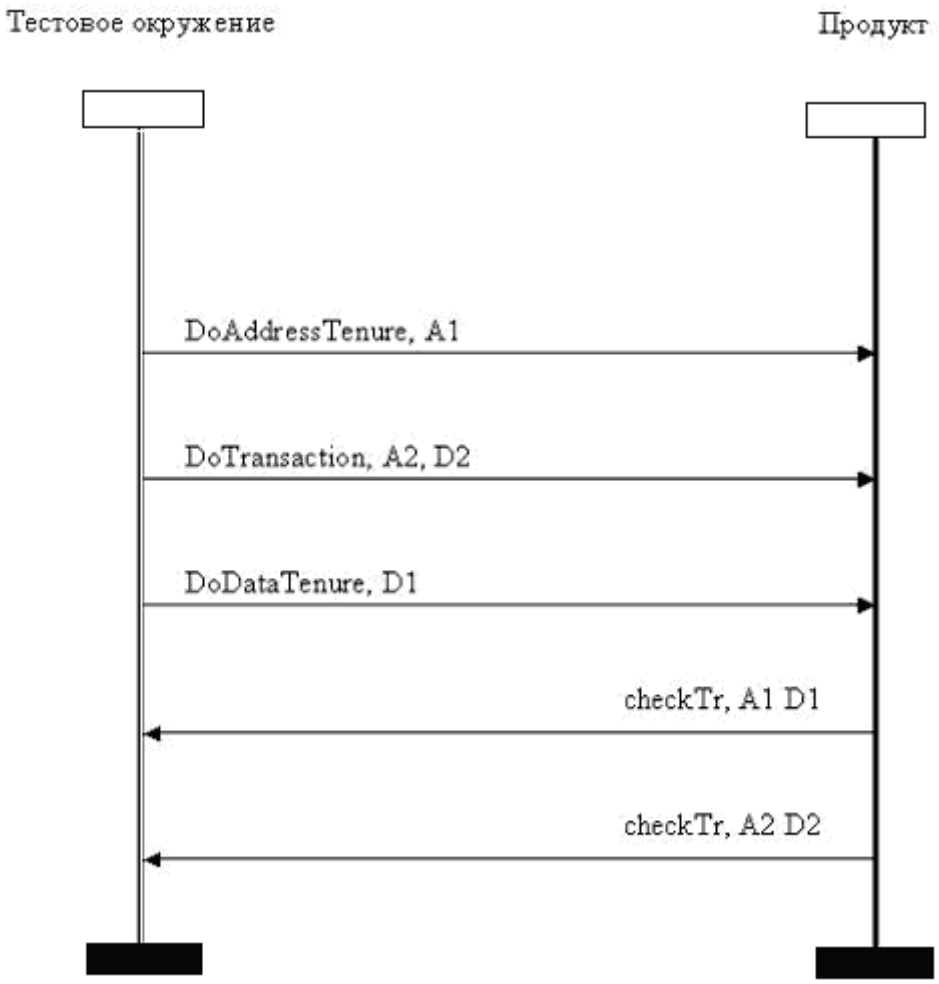


Рис. 9.3. Формальна запис сценарного тесту на MSC

Слід зазначити, що стрілки, відповідні перевірки транзакцій, можуть після генерації перетворитися в досить складний код, який буде виконувати очікування появи транзакції на шині протягом заданого при генерації часу - тайм-ауту, перевіряти фази транзакції і звіряти обчислені значення параметрів із заданими еталонними значеннями .

У результаті в розглянутому прикладі виграш від застосування генераційні підходу досягається в основному за рахунок використання наочного візуального представлення тестів, що може бути нівельовано витратами на створення генераційні сценарію на MSC.

Можлива куди більш ефективна формалізація MSC сценарію для генерації тестів. Рис. 9.4 представляє інший спосіб формалізації для тіста, що виконує ті ж самі перевірки.

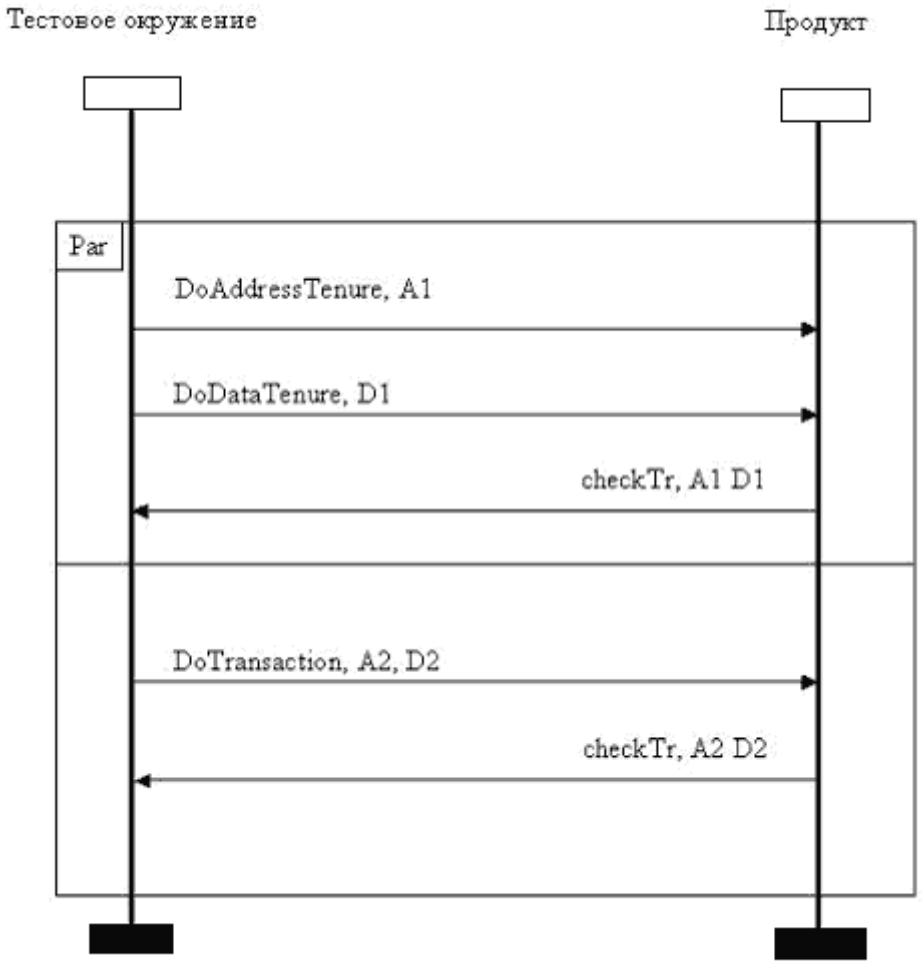


Рис. 9.4. Формальна запис сценарного тесту на MSC з використанням паралелізму.

У MSC на Рис. 9.4 перевірки транзакцій згруповані з породжують їх викликами в окремі фрагменти, а паралелізм, використовуваний при виконанні фрагментів, заданий через Par - формальну конструкцію, яка застосовується для зображення паралелізму в мові MSC. При генерації тестів по діаграмі Рис. 9.4 тестовий генератор перебирає всі можливі і неповторювані варіанти виклику тестованих функцій, зберігаючи при цьому коректність порядку перевірок, що в даному прикладі дає три згенерованих тесту.Нескладно бачити, що витрати на створення

діаграми Рис. 9.4 не сильно відрізняються від витрат на діаграму Рис. 9.3 , в той час як кількість тестів збільшується в три рази.

Таким чином, використання методики генерації тестового коду по формалізованим MSC диаграммам дозволяє значно підняти продуктивність тестування, а також перетворити формалізацію (кодування) сценаріїв у досить інтелектуальну діяльність.

Документування та оцінка індустріального тестування

Анотація: Описуються особливості документування тестових процедур для ручних і автоматизованих тестів, описів тестових наборів і тестових звітів. Розглядається життєвий цикл дефекту. Обговорюються метрики, використовувані при тестуванні.

Ключові слова: Ручне тестування , прогін тестів , інтерфейс командного рядка , тестовий цикл , тестовий набір , тестова процедура , Автоматизації тестування , тестовий звіт , якість даних , оглядів тестової стратегії

Виконання тестів

Розглянемо два основні підходи до виконання тестів: підхід *ручного тестування* і підхід автоматичного виконання (*прогін) тестів.*Підходи розглянуті на прикладі тестування продукту, підтримуючого *інтерфейс командного рядка.* Тести описують виклик продукту з параметрами і перевірку значення, що повертається у вигляді фіксованих при прогоні - тексту з STDOUT і стану деяких файлів, залежного від вхідних параметрів.

Ручне тестування

Ручне тестування полягає у виконанні задокументованої процедури, де описана методика виконання тестів, що задає порядок тестів і для кожного тесту - список значень параметрів, який подається на вхід, і список результатів, очікуваних на виході. Оскільки процедура призначена для виконання людиною, в її описі для стислості можуть використовуватися деякі значення за замовчуванням,

орієнтовані на здоровий глузд, або посилання на інформацію, що зберігається в іншому документі.

Приклад фрагмента процедури

1. Подати на вхід три різних цілих числа.
2. Запустити тестове виконання.
3. Перевірити, чи відповідає отриманий результат таблиці [посилання на документ1] з урахуванням поправок [посилання на документ2].
4. Переконатися в зрозумілості та коректності видаваної супровідної інформації.

У наведеній процедурі тестувальник використовує два додаткових документа, а також власне розуміння того, яку супровідну інформацію вважати "зрозумілою і коректної". Успіх від використання процедурного підходу досягається у випадку однозначного розуміння тестувальником всіх пунктів процедури. Наприклад, в п.1 наведеної процедури не уточнюється, з якого діапазону повинні бути задані три цілих числа, і не описується додатково, які числа вважаються "різними".

Автоматизоване тестування

Спроба автоматизувати Наведений вище тест призводить до створення скрипта, що задає тестируемому продукту три конкретних числа і перенаправляє висновок продукту у файл з метою його аналізу, а також містить конкретне значення бажаного результату, з яким звіряється одержуване при прогоні тесту значення. Таким чином, вся необхідна інформація повинна бути явно поміщена в текст (скрипт) тесту, що вимагає додаткових порівняно з ручним підходом зусиль. Також додаткових зусиль і часу вимагає створення розбирача виводу (програми узгодження форматів представлення еталонних значень з тіста і обчислюваних при прогоні результатів) і, можливо, створення бази зберігання станів еталонних даних.

Приклад скрипта

Наведемо приклад послідовності дій, які закладаються в скрипт:

1. Видати на консоль ім'я або номер тесту і час його початку.
2. Викликати продукт з фіксованими параметрами.
3. Перенаправити висновок продукту в файл.
4. Перевірити повернене продуктом значення. Воно має дорівнювати очікуваному (еталонному) результату, зафіксованому в тісті.
5. Перевірити висновок продукту, збережений у файлі (п.3), на рівність заздалегідь приготовленого еталону.
6. Видати на консоль результати тесту у вигляді вердикту PASS / FAIL і у випадку FAIL - короткого пояснення, яка саме перевірка не пройшла.
7. Видати на консоль час закінчення тесту.

Порівняння ручного та автоматизованого тестування

Результати порівняння наведені в Табл. 10.1 . Порівняння показує тенденцію сучасного тестування, ориентирующую на максимальну автоматизацію процесу тестування і генерацію тестового коду, що дозволяє справлятися з великими обсягами даних і тестів, необхідних для забезпечення якості при виробництві програмних продуктів.

Таблиця 10.1. Порівняння ручного та автоматизованого підходу

	Ручне	**Автоматизоване**
Завдання вхідних значень	Гнучкість у завданні даних. Дозволяє використовувати різні значення на різних циклах прогону тестів, розширюючи покриття	Вхідні значення строго задані

Перевірка результату	Гнучка, дозволяє тестувальника оцінювати нечітко сформульовані критерії	Сувора. Нечітко сформульовані критерії можуть бути перевірені тільки шляхом порівняння з еталоном
Повторюваність	Низька. Людський фактор і нечітке визначення даних призводять до неповторяемости тестування	Висока
Надійність	Низька. Тривалі *тестові цикли* призводять до зниження уваги тестувальника	Висока, не залежить від довжини*тестового циклу*
Чутливість до незначних змін в продукті	Залежить від детальності опису процедури. Зазвичай тестувальник в змозі виконати тест, якщо зовнішній вигляд продукту і текст повідомлень дещо змінилися	Висока. Незначні зміни в інтерфейсі часто ведуть до корекції еталонів
Швидкість виконання*тестового набору*	Низька	Висока
Можливість генерації тестів	Відсутня. Низька швидкість виконання зазвичай не дозволяє виконати згенерований набір тестів	

Тестові процедури

Тестові процедури - це формальний документ, що містить опис необхідних кроків для виконання *тестового набору*. У разі ручних тестів *тестові процедури* містять повний опис всіх кроків і перевірок, що дозволяють протестувати продукт і винести вердикт PASS / FAIL.

2.2 Hand Test

2.2.1 INST0021 - check license and quality status contents for the all targets.

2.2.1.1 Setup

Login to the Unix C shell environment as a new created test user.

2.2.1.2 Execution

[1] Change the current directory in Unix terminal window to the directory with Product files of yet untested target.
[2] Enter the following command at the Unix prompt: *Product.ins*.
[3] Enter the following answer at the Unix prompt: *y*.
[4] Enter the following command at the Unix prompt: *accept*.
[5] Enter the following command at the Unix prompt: *accept*.

Following steps are only to finish the installation correctly. Just press *Enter* to chose the default answer.

[6] Press *Enter*.
[7] Press *Enter*.
[8] Press *Enter*.

2.2.1.3 Validation

[1] The brief description of installation shall appear after step [2]. Prompt *OK to continue? [y]:* shall be active.
[2] The license agreement shall appear after step [3]. Check that the contents of the license agreement is correct.
[3] The quality status shall appear after step [4]. Check that the contents of the quality status is correct.
[4] Resume and "INSTALLATION IS SUCCESSFULLY COMPLETED" message shall appear after step [8].

2.2.1.4 Cleanup

Remove current Product target by executing *Product.ins uninstall* from the directory with just tested Product files.

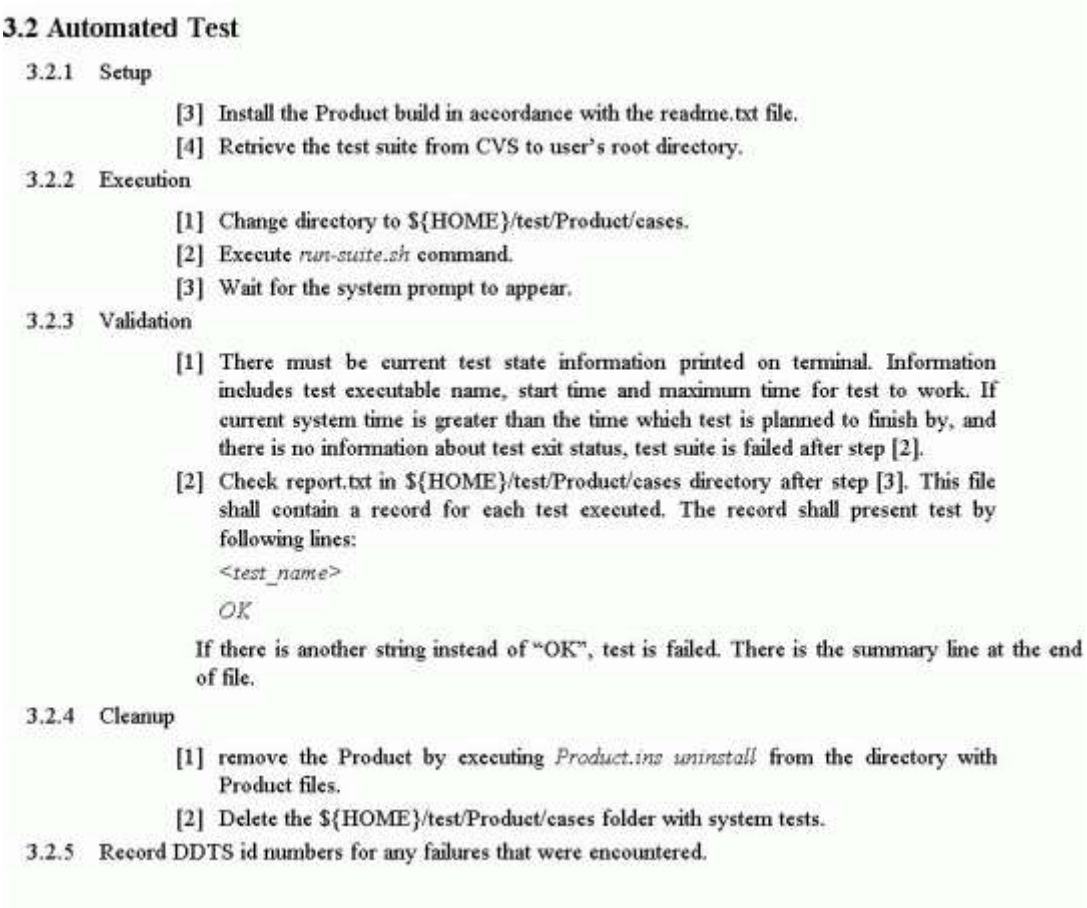
2.2.1.5 Log

Record DDTS id numbers for any failures that were encountered.

Рис. 10.1. Приклад фрагмента тестової процедури для ручного тестування

Процедури повинні бути складені таким чином, щоб будь-який інженер, не пов'язаний з даним проектом, був здатний адекватно провести цикл тестування, володіючи тільки самими базовими знаннями про застосовувані інструментарії. Приклад фрагмента*тестової процедури* для *ручного тестування* наведено на Рис. 10.1

У разі опису автоматизованих тестів *тестові процедури* повинні містити достатню інформацію для запуску тестів і аналізу результатів.Приклад фрагмента такої процедури приведений на Рис. 10.2



3.2 Automated Test

3.2.1 Setup

[3] Install the Product build in accordance with the readme.txt file.

[4] Retrieve the test suite from CVS to user's root directory.

3.2.2 Execution

[1] Change directory to ${HOME}/test/Product/cases.

[2] Execute *run-suite.sh* command.

[3] Wait for the system prompt to appear.

3.2.3 Validation

[1] There must be current test state information printed on terminal. Information includes test executable name, start time and maximum time for test to work. If current system time is greater than the time which test is planned to finish by, and there is no information about test exit status, test suite is failed after step [2].

[2] Check report.txt in ${HOME}/test/Product/cases directory after step [3]. This file shall contain a record for each test executed. The record shall present test by following lines:

<test_name>

OK

If there is another string instead of "OK", test is failed. There is the summary line at the end of file.

3.2.4 Cleanup

[1] remove the Product by executing *Product.ins uninstall* from the directory with Product files.

[2] Delete the ${HOME}/test/Product/cases folder with system tests.

3.2.5 Record DDTS id numbers for any failures that were encountered.

Рис. 10.2. Приклад фрагмента автоматизованої тестової процедури

Опис тестів

Опис тестів розробляється для полегшення аналізу та підтримки *тестового набору.* Опис може бути реалізовано в довільній формі, але при цьому повинні виконувати наступні завдання:

1. Аналізувати ступінь покриття продукту тестами на підставі опису *тестового набору.*
2. Для будь-якої функції тестованого продукту знайти тести, в яких функція використовується.

3. Для будь-якого тесту визначити всі функції і їх поєднання, які даний тест використовує (зачіпає).
4. Зрозуміти структуру і взаємозв'язки тестових файлів.
5. Зрозуміти принцип побудови системи *автоматизації тестування.*

Документування і життєвий цикл дефекту

Кожен дефект, виявлений в процесі тестування, повинен бути задокументований і відстежено. При виявленні нового дефекту його заносять до бази дефектів. Для цього краще всього використовувати спеціалізовані бази, що підтримують зберігання і відстеження дефектів - типу DDTS. При занесенні нового дефекту рекомендується вказувати, як мінімум, таку інформацію:

1. Найменування підсистеми, в якій виявлений дефект.
2. Версія продукту (номер build), на якому дефект був знайдений.
3. Опис дефекту.
4. Опис процедури (кроків, необхідних для відтворення дефекту).
5. Номер тесту, на якому дефект був виявлений.
6. Рівень дефекту, тобто ступінь його серйозності з точки зору критеріїв якості продукту або замовника.

Занесений до бази дефектів новий дефект знаходиться в стані **"New".** Після того, як команда розробників проаналізує дефект, він переводиться в стан **"Open"** із зазначенням конкретного розробника, відповідального за виправлення дефекту. Після виправлення дефект перекладається розробником в стан **"Resolved".** При цьому розробник повинен вказати наступну інформацію:

1. Причину виникнення дефекту.
2. Місце виправлення, як мінімум, з точністю до виправленого файлу.
3. Короткий опис того, що було виправлено.

4. Час, витрачений на виправлення.

Після цього тестувальник перевіряє, чи дійсно дефект був виправлений і якщо це так, переводить його в стан **"Verified".** Якщо тестувальник не підтвердить факт виправлення дефекту, то стан дефекту змінюється знову на **"Open".**

Якщо проектна команда приймає рішення про те, що деякий дефект виправлятися не буде, то такий дефект переводиться у стан**"Postponed"** із зазначенням осіб, відповідальних за це рішення, і причин його прийняття.

Тестовий звіт

Тестовий звіт оновлюється після кожного циклу тестування і повинен містити наступну інформацію для кожного циклу:

1. Перелік функціональності відповідно до пунктів вимог, запланований для тестування на даному циклі, і реальні дані по ньому.
2. Кількість виконаних тестів - заплановане і реально виконане.
3. Час, витрачений на тестування кожної функції, і загальний час тестування.
4. Кількість знайдених дефектів.
5. Кількість повторно відкритих дефектів.
6. Відхилення від запланованої послідовності дій, якщо такі мали місце.
7. Висновки про необхідні коригуваннях в системі тестів, які повинні бути зроблені до наступного *тестового циклу.*

Приклад фрагмента з *тестового звіту* представлений на Рис. 10.3 . Наведений фрагмент звіту містить приблизні дані для чотирьох циклів тестування та ілюструє структуру звіту. Такий вид звіт має після тестування, перед початком циклу тестування поля не заповнено, заповнення здійснюється після закінчення відповідного циклу.

Defect Activity

Number of Defects	Enhance ments	Severity 1	Severity 2	Severity 3	Total Enhan.&1	Total 2&3	Total
Newly Opened	2	5	4	0	7	4	11
Reopened	0	1	2	0	1	2	3
Newly Closed	1	3	8	2	4	10	14
Total Postponed	3	0	1	0	3	1	4

Test Coverage

Type	Planned Coverage	Current Version 3	Current Version 2	Current Version 1	Current version	Measurement
Functional	100%	60%	70%	90%	100%	Traceability Matrix with FS
Statement	75%	-	70%	-	76%	Log-file

Test Cases Summary

Test Category	Number of Requirements	Estimated Number of Tests	Number of Written Tests
Functional	68	74	103
Installation	12	20	22
Boundary	5	11	11
Stress	2	15	14
Performance	1	2	3
Total	88	122	153

Summary of Activities
Functional tests

Function	Number of tests executed	Number of tests passed	Tester	Execution & Analysis Time (hours)	Number of Severity 2 & 3 Defects
API	25	21	Alexandrov	2,5	2
Total	25	21		2,5	2

Рис. 10.3. Фрагмент тестового звіту

Оцінка якості тестів

Тести потребують контролю якості так само, як і тестований продукт. Оскільки тести для продукту є свого роду еталоном його структурних і поведінкових характеристик, закономірне питання про те, наскільки адекватний еталон. Для оцінки якості тестів використовуються різні методи, найбільш популярні з яких коротко розглянуті нижче.

Тестові метрики

Існує усталений набір тестових метрик, який допомагає визначити ефективність тестування і поточний стан продукту. До таких метрикам належать такі:

1. Покриття функціональних вимог.
2. Покриття коду продукту. Найбільш застосовно для модульного рівня тестування.
3. Покриття безлічі сценаріїв.
4. Кількість чи щільність знайдених дефектів. Поточне кількість дефектів порівнюється із середнім для даного типу продуктів з метою встановити, чи знаходиться воно в межах допустимого статистичного відхилення. При цьому виявлені відхилення як у більшу, так і в меншу сторону призводять до аналізу причин їх появи і, якщо необхідно, до вироблення коригувальних дій.
5. Співвідношення кількості знайдених дефектів з кількістю тестів на дану функцію продукту. Сильна розбіжність цих двох величин говорить або про неефективність тестів (коли велика кількість тестів знаходить мало дефектів) або про погане *якості даної*ділянки коду (коли знайдено велику кількість дефектів на не дуже великій кількості тестів).
6. Кількість знайдених дефектів, співвіднесені за часом, або швидкість пошуку дефектів. Якщо похідна такої функції близька до нуля, то продукт володіє якістю, достатньою для закінчення тестування і постачання замовнику.

Огляди тестів і стратегії

Тестовий код і стратегія тестування, зафіксовані у вигляді документів, помітно поліпшуються, якщо піддаються колективного обговорення. Такі обговорення називаються оглядами **(review).** Існує прийнята в організації процедура проведення та оцінки результатів огляду. Огляди поряд з тестуванням утворюють потужний набір методів боротьби з помилками з метою підвищення якості продукту. Цілі *оглядів тестової стратегії* і тестового коду різні.

Цілі *огляду тестової стратегії:*

1. Встановити достатність перевірок, забезпечуваних тестуванням.
2. Проаналізувати оптимальність покриття або адекватність розподілу кількості планованих тестів по функціональності продукту.
3. Проаналізувати оптимальність підходу до розробки коду, генерації коду, *автоматизації тестування.*

Цілі огляду тестового коду:

1. Встановити відповідність *тестового набору* тестової стратегії.
2. Перевірити вірність кодування тестів.
3. Оцінити досягнуту ступінь якості коду, виходячи з вимог за стандартами, простоті підтримки, наявності коментарів і т.п.
4. Якщо необхідно, проаналізувати оптимальність тестового коду з метою задоволення вимог до швидкодії і обсягом.

Лекція 9: Регресійне тестування: цілі та завдання, умови застосування, класифікація тестів і методів відбору

Анотація: Розглядаються цілі, завдання та види регресійного тестування. Перераховуються необхідні і достатні умови застосування методів вибіркового регресійного тестування. Дається класифікація методів вибіркового регресійного тестування і самих тестів при відборі. Розглядаються можливості повторного використання тестів.

Ключові слова: регресійне тестування , діяльність , модифікований код , регресійний дефект , мета регресійного тестирования ,подмножество , информация , запуск , множества , новый тест , програма , вихідні дані , вибіркове регресійне тестування , відбір тестів , ставлення , завдання регресійного тестування , циклу , вид регресійного тестування , чергу , коригуючий супровід ,адаптивне супровід , специфікація програми , час тестування , активний , висновок , кероване регресійне тестування , безпечний метод , граф викликів , прогін тестів , керуючий граф , оператори , Java , розподіл пам'яті , операції , ресурс , вхідні дані ,застарілий тест , уявлення , бази даних , повторне використання тестів , клас , траєкторія , компонент , формат виведення ,очікуване значение , полнота , точность , универсальность , мера , пр оцент

Цілі і завдання регресійного тестування

При коригуваннях програми необхідно гарантувати збереження якості. Для цього використовується *регресійне тестування* - дорога, але необхідна *діяльність* в рамках етапу супроводу, спрямована на повторну перевірку коректності зміненої програми. У відповідності зі стандартним визначенням, ***регресійне тестування*** - це вибіркове тестування, що дозволяє переконатися, що зміни не викликали небажаних побічних ефектів, або що змінена система як і раніше відповідає вимогам.

Головним завданням етапу супроводу є реалізація систематичного процесу обробки змін в коді. Після кожної модифікації програми необхідно упевнитися, що на функціональність програми не зробив впливу *модифікований код.* Якщо такий вплив виявлено, говорять про *регрессионном дефекті.* Для *регресійного тестування* функціональних можливостей, зміна яких не планувалося, використовуються раніше розроблені тести. Одна з *цілей регресійного тестування* полягає в тому, щоб, відповідно до використовуваним критерієм покриття коду (наприклад, критерієм покриття потоку операторів або потоку даних), гарантувати той же рівень покриття, що і при повному повторному тестуванні програми. Для цього необхідно запускати тести, що відносяться до змінених областям коду або функціональних можливостей.

Нехай T = {t1, t2, ..., tN} - безліч з N тестів, що використовується при первинній розробці програми P, а $T' \subseteq T$ -*підмножина* регресійних тестів для тестування нової версії програми P '. *Інформація* про покриття коду, обеспечиваемом T ',дозволяє вказати блоки P', що вимагають додаткового тестування, для чого може знадобитися повторний *запуск* деяких тестів з*безлічі* $T \supseteq T'$, Або навіть створення T" - набору *нових тестів* для P '- і оновлення T.

Інша *мета регресійного тестування* полягає в тому, щоб упевнитися, що *програма* функціонує у відповідності зі своєю специфікацією, і що зміни не призвели до внесення нових помилок у раніше протестований код. Ця мета завжди може бути досягнута повторним виконанням всіх тестів регресійного набору, але більш перспективно відсівати тести, на яких *вихідні дані* модифікованої і старої програми не можуть різнитися. Результати порівняння вибіркових методів і методу повторного прогону всіх тестів наведені втаблиця 11.1 .

Таблиця 11.1. Вибіркове регресійне тестування і повторний прогін всіх тестів.

Повторний прогін всіх тестів	***Вибіркове регресійне тестування***

Простий в реалізації	Вимагає додаткових витрат при впровадженні
Дорогий і неефективний	Здатне зменшувати витрати за рахунок виключення зайвих тестів
Виявляє всі помилки, які були б знайдені при вихідному тестуванні	Може призводити до пропуску помилок

Завдання *відбору тестів* з набору T для заданої програми P і зміненої версії цієї програми P 'полягає у виборі підмножини $T'_{\text{идеальное}} \subseteq T$ для повторного запуску на зміненій програмі P ', де $T'_{\text{идеальное}} = \{t \in T | P'(t) \neq P(t)\}$. Так як *вихідні дані* P і P 'для тестів з *безлічі* $T \supseteq T'_{\text{идеальное}}$ завідомо однакові, немає необхідності виконувати жоден з цих тестів на P '. У загальному випадку, за відсутності динамічної інформації про виконання P і P 'не існує методики обчислення *безлічі* T 'ідеальне для довільних множин P, P' і T. Це випливає з відсутності спільного рішення проблеми зупину, що складається в неможливості створення в загальному випадку алгоритму, що дає відповідь на питання, завершується чи коли-небудь довільна *програма* P для заданих значень вхідних даних. На практиці створення T 'ідеальне можливо тільки шляхом виконання на інструментованого версії P' кожного регресійного тесту, чого і хочеться уникнути.

Реалістичний варіант розв'язання задачі *вибіркового регресійного тестування* полягає в отриманні корисної інформації за результатами виконання P та об'єднання цієї інформації з даними статичного аналізу для отримання *безлічі* T 'реальне у вигляді апроксимації T' ідеальне. Цей підхід застосовується у всіх відомих вибіркових методах *регресійного тестування,* заснованих на аналізі коду. Безліч T 'реальне повинно включати всі тести з T, що активують змінений код, і не включати ніяких інших тестів, тобто тест $t \in T$ входить в T 'реальне тоді і тільки тоді, коли t задіє код P в точці, де в P' код був видалений або змінений, або де був доданий новий код.

Якщо деякий тест t задіє в P той же код, що і в P ', *вихідні дані* P і P 'для t відрізнятися не будуть. З цього випливає, що якщо $P(t) \neq P'(t)$, T повинен задіяти деякий код, змінений в P 'по відношенню до P, тобто повинно виконуватися*ставлення* $t \in T'_{\text{реальное}}$. З іншого боку, оскільки не кожне виконання зміненого коду відбивається на вихідних значеннях тесту, можуть існувати деякі такі $t \in T'_{\text{реальное}}$, Що P (t) = P '(t). Таким чином, T $_{\text{'реальне}}$ містить T' $_{\text{ідеальне}}$цілком і може використовуватися в якості його альтернативи без шкоди для якості тестованого програмного продукту.

Важливою *завданням регресійного тестування* є також зменшення вартості і скорочення часу виконання тестів.

Розглянемо *відбір тестів* на прикладі рис. 11.1 . Код, що покриваються тестами, виділений кольором і штрихуванням. Легко помітити, що код, що покриваються тестом 1, не змінився з попередньою версією, отже, повторне виконання тесту 1 не потрібно.Навпаки, код, що покриваються тестами 2, 3 і 4, змінився; отже, потрібно їх повторний *запуск.*

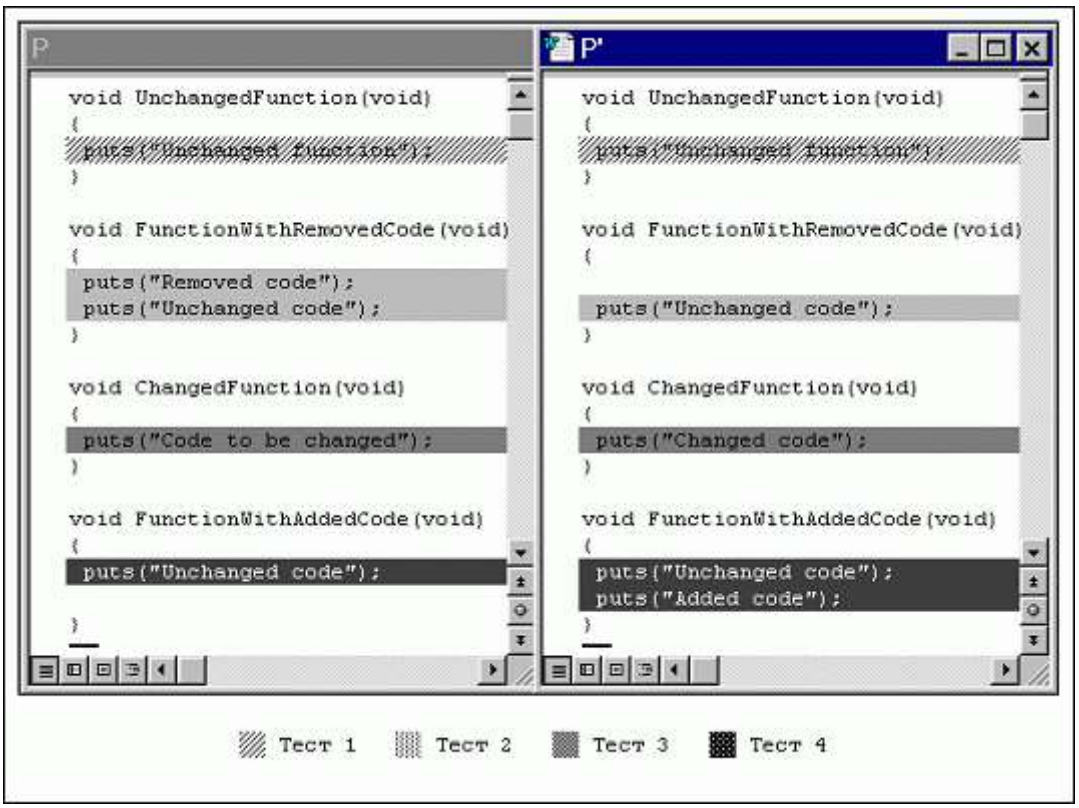


Рис. 11.1. Відбір тестів для безлічі T '.

Види регресійного тестування

Оскільки *регресійне тестування* являє собою повторне проведення *циклу* звичайного тестування, *види регресійного тестування*збігаються з видами звичайного тестування. Можна говорити, наприклад, про модульному *регрессионном тестуванні* або про функціональне *регрессионном тестуванні.*

Інший спосіб класифікації *видів регресійного тестування* пов'язує їх з типами супроводу, які, в свою *чергу,* визначаються типами модифікацій. Виділяють три типи супроводу:

- ***Коригуючий супровід,*** зване зазвичай виправленням помилок, виконується у відповідь на виявлення помилки, що не вимагає зміни специфікації вимог. При *коригуючому супроводі* проводиться діагностика і коректування дефектів у програмному забезпеченні з метою підтримки системи в працездатному стані.
- ***Адаптивне супровід*** здійснюється у відповідь на вимоги зміни даних або середовища виконання. Воно застосовується, коли існуюча система поліпшується або розширюється, а специфікація вимог змінюється з метою реалізації нових функцій.
- **Вдосконалять (прогресивне) супровід** включає будь-яку обробку з метою підвищення ефективності роботи системи або ефективності її супроводу.

У процесі *адаптивного* або вдосконалять супроводу звичайно вводяться нові модулі. Щоб відобразити те чи інше удосконалення або адаптацію, змінюється специфікація системи. При *коригуючому супроводі,* як правило, специфікація не змінюється, і нові модулі не вводяться. Модифікація програми на фазі розробки подібна модифікації при *коригуючому супроводі,* так як через виявлення помилки

навряд чи потрібно міняти *специфікацію програми.* За винятком рідкісних моментів великих змін, на фазі супроводу зміни системи зазвичай невеликі і виробляються з метою усунення проблем або поступового розширення функціональних можливостей.

Відповідно, визначають два типи *регресійного тестування:* прогресивне і коригуючий.

- Прогресивне *регресійне тестування* передбачає модифікацію технічного завдання. У більшості випадків при цьому до системи програмного забезпечення додаються нові модулі.
- При коригуючому *регрессионном тестуванні* технічне завдання не змінюється. Модифікуються тільки деякі оператори програми і, можливо, конструкторські рішення.

Прогресивне *регресійне тестування* зазвичай виконується після *адаптивного* або вдосконалять супроводу, тоді як коригуючий*регресійне тестування* виконується під *час тестування* в циклі розробки і після *коригуючого супроводу,* тобто після того, як над програмним забезпеченням були виконані деякі коригувальні дії. Взагалі кажучи, коригуючий *регресійне тестування* має бути більш простим, ніж прогресивне *регресійне тестування,* оскільки допускає повторне використання більшої кількості тестів.

Підхід до відбору регресійних тестів може бути активним або консервативним. *Активний* підхід у центр ставить зменшення обсягу*регресійного тестування* і нехтує ризиком пропустити дефекти. *Активний* підхід застосовується для тестування систем з високою вихідною надійністю, а також у випадках, коли ефект змін невеликий. Консервативний підхід вимагає відбору всіх тестів, які з ненульовою ймовірністю можуть виявляти дефекти. Цей підхід дозволяє виявляти більшу кількість помилок, але призводить до створення більш обширних наборів регресійних тестів.

Кероване регресійне тестування

Протягом життєвого *циклу* програми період супроводу триває довго. Коли змінена *програма* тестується набором тестів T, ми зберігаємо без змін по відношенню до тестування вихідної програми P всі фактори, які могли б впливати на *висновок* програми.Тому атрибути конфігурації, в якій *програматестувалася* останній раз (наприклад, план тестування, тести t_j і покриваються елементи MT (P, C, $t_{j)}$, підлягають управлінню конфігурацією. Практика тестування зміненої версії програми P 'в тих же умовах, в яких тестувалася вихідна *програма* P, називається *керованим регресійним тестуванням.* При некерованому*регрессионном тестуванні* деякі властивості методів *регресійного тестування* можуть змінюватися, наприклад, *безпечний методвідбору тестів* може перестати бути безпечним. У свою *чергу,* для забезпечення *керованості регресійного тестування* необхідно виконання ряду умов:

- Як при модульному, так і при інтеграційному *регрессионном тестуванні* в якості модулів, що викликаються тестируемим модулем безпосередньо або побічно, повинні використовуватися реальні модулі системи. Це легко здійснити, оскільки на етапі*регресійного тестування* всі модулі доступні в завершеному вигляді.
- Інформація про зміни коректна. Інформація про зміни вказує на змінені модулі та розділи специфікації вимог, не маючи на увазі при цьому коректність самих змін. Крім того, при зміні специфікації вимог необхідно посилене *регресійне тестування* змінених функцій цієї специфікації, а також всіх функцій, які могли бути порушені з необережності. Єдиним випадком коли ми змушені покластися на правильність зміненого технічного завдання, є зміна технічного завдання для всієї системи або для модуля верхнього (в *графі викликів)* рівня, за умови, що окрім технічного завдання, не існує ніякої додаткової документації та / або якої іншої

інформації, за якою можна було б судити про помилку в технічному завданні.

- У програмі немає помилок, крім тих, які могли виникнути через її зміни.
- Тести, що застосовувалися для тестування попередніх версій програмного продукту, доступні, при цьому протокол *прогону тестів* складається з вхідних даних, вихідних даних і траєкторії. Траєкторія являє собою шлях в *керуючому графі* програми, проходження якого викликається використанням деякого набору вхідних даних. Її можна застосовувати для оцінки структурного покриття, забезпечуваного набором тестів.
- Для проведення *регресійного тестування* з використанням існуючого набору тестів необхідно зберігати інформацію про результати виконання тестів на попередніх етапах тестування.

Припустимо, що ніякі *оператори* програми, крім тих, чия поведінка залежить від змін, не можуть несприятливо впливати на програму. Навіть за такої умови існують деякі ситуації, що вимагають особливої уваги, наприклад проблема витоку пам'яті і їй подібні.Ситуації такого роду в різних системах програмування обробляються по-різному. Наприклад, мова *Java* сам по собі включає систему управління пам'яттю. Якщо ж система не контролює *розподіл пам'яті* автоматично, ми повинні вважати, що всі *оператори* роботи з пам'яттю також володіють поведінкою, залежних від змін.

Проблема мов типу C і C + +, які допускають довільні арифметичні *операції* над покажчиками, полягає в тому, що покажчики можуть порушувати межі областей пам'яті, на які вони вказують. Це означає, що змінні можуть оброблятися способами, які не піддаються аналізу на рівні вихідного коду. Щоб врахувати такі порушення кордонів пам'яті, висуваються наступні гіпотези:

- Гіпотеза 1 (чітко визначена пам'ять). Кожен сегмент пам'яті, до якого звертається система програмного забезпечення, відповідає деякій символічно певної змінної.

- Гіпотеза 2 (строго обмежений покажчик). Кожна змінна або вираз, що використовується як покажчик, повинно посилатися на деяку базову змінну і обмежуватися використанням сегмента пам'яті, що визначається цієї змінної.

Щоб гарантувати покриття всіх залежних від змін компонентів, для яких можна показати, що вони зачіпаються існуючими тестами, достатньо одного тесту для кожного з таких компонентів. Безліч тестів досить великого розміру (як правило сценарних), може сприяти виявленню помилок, викликаних порушеннями умов *керованого регресійного тестування.*

Існують і організаційні умови проведення *регресійного тестування.* Це *ресурс* (час), необхідний тестового аналітику для ознайомлення зі специфікацією вимог системи, її архітектурою і, можливо, самим кодом.

Обгрунтування коректності методу відбору тестів

Перерахуємо деякі особливості реалізації *регресійного тестування.*

- Деякі ділянки коду програми не отримують управління при виконанні деяких тестів.
- Якщо ділянка коду реалізує вимогу, але змінений фрагмент коду не отримує управління при виконанні тесту, то він і не може впливати на значення вихідних даних програми при виконанні даного тесту.
- Навіть якщо ділянка коду, який реалізує вимога, отримує управління при виконанні тесту, це далеко не завжди відбивається на вихідних даних програми при виконанні даного тесту. Дійсно, якщо змінюється перший блок програми, наприклад, шляхом додавання ініціалізації змінної, всі шляхи в програмі також змінюються, і, як наслідок, вимагають повторного тестування. Однак може так статися, що тільки на невеликому підмножині шляхів дійсно використовується ця ініціалізованих змінна.
- Не кожен тест $t_k \in T$, Перевіряючий код, що знаходиться на одній колії зі зміненим кодом, обов'язково покриває цей змінений код.

- Код, що знаходиться на одній колії зі зміненим кодом, може не впливати на значення вихідних даних змінених модулів програми.
- Не завжди кожен оператор програми впливає на кожен елемент її вихідних даних.

Припустимо, що зміни в програмі обмежуються одним оператором. Якщо при виконанні якого-небудь тесту на вихідній програмі цей оператор ніколи не отримує управління, можна з упевненістю сказати, що він не отримає управління і в ході виконання тесту на новій програмі, а результати тестування нової та старої програм будуть збігатися. Отже, немає необхідності виконувати цей тест на новій програмі. Зазначений метод легко можна узагальнити для випадку декількох змін: якщо тест не задіює жодного зміненого оператора, і його *вхідні дані* не змінилися, код, що виконується їм у зміненій програмі, буде в точності таким же, як в первісній версії. Такий тест не виявляє відмінностей між двома версіями системи; отже, немає необхідності проганяти його повторно. Якщо тест не зачіпає жодного оператора виведення, поведінка якого залежить від змінених операторів, це означає, що, незважаючи на зміни в програмі, все *оператори,* які отримують управління при виконанні цього тесту, не змінять *висновок* системи по відношенню до попередньої версії. Таким чином, немає необхідності повторно проганяти і тести такого роду.

Отже, необхідно орієнтуватися на вибір тільки тих тестів, які покривають змінений код, що впливає, у свою *чергу,* на *висновок*програми. Такий підхід гарантує, що будуть обрані тільки тести, які виявляють зміни, і метод буде, як кажуть, точним.

Класифікація тестів при відборі

Створення наборів регресійних тестів рекомендується починати з *безлічі* вихідних тестів. При заданому критерії *регресійного тестування* всі вихідні тести t ($t \in T$) Поділяються на три підмножини:

1. Безліч тестів, придатних для повторного використання. Це тести, які вже запускалися і придатні до використання, але зачіпають тільки покриваються елементи програми, що не зазнали змін. При повторному виконанні вихідні дані таких тестів збіжаться з вихідними даними, отриманими на вихідній програмі. Отже, такі тести не вимагають перезапуску.

2. Безліч тестів, що вимагають повторного запуску. До них відносяться тести, які вже запускалися, але вимагають перезапуску, оскільки зачіпають, принаймні, один змінений покриваються елемент, який підлягає повторному тестуванню. При повторному виконанні такі тести можуть давати результат, відмінний від результату, показаного на вихідній програмі. Безліч тестів, що вимагають повторного запуску, забезпечує хороше покриття структурних елементів навіть за наявності нових функціональних можливостей.

3. Безліч *застарілих тестів.* Це тести, паче не застосовні до зміненої програмі та непридатні для подальшого тестування, оскільки вони зачіпають тільки покриваються елементи, які були видалені при зміні програми. Їх можна видалити з набору регресійних тестів.

4. *Нові тести,* які ще не запускалися і можуть бути використані для тестування.

Рис. 11.2 дає *уявлення* про життєвий цикл тесту. Відразу після створення тест вводиться в структуру *бази даних* як новий. Після виконання *новий тест* переходить у категорію тестів, придатних для *повторного використання* або *застарілих.* Якщо виконання тесту сприяло збільшенню поточної ступеня покриття коду, тест позначається як придатний для *повторного використання.* В іншому випадку він позначається як застарілий і відкидається. Існуючі тести, повторно запущені після внесення зміни в код, також класифікуються заново як придатні для *повторного використання* або

застарілі залежно від тестових траєкторій і використовуваного критерію тестування.

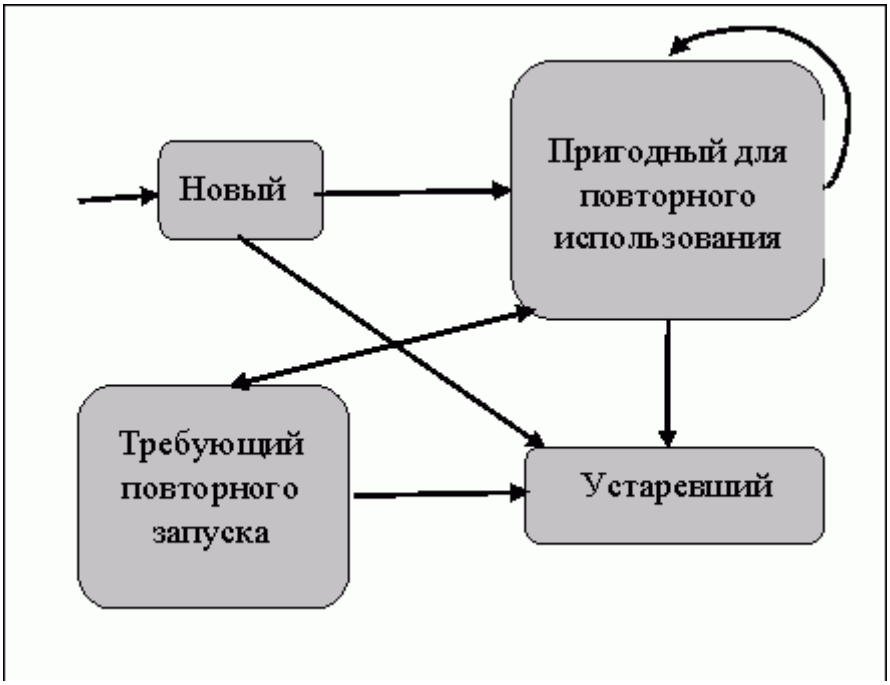


Рис. 11.2. Життєвий цикл тесту

Класифікація тестів по відношенню до змін в коді вимагає аналізу наслідків змін. Тести, що активують код, порушене змінами, можуть вимагати повторного запуску або виявитися застарілими. Щоб тест був включений в *клас* тестів, що вимагають повторного запуску, він має бути порушений змінами в коді, а також має сприяти збільшенню ступеня покриття зміненого коду по використовуваному критерію. Порушених елементом тесту може бути *траєкторія,* вихідні значення, або і те, і інше. Щоб тест був включений в *клас*тестів, придатних для *повторного використання,* він повинен вносити вклад у збільшення ступеня покриття коду і не вимагати повторного запуску.

Ступінь покриття коду визначається для тестів, придатних для *повторного використання,* оскільки до цього класу відносяться тести, які не потребують повторного запуску і які б збільшення ступеня покриття до бажаної величини. Якщо є *компонент*програми, що не задіяний придатними для *повторного використання тестами,* то замість них вибираються і

виконуються з метою збільшення ступеня покриття тести, що вимагають повторного запуску. Після запуску такий тест стає придатним для *повторного використання* або *застарілим.* Якщо тестів, що вимагають повторного запуску, більше не залишилося, а необхідна ступінь покриття коду ще досягнуто, породжуються додаткові тести і тестування повторюється.

Остаточний набір тестів збирається з тестів, придатних для *повторного використання,* тестів, що вимагають повторного запуску, і*нових тестів.* Нарешті, *застарілі* і надлишкові тести видаляються з набору тестів, оскільки надлишкові тести не перевіряють нові функціональні можливості і не збільшують покриття.

Можливості повторного використання тестів

До зміни існуючих тестів можуть привести три наступних виду діяльності програмістів:

- Створення *нових тестів.*
- Виконання тестів.
- Зміна коду.

Оскільки кожен тест містить *вхідні дані, вихідні дані* і траєкторію, ці компоненти можуть піддатися зміні в будь-якій комбінації. При зміні вхідних даних існуючого тесту будемо вважати, що старий тест припиняє існування, і створюється *новий тест.* Таким чином, до числа дозволених змін тесту відносяться всілякі пертурбації вихідних даних або траєкторій. Зміна вихідних даних без зміни траєкторії і / або вхідних даних неможливо. Отже, існує тільки два можливих варіанти зміни тесту: зміна траєкторії або зміна траєкторії і вихідних даних.

Згідно з наведеними вище міркуваннями можна виділити чотири рівня *повторного використання тесту:*

- Рівень 1. Тест не допускає *повторного використання.* Потрібно створення нового набору тестів (наприклад, шляхом видалення або зміни цього тесту).
- Рівень 2. *Повторне використання* можливе тільки вхідних даних тесту. У багатьох випадках мета тестування полягає в активізації деяких покриваються елементів програми. Якщо з траєкторії існуючого тесту видно, що елементи програми, підлягають покриттю, задіюються до змінених команд, вхідні дані тесту можуть бути використані повторно для покриття цих елементів. В результаті змін у програмі та / або технічному завданні нова траєкторія і вихідні дані тесту можуть відрізнятися від результатів попереднього виконання. Таким чином, тести першого рівня повинні бути запущені повторно для отримання нових вихідних даних і траєкторій.
- Рівень 3. Можливо *повторне використання* як вхідних, так і вихідних даних тесту. Очевидно, що на цьому рівні зазвичай розташовуються функціональні тести. Якщо модуль піддався тільки зміні коду із збереженням функціональності, можливо*повторне використання* існуючих функціональних тестів для перевірки правильності реалізації. Оскільки траєкторія може змінитися, а вихідні дані - зазнати впливу з боку змін коду, такі тести повинні бути запущені повторно, але очікується отримання ідентичних результатів.
- Рівень 4. Найвищий рівень *повторного використання тесту,* що передбачає повторне використання вхідних даних, вихідних даних і траєкторії тесту. У цьому випадку на траєкторії тесту не змінюється ні один оператор. Отже, в повторному запуску цих тестів необхідності немає, так як вихідні дані і траєкторія залишаться незмінними.

Приклад регресійного тестування функції вирішення квадратного рівняння.

Код цієї функції наведено на приклад 11.1 . Вхідними параметрами є коефіцієнти квадратного рівняння A, B і C, а також прапорPrint, ненульове значення якого вказує, що отримане рішення необхідно вивести на екран. До вихідних параметрів належатьX1 і X2, призначені для зберігання коренів рівняння, і повертається значення функції - дискримінант рівняння. У початковому вигляді функція містить дефект, в результаті чого рівняння з негативним дискриминантом породжують помилку часу виконання. У новій версії функції дефект повинен бути виправлений; крім того, необхідно реалізувати запит користувача на зміну *формату виводу*рішення. Код нової версії функції Equation наводиться на приклад 11.2 .

Існуючі тести для функції Equation наведені в таблиця 11.2 . Вхідні дані тестів являють собою сукупність значень Print, A, B iC, що подаються на вхід функції. Вихідними даними для тесту є значення X1 і X2, повертається значення функції, а також рядок, що виводиться на екран; в таблиця 11.2 наведені *очікувані значення* вихідних даних. Крім того, для кожного тесту обчислюється траєкторія його проходження по коду.

Таблиця 11.2. Вхідні і вихідні дані тестів

Тест	**Вхідні дані**				**Очікувані вихідні дані**			
	A	**B**	**C**	**Print**	**X1**	**X2**	**Значення, що повертається**	**Що виводиться рядок**
1	1	1	-16	1	2	-3	25	Solution: X1 = 2, X2 = -3
2	2	-5	13	1	0.75	5.567764	-31	Solution: X1 = 0.75 +5.567764 i, X2 = 0.75-5.567764I
3	1	2	0	0	0	-2	4	
4	1	2	1	0	-1	-1	0	

5 1 2 2 0 -1 2 -4

```
 double Equation (int Print, float A, float B, float C,
                  float & X1, float & X2)
{
 float D = B * B - 4.0 * A * C;
 if (D> = 0)
 {
  X1 = (-B + sqrt (D)) / 2.0 / A;
  X2 = (-B - sqrt (D)) / 2.0 / A;
 }
 else
 {
  X1 =-B / 2.0 / A;
  X2 = sqrt (D) / 2.0 / A;
 }
 if (Print)
  printf ("Solution:% f,% f \ n", X1, X2);
 return D;
}
```

11.1. Функція Equation - початкова версія.

```
 double Equation (int Print, float A, float B, float C,
                  float & X1, float & X2)
{
```

```
 float D = B * B - 4.0 * A * C;
 if (D> = 0)
 {
  X1 = (-B + sqrt (D)) / 2.0 / A;
  X2 = (-B - sqrt (D)) / 2.0 / A;
 }
 else
 {
  X1 =-B / 2.0 / A;
  X2 = sqrt (D);
 }
 if (Print)
  printf ("Solution:% f,% f \ n", X1, X2);
 return D;
}
```

11.1.1. Функція Equation - початкова версія.

```
 double Equation (int Print, float A, float B,
       float C, float & X1, float & X2)
{
 float D = B * B - 4.0 * A * C;

 if (D> = 0)
 {
```

```
 X1 = (-B + sqrt (D)) / 2.0 / A;
 X2 = (-B - sqrt (D)) / 2.0 / A;
}
else
{
 X1 =-B / 2.0 / A;
 X2 = sqrt (-D);
}
if (Print)
{
 if (D> = 0)
  printf ("Solution: X1 =% f, X2 =% f \ n", X1, X2);
 else
  printf ("Solution: X1 =% f +% fi, X2 =% f-% fi \ n",
        X1, X2, X1, X2);
}

return D;
}
```

11.2. Функція Equation - змінена версія.

```
double Equation (int Print, float A, float B,
      float C, float & X1, float & X2)
{
```

```
float D = B * B - 4.0 * A * C;


if (D> = 0)
{
 X1 = (-B + sqrt (D)) / 2.0 / A;
 X2 = (-B - sqrt (D)) / 2.0 / A;
}
else
{
 X1 =-B / 2.0 / A;
 X2 = sqrt (-D);
}
if (Print)
{
 if (D> = 0)
  printf ("Solution: X1 =% f, X2 =% f \ n", X1, X2);
 else
  printf ("Solution: X1 =% f +% fi, X2 =% f-% fi \ n",
         X1, X2, X1, X2);
}


return D;
}
```

11.2.1. Функція Equation - змінена версія.

При зміні функції Equation від приклад 11.1 до приклад 11.2 змінюється формат виведених на екран даних, так що тести 1 і 2, перевіряючі висновок на екран, можуть бути повторно використані тільки на рівні 2. Тести 3, 4 і 5 можуть бути використані на рівні 3 або 4 залежно від результатів аналізу їх траєкторії.

Класифікація вибіркових методів

Для перевірки коректності різних підходів до *регрессионному тестуванню* використовується модель оцінки методів *регресійного тестування.* Основними об'єктами розгляду стали *повнота, точність,* ефективність і *універсальність.*

Повнота відображає міру *відбору тестів* з T, на яких результат виконання зміненої програми різниться від результату виконання вихідної програми, внаслідок чого можуть бути виявлені помилки в P '. Метод, повний на 100%, називається безпечним.

Точність - *міра* здатності методу уникати вибору тестів з T, на яких результат виконання зміненої програми не буде відрізнятися від результату її первісній версії, тобто тестів, нездатних виявляти помилки в P '. Припустимо, що набір T містить rрегресійних тестів. З них для n тестів (n <= r) поведінку і результати виконання старої програми P відрізняються від поведінки і результатів виконання нової програми P '. Набір тестів $T' \subseteq T$ містить m ($m \neq 0$) Тестів, отриманих з використанням методу відбору регресійних тестів M. З цих m тестів для l тестів поведінку P 'і P розрізняється. *Точність* T 'відносно P, P', T і M, виражена у відсотках, визначається виразом 100 * (l / m), тоді як відповідний *відсоток* обраних тестів визначається виразом 100 * (l / n), якщо $n \neq 0$ або дорівнює 100%, якщо n = 0.

Виходячи з наведеного визначення, *точність безлічі* тестів - це *відношення* числа тестів даного *множини,* на яких результати виконання нової та старої програм розрізняються, до загального числа тестів *множини. Точність* є важливим

атрибутом методу*регресійного тестування.* Неточний метод має тенденцію відбирати тести, які не повинні були бути обрані. Чим менш точний метод, тим ближче обсяг обраного набору тестів до обсягу вихідного набору тестів.

Ефективність - оцінка обчислювальної вартості стратегії *вибіркового регресійного тестування,* тобто вартості реалізації її вимог за часом і пам'яті, а також можливості автоматизації. Відносною ефективністю називається ефективність методу тестування за умови наявності не більше однієї помилки в програмі, що тестується. Абсолютною ефективністю називається ефективність методу в реальних умовах, коли оцінка кількості помилок у програмі не обмежена.

Універсальність відображає міру здатності методу до застосування в досить широкому діапазоні ситуацій, що зустрічаються на практиці.

Для програми P, її зміненої версії P 'і набору тестів T для P потрібно, щоб методика вибіркового повторного тестування задовольняла таким критеріям оцінки:

- Критерій 1. **Безпека.** Методика вибіркового повторного тестування повинна бути безпечною, тобто повинна вибирати всі тести з T, які потенційно можуть виявляти помилки (всі тести, чия поведінка на P 'і P може бути різним). Безпечна методика повинна розглядати наслідки додавання, видалення і зміни коду. При додаванні нового коду в P в T можуть вже міститися тести, що покривають цей новий код. Такі тести необхідно виявляти і враховувати при відборі.
- Критерій 2. **Точність.** Стратегія повторного прогону всіх тестів є безпечною, але неточною. На додаток до вибору всіх тестів, потенційно здатних виявляти помилки, вона також вибирає тести, які в жодному разі не можуть демонструвати змінений поведінка. В ідеалі, методика вибіркового повторного тестування повинна бути точною, тобто повинна вибирати тільки тести з зміненим поведінкою. Однак для довільно взятого тесту, не запускаючи його, неможливо визначити, чи зміниться його

поведінку. Отже, в кращому випадку ми можемо розраховувати лише на деяке збільшення точності.

Всілякі існуючі вибіркові методи *регресійного тестування* розрізняються не в останню чергу вибором об'єкта або об'єктів, для яких виконується аналіз покриття та аналіз змін. Наприклад, при аналізі на рівні функції при зміні будь-якого оператора функції вся функція вважається зміненою; при аналізі на рівні окремих операторів ми можемо виключити частину тестів, що містять виклик функції, але не активують змінений оператор. Вибір об'єктів для аналізу покриття відбивається на рівні подробиці аналізу, а значить, і на його точності та ефективності. Абсолютні величини точності та кількості вибраних тестів для заданих набору тестів і безлічі змін повинні розглядатися тільки разом зі зменшенням розміру набору тестів. Невеликий відсоток обраних тестів може бути прийнятним, тільки якщо рівень точності залишається досить високим.

- Критерій 3. **Ефективність.** Методика вибіркового повторного тестування повинна бути ефективною, тобто повинна допускати автоматизацію і виконуватися досить швидко для практичного застосування в умовах обмеженого часу *регресійного тестування.*Методика повинна також передбачати зберігання інформації про хід виконання тестів у мінімально можливому обсязі.
- Критерій 4. **Універсальність.** Методика вибіркового повторного тестування повинна бути універсальною, тобто застосовної до всіх мов і мовних конструкціях, ефективної для реальних програм і здатною до обробки як завгодно складних змін коду.

У загальному випадку існує певний компроміс між безпекою, точністю і ефективністю. При *відборі тестів* аналіз необхідно провести за час, менший, ніж потрібно для виконання та перевірки результатів тестів з T, що не увійшли до T '. З урахуванням цього обмеження рішенням *завдання регресійного тестування* буде *безпечний метод* з хорошим балансом дешевизни і високої точності.

Регресійне тестування: різновиди методу відбору тестів

Анотація: Розглядаються випадкові методи, безпечні методи, методи мінімізації, методи, засновані на покритті коду. Також розглядається інтеграційне регресійне тестування і регресійне тестування об'єктно-орієнтованих програм.

Ключові слова: програмні засоби , відбір тестів , випадковий метод , відсоток , вхідні данные , пользователь , метрика , множества, подмножество , исключение , произвольное , стоимость , интервал , формат виводу , метод експертних оцінок , витрати ,вибіркове регресійне тестування , безпечний метод , вихідні данные , исполнение , программа , вероятность , матрица ,информация , ячейка , метод мінімізації , коректність програми , детермінований метод , аналіз , функціональні вимоги , модульне тестування , значення , метод покриття коду , атрибут , завдання регресійного тестування

Випадкові методи

Коли через обмеження за часом використання методу повторного прогону всіх тестів неможливо, а *програмні засоби відбору тестів*недоступні, інженери, відповідальні за тестування, можуть вибирати тести випадковим чином або на підставі "здогадок", тобто предположительного співвіднесення тестів з функціональними можливостями на підставі попередніх знань або досвіду. Наприклад, якщо відомо, що деякі тести задіють особливо важливі функціональні можливості або виявляли помилки раніше, їх було б непогано використовувати також і для тестування зміненої програми. Один простий метод такого роду передбачає випадковий *відбір*зумовленого відсотка тестів з Т. Подібні *випадкові методи* прийнято позначати random (x), де x - *відсоток* обираних тестів.

Випадкові методи виявляються напрочуд дешевими і ефективними. Випадково обрані *вхідні дані* можуть давати більший розкид по покриттю коду, ніж *вхідні дані,* які використовуються в наборах тестів, заснованих на покритті, в одних випадках дублюючи покриття, а в інших, не забезпечуючи його. При невеликих

інтервалах тестування їх ефективність може бути як дуже високою, так і дуже низькою. Це призводить і до більшого розкиду статистики *відбору тестів* для таких наборів. Однак при збільшенні інтервалу тестування цей розкид стає значно менше, і середня ефективність *випадкових методів* наближається до ефективності методу повторного прогону всіх тестів з невеликими відхиленнями для різних спроб. Таким чином, в останньому випадку *користувачвипадкових методів* може бути більш впевнений в їх ефективності. Взагалі, детерміновані методи ефективніше *випадкових методів,*але набагато дорожче, оскільки вибіркові стратегії вимагають великої кількості часу і ресурсів при *відборі тестів.*

Якщо зміни в новій версії зачіпають код, що виконується відносно часто, при випадкових вхідних даних змінений код може в середньому активуватися навіть частіше, ніж при виконанні тестів, заснованих на покритті коду. Це призведе до збільшення метрики кількості відібраних тестів для випадкових наборів. Навпаки, відносно рідко виконуваний змінений код активується випадковими тестами рідше, і відповідна *метрика* знижується. При зменшенні потужності *безлічі* відібраних тестів падає ефективність виявлення помилок.

Коли вбрання *підмножина,* хоча і досконале з погляду повноти і точності, все ще занадто дорого для регресійного тестування, особливо важлива гнучкість при *відборі тестів.* Які додаткові процедури можна застосувати для подальшого зменшення числа вибраних тестів? Одне з можливих рішень - випадкове *виняток* тестів. Однак, оскільки таке рішення допускає *довільне* видалення тестів, що активують зміни в коді, існує високий ризик виключення всіх тестів, що виявляють помилку в цьому коді. Проте, якщо*вартість* пропуску помилок незначна, а *інтервал* тестування великий, доцільним буде використання *випадкового методу* з невеликим відсотком обираних тестів **(25-30%),** наприклад, random (25).

Повернемося до прикладу регресійного тестування функції вирішення квадратного рівняння. *Випадковий метод,* такий, як random (40), може відібрати

для повторного виконання будь-які 2 тесту з 5. Наприклад, якщо будуть обрані тести 4 і 5, зміни *формату виводу* на екран не будуть протестовані зовсім, що навряд чи може влаштувати розробника.

При використанні іншого *випадкового методу* - *методу експертних оцінок* - в даному випадку найбільш вірогідний вибір всіх тестів, так як *витрати* на прогін невеликі. Однак при регресійному тестуванні великих програмних систем, коли повторний прогін всіх тестів неприйнятний, експерт змушений відсівати деякі тести, що також може призводити до того, що частина змін не буде протестована повністю.

Безпечні методи

Метод *вибіркового регресійного тестування* називається ***безпечним,*** якщо при деяких чітко визначених умов він не виключає тестів (з доступного набору тестів), які виявили б помилки у зміненій програмі, тобто забезпечує вибір всіх тестів, що виявляють зміни.Тест називається виявляє зміни, якщо його *вихідні дані* при прогоні на P 'відрізняються від вихідних даних при прогоні на P: $P(t) \neq P'(t)$. Тести, що активізують змінений код, називаються виконують зміна.

Вибір всіх виконують зміна тестів є *безпечним,* але при цьому відбираються деякі тести, які не виявляють змін. *Безпечний метод*може включати в T ' *підмножина* тестів, *вихідні дані* яких для P і P 'ні за яких умов не відрізняються. Оскільки не існує методики, окрім власне виконання тесту, що дозволяє для будь P 'визначити, чи будуть *вихідні дані* тесту відрізнятися для P іP ', жоден метод не може бути *безпечним* і абсолютно точним одночасно. T 'є безпечним підмножиною T тоді і тільки тоді, коли:

$$P(t) \neq P'(t) \Rightarrow t \in T'$$

Якщо P і P 'виконуються в ідентичних умовах і T' є безпечним підмножиною T, *виконання* T 'на P' завжди виявляє будь-які пов'язані з змінами помилки в P, які можуть бути знайдені шляхом виконання T. Якщо існує тест,

який виявляє помилку,*безпечний метод* завжди знаходить її. Таким чином, жоден *випадковий метод* не володіє такою ж ефективністю виявлення помилок, як *безпечний метод.*

За деяких умов *безпечні методи* в силу визначення "безпеки" гарантують, що всі "виявляються" помилки будуть знайдені. Тому відносна ефективність всіх *безпечних методів* дорівнює ефективності методу повторного прогону всіх тестів і становить 100%.Проте їх абсолютна ефективність падає із збільшенням інтервалу тестування. Відзначимо, що *безпечний метод* дійсно безпечний тільки в припущенні коректності вихідного *безлічі* тестів T, тобто коли при виконанні всіх $t \in T$ вихідна *програма* Рзавершилася з коректними значеннями вихідних даних, а всі застарілі тести були з T видалені.

Існують програми, змінені версії та набори тестів, для яких застосування безпечного *відбору* не дає великого виграшу в розмірі набору тестів. Характеристики вихідної програми, зміненої версії і набору тестів можуть спільно або незалежно впливати на результати *відбору тестів.* Наприклад, при ускладненні структури програми *ймовірність* активації довільним тестом довільної зміни в програмі зменшується. *Безпечний метод* переважніше виконання всіх тестів набору тоді і тільки тоді, коли *вартість* аналізу менше, ніж *вартість* виконання невибраних тестів. Для деяких систем, критичних з точки зору безпеки, *вартість* пропуску помилки може бути настільки висока, що небезпечні методи *вибіркового регресійного тестування* використовувати не можна.

Прикладом *безпечного методу* може служити метод, який вибирає з T кожен тест, що виконує, принаймні, один оператор, доданий або змінений в P 'або видалений з P. Застосування цього методу для регресійного тестування функції вирішення квадратного рівняння потребують побудови матриці покриття, приклад якої наведено в таблиці на Рис. 12.1 . Слід зазначити, що*матриця* покриття відповідає вихідної версії програми, оскільки аналогічна *інформація* для нової версії програми поки не зібрана.Зірочка в

комірці таблиці означає, що відповідний тест покриває певну рядок коду; якщо тест не покриває рядок коду, *осередок*залишена порожньою. Рядки, змінені по відношенню до вихідної версії, виділені кольором. Легко помітити, що відповідно до вимог запропонованого *безпечного методу* для повторного виконання повинні бути відібрані тести 1, 2 і 5.

Методи мінімізації

Процедура мінімізації набору тестів ставить метою *відбір* мінімального (у термінах кількості тестів) підмножини T, необхідного для покриття кожного елемента програми, залежного від змін. Для перевірки *коректності програми* використовуються тільки тести з мінімального підмножини.

№	Строка кода	Тест				
		1	2	3	4	5
1	`Double Equation(int Print, float A, float B, float C, float& X1, float& X2) {`	*	*	*	*	*
2	`float D = B * B - 4.0 * A * C;`	*	*	*	*	*
3	`if (D >= 0) {`	*	*	*	*	*
4	`X1 = (-B + sqrt(D)) / 2.0 / A;`	*		*	*	
5	`X2 = (-B - sqrt(D)) / 2.0 / A; } else {`	*		*	*	
6	`X1 = -B / 2.0 / A;`		*			*
7	`X2 = sqrt(D); }`		*			*
8	`if (Print)`	*	*	*	*	*
9	`printf("Solution: %f, %f\n", X1, X2);`	*	*			
10	`return D;`	*	*	*	*	*
11	`}`	*	*	*	*	*

Рис. 12.1. Матриця покриття тестованого коду

Обгрунтування застосування *методів мінімізації* полягає в наступному:

- Кореляція між ефективністю виявлення помилок і покриттям коду вище, ніж між ефективністю виявлення помилок і розміром безлічі тестів. Неефективне тестування, наприклад багатогодинне виконання тестів, не збільшують покриття коду, може призвести до помилкового висновку про *коректності програми.*
- Незалежно від способу породження вихідного набору тестів, його мінімальні підмножини мають перевагу в розмірі та ефективності, оскільки

складаються з меншої кількості тестів, не послаблюючи при цьому здатності до виявлення помилок або знижуючи її незначно.

- Взагалі кажучи, скорочений набір тестів, відібраний при *мінімізації,* може виявляти помилки, що не виявляються скороченим набором того ж розміру, обраним випадковим або яким-небудь іншим способом. Така перевага *мінімізації* перед *випадковими методами* в ефективності є закономірним. Проте з усіх *детермінованих методів мінімізація* призводить до створення найменш ефективних наборів тестів, хоча і найменших. Зокрема, *безпечні методи* ефективніше *методів мінімізації,* хоча і набагато дорожче.

Мінімізація набору тестів вимагає певних витрат на *аналіз.* Якщо *вартість* цього аналізу більше витрат на виконання деякого порогового числа тестів, існує дешевший *випадковий метод,* що забезпечує таку ж ефективність виявлення помилок.

Хоча мінімальні набори тестів можуть забезпечувати структурний покриття зміненого коду, найчастіше вони не є безпечними, оскільки очевидно, що деякі тести, потенційно здатні виявляти помилки, можуть залишитися за межею *відбору.* Набір функціональних тестів зазвичай не володіє надмірністю в тому сенсі, що ніякі два тести не покривають одні й ті ж *функціональні вимоги.* Якщо тести початково створювалися за критерієм структурного покриття, *мінімізація* приносить плоди, але коли ми маємо справу з функціональними тестами, переважніше не відкидали тести, потенційно здатні виявляти помилки. В існуючій практиці тестування інженери воліють не займатися *мінімізацією* набору тестів.

Багато критерії покриття коду фактично не вимагають вибору мінімального *безлічі* тестів. У певному сенсі, про безпечні стратегіях і стратегіях мінімізації можна думати як про знаходяться на двох полюсах *безлічі* стратегій. На практиці, використання "майже мінімальних" наборів тестів може бути задовільним. Прагнення до скорочення обсягу набору

тестів заснована на інтуїтивному припущенні, що кількаразове повторне виконання коду в ході *модульного тестування* "марнотратно". Однак зусилля, необхідні для*мінімізації* набору тестів, можуть бути істотні, і, отже, можуть не виправдовувати витрат. Відзначимо, що більшість стратегій*вибіркового регресійного тестування,* описаних в літературі, в общем-то, не залежить від критерію покриття, можливо, що використовувався при створенні вихідного набору тестів. Інженери, що займаються регресійним тестуванням, часто не мають інформації про те, як розроблявся початковий набір тестів.

Виявлення помилок важливо для додатків, де *вартість* виконання тестів дуже висока, в той час як *вартість* пропуску помилок вважається незначною. У цих умовах використання *методів мінімізації* доцільно, оскільки вони пов'язані з *відбором* невеликої кількості тестів. Прикладом застосування *методів мінімізації* служить метод, що вибирає з T не менше одного тесту для кожного оператора програми, доданого або зміненого при створенні P '. У таблиці на Рис. 12.1 для випадку регресійного тестування функціїEquation даний метод обмежиться *відбором* одного тесту - тесту 2, так як цей тест покриває обидві змінені рядки.

Методи, засновані на покритті коду

Значення методів, заснованих на покритті коду, полягає в тому, що вони гарантують збереження обраним набором тестів необхідного ступеня покриття елементів P 'щодо деякого критерію структурного покриття C, що використовувався при створенні первинного набору тестів. Це не означає, що якщо *атрибут* програми, певний C, покривається початковим безліччю тестів, він буде також покритий і вибраним безліччю; гарантується тільки збереження відсотка покривається коду. *Методи, засновані на покритті,* зменшують розкид по покриттю, вимагаючи *відбору тестів,* що активують важкодоступний код, і виключення тестів, які тільки дублюють покриття. Оскільки на практиці критерії покриття коду зазвичай застосовуються для *відбору* єдиного тесту для кожного

що покривається елемента, *підходи, засновані на покритті коду,* можна розглядати як специфічний вид *методів мінімізації.*

Різновидом *методів, заснованих на покритті коду,* є *методи, які базуються на покритті* потоку даних. Ці методи ефективніше *методів мінімізації* і майже настільки ж ефективні, як *безпечні методи.* У той же час, вони можуть вимагати, принаймні, такого ж часу на*аналіз,* як і найбільш ефективні *безпечні методи,* і, отже, можуть обходитися дорожче *безпечних методів* і набагато дорожче інших*методів мінімізації.* Вони мають тенденцію до включення надлишкових тестів в набір регресійних тестів для покриття залежать від змін пар визначення-використання, що, в деяких випадках, веде до великого числа відібраних тестів. Цей факт зафіксовано експериментально.

Методи, засновані на використанні потоку даних, можуть бути корисні і для інших *задач регресійного тестування,* крім *відбору тестів,* наприклад, для знаходження елементів P, недостатньо тестованих T '.

Метод стовідсоткового покриття зміненого коду аналогічний *методу мінімізації.* Так, для прикладу таблиці с Рис. 12.1 існує 4 способи відібрати 2 тесту відповідно до цього критерію. Одного тесту недостатньо. Результати порівняння методів *вибіркового регресійного тестування* наведені в Табл. 12.1 .

Таблиця 12.1. Порівняння методів *вибіркового регресійного тестування*

Клас методів	**Випадкові**	**Безпечні**	**Мінімізації**	**Покриття**
Повнота	Від 0% до 100%	100%	<100%	<100%
Розмір набору тестів	Налаштовується	Великий	Невеликий	Залежить від параметрів методу
Час виконання методу	Нехтує мало	Значне	Значне	Значне

Перспективні властивості методів регресійного тестування	Відсутність засоби підтримки регресійного тестування	Високі вимоги щодо якості	Вартість пропуску помилки невелика	Набір вихідних тестів створюється за критерієм покриття

Лекція 10: Регресійне тестування: методики, не пов'язані з відбором тестів і методики породження тестів

Анотація: Розглядається метод зменшення обсягу що тестується, методи упорядкування тестів, а також коло питань, пов'язаних з доцільністю регресійного тестування, а також методика породження нових тестів на основі аналізу підозрілих станів і сценарій її застосування.

Ключові слова: регресійне тестирование , ПО , значение , ресурс , операторы , брандмауэр , подмножество , граф викликів ,відбір тестів , інтеграційне тестування , об'єктно-орієнтований підхід , безлічі , ймовірність , модульне тестування , клас , тест здібностей , виклик методу , час выполнения , анализ , механизмы , путь , программа , место , управляющие , стоимость ,нумерация , вычисление , запуск , время компіляції , метод зменшення обсягу що тестується , метод впорядкування тестів ,функція , вага , вибіркове регресійне тестування , висновок , компроміс , правильна программа , затраты , мощность , минимизация, детерминированный метод , потужність множини , відношення , компіляція , завантаження , програмне забезпечення , прогін тестів , витрати , інформація , мінімум , тип сущности , модуль , переменная , макроопределения , безопасный метод , доцільність регресійного тестирования , точность , сценарий , выход , цикла , завершение работы , список , суперпозиція , методика породження тестів , аналіз підозрілих станів , система контролю версій

Інтеграційне регресійне тестування

З появою нових напрямків у розробці програмного забезпечення (наприклад, об'єктно-орієнтованого програмування), що заохочують використання великої кількості маленьких процедур, підвищується важливість обробки межмодульного впливу змін коду для методик зменшення вартості *регресійного*

тестування. Для вирішення цього завдання необхідно розглядати залежно *з*глобальних змінним, коли змінної в одній або декількох процедурах присвоюється *значення,* яке потім використовується в багатьох інших процедурах. Таку залежність можна розглядати як залежність між процедурами *по* потоку даних. Також можливі міжмодульні залежності *по* ресурсах, наприклад *по* пам'яті, коли *ресурс* поділяється між кількома процедурами. Відзначимо, що при системному регрессионном тестуванні залежності такого роду можна ігнорувати.

Якщо зміна специфікації вимог зачіпає глобальну змінну, можуть знадобитися нові модульні тести. В іншому випадку, повторному виконанню підлягають тільки модульні тести, що зачіпають як змінений код, так і *оператори,* що містять посилання на глобальну змінну.

Брандмауер можна визначити як *підмножина графа викликів,* що містить змінені і залежні від змін процедури та інтерфейси. Методи*відбору тестів,* що використовують *брандмауер,* вимагають повторного *інтеграційного тестування* тільки тих процедур і інтерфейсів, які безпосередньо викликають або викликаються з змінених процедур.

Регресійне тестування об'єктно-орієнтованих програм

Об'єктно-орієнтований підхід стимулює нові додатки методик вибіркового повторного тестування. Дійсно, при зміні класу необхідно виявити в наборі тестів класу тільки тести, що вимагають повторного виконання. Точно так само при породженні нового класу з існуючого необхідно визначити тести з *безлічі* тестів базового класу, що вимагають повторного виконання на класі-нащадку. Хоча завдяки інкапсуляції *ймовірністьпомилкового* взаємодії об'єктно-орієнтованих модулів коду зменшується, проте, можливо, що тестування прикладних програм виявить помилки в методах, що не знайдені при *модульному тестуванні* методів. У цьому випадку необхідно розглянути всі прикладні програми, що використовують змінений *клас,* щоб продемонструвати, що всі існуючі *тести, здатні*виявляти помилки в змінених класах, були запущені

повторно і було вибрано безпечне безліч тестів. При повторному тестуванні прикладних програм, класів або їх спадкоємців застосування методик вибіркового повторного тестування до існуючих наборам тестів може принести чималу користь.

В об'єктно-орієнтованої програми *виклик методу* під *час виконання* може бути зіставлений будь-якого з ряду методів. Для заданого виклику ми не завжди можемо статично визначити метод, за яким він буде пов'язаний. Вибіркові методи повторного тестування, які покладаються на статичний *аналіз,* повинні забезпечувати *механізми* для вирішення цієї невизначеності.

Зменшення обсягу тестується

Ще один *шлях* скорочення витрат на *регресійне тестування* полягає в тому, щоб замість повторного тестування (великий) зміненої програми з використанням відповідно великого числа тестів довести, що змінена *програма* адекватно тестується за допомогою виконання деякого (меншого) числа тестів на остаточний програмі. Залишкова *програма* створюється шляхом використання графа залежності системи замість графа потоку керування, що дозволяє виключити непотрібні залежності між компонентами в межах одного шляху графа потоку керування. Так, коректування-якого оператора в ідеалі повинна приводити до необхідності тестувати залишкову програму, що складається з усіх операторів вихідної програми, здатних вплинути на цей оператор або опинитися у сфері його впливу. Для отримання залишкової програми необхідно знати *місце* коригування (в термінах операторів), а також інформаційні та *керуючі* зв'язку в програмі. Цей підхід працює найкраще для малих і середніх змін великих програм, де висока *вартістьрегресійного тестування* може змусити взагалі відмовитися від його проведення. Наявність дешевого методу залишкових програм, що забезпечує таку ж ступінь покриття коду, робить *регресійне тестування* успішним навіть у таких випадках.

Метод залишкових програм має ряд обмежень. Зокрема, він не працює при перенесенні програми на машину з іншим процесором або об'ємом

пам'яті. Більше того, він може давати невірні результати і на тій же самій машині, якщо поведінка програми залежить від адреси її початкового завантаження, або якщо для залишкової програми потрібно менше пам'яті, ніж для зміненої, і, відповідно, на остаточний програмі проходить тест, який для зміненої програми викликав би помилку нестачі пам'яті. Дослідження методу на програмах невеликого обсягу показали, що виконання меншої кількості тестів на остаточний програмі не виправдовує витрат на*відбір тестів* і зменшення обсягу програми. Однак для програм з великими наборами тестів це не так.

Для тіста 1 рис. 12.1 для функції Equation залишкова *програма* виглядає так, як показано в Табл. 13.1 . *Нумерація* рядків залишена такою ж, як у вихідній програмі. Таким чином, можна помітити, що були видалені рядки 6 і 7, які не будуть зачіпатися тестом 1 в ході його виконання, а також рядки 3 і 8, що містять *обчислення* предикатів, які в ході виконання тесту завжди істинні.*Запуск* тесту на повній зміненої програмі і на остаточний програмі призводить до активізації одних і тих же операторів, тому виграшу в часі отримати не вдається, однак за рахунок скорочення обсягу програми зменшується *час компіляції.* Для нашого прикладу цей виграш незначний і не виправдовує витрат на *аналіз,* необхідний для зменшення об'єму. Таким чином, розглянута технологія рекомендується до застосування, перш за все, у випадках, коли *вартість* компіляції відносно висока.

Таблиця 13.1. Залишкова програма

№	**Рядок коду**
1	double Equation (int Print, float A, float B, float C, float & X1, float & X2) {
2	float D = B * B - 4.0 * A * C;
4	X1 = (-B + sqrt (D)) / 2.0 / A;
5	X2 = (-B - sqrt (D)) / 2.0 / A;

9	printf ("Solution:% f,% f \ n", X1, X2);
10	return D;
11	}

Відомості про *методикою зменшення обсягу тестируемой програми* наведені в Табл. 13.2 .

Таблиця 13.2. Результати застосування методики зменшення обсягу	
Характеристика	**Зміна в результаті застосування методики**
Час компіляції програми, що тестується	Зменшується
Час виконання програми, що тестується	Не змінюється
Час роботи методу відбору	Збільшується
Ризик пропуску помилок	Збільшується
Результати застосування методики на практиці	Негативні

Методи впорядкування

Методи впорядкування дозволяють інженерам-тестувальникам розподілити тести так, що тести з більш високим пріоритетом виконуються раніше, ніж тести з більш низьким пріоритетом, щоб потім обмежитися вибором перших n тестів для повторного виконання. Це особливо важливо для випадків, коли тестувальники можуть дозволити собі повторне виконання тільки невеликої кількості регресійних тестів.

Однією з проблем упорядкування тестів є відсутність уявлень про те, скількох тестів в конкретному проекті достатньо для відбору.На відміну від підходу мінімізації, що використовує всі тести мінімального набору, оптимальне число

тестів в упорядкованому наборі невідомо. Виникає проблема балансу між тим, що необхідно робити в ході *регресійного тестування,* і тим, що ми можемо собі дозволити. У результаті кількість запускаються тестів визначається обмеженнями *за* часом і бюджету і порядком тестів в наборі. З урахуванням цих чинників слід запускати повторно якомога більше тестів, починаючи з верхнього рядка списку упорядкованих тестів.

Можливе перевагу впорядкування тестів полягає в тому, що складність відповідного алгоритму, O (n $^{2)}$ у найгіршому випадку, менше, ніж складність алгоритму мінімізації, який в деяких випадках може вимагати експоненціального часу виконання.

Проблему впорядкування тестів можна сформулювати наступним чином:

- Дано: T - набір тестів, PT - набір перестановок T, f - функція з PT на безліч дійсних чисел.
- Знайти: набір $T' \in PT$ такий, що:

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$$

У наведеному визначенні PT являє собою безліч всіх можливих варіантів упорядкування - T, а f - *функція,* яка, будучи застосована до будь-якого такого впорядкування, видає його *вагу.* (Припустимо, що великі значення ваг переважніше малих.)

Впорядкування може переслідувати різні цілі:

- Збільшення частоти виявлення помилок наборами тестів, тобто збільшення ймовірності виявити помилку раніше при виконанні регресійних тестів з цих наборів.
- Прискорення процесу покриття коду тестованої системи і досягнення необхідного ступеня покриття коду на більш ранніх етапах процесу тестування.
- Найшвидший зростання ймовірності того, що тестована система надійна.

- Збільшення ймовірності виявлення помилок, пов'язаних з конкретними змінами коду, на ранніх етапах процесу тестування і т.п.

Методи впорядкування планують виконання тестів в процесі *регресійного тестування* в порядку, що збільшує їх ефективність у термінах досягнення заданої заходи продуктивності. Обгрунтування використання упорядочивающего методу полягає в тому, що впорядкований набір тестів має велику ймовірність досягнення мети, ніж тести, розташовані по якомусь іншому правилу або у випадковому порядку.

Розрізняють два типи впорядкування тестів: загальне і залежно від версії. Загальне впорядкування тестів за даних програмі P і наборі тестів T передбачає перебування порядку тестів T, який виявиться корисним для тестування декількох послідовних змінених версій. Вважається, що для цих версій підсумковий впорядкований набір тестів дозволяє в середньому швидше досягти мети, заради якої проводилося впорядкування, ніж вихідний набір тестів. Однак є значний обсяг статистичних свідчень на користь наявності зв'язку між частотою виявлення помилок і конкретною версією програми, що тестується: різні версії програми надають різні можливості *з* упорядкування тестів. При регресійному тестуванні ми маємо справу з конкретною версією програмного продукту і хочемо впорядковувати тести найбільш ефективно *по* відношенню саме до цієї версії.

Наприклад, впорядковувати тести можна *за* кількістю покриваються ними змін коду. При безпечному відборі тестів з рис. 12.1будуть обрані тести 1, 2 і 5, з яких найбільш пріоритетним є тест 2, так як він зачіпає обидві зміни, тоді як тести 1 і 5 - тільки одне.

Відомості про *методикою упорядкування тестів* підсумовані в Табл. 13.3 .

Таблиця 13.3. Результати застосування методики упорядкування тестів	
Характеристика	**Зміна в результаті застосування методики**

Час роботи методу відбору	Збільшується незначно
Частота виявлення помилок	Збільшується
Швидкість покриття коду	Збільшується
Результати застосування методики на практиці	Позитивні

Доцільність відбору тестів

Оскільки в загальному випадку оптимальний *відбір тестів* (тобто вибір в точності тих тестів, які виявляють помилку) неможливий, співвідношення між витратами на застосування методів *вибіркового регресійного тестування* і виграшем від їх використання є основним питанням практичного застосування *вибіркового регресійного тестування.* На підставі оцінки цього співвідношення робиться *висновок* про доцільність *відбору тестів.*

Ефективне *регресійне тестування* являє собою *компроміс* між якістю програми, що тестується і витратами на тестування. Чим більше регресійних тестів, тим повніше перевірка *правильності програми.* Однак більшу кількість виконуваних тестів зазвичай означає збільшення фінансових витрат і часу на тестування, що на практиці не завжди прийнятно. Виконання меншої кількості регресійних тестів може виявитися дешевше, але не дозволяє гарантувати збереження якості.

Коли окремі модулі невеликі і нескладні, а пов'язані з ними набори тестів також невеликі, простий повторний *запуск* всіх тестів досить ефективний. При *інтеграційному тестуванні* це менш імовірно. У той час як тести для окремих модулів можуть бути невеликими, тести для груп модулів і підсистем досить великі, що створює передумови для зменшення витрат тестування. З іншого боку, із зростанням розміру додатків *вартість* застосування вибіркової стратегії повторного тестування може зрости до неприйнятною величини. *Витрати* на необхідний для

відбору *аналіз* можуть переважувати економію від прогону скороченого набору тестів та аналізу результатів прогону. Однак в області тестування досить великих програм позитивний баланс витрат і вигод цілком досяжний.

Модель витрат і вигод при використанні вибіркових стратегій *регресійного тестування* повинна враховувати прямі і непрямі *витрати.*Прямі *витрати* включають відбір і виконання тестів і *аналіз* результатів. Непрямі *витрати* включають *витрати* на управління, супровід баз даних і розробку програмних засобів. Вигоди - це *витрати,* яких вдалося уникнути, не виконуючи частину тестів. Щоб метод*вибіркового регресійного тестування* був ефективнішим методу повторного прогону всіх тестів, *вартість* аналізу при відборі підмножини тестів укупі з вартістю їх виконання та перевірки результатів повинна бути менше, ніж *вартість* виконання та перевірки результатів вихідного набору тестів.

Нехай T '- *підмножина* T, відібране деякої стратегією *вибіркового регресійного тестування* M для програми P, | T '| - позначає *потужність* T ', s - середня *вартість* відбору одного тесту в результаті застосування M до P для створення T ', ar - середня *вартість* виконання одного тесту з T на P і перевірки його результату. Тоді для того, щоб вибіркове *регресійне тестування* було доцільним, потрібне виконання нерівності:

$$s \mid T' \mid < r (\mid T \mid - \mid T' \mid)$$

Застосовуючи вищезгадану модель вартості з метою аналізу витрат, корисно умовно розділяти *регресійне тестування* на дві фази - попередню і критичну. Попередня фаза *регресійного тестування* починається після випуску чергової версії програмного продукту;під час цієї фази розробники розширюють функціональність програми і виправляють помилки, готуючись до випуску наступної версії.Одночасно тестувальники можуть планувати майбутнє тестування або виконувати завдання, що вимагають наявності тільки попередній версії програми, такі як збір тестових траєкторій і *аналіз* покриття. Як тільки в програму внесені виправлення, починається критична фаза *регресійного*

тестування. Протягом цієї фази *регресійне тестування* нової версії програми є домінуючим процесом, час якого зазвичай обмежена моментом поставки замовнику. Саме на критичній фазі *регресійного тестування* найбільш важлива *мінімізація* витрат. При використанні вибіркового методу регресійного тестування важливо використовувати факт наявності цих двох фаз, приділяючи якомога більше уваги виконанню завдань, пов'язаних з аналізом, протягом попередньої фази, щоб на критичній фазі займатися тільки прогоном тестів і зменшити *ймовірність* зриву термінів постачання. Тим не менш, важливо розуміти, що до внесення останньої зміни в код *аналіз* може бути виконаний тільки частково.

Якщо не враховувати не дуже великих витрат на *аналіз* при використанні *детермінованих методів,* рішення про застосування конкретного методу *відбору тестів* залежатиме від ставлення вартості виконання більшої кількості тестів до ціни пропуску помилки, що залежить від *безлічі* факторів, специфічних для кожного конкретного випадку. При відсутності помилок заощадження пропорційні зменшення розміру набору тестів і можуть бути виміряні в термінах відсотка вибраних тестів, | T '| / | T |.

Моделі вартості можуть використовуватися як при виборі найкращої, так і для оцінки придатності конкретної стратегії. При аналізі враховуються такі чинники, як розмір програми (у рядках коду), *потужність множини* регресійних тестів і кількість покриваються елементів, задіяних вихідним безліччю тестів.

Загальний метод дослідження проблеми доцільності *відбору тестів* полягає в знаходженні або створенні вихідної і зміненої версій деякої системи і відповідного набору тестів. У цих умовах застосовується методика *відбору тестів,* і розмір і ефективність обраного набору тестів порівнюється з розміром і ефективністю первинного набору тестів. Результати показують, що застосування методів відбору регресійних тестів, у тому числі і безпечних, не завжди доцільно, оскільки *витрати* і вигоди від їх використання змінюються в

широкому діапазоні в залежності від багатьох факторів. На практиці набори, засновані на покритті, забезпечують кращі результати*відбору тестів.*

Зрозуміло, *ставлення* покриття - не єдиний фактор, який може відбитися на доцільності застосування *вибіркового регресійного тестування.* Для деяких додатків створення умов для тестування (у тому числі *компіляція* і *завантаження* модулів і введення даних) може обходитися набагато дорожче, ніж обчислювальні ресурси для безпосереднього виконання тестованої системи. Наприклад, в телекомунікаційній промисловості *вартість* створення тестової лабораторії для моделювання реальної мережі зв'язку може досягати декількох мільйонів доларів.

Підрахунок порога доцільності допомагає визначити, чи може *відбір тестів* взагалі бути доцільний для даного програмного виробу і набору тестів. Однак навіть у випадках, коли *значення* порога доцільності вказує, що *відбір тестів* може бути доцільний, він не обов'язково буде таким; результат залежить від параметрів набору тестів, таких як розмір набору, характеристики покриття коду, рівень подробиці і *час виконання* тестів, а також від місця розташування змін. Істотно вплинути на загальну оцінку можуть *витрати*на оплату праці тестового персоналу, доступність вільного машинного часу для *регресійного тестування,* доступність стенду, на якому розгорнуто *програмне забезпечення* додатки і т.п. Відзначимо, що *вартість прогону тестів* пов'язана не стільки з розміром програми, скільки з обмеженнями на допустимий час прогону.

У деяких випадках, коли число тестів, відкинутих вибірковим методом *регресійного тестування* незначно, але його застосування проте заслуговує на увагу. Справа в тому, що будь-яке скорочення високовитратного часу використання тестової лабораторії особливо важливо, а для *відбору тестів* використовуються інші ресурси. Подібні обставини необхідно включати в оцінку вартості аналізу шляхом врахування не тільки вартості експлуатації ресурсу, а й таких факторів як час доби, день тижня, час, що залишився до

випуску чергової версії продукту тощо У цьому випадку модель вартості повинна дотримувати баланс між високою вартістю*прогону тестів* в тестовій лабораторії і відносно невеликою вартістю проведення аналізу на незайнятих комп'ютерах.

Для деяких програм і наборів тестів вибіркове тестування неефективно, так як поріг доцільності перевищує число тестів в наборі. У таких випадках методи *відбору тестів,* незалежно від того, наскільки успішно вони зменшують число тестів, що вимагають повторного виконання, не можуть давати економію. Цей результат відображає той факт, що доцільність відбору залежить як від вартості аналізу, так і від вартості виконання тестів. Можливість досягнення економії при відборі регресійних тестів для конкретної системи програмного забезпечення і конкретного набору тестів повинна оцінюватися комплексно з урахуванням всіх впливають на рішення чинників.

Варто зауважити, що доцільність застосування вибіркового методу *регресійного тестування* не можна сприймати як щось само собою зрозуміле. Слід дуже обережно підходити до оцінки доцільності відбору повторно проганяють тестів. У ряді випадків, коли або одержуване число залишкових тестів близько до первісного їх кількості, або накладні *витрати* на повторне тестування незначні, вигідніше проганяти заново всі тести, особливо якщо *прогін тестів* повністю автоматизований.

Функції передбачення доцільності

На практиці не існує способу в точності передбачити, скільки тестів буде вибрано при мінімізації (або *за* результатами застосування будь-якої іншої методики *відбору тестів)*. Коли кількість тестів, відсіяних *за* результатами відбору, незначно, ресурси, витрачені на*відбір тестів,* пропадають даремно. У таких випадках говорять, що вибіркове *регресійне тестування* недоцільно. Щоб з'ясувати, чи заслуговує на увагу спроба застосування вибіркового методу *регресійного тестування,* необхідно використовувати прогнозуючу функцію.

Прогнозуючі функції засновані на покритті коду. Для їх обчислення використовується *інформація* про частку тестів, що активують покриваються сутності - *оператори,* гілки або функції, - що дозволяє передбачити кількість тестів, які будуть обрані при зміні цих сутностей. Існує як *мінімум* одна прогнозуюча *функція,* яка може бути використана для передбачення доцільності застосування безпечної стратегії *вибіркового регресійного тестування.*

Нехай P - тестована система, S - її специфікація, T - набір регресійних тестів для P, а | T | означає число окремих тестів вT. Нехай M - вибірковий метод *регресійного тестування,* використовуваний для відбору підмножини T при тестуванні зміненої версії P; M може залежати від P, S, T, інформації про виконання T на P та інших факторів. Через E позначимо набір розглянутих M сутностей тестованої системи. Передбачається, що T і E непорожні, і що кожен синтаксичний елемент Рналежить, *принаймні,* однієї сутності з E. *Ставлення* covers $_{M}$ (t, E) визначається як *ставлення* покриття, що досягається методом M для P. Це *ставлення* визначено над TxE і справедливо тоді і тільки тоді, коли виконання тесту t на Рпризводить до виконання сутності e як *мінімум* один раз. *Значення* терміна "виконання" визначено для всіх *типів сутностей* Р.Наприклад, якщо e - *функція* або *модуль* P, e виконується при виклику цієї функції або модуля. Якщо e - простий оператор, умовний оператор, пара визначення-використання або інший вид елемента шляху в графі виконання P, e виконується при виконанні цього елемента шляху. Якщо e - *змінна* P, e виконується при читанні або запису цієї змінної. Якщо e - тип P, евиконується при виконанні будь-якої змінної типу e. Якщо e - *макровизначення* P, e виконується при виконанні розширення цього *макроозначення.* Якщо e - сектор P, e виконується при виконанні всіх складових його операторів. Відповідні значення терміна "виконання" можуть бути визначені *за* аналогією для інших *типів сутностей* P.

Для даної тестованої системи P, набору регресійних тестів T і вибіркового методу *регресійного тестування* M можна передбачити, чи варто

задіяти M для *регресійного тестування* майбутніх версій P, використовуючи інформацію про ставлення покриття covers $_{M,}$ що досягається при використанні M для T і P. Прогноз заснований на метриці вартості, відповідної P іT. Щодо витрат приймаються деякі спрощують припущення.

Нехай E ^{C} позначає безліч покритих сутностей:

$$E^C = \{e \in E | (\exists t \in T)(covers_M(t, E))\}.$$

Позначення | E ^{C} | використовується для числа покритих сутностей. Іноді зручно представити залежність covers $_{M}$ (t, E) у вигляді бінарної матриці C, рядки якої представляють елементи T, а стовпці - елементи E. При цьому елемент C $_{i, j}$матриці C визначається наступним чином:

C $_{i, j}$ = 1, якщо covers $_{M}$ (i, j)

C $_{i, j}$ = 0, інакше

Ступінь накопиченого покриття, забезпечуваного T, тобто загальне число одиниць в матриці C, позначається CC:

$$|T||E|CC = \Sigma\Sigma C_{i,j} i = 1\ j = 1$$

Відзначимо, що якщо обмежитися включенням в C тільки стовпців, відповідних покритим сутностей E $_{C,}$ накопичене покриттяCC залишиться незмінним. Зокрема, для всіх непокритих сутностей u C $_{i, u}$ дорівнює нулю для всіх тестів i (так як covers $_{M}$(i, u) помилково для всіх таких випадків). Отже, обмеження на E ^{C} при обчисленні суми, визначальною CC, призводить тільки до виключення доданків, рівних нулю.

Нехай T $_{M}$ - *підмножина* T, вбрання M для P, і нехай | T $_{M}$ | позначає його *потужність,* тоді $T_M = \{t \in T | M \text{выбирает } t\}$. Нехай s $_{M}$ - питома *вартість* відбору одного тесту для T $_{M}$ при застосуванні Мдо P, і нехай r - питома *вартість* виконання одного тесту з T на P і перевірки його результату. M доцільно використовувати як методу *відбору тестів* тоді і тільки тоді, коли:

s $_{м}$ | T $_{м}$ | <r (| T | - | T $_{м}$ |), тобто *вартість* аналізу, необхідного для відбору T $_{м,}$ повинна бути менше вартості прогону невибраних тестів, $T \supseteq T_M$.

Оцінка очікуваного числа тестів, що вимагають повторного запуску, позначається N $_{м}$ і обчислюється таким чином:

N $_{m}$ = CC / | E |

Використання цієї прогнозуючої функції передбачається тільки у випадках, коли мета вибіркової стратегії *регресійного тестування*полягає в повторному виконанні всіх тестів, порушених змінами, тобто використовується *безпечний метод відбору тестів.* Кілька вдосконалений варіант оцінки N $_{м,}$ що використовує як простору сутностей E ^{C} замість E:

N ^{c} $_{m}$ = CC / | E ^{c} |

Прогнозуюча *функція* для частки набору тестів, що вимагає повторного виконання, тобто для | T $_{м}$ | / | T |, позначається π_{M} :

$$\pi_m = N_m^c/|T| = CC/|E^c||T|$$

Прогнозуюча *функція* π_{M} покладається безпосередньо на інформацію про покриття. Головні передумови, що лежать в основі застосування прогнозуючої функції, такі:

- *Доцільність застосування вибіркового методу регресійного тестування* і, як наслідок, наша здатність до передбачення доцільності, безпосередньо залежить від частки тестового набору, обраній для виконання методом *регресійного тестування.*
- Ця частка в свою чергу безпосередньо залежить від ставлення покриття.

Точність прогнозуючої функції на практиці може значно змінюватися від версії до версії. Проблема точності може виявитися досить серйозною, проте, оскільки прогнозуюча *функція* використовується для довготривалого передбачення поведінки методу протягом декількох версій, застосування середніх значень вважається допустимим. *Ставлення* covers $_{м}$ (t, e) у ході супроводження

змінюється дуже слабо. *З* цієї причини *інформація,* отримана в результаті аналізу єдиної версії, може виявитися достатньою для управління *відбором тестів* протягом декількох послідовних нових версій.

Існують фактори, що впливають на доцільність *відбору тестів,* але не враховуються прогнозуючої функцією. Один з підходів поліпшення якості прогнозу полягає у використанні інформації про історію змін програми, яку часто можна отримати з системи управління конфігурацією.

Наприклад, в рис. 12.1 прогнозуюча *функція* може бути підрахована як *відношення* загальної кількості зірочок в таблиці до кількості рядків таблиці, тобто числу покриваються сутностей. Ця величина складає 42/11 = 3.8, тобто *безпечний метод* відбиратиме в середньому близько 4 тестів. Відомості про методику передбачення підсумовані в Табл. 13.3 .

Таблиця 13.4. Результати застосування методики передбачення	
Характеристика	**Зміна в результаті застосування методики**
Час роботи методу відбору в разі, якщо вибіркове тестування доцільно	Збільшується незначно
Час роботи методу відбору в разі, якщо вибіркове тестування недоцільно	Зменшується до пренебрежимо малої величини
Зниження точності передбачення від версії до версії	Залежить від обсягу змін
Результати застосування методики на практиці	Позитивні (помилка в 0.8%)

Породження нових тестів

Породження нових тестів при структурному регрессионном тестуванні зазвичай обумовлено недостатнім рівнем покриття. Нові тести розробляються так, щоб

задіяти ще не покриті ділянки вихідного коду. Процес припиняється, коли рівень покриття коду досягає необхідної величини (наприклад, 80%). Розробка нових тестів при функціональному регрессионном тестуванні є менш тривіальним завданням і зазвичай пов'язана з введенням нових вимог або з бажанням перевірити деякі сценарії роботи системи додатково.

Основою більшості програмних продуктів для керуючих застосувань, що знаходяться в промисловому використанні, є цикл обробки подій. *Сценарій* роботи з системою, побудованої *по* такій архітектурі, складається з послідовності транзакцій, управління після обробки кожної транзакції знову передається циклу обробки подій. Виконання транзакції призводить до зміни стану програми; в результаті деяких транзакцій відбувається *вихід* з *циклу* і *завершення роботи* програми. Тести для таких програм являють собою послідовність транзакцій.

Розвиток програмного продукту від версії до версії тягне за собою появу нових станів. Оскільки більшість тестів легко може бути розширене шляхом додавання додаткових транзакцій в *список,* нові тести можна створювати шляхом *суперпозиції* вже наявних, з урахуванням інформації про зміну стану тестованої системи в результаті *прогону тесту.* Цей підхід дозволяє вказати, якого роду нові тести з найбільшою ймовірністю виявлять помилки.

Позначимо тестовану програму P, а безліч її тестів $T = \{t_1, t_2, ..., t_n\}$. Будемо вважати, що стан тестується sвизначається сукупністю значень деякогопідмножини глобальних і локальних змінних. При створенні нових тестів будемо розглядати стану програми перед запуском тесту (s_0) і після його закінчення (s_j). Інформацію про ці станах необхідно збирати для кожного тесту *за* результатами запуску на попередній версії продукту. *Методика породження нових тестів* на основі*аналізу "підозрілих" станів* зводиться до описаної нижче послідовності дій.

1. Обчислення списку глобальних і локальних змінних, що визначають стан програми s.

2. Збір інформації (на основі аналізу профілю програми, отриманого на попередній версії продукту i-1, для кожного існуючого тесту $t_{j)}$ про стани програми перед запуском тесту і після його закінчення (тобто s_0 і $s_{j)}$. Безліч таких станів позначається $S_{i-1} : S_{i-1} = s_0 \cup \{s_j | \forall j\}$

3. Виконання на поточній версії продукту і нових та вибраних регресійних тестів з безлічі T '. За аналогією з S_{i-1}обчислюється безліч $S_{i,}$ яке зберігається під управлінням *системи контролю версій.*

4. Оцінка "підозрілих" з точки зору наявності помилок безлічі нових порівняно з попередніми версіями станів N_i у відповідності з наступною формулою: $N_i = S_i \setminus S_{i-1}$

5. Аналіз станів безлічі $N_{i,}$ в яких подальша робота продукту неможлива відповідно до специфікації. Предмет аналізу - визначити чи створюються ці стани в результаті виконання тестів, перевіряючих нештатні режими роботи продукту, або будь-яких інших тестів. В останньому випадку фіксується помилка.

6. Виняток нештатних станів з безлічі $N_{i.}$

7. Перехід до кроку 10, якщо нових станів, що допускають продовження виконання програми, не виявлено, тобто $N_i = \varnothing$.

8. Для кожного стану безлічі N_i обчислення вектора відмінності від вихідного стану $s_{0,}$ тобто безлічі змінних, змінених порівняно з $s_{0.}$

9. Модифікація безлічі змінених рядків вихідного коду P на основі інформації про змінені змінних і використання будь-якої методики *відбору тестів* для *вибіркового регресійного тестування.*

10. Повторне виконання кроків 3-9 до досягнення стану $N_i = \varnothing$ або до закінчення часу, відведеного на *регресійне тестування.* Використання методів розбиття на класи еквівалентності для дострокового прийняття

рішення про припинення циклу тестування, якщо жоден з тестів, створених на черговому етапі, не належить до нового класу еквівалентності.

Для наведеної методики організації тестування, коли нові тести виходять в результаті *суперпозиції* вже наявних, доцільно в якості вихідних тестів, тобто тих "цеглинок", з яких будуть будуватися тести надалі, брати тести, що укладають всього одну елементарну перевірку. Це допомагає уникнути надмірності при багаторазовому злитті тестів, коли шукані "підозрілі" ситуації виникають в ході роботи тесту, але не аналізуються, так як не повторюються при його завершенні.

Використання описаної методики дозволяє в програмному комплексі знаходити помилки, не виявляються вихідним набором тестів.Відзначимо, що застосування запропонованого підходу неможливо для програм, поняття стану для яких не визначено.

Регресійне тестування: алгоритм і програмна система підтримки

Анотація: Розглядаються методики регресійного тестування, повний алгоритм регресійного тестування і програмна система його підтримки.

Ключові слова: методика регресійного тестування , поєднання , ПО , структура системы , файл , список , множества , п оиск ,макроопределения , функция , запуск , выходные дані , архітектура , моду ль , сценарій , автоматизація

Методика регресійного тестування

Методика призначена для ефективного вирішення завдання вибіркового повторного тестування. Її вихідними даними є: програмаР і її модифікована версія Р ', критерій тестування С, безліч (набір) тестів Т, що раніше використовувалися для тестуванняР, інформація про покриття елементів Р (М (Р, С)) тестами з Т. Необхідно реалізувати ефективний спосіб, який гарантує достатній ступінь впевненості в правильності Р ', використовуючи тести з Т.

Методика будується на основі *поєднання* процедур звичайного і регресійного тестування

Розглянемо процедуру звичайного тестування. У ній для отримання інформації про тестованих об'єктах в ході тестування необхідно встановити відповідність між покриваються елементами і тестами для їх перевірки. Відповідно, процедура тестування повинна включати наведену нижче послідовність дій:

1. Визначити необхідні функціональні можливості програми з використанням, наприклад, методу розбиття на класи еквівалентності.
2. Створити тести для необхідних функціональних можливостей.
3. Виконати тести.
4. У разі необхідності - створити і виконати додаткові тести для покриття залишилися (ще не покритих) структурних елементів (попередньо встановивши їх відповідність функціональним вимогам).
5. Створити базу даних тестів програми.

За аналогією зі звичайним тестуванням, процедура регресійного тестування в процесі супроводу складається з наступних етапів:

1. Використовувати передбачення доцільності. Якщо прогнозована кількість вибраних тестів більше, ніж поріг доцільності, провести повторний прогін всіх тестів. В іншому випадку перейти до кроку 2.
2. Ідентифікація змін ΔP в програмі P '(і безлічі $\Delta \mathrm{M}$ змінених покриваються елементів) і встановлення взаємно однозначної відповідності між покриваються елементами M (P, C) і M (P ', C) відповідно до змін: $\Delta M = (M(P,C) \setminus M(P',C)) \cup (M(P',C) \setminus M(P,C))$
3. Вибір $T' \subseteq T$ - Підмножини вихідних тестів, потенційно здатних виявити пов'язані із змінами помилки в P ', для повторного виконання на P', з використанням результатів, отриманих в пункті 2. Це підмножина можна впорядкувати, а також вказати число тестів, виконання яких достатньо для відповідності якому- або критерієм мінімізації. Для

безпечних методів відбору тестів безліч T 'задовольняє наступним обмеженням: $t_i \in T, t_i \notin T' \Rightarrow P(t_i) \equiv P'(t_i)$

4. Застосування підмножини T 'для регресійного тестування зміненої програми P' з метою перевірки результатів і встановлення факту коректності P 'по відношенню до T' (відповідно до зміненим технічним завданням), а також оновлення інформації про проходження тестів з T 'на P'.

5. У разі необхідності - створення додаткових тестів для доповнення набору регресійних тестів. Це можуть бути нові функціональні тести, необхідні для тестування змін у технічному завданні або нових функціональних можливостей зміненої програми; нові структурні тести для активізації залишилися (непокритих) структурних елементів (попередньо встановивши їх відповідність перевіряється функціональним вимогам).

6. Створення T" - нового набору тестів для P ', застосування його для тестування зміненої програми, перевірка результатів та встановлення факту коректності P' по відношенню до T", оновлення інформації про хід виконання тесту та створення бази даних тестів зміненої програми для зберігання цієї інформації та вихідних даних тестів. Видалення застарілих тестів. T"формується за наступним правилом: $T'' = (T \cup T_{\text{новые}}) \setminus T_{\text{устаревшие}}$

Система підтримки регресійного тестування

Структура системи підтримки регресійного тестування представлена на рис. 14.1 . Вихідний код обох версій програми, що тестується зберігається під управлінням системи контролю версій. Для відбору тестів *за* методом покриття точок використання неісполняемих визначень засобами системи контролю версій створюється *файл* відмінностей, на підставі якого обчислюється*список* доданих, змінених і віддалених рядків вихідного коду, який є зручною формою подання *безлічі* ΔP . Потім проводиться перебір цього

списку. Якщо будь-який рядок списку являє собою макровизначення, здійснюється *пошук* рядків коду, що містять використання цього *макроозначення по* всьому тексту програми, що тестується; знайдені рядки приєднуються до безлічі ΔP .Розширене безліч ΔP зіставляється з результатами прогону тестів з *безлічі* T на попередній версії програми. Якщо в ході виконання будь-якого тесту t_i отримувала управління хоча б один рядок, що входить до безліч ΔP , Тест t_i відбирається для повторного запуску.

При створенні нових тестів *за* методом "підозрілих" станів *функція* тестується, що містить цикл обробки подій, доповнюється операторами виведення значень глобальних і видимих локальних змінних. *Запуск* тестів з *безлічі* T 'на профільованої версії програми дозволяє отримати *список* її станів. Цей *список* аналізується, і для кожного раніше не спостерігалося стану обчислюється*список* змінних, що змінилися *в* порівнянні *з* яким-небудь відомим станом. Безліч ΔP доповнюється рядками коду, де використовуються змінні з цього списку. Для кожного стану вказуються тести, *запуск* яких необхідний. Нарешті, створюється *список*рекомендованих нових тестів у формі, зручній для сприйняття людиною.

Вихідні дані кожної програми-обробника доступні користувачеві, що дозволяє контролювати проміжні результати роботи системи.Наприклад, можна виключити з розгляду змінні, які, хоча і змінюються в ході виконання програми, на її стан не впливають.*Архітектура* системи дозволяє легко розширювати функціональність; наприклад, для підтримки якої нової системи контролю версій досить створити один новий *модуль* об'ємом близько 100 рядків коду. Інші модулі можна використовувати без змін.

Типовий *сценарій* проведення регресійного тестування програм, написаних мовою C, із застосуванням описаної вище системи складається з наступних етапів:

1. Обчислюється безліч ΔP рядків вихідного коду, доданих, віддалених або змінених у порівнянні з попередньою версією.

2. Безліч ΔP доповнюється рядками, безпосередньо не изменявшимися, але містять посилання на змінені макроозначення
3. Обчислюється упорядкований безліч регресійних тестів T ', для яких $\forall i \in T' T_{i,j-1} \cap \Delta P \neq \varnothing$, Де $T_{i, j-1}$ - безліч рядків вихідного коду продукту, які отримують управління в ході виконання тесту і на версії системи j-1. Тести упорядковуються за спаданням кількості змінених рядків в дорозі їх виконання.
4. Обчислюється список глобальних і локальних змінних, що визначають стан програми s. Вихідний код програми, що тестується модифікується так, що інформація про стан s (значення глобальних, статичних і локальних змінних) виводиться в зовнішній файл перед запуском тесту і після його закінчення.
5. Нові тести і тести з безлічі T '(регресійні тести) виконуються на поточній версії продукту j.
6. Тести, перевіряючі нештатні режими роботи продукту, тобто створюють стану, в яких подальша робота продукту неможлива відповідно до специфікації вимог, виключаються з розгляду. Якщо відомо, що жоден тест не призводить до виникнення нештатних станів, даний етап може бути опущений
7. Для кожного тесту і обчислюється безліч $T_{ij.}$
8. Обробляються результати виконання тестів, і створюється безліч $S_{j,}$ що складається з початкового стану s_0 і всіх спостерігалися кінцевих станів. Обчислюється безліч $N_j = S_j \setminus S_{j-1}$ всіх нових у порівнянні з попередніми версіями станів, яка є "підозрілим" з точки зору наявності помилок, і вектора їх відмінностей від вихідного стану $s_{0,}$ тобто змінні, змінені в порівнянні з $s_{0.}$
9. Безліч змінених рядків вихідного коду ΔP доповнюється номерами рядків, де використовуються задані змінні

10. Якщо $N_j = \varnothing$, Цикл роботи завершується. Якщо $N_j \neq \varnothing$, слід перейти до кроку 4 або, якщо лічильник кількості ітерацій роботи системи перевищує деяке задане граничне значення, сповістити про це користувача. Користувач може прийняти рішення про припинення тестування чи пропуску деякого числа циклів.

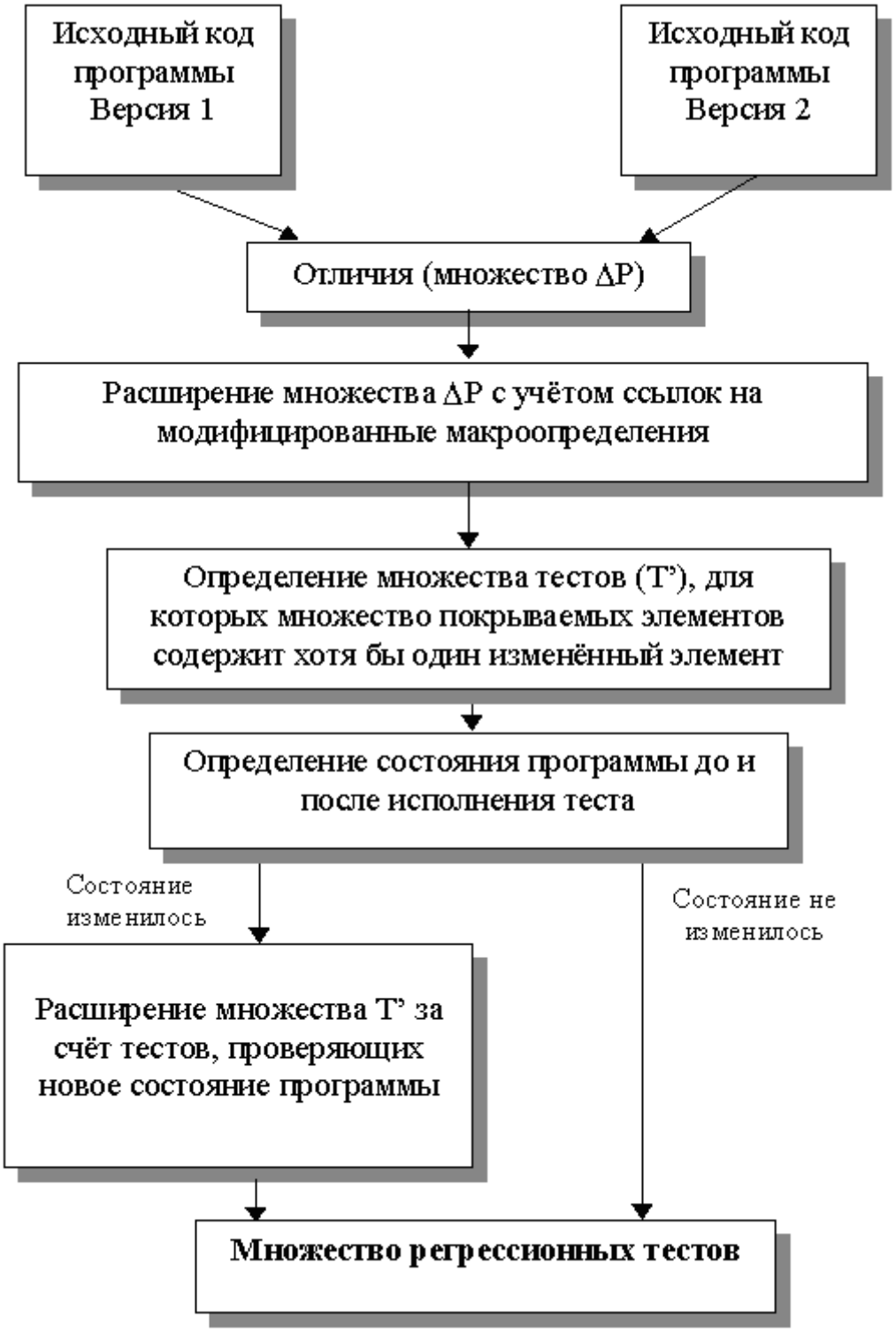


Рис. 14.1. Структура системи підтримки регресійного тестування.

Якщо етап 6 виконується автоматично, а етап 7 можна опустити, можлива повна *автоматизація* регресійного тестування.

Опис тестованої системи та її оточення. Планування тестування

Ключові
слова: БД , програма , Високорівнева команда , управляющие , связь , store , пр иложение , планировщик , системноетестування , чорний ящик , вхідні данные , сеанс , испытание , диаметр , вывод , manual , log-файл , представление , ролирозробників , інтеграційне тестування , тестова стратегія

Практикум базується на тестуванні моделі реальної системи управління автоматизованим комплексом зберігання підшипників. Вона забезпечує прийом підшипників на склад, збереження характеристик надійшли підшипників в базі даних (*БД),* а при надходженні заявки на підшипники разом з параметрами осі - підбір відповідних підшипників та їх видачу. У кожного з елементів комплексу (складу, терміналу підшипника і терміналу осі) існує *програма* низкоуровневого управління, реалізована у вигляді динамічно підключається бібліотеки (dll), що приймає на вхід *високорівневі команди,* і перетворююча їх у *керуючі* впливу на даний елемент тестованої системи. Таким чином, є реальне **оточення** - апаратура та dll, які здійснюють *зв'язок* з апаратурою. Система викликає такі функції з dll для своїх елементів (див. рис. 1.1):

GetStoreStat, GetStoreMessage, SendStoreCom (*Store.* Dll) для складу.

GetAxlePar (Axle.dll) для терміналу осі.

GetRollerPar (Bearing.dll) для терміналу підшипника.

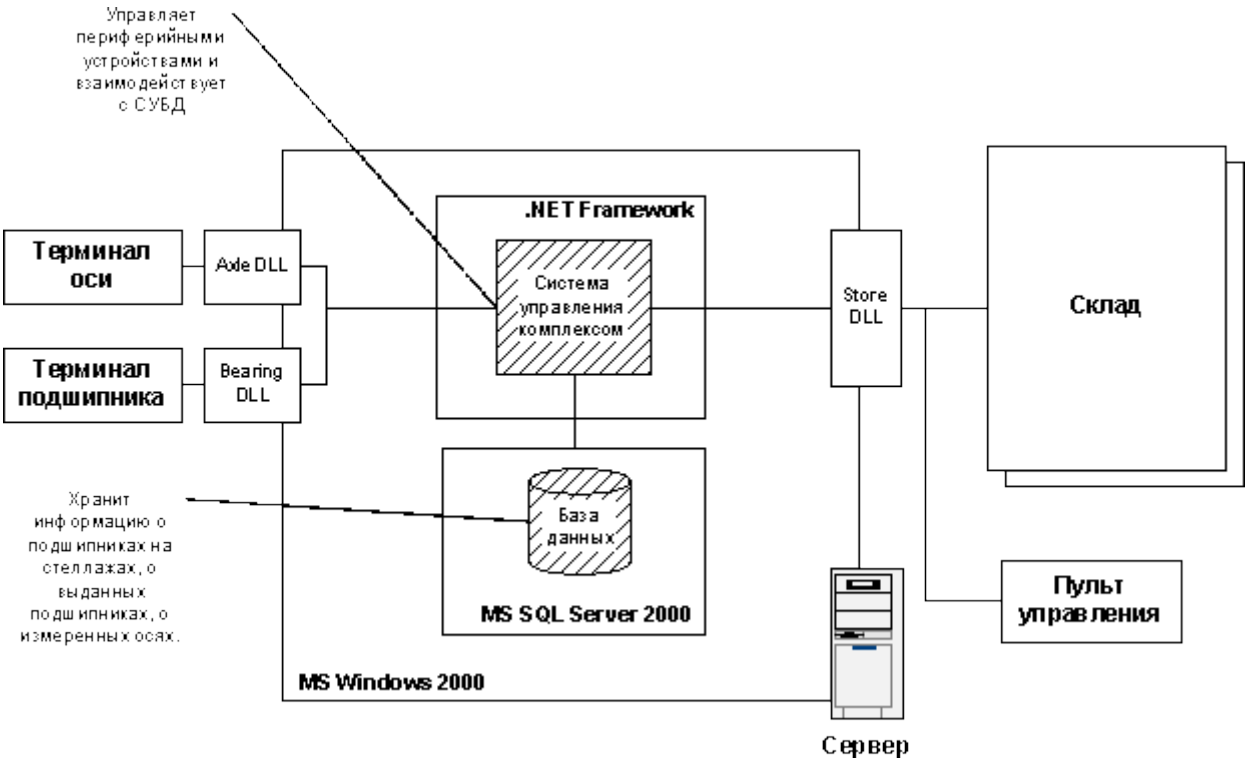


Рис. 1.1. Система і її оточення

Оскільки для тестування використовується модель системи, в її складі реальне оточення замінено на модельне, що забезпечується спеціальною бібліотекою dll-функцій оточення.

Тестируемая система реалізована як багатопотокове *додаток.* Нить породжує недетерминированность поведінки системи в часі.Тому при тестуванні необхідно враховувати, що можливі різні варіанти допустимих часових послідовностей подій системи. Крім того, зовсім не просто точно відтворити прогін конкретного тесту, коли система містить багато паралельних потоків, так як *планувальник*операційної системи сам визначає порядок подій. Зміни, що вносяться до програми, не пов'язані з тестованої системою, можуть вплинути на порядок, в якому будуть виконуватися (відтворюватися) потоки подій тестованої системи. Це може призвести до того, що після виявлення та виправлення дефекту при проведенні повторного тестування далеко не завжди вдасться переконатися в тому, що дефект дійсно усунений, якщо помилка не виявлено під час прогону.

При *системному тестуванні* ми розглядаємо систему як *чорний ящик.* **Тестовий випадок (test case)** являє собою пару **(вхідні дані, очікуваний результат),** в якій *вхідні дані* - це опис даних, що подаються на вхід нашої системи, а очікуваний результат - це опис **вихідних даних,** які система повинна пред'явити у відповідь на відповідний введення. **Виконання (прогін) тестового випадку** - це *сеанс* роботи системи, в рамках якого на вхід системи подаються набори даних, передбачені специфікацією тестового випадку, і фіксуються результати їх обробки, які потім порівнюються з очікуваними результатами, зазначеними в тестовому випадку. Якщо фактичний результат відрізняється від очікуваного, значить, виявлений відмова, тобто тестована система не пройшла *випробування* на заданому **тестовому випадку.** Якщо отриманий результат збігається з очікуваним, значить, тестована система пройшла *випробування* на заданому тестовому випадку. З тестових випадків формуються **тестові набори (test suits). Тестові набори** організовані в певному порядку, що відбиває властивості **тестових випадків.** Якщо система успішно впоралася з усіма **тестовими випадками з набору,** то вона успішно пройшла випробування на **тестовому наборі.**

Для нашої системи вхідними даними є стан оточення її компонентів:

Склад. Стан складу характеризуватиметься наступними параметрами:

Статус складу (StoreStat).

Повідомлення від складу про результати виконання команди (StoreMessage).

Повідомлення від складу про результати отримання команди - статус команди (CommandStatus).

Термінал підшипника. Стан терміналу підшипника задається наступними параметрами:

Статус обміну з терміналом підшипника.

Характеристики (параметри) підшипника (RollerPar):

ПІБ майстра, яка провадила вимірювання.

Назва депо.

Номер робочої зміни.

Номер підшипника.

Номер групи підшипника.

Тип сепаратора підшипника.

Термінал осі. Стан терміналу осі задається наступними параметрами:

Статус обміну з терміналом осі.

Характеристики (параметри) осі (AxlePar):

ПІБ майстра, яка провадила вимірювання.

Назва депо.

Номер осі.

Сторона осі: права чи ліва.

Посадковий *діаметр* задній.

Посадковий *діаметр* передній.

База даних (БД). У *БД* зберігаються характеристики надійшли на склад підшипників. При виборі відповідного для осі підшипника система звертається за цією інформацією до *БД*. Тому має сенс заздалегідь очистити *БД* або заповнити її певними даними.

У специфікації **тестового випадку** повинні бути задані стан оточення **(вхідні дані)** і очікувана послідовність подій в системі**(очікуваний результат).** Після прогону **тестового випадку** ми отримаємо реальну послідовність подій в системі **(вихідні дані)**при заданому стані оточення. Порівнюючи фактичний результат і очікуваний, можна зробити *висновок* про те, чи пройшла тестована система *випробування* на заданому **тестовому випадку.** В якості **очікуваного**

результату будемо використовувати покроковий опис **випадку використання (use case),** так як воно визначає, як при заданому стані оточення система повинна функціонувати. Задаючи очікуваний результат, дуже важливо пам'ятати про те, що при заданому стані оточення можливі різні варіанти послідовності подій системи, які все є правильними.

У процесі роботи послідовність подій (команд) системи, або історія системи, записується в **журнал (log) системи.** Ви можете використовувати SystemLogAnimator (див. п.14 SysLog Animator *Manual)* для візуалізації журналу системи. Вибираючи різні *log-файли*системи для візуалізації, можна отримати наочне і досить повне *уявлення* про функціонування системи і про те, які події і в якому порядку можуть відбуватися в системі.

Планування тестування

Процес тестування

Процес тестування знаходиться в прямій залежності від процесу розробки програмного забезпечення, але при цьому сильно відрізняється від нього, оскільки переслідує інші цілі. Розробка орієнтована на побудову програмного продукту, тоді як тестування відповідає на питання, чи відповідає розроблюваний програмний продукт вимогам, в яких зафіксований початковий задум вироби (тобто те, що замовив замовник).

Разом обидва процеси охоплюють види діяльності, необхідні для отримання якісного продукту. Помилки можуть бути привнесені на кожній стадії розробки. Отже, кожному етапу розробки повинен відповідати етап тестування. Відносини між цими процесами такі, що якщо щось розробляється, то воно піддається тестуванню, а результати тестування використовуються для визначення, чи відповідає це "щось" набору пропонованих вимог. Процес тестування повертає виявлені ним помилки в процес розробки. Процес розробки передає процесу тестування нові і виправлені проектні версії.

Планування тестування

Як було зазначено вище, процес тестування тісно пов'язаний з процесом розробки. Відповідно планування тестування теж залежить від обраної моделі розробки. Однак незалежно від моделі розробки при плануванні тестування необхідно відповісти на п'ять питань, що визначають цей процес:

Хто буде тестувати і на яких етапах?

Розробники продукту, незалежна група тестувальників або спільно?

Які компоненти треба тестувати?

Чи будуть піддані тестуванню всі компоненти програмного продукту або тільки компоненти, які загрожують найбільшими втратами для всього проекту?

Коли треба тестувати?

Чи буде це безперервний процес, вид діяльності, що виконується в спеціальних контрольних точках, або вид діяльності, що виконується на завершальній стадії розробки?

Як треба тестувати?

Чи буде тестування зосереджена тільки на перевірці того, що даний продукт повинен виконувати, або також на тому, як це реалізовано?

У якому обсязі тестувати?

Як визначити, в достатній Чи обсязі виконано тестування, або як розподілити обмежені ресурси, виділені під тестування?

Хто буде тестувати?

Розробник - це роль, для якої характерні види діяльності, орієнтовані на створення програмного продукту (ПП). Тестувальник - це роль, для якої характерні види діяльності, орієнтовані на поліпшення / забезпечення якості програмного продукту. Ця роль передбачає вибір тестів, необхідних для конкретних цілей, побудова тестів, виконання тестів і оцінку результатів. Конкретний виконавець проекту може виступати як в *ролі*

розробника, так і в ролі тестувальника. Момент початку тестування в проекті можна регулювати (рис. 1.2).

Рис. 1.2. Хто тестує

В рамках даного практикуму студенту призначена роль тестувальника.

Які компоненти треба тестувати?

Можуть бути варіанти, коли нічого не треба тестувати (оскільки всі компоненти були протестовані раніше), а може потрібно тестувати кожен компонент з точністю до рядка коду. В об'єктно-орієнтованому програмуванні базовим компонентом є клас. У цьому випадку область тестування визначається класами. Область тестування на рівні класів підлягає вибору (рис. 1.3). Класи, запозичені з інших проектів або взяті з бібліотек, найчастіше в повторному тестуванні не потребують. Існують різні стратегії щодо вибору підмножини класів для тестування.

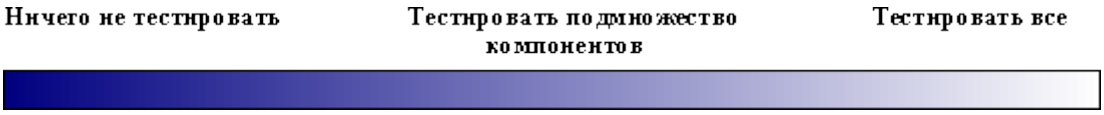


Рис. 1.3. Що тестувати

У нашому випадку буде проводитися тестування всіх класів додатки.

Коли треба тестувати?

Компоненти можна тестувати на завершальному етапі, коли вони будуть інтегровані в єдиний виконуваний модуль. Частота тестування визначається різними міркуваннями. Можна проводити тестування кожен день, враховуючи той факт, що чим раніше виявлена проблема, тим легше і дешевше її рішення. Можна тестувати програмний компонент у міру завершення його

розробки (рис. 1.4). Часте тестування компонентів трохи сповільнює ранні етапи розробки, проте пов'язані з цим втрати з лишком заповнюються за рахунок меншого числа проблем на більш пізніх етапах розробки проекту, коли окремі модулі об'єднуються в більші компоненти системи.

У разі, коли компоненти системи не відрізняються великою складністю, можна спочатку здійснювати інтегрування не піддавалися автономному тестуванню компонентів, а потім тестувати об'єднаний код як єдине ціле. Такий підхід корисний при тестуванні компонентів, для яких реалізація тестових драйверів вимагає істотних зусиль. Тестовий драйвер являє собою програму, яка виконує прогін тестових випадків і збір отриманих при цьому результатів.

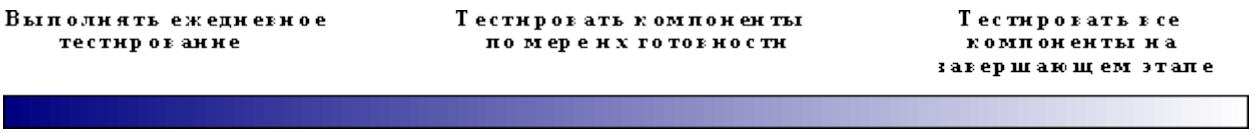


Рис. 1.4. Коли тестувати

Студентам пропонується тестувати компоненти по мірі їх готовності.

Як треба тестувати?

Основні підходи до тестування ПО засновані на **специфікації** і **реалізації** (рис. 1.5).

Специфікація модуля (або класу) ПП визначає, що цей модуль повинен робити, тобто вона описує допустимі набори вхідних даних, що подаються на вхід модуля, включаючи обмеження на те, як багаторазові вводи даних повинні співвідноситися один з одним, і які вихідні дані відповідають різним наборам вхідних даних.

Реалізація модуля ПП є вираз алгоритму, що породжує вихідні результати для різних наборів вхідних даних з дотриманням вимог специфікації. **Специфікація** вказує, що робить модуль ПП, а **реалізація** показує, **як** модуль ПП це робить. Повний облік вимог специфікації дає гарантію того, що ПП виконує все, що від нього вимагається. Повний облік

вимог до реалізації дає гарантію того, що ПП не робитиме того, що від нього не потрібно.

Специфікація відіграє важливу роль у тестуванні. Зазвичай для безлічі компонентів ПП створюються специфікації, що забезпечують розробку і тестування, включаючи специфікації систем, підсистем і класів.

Поряд з автономним тестуванням компонентів (класів) системи **(модульним рівнем тестування),** необхідно тестувати взаємодію між різними компонентами **(інтеграційний рівень тестування).** Мета **інтеграційного тестування** полягає у виявленні відмов, що виникають внаслідок помилок в інтерфейсах або в силу невірних припущень щодо інтерфейсів. Після *інтеграційного тестування*проводиться **системне тестування** ПП **(системний рівень).** На цьому рівні тестуванню піддається система як єдине ціле.

Тестування слід здійснювати в достатніх обсягах, щоб бути більш-менш впевненим у тому, що ПП функціонує відповідно до пред'явленими до нього вимогами, тобто виконується принцип **адекватності** тестування ПП. **Адекватність** можна виміряти, використовуючи поняття **покриття. Покриття** можна виміряти двома способами. Перший полягає в підрахунку кількості вимог, сформульованих у специфікації, які піддалися тестуванню. Другий спосіб полягає в підрахунку виконаних компонентів ПП в результаті прогону тестового набору. Набір тестів можна вважати адекватним, якщо певна частина рядків вихідного коду або виконуваних гілок вихідного коду була виконана, принаймні, один раз під час прогону тестового набору. Ці два способи вимірювання відображають два базові підходи до тестування:

- при використанні першого підходу перевіряється, що повинен виконувати ПП;
- при використанні другого підходу перевіряється, як фактично працює ПП.

При **тестуванні відповідно до специфікації (функціональному тестуванні або тестуванні "чорного ящика")** побудова тестових випадків проводиться у відповідності зі специфікацією і не залежить від того, як реалізований ПП. Ефективність залежить від якості специфікації і здібності тестувальника коректно її інтерпретувати.

При **структурному тестуванні (тестуванні відповідно до реалізації або тестуванні "білого ящика")** побудова тестових випадків проводиться на основі програмного коду, що представляє собою реалізацію ПП. Вхідні дані кожного тестового випадку мають бути визначені специфікацією ПП, однак вони можуть бути обрані на основі аналізу самого програмного коду для проходження тієї чи іншої гілки програми. При цьому покриття збільшується.

Рис. 1.5. Як тестувати

Ми будемо використовувати обидва підходи. При тестуванні класів ми будемо прагнути покрити як специфікації класів, так і код їх реалізації. При тестуванні взаємодій будемо покривати специфікацію. При *системному тестуванні* також будемо прагнути покрити специфікацію системи.

У якому обсязі тестувати?

Різні рівні адекватного тестування зображені на рис. 1.6 , який охоплює випадки від відсутності тестування до вичерпного тестування, коли виконується прогін всіх можливих тестових випадків. Обсяг необхідного тестування слід визначати виходячи з короткострокових і довгострокових цілей проекту і відповідно до особливостей розроблюваного ПП. **Покриття** - це міра повноти використання можливостей програмного компонента тестовим набором.

Наприклад, один із заходів - задіяна Чи кожен рядок програмного коду продукту хоча б один раз при прогоні даного тестового набору. Інша міра - кількість вимог

специфікації, перевірених даними тестовим набором. Якщо вимоги сформульовані в термінах**випадків використання,** то **покриття** вимірюється кількістю випадків використання і числом сценаріїв, побудованих для кожного випадку використання.

Аналіз ризиків в процесі тестування застосовується для визначення рівня деталізації і часу, що витрачається на тестування конкретного компонента. Наприклад, на тестування класів, важливіших для програми, відводиться більше часу.

Рис. 1.6. В якому обсязі тестувати

Ми будемо використовувати всі перераховані заходи.

Відповіді на поставлені питання і, можливо, на багато інших, оформляються у вигляді набору документів, прийнятого в компанії.Наприклад, **тестовий план** може містити наступну інформацію:

1. Перелік тестових ресурсів.
2. Перелік функцій і підсистем, що підлягають тестуванню.
3. *Тестову стратегію:*
    - Аналіз функцій і підсистем з метою визначення слабких місць, що вимагають вичерпного тестування, тобто ділянок функціональності, де поява дефектів найбільше ймовірно.
    - Визначення стратегії вибору вхідних даних для тестування. Оскільки в реальних застосуваннях безліч вхідних даних програмного продукту практично нескінченно, вибір кінцевого підмножини для проведення тестування є складним завданням. Для її рішення можуть бути застосовані методи покриття класів вхідних і вихідних даних,

аналіз крайніх значень, покриття випадків використання тощо. Обрана стратегія повинна бути обгрунтована і задокументована.

- Визначення потреби автоматизації процесу тестування. При цьому рішення про використання існуючої, або про створення нової автоматизованої системи тестування повинно бути обгрунтовано, а також продемонстрована оцінка витрат на створення нової системи або на впровадження вже існуючої.

4. Графік (розклад) тестових циклів.
5. Вказівка конкретних параметрів апаратури і програмного оточення.
6. Визначення тестових метрик, які необхідно збирати та аналізувати, таких як покриття набору вимог, покриття коду, кількість і рівень серйозності дефектів, обсяг тестового коду і т.п.

Модульне тестування на прикладі класів

Ключові слова: модуль , функція , група , тип даних , клас , ПО , тестировщик , програма , регресійне тестування , драйвер ,Вихід , статичний элемент , адекватность , приложение , t ester , значение , log-файл , тестовый звіт , crash

Мета тестування програмних модулів полягає в тому, щоб упевнитися, що кожен *модуль* відповідає своїй специфікації. Якщо це так, то причиною будь-яких помилок, які виникають при їх об'єднанні, є неправильна стиковка модулів. У процедурно-орієнтованому програмуванні модулем називається процедура або *функція,* іноді *група* процедур, яка реалізує абстрактний *тип даних.* Тестування модулів звичайно являє собою деяке сполучення перевірок і **прогонів тестових випадків.** Можна скласти план тестування модуля, в якому врахувати тестові випадки і побудова **тестового драйвера.**

Тестування класів аналогічно тестуванню модулів. Основним елементом об'єктно-орієнтованої програми є *клас.* Розглянемо методику тестування

окремого класу. Тестування класів охоплює види діяльності, асоційовані з перевіркою реалізації класу на точну відповідність специфікації класу. Якщо реалізація коректна, то кожен екземпляр цього класу поводиться належним чином.

Ефективного тестування класів можна досягти за допомогою **рев'ю** і **тестових прогонів. Рев'ю** являє собою перегляд вихідного коду *ПО* з метою виявлення помилок і дефектів, можливо, до того, як це *ПО* запрацює. **Ревьюірованіе** призначене для виявлення таких помилок, як нездатність виконувати ту чи іншу **вимогу специфікації** або її неправильне розуміння, а також алгоритмічних помилок в реалізації. **Тестовий прогін** забезпечує тестування *ПЗ* в процесі виконання програми. Здійснюючи прогін програми,*тестувальник* прагне визначити, чи здатна *програма* вести себе у відповідності зі специфікацією. *Тестувальник* повинен вибрати набори вхідних даних, визначити відповідні їм правильні набори вихідних даних і зіставити їх з реально одержуваними вихідними даними.

Розглянемо тестування класів у режимі **прогону тестових випадків.** Після ідентифікації **тестових випадків** для класу потрібно реалізувати **тестовий драйвер,** що забезпечує прогін кожного тестового випадку, і запротоколювати результати кожного прогону.При тестуванні класів **тестовий драйвер** створює один або більше число примірників тестованого класу і здійснює прогін тестових випадків. **Тестовий драйвер** може бути реалізований як автономний тестуючий *клас.*

Хто, що, коли, як і в якому обсязі?

Розглянемо ці питання в контексті тестування класів.

Хто виконує тестування? Зазвичай тестування класів виконують їх розробники. У цьому випадку час на вивчення специфікації і реалізації зводиться до мінімуму. Недоліком підходу є те, що якщо розробник неправильно зрозумів специфікації, то він для своєї неправильної реалізації розробить і "помилкові" тестові набори.

Що тестувати? Необхідно впевнитися, що програмний код класу в точності відповідає вимогам, сформульованим у його специфікації, і що він не робить нічого більше.

У який момент слід виконувати тестування? План тестування або хоча б тестові випадки повинні розроблятися після складання повної специфікації класу. Розробка тестових випадків *по* мірі реалізації класу допомагає розробнику краще зрозуміти специфікацію.Тестування класу повинно проводитися до того, як виникне необхідність використовувати цей *клас* в інших компонентах *ПЗ.Регресійне тестування* класу повинно виконуватися щоразу, коли змінюється реалізація класу. *Регресійне тестування* дозволяє переконатися в тому, що розроблені й відтестовані функції продовжують задовольняти специфікації після виконання модифікації *ПЗ.*

Як буде виконуватися тестування? Тестування класів зазвичай виконується шляхом розробки тестового драйвера, який створює екземпляри класів і оточує ці екземпляри відповідної середовищем (тестовим оточенням), щоб став можливий прогін відповідного тестового випадку. *Драйвер* посилає повідомлення екземпляру класу відповідно до специфікації тестового випадку, а потім перевіряє *результат* цих повідомлень. Тестовий *драйвер* повинен видаляти створені ним екземпляри тестованого класу. *Статичні елементи* даних класу також необхідно тестувати.

Які обсяги тестування слід вважати адекватними? *Адекватність* може бути виміряна повнотою охоплення тестами специфікації або реалізації. Будемо використовувати обидва способи.

Що тестувати?

Можна виділити два типи класів з точки зору їх взаємодії з іншими класами:

- примітивні класи;
- непрімітівние класи.

Примітивний клас може породжувати екземпляри, і ці екземпляри можна використовувати без необхідності створення екземплярів яких інших класів, у тому числі і даного класу. Такі об'єкти являють собою найпростіші компоненти системи і, безсумнівно, грають важливу роль при виконанні будь-якої програми. Тим не менш, в об'єктно-орієнтованої програми існує порівняно невелика кількість примітивних класів, які реалістично моделюють об'єкти завдання і всі відносини між цими об'єктами.Звичайним явищем для добре спроектованих об'єктно-орієнтованих програм є використання непрімітівних класів. Грунтуючись на цій інформації, визначимо, до якого типу належить кожен клас в нашому додатку (табл. 2.1).

Таблиця 2.1. Типи Класів

Клас	**Тип**
TBearingParam	Примітивний
TAxleParam	Примітивний
TCommand	Примітивний
TLog	Примітивний
TCommandQueue	Непрімітівний
TStore	Непрімітівний
TTerminalBearing	Непрімітівний
TTerminalAxle	Непрімітівний
TModel	Непрімітівний
MainForm	Непрімітівний

У більшості об'єктно-орієнтованих мов члени класу мають один з трьох рівнів доступу:

Public. Члени з доступом **public** доступні з будь-яких класів. Вони утворюють інтерфейс класу, яким користуватиметься будь-який розробник, що використовує даний клас в своєму додатку.

Private. Члени з доступом **private** доступні тільки всередині самого класу, тобто з його методів. Вони є частиною внутрішньої реалізації класу і недоступні сторонньому розробнику.

Protected. Члени з доступом **protected** доступні з самого класу і з класів, які є його нащадками, але недоступні ззовні.Використання цих методів можливо тільки при створенні класу-нащадка, що розширює функціональність базового класу.

Таким чином, необхідність тестування функціональності класу залежить від того, чи надається їм можливість наслідування. Якщо клас є закінченим **(final)** і не припускає спадкування, необхідно тестування його **public** частини (втім, класи **final** не містять**protected** членів). Якщо ж клас розрахований на розширення за рахунок успадкування, необхідно тестування також його**protected** частини.

Крім того, в багатьох мовах клас може містити статичні **(static)** члени, які належать класу в цілому, а не його конкретним екземплярам. При наявності **public static** або **protected static** членів, окрім тестування об'єктів класу, повинно окремо виконуватися тестування статичної частини класу.

Як тестувати?

Як уже згадувалося, для тестування класів застосовуються **тестові драйвери.** Існує кілька способів реалізації тестового драйвера:

Тестовий драйвер реалізується у вигляді окремого класу. Методи цього класу створюють об'єкти тестованого класу і викликають їхні методи, в тому числі статичні методи класу. Таким способом можна тестувати public частина класу.

Тестовий драйвер реалізується у вигляді класу, успадкованого від тестованого. На відміну від попереднього способу, такому тестового драйверу доступна не тільки public, а й protected частину.

Тестовий драйвер реалізується безпосередньо всередині тестованого класу (в клас додаються діагностичні методи). Такий тестовий драйвер має доступ до всієї реалізації класу, включаючи private члени. У цьому випадку в методи класу включаються виклики налагоджувальних функцій і агенти, які відстежують деякі події при тестуванні.

Надалі ми будемо використовувати перший спосіб при реалізації драйверів.

При розробці **специфікації** класу можна задіяти один з наступних підходів:

Контрактний підхід. Інтерфейс визначається у вигляді зобов'язань відправника і одержувача, що вступили у взаємодію. Операція визначається як набір зобов'язань кожної сторони, причому відповідальність по відношенню один до одного дотримується як відправником, так і одержувачем.

Підхід захисного програмування. Інтерфейс визначається головним чином в термінах одержувача. Операція повертає результат запиту - успішне або невдале виконання по конкретній причини (наприклад, по неприпустимого вхідному значенню). Іншими словами, відповідний одержувач стежить за тим, щоб на вхід не влучили некоректні дані, тобто перевіряє правильність і допустимість вхідних даних, і після отримання запиту повідомляє відправнику результат обробки запиту.

Різниця між **контрактним** і **захисним методами проектування** поширюється і на тестування. **Контрактне проектування**покладає велику відповідальність на проектувальника, ніж на програми пошуку помилок. Основна увага під час тестування взаємодій в умовах контактного підходу приділяється перевірці того, чи виконані об'єктом-відправником предусловия методів одержує об'єкта. Не допускається побудова тестових випадків, що порушують ці предусловия. Зазвичай практикується переведення об'єкта-одержувача в деякий

заданий стан, після чого ініціюється виконання тестового драйвера, за умовами якого об'єкт-відправник вимагає, щоб об'єкт-одержувач перебував в іншому стані. Сенс подібної перевірки полягає в тому, щоб встановити, чи виконує об'єкт-відправник перевірку передумов об'єкта-одержувача, перш ніж відправити заздалегідь неприйнятну повідомлення, і чи коректно він припиняє свою роботу.

Детальний опис тестового випадку

Розглядається приклад тестів на C # для класу TCommand (додаток 3 (HLD)). При виконанні завдань необхідно буде самостійно написати тести для інших класів додатки. Паралельно з вивченням цього розділу корисно відкрити проект ModuleTesting \ ModuleTests.sln.

Розглянемо тестування класу TCommand. Цей *клас* реалізує єдину операцію GetFullName (), яка повертає повна назва команди у вигляді рядка. Розробимо **специфікацію тестового випадку** для тестування методу GetFullName на основі специфікації цього класу (*додаток* 3):

Назва класу: TCommand Назва тестового випадку: TCommandTest1

Опис тестового випадку: Тест перевіряє правильність роботи методу GetFullName - отримання повної назви команди на основі коду команди. У тесті подаються наступні значення кодів команд (вхідні значення): -1, 1, 2, 4, 6, 20, де -1 - заборонене значення

Початкові умови: Ні

Очікуваний результат:

Зазначеним вхідним значенням повинні відповідати наступні вихідні:

Коду команди -1 має відповідати повідомлення "ПОМИЛКА: Невірний код команди"

Коду команди 1 має відповідати повна назва команди "ОТРИМАТИ ІЗ ВХІДНИЙ ОСЕРЕДКУ"

Коду команди 2 має відповідати повна назва команди "ВІДПРАВИТИ ІЗ ОСЕРЕДКУ У вихідні осередку"

Коду команди 4 має відповідати повна назва команди "ПОКЛАСТИ У РЕЗЕРВ"

Коду команди 6 має відповідати повна назва команди "СПРАВИТИ занулення"

Коду команди 20 повинно відповідати повна назва команди "ЗАВЕРШЕННЯ КОМАНД ВИДАЧІ"

На основі специфікації був створений тестовий *драйвер* - *клас* TCommandTester, що успадковує функціональність абстрактного класу *Tester*.

```
public class Log
{
        static private StreamWriter log = new
          StreamWriter ("log.log");  / / Створення лог файлу
        static public void Add (string msg)
          / / Додавання повідомлення в лог файл
        {
          log.WriteLine (msg);
        }
        static public void Close () / / Закрити лог файл
        {
          log.Close ();
        }
```

```
}
abstract class Tester
{
protected void LogMessage (string s)
/ / Додавання повідомлення в лог-файл
        {
          Log.Add (s);
        }
}
class TCommandTester: Tester / / Тестовий драйвер
{
        TCommand OUT;
        public TCommandTester ()
        {
          OUT = new TCommand ();
          Run ();
        }
        private void Run ()
        {
          TCommandTest1 ();
        }
        private void TCommandTest1 ()
        {
```

```
    int [] commands = {-1, 1, 2, 4, 6, 20};
    for (int i = 0; i <= 5; i + +)
        {
          OUT.NameCommand = commands [i];
          LogMessage (commands [i]. ToString () + ":
           "+ OUT.GetFullName ());
        }
  }
  [STAThread]
  static void Main ()
  {
    TCommandTester CommandTester = new TCommandTester ();
    Log.Close ();
  }
}
```

Лістинг 2.1. Тестовий драйвер

Клас TCommandTester містить метод TCommandTest1 (), в якому реалізована вся функціональність тесту. У даному випадку для покриття специфікації досить перебрати наступні значення кодів команд: -1, 1, 2, 4, 6, 20, де -1 - заборонене *значення,* і отримати відповідні їм повна назва команди за допомогою методу GetFullName (). Пари відповідних значень заносяться в *log-файл* для подальшої перевірки на відповідність специфікації.

Таким чином, для тестування будь-якого методу класу необхідно:

- Визначити, яка частина функціональності методу повинна бути протестована, тобто за яких умов він повинен викликатися. Під умовами тут розуміються параметри виклику методів, значення полів і властивостей об'єктів, наявність і вміст використовуваних файлів і т. д.
- Створити тестове оточення, що забезпечує необхідні умови.
- Запустити тестове оточення на виконання.
- Забезпечити збереження результатів у файл для їх подальшої перевірки.
- Після завершення виконання порівняти отримані результати зі специфікацією.

Опис тестових процедур

Як запустити тест?

Для того щоб запустити тест, потрібно:

- У методі `Run` тестового драйвера `TCommandTester` викликати метод `TCommandTest1`, який реалізує тест.
- Зібрати і запустити програму.

Перевірка результатів виконання тестів (порівняння з очікуваним результатом)

Після завершення тесту слід переглянути текстовий журнал тесту `(.. \ ModuleTesting \ bin \ Debug \ log.log)`, щоб порівняти отримані результати з очікуваними результатами, заданими в специфікації тестового випадку `TCommandTest1`. Журнал тіста:

```
 -1: ПОМИЛКА: Невірний код команди
1: ОТРИМАТИ ІЗ ВХІДНИЙ ОСЕРЕДКУ
2: НАДІСЛАТИ ІЗ ОСЕРЕДКУ У вихідні комірки
4: ПОКЛАСТИ У РЕЗЕРВ
6: СПРАВИТИ занулення
20: ЗАВЕРШЕННЯ КОМАНД ВИДАЧІ
```

Завдання 1

Для інших примітивних класів (табл. 2.1) згідно з наведеним прикладом необхідно самостійно розробити специфікації тестових випадків, відповідні тести і провести тестування. Звіт потрібно скласти в наступній формі (табл. 2.2):

Таблиця 2.2. *Тестовий звіт*

Назва тестового випадку:

Тестувальник:

Тест пройдено: Так / Ні (PASS / FAIL)

Ступінь важливості помилки:

Фатальна (3 рівень - *crash)*

Серйозна (2 рівень - розбіжність в специфікації)

Незначна (1 рівень - незначна помилка)
Опис проблеми:

Як відтворити помилку:

Пропоноване виправлення (необов'язково):

Коментар тестувальника (необов'язково):

Інтеграційне тестування

Ключові слова: обмін повідомленнями , інтеграційне тестування , запит , модульне тестирование , операции , класс , параметр ,объект , систематический , подмножество , множества , конструктор , драйвер , tester , log-файл , команда

Основне призначення тестування взаємодій полягає в тому, щоб переконатися, що відбувається правильний *обмін повідомленнями*між об'єктами, класи яких вже пройшли тестування в автономному режимі (на модульному рівні тестування).

Тестування взаємодії або *інтеграційне тестування* являє собою тестування зібраних разом, взаємодіючих модулів (об'єктів). У інтеграційному тестуванні можна об'єднувати різну кількість об'єктів - від двох до всіх об'єктів тестованої системи. *Інтеграційне тестування* відрізняється від системного тим, що:

- при *інтеграційному тестуванні* використовується підхід "білого ящика", а при системному - "чорної скриньки";
- метою інтеграційного тестування є тільки перевірка правильності взаємодії об'єктів, тоді як метою системного - перевірка правильності функціонування системи в цілому.

Ідентифікація взаємодій

Взаємодія об'єктів являє собою просто *запит* одного об'єкта **(відправника)** на виконання іншим об'єктом **(одержувачем)** однієї з операцій одержувача і всіх видів обробки, необхідних для завершення цього запиту.

У ситуаціях, коли в якості основи тестування взаємодій об'єктів обрані тільки специфікації загальнодоступних операцій, тестування набагато простіше, ніж коли такою основою служить реалізація. Ми обмежимося тестуванням загальнодоступного інтерфейсу.Такий підхід цілком виправданий, оскільки ми вважаємо, що класи вже успішно пройшли *модульне тестування.* Проте, вибір такого підходу аж ніяк не означає, що не потрібно повертатися до специфікаціям класів, щоб переконатися в тому, що той чи інший метод виконав усі необхідні обчислення. Це обумовлює необхідність перевірки значень атрибутів внутрішнього стану одержувача, в тому числі будь-яких агрегованих атрибутів, тобто атрибутів, які самі є об'єктами. Основна увага приділяється відбору тестів на основі специфікації кожної *операції* з загальнодоступного інтерфейсу класу.

Взаємодії неявно передбачаються в специфікації класу, в якій встановлені посилання на інші об'єкти. У розділі 4 розглядалося тестування примітивних класів. Такі об'єкти являють собою найпростіші компоненти системи і, безсумнівно, грають важливу роль при виконанні будь-якої програми. Тим не менш, в об'єктно-орієнтованої програми існує порівняно невелика кількість примітивних класів, які реалістично моделюють об'єкти завдання і всі відносини між цими об'єктами. Звичайним явищем для добре спроектованих об'єктно-орієнтованих програм є використання непрімітівних класів; в цих програмах їм відводиться головна роль.

Виявити такі взаємодіючі класи можна, використовуючи відносини асоціації (у тому числі відносини агрегування і композиції), представлені на діаграмі класів. Асоціації такого роду перетворюються в інтерфейси класу, а той чи інший *клас* взаємодіє з іншими класами допомогою одного або декількох способів:

Тип 1. Загальнодоступна операція має один або більше число формальних параметрів об'єктного типу. Повідомлення встановлює асоціацію між одержувачем і параметром, яка дозволяє одержувачеві взаємодіяти з цим параметричним об'єктом.

Тип 2. Загальнодоступна операція повертає значення об'єктного типу. На *клас* може бути покладено завдання створення об'єкта, що повертається, або він може повертати модифікований *параметр.*

Тип 3. Метод одного класу створює екземпляр іншого класу як частину своєї реалізації.

Тип 4. Метод одного класу посилається на глобальний екземпляр деякого іншого класу. Зрозуміло, принципи хорошого тону в проектуванні рекомендують мінімальне використання глобальних об'єктів. Якщо реалізація якого класу посилається на деякий глобальний *об'єкт,* розглядайте його як неявний *параметр* в методах, які на нього посилаються.

Наведемо ще раз таблицю поділу класів на примітивні і непрімітівние типи (табл. 3.1):

Таблиця 3.1. Типи класів

Клас	**Тип**
TBearingParam	Примітивний
TAxleParam	Примітивний
TCommand	Примітивний
TLog	Примітивний
TCommandQueue	Непрімітівний
Клас	Непрімітівний

TStore	Непрімітівний
TTerminalBearing	Непрімітівний
TTerminalAxle	Непрімітівний
TModel	Непрімітівний
MainForm	Непрімітівний

Таблиця 3.2. Типи взаємодії класів

Непрімітивные типи	**Tbearing Param**	**Taxle Param**	**Tcommand**	**TCommand Queue**	**TStore**	**TTerminal Bearing**	**TTerminal Axle**	**TLog**	**Tmodel**
TCommand Queue	1	1	3		1	1			
TStore	3	1	1						
TTerminalBearing	3								
TTerminalAxle		3							
TModel				3	3	3	3	3	
MainForm									3

Таким чином, *інтеграційному тестуванню* будуть піддані взаємодії перерахованих непрімітівних класів.

Вибір тестових випадків

Повне тестування, іншими словами, прогін кожного можливого тестового випадку, покриває кожне поєднання значень - це, поза всяких сумнівів, надійний підхід до тестування. Однак у багатьох ситуаціях кількість тестових випадків досягає таких великих значень, що звичайними методами з ними впоратися попросту неможливо. Якщо є принципова можливість побудови такої великої кількості тестових випадків, на побудову і виконання яких не вистачить ніякого часу, повинен бути розроблений *систематичний*метод визначення, якими з тестових випадків слід скористатися. Якщо є вибір, то ми віддаємо перевагу таким тестовим випадкам, які дозволяють знайти помилки, у виявленні яких ми зацікавлені найбільше.

Існують різні способи визначення, яке *підмножина* з *безлічі* всіх можливих тестових випадків слід вибирати. При будь-якому підході ми зацікавлені в тому, щоб систематично підвищувати рівень покриття.

Детальний опис тестового випадку

Продемонструємо тестування взаємодій на прикладі класу TCommandQueue. З табл. 3.2 , яка була складена на основі специфікацій класів, описаних у додатку 2, видно, що *клас* черги команд взаємодіє з наступними класами:

TBearingParam,

TAxleParam,

TCommand,

TStore,

TterminalBearing.

З об'єктом TCommand здійснюється взаємодія третього типу, тобто TCommandQueue створює об'єкти класу TCommand як частину своєї внутрішньої реалізації. З іншими класами здійснюється взаємодія першого типу: посилання на об'єкти класів TStore іTTerminalBearing передаються як параметри

в *конструктор* TCommandQueue, а посилання на об'єкти класів TBearingParam іTAxleParam передаються в метод TCommandQueue.AddCommand.

Одночасно з вивченням цього розділу можна відкрити проект IntegrationTesting \ IntegrationTests.sln.

Для тестування взаємодії класу TCommandQueue і класу TCommand, так само, як і при модульному тестіруванii, розробимо специфікацію тестового випадку:

Таблиця 3.3. Специфікація тестового випадку

Назви взаємодіючихНазва тесту: TCommandQueueTest1 класів: TCommandQueue, TCommand

Опис тесту: тест перевіряє можливість створення об'єкта типу TCommand і додавання його в чергу при виклику методуAddCommand

Початкові умови: черга команд порожня

Очікуваний результат: в чергу буде додана одна команда

На основі цієї специфікації був розроблений тестовий *драйвер* - *клас* TCommandQueueTester, який успадковується від класу*Tester*. Цей *клас* містить:

- Метод Init, в якому створюються об'єкти класів TStore, TterminalBearing і об'єкт типу TcommandQueue. Цей метод необхідно викликати на початку кожного тесту, щоб тестовані об'єкти створювалися знову:
- private void Init ()
- {
- TB = new TTerminalBearing ();
- S = new TStore ();
- CommandQueue = new TCommandQueue (S, TB);

- S.CommandQueue = CommandQueue;

}

Приклад 3.1. Метод Init

- Методи, що реалізують тести. Кожен тест реалізований в окремому методі.
- Метод Run, у якому викликаються методи тестів.
- Метод dump, який зберігає в *log-файлі* тесту інформацію про всі команди, що знаходяться в черзі в форматі - номер позиції в черзі: повна назва команди.
- Точку входу в програму - метод Main, в якому відбувається створення екземпляра класу TCommandQueueTester і запуск методу Run.

Спочатку створимо тест, який перевіряє, чи створюється *об'єкт* типу TCommand, і додається чи *команда* в кінець черги.

```
private void TCommandQueueTest1 ()
{
Init ();
LogMessage ("/ / / / / / / / / TCommandQueue Test1 / / / / / / / / / / / / /");
LogMessage ("Перевіряємо, чи створюється об'єкт типу TCommand");
/ / В черзі немає команд
dump ();
/ / Додаємо команду
/ / Параметр = -1 означає, що команда повинна бути додана
/ / В кінець черги
CommandQueue.AddCommand (TCommand.GetR, 0,0,0, new
TBearingParam (), new TAxleParam (), -1);
```

```
LogMessage ("Command added");
/ / В черзі одна команда
dump ();
}
```

Приклад 3.2. Тест, перевіряючий створення об'єкта типу TCommand

У цей *клас* включені ще два розроблених тесту.

Опис тестових процедур

Як запустити тест?

Для виконання цього тесту в методі Run необхідно викликати метод TCommandQueueTest1 () і запустити програму на виконання:

```
private void Run ()
{
  TCommandQueueTest1 ();
}
```

Приклад 3.3. Метод Run

Перевірка результатів виконання тестів (порівняння з очікуваним результатом)

Після завершення тесту слід переглянути текстовий журнал тесту (.. \ IntegrationTesting \ bin \ Debug \ test.log),щоб порівняти отримані результати з очікуваними результатами, заданими в специфікації тестового випадку TCommandQueueTest1.

```
/ / / / / / / / / / / / / / / / / / / TCommandQueue Test1 / / / / / / / / / / / / / / / / /
Перевіряємо, чи створюється об'єкт типу TCommand
0 commands in command queue
```

Command added

1 commands in command queue

0: ОТРИМАТИ ІЗ ВХІДНИЙ ОСЕРЕДКУ

Приклад 3.4. Журнал тесту

Завдання 2

Для тестування взаємодії інших непрімітівних класів (табл. 2.1) за аналогією з наведеним прикладом вимагається самостійно розробити специфікації тестових випадків, відповідні тести, провести тестування і скласти тестові звіти (табл. 2.2).

Системне тестування

Ключові слова: системне тестування , інтеграційне тестування , завдання системного тестування , Автоматизації тестування ,уявлення , вхідні дані , команда , вихідні дані , виклик функції , чергу , місце , мову програмування

Системне тестування якісно відрізняється від інтеграційного і модульного рівнів. *Системне тестування* розглядає систему в цілому і застосовується на рівні користувача інтерфейсів, на відміну від останніх фаз *інтеграційного тестування,* яке оперує на рівні інтерфейсів модулів, хоча набір модулів може бути аналогічним. Різні і цілі цих рівнів тестування. На рівні системи часто складно і малоефективно аналізувати проходження тестових траєкторій усередині програми, а також відстежувати правильність роботи конкретних функцій. Основний *завданням системного тестування* є виявлення дефектів, пов'язаних з роботою системи в цілому, таких як невірне використання ресурсів системи, непередбачені комбінації даних користувача рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність у використанні тощо.

Оскільки *системне тестування* проводиться на рівні користувача інтерфейсів, то побудова спеціальної тестової системи стає технічно необов'язковим. Однак

обсяги даних на цьому рівні такі, що зазвичай більш ефективним підходом є повна або частковаавтоматизація тестування, що може зажадати створення тестової системи набагато більш складною, ніж система тестування на рівні модулів або їх комбінацій.

Необхідно підкреслити, що існує два принципово різних підходи до системного тестування.

У першому варіанті для побудови тестів використовуються вимоги до системи, наприклад, для кожної вимоги будується тест, який перевіряє виконання даної вимоги в системі. Цей підхід особливо широко застосовується при розробці військових і наукових систем, коли замовник цілком усвідомлює, яка функціональність йому потрібна, і становить повний набір **формальних вимог.**Тестировщик в даному випадку тільки перевіряє, чи відповідає розроблена система цього набору. Такий підхід передбачає довгу і дорогу фазу збору вимог, виконувану до початку власне проекту. У цьому випадку для визначення вимог звичайно розробляється прототип майбутньої системи.

У другому підході основою для побудови тестів служить *уявлення* про способи використання продукту і про завдання, які він вирішує. На основі більш-менш формальної моделі користувача створюються **випадки використання** системи, по яких потім будуються власне **тестові випадки. Випадок використання (use case)** описує, як суб'єкт використовує систему, щоб виконати ту чи іншу задачу. **Суб'єкти** або **актори (actors)** можуть виконувати різні ролі при роботі з системою. **Випадки використання**можуть описуватися з різним ступенем абстракції. **Випадки використання** не обов'язково охоплюють кожну вимогу. Можна конкретизувати **випадки використання** і розширювати їх в набори більш **специфічних випадків використання (покроковий опис випадку використання).** У контексті конкретного **випадку використання** можна визначити один або більше число**сценаріїв. Сценарій** представляє конкретний екземпляр **випадку використання - шлях в**

покроковому описі випадку використання. Кожен **шлях (сценарій)** у **разі використання** повинен бути протестований (рис. 4.1).

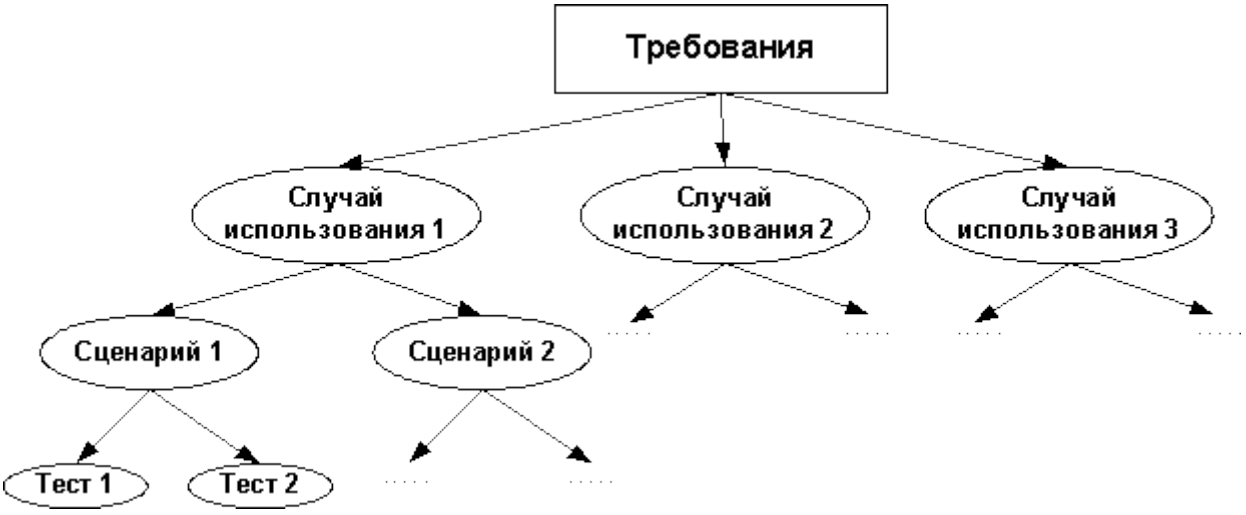


Рис. 4.1. Тестування випадків використання

Вхідні дані для кожного **сценарію** треба вибирати таким чином:

- Ідентифікувати всі значення (вхідні дані), які можуть задавати суб'єкти для **випадку використання.**
- Визначити **класи еквівалентності** для кожного типу вхідних даних.
- Побудувати таблицю зі списком значень з різних класів еквівалентності.
- Побудувати **тестові випадки** на базі таблиці з урахуванням зовнішніх обмежень.

Далі при побудові **тестових випадків** застосовувалися обидва підходи і при виконанні завдань необхідно діяти наступним чином:

- На основі вимог визначити випадки використання (use case)
- На основі кожного випадку використання (use case) побудувати сценарії.
- Для кожного сценарію розробити тестові випадки (набір тестів).

Випадки використання (use cases)

Опис випадку використання (use case) "підбір підшипників для осі"

Послідовно приходять два підшипника, надходить запит від осі. Під час отримання запиту від осі система підбирає два підшипника з наявних на складі і видає їх у вихідну комірку.

Розглянемо цей **випадок використання** докладніше. Згідно специфікації, система постійно опитує склад і термінал осі. При надходженні підшипника (статус складу 32) система опитує термінал підшипника, формує і посилає команду складу "прийняти підшипник" і отримує відповідь від складу про результати виконання команди. При надходженні осі (надходженні параметрів осі при опитуванні терміналу осі) система повинна підібрати підшипники з наявних на складі, сформувати команди для їх видачі, послати їх складу і отримати відповідь про результат виконання команд.

Далі наводиться покроковий опис цього випадку використання:

Прийняли на склад першого підшипник (1-10)

Прийняли на склад другого підшипник (11-20)

Надійшла вісь (21-26)

Підбираємо перший підшипник для осі (27-30)

Підбираємо другий підшипник для осі (31-34)

Завершення видачі команд (35-39).

Покроковий опис випадку використання

Покроковий опис наведено на рис. 4.2 .

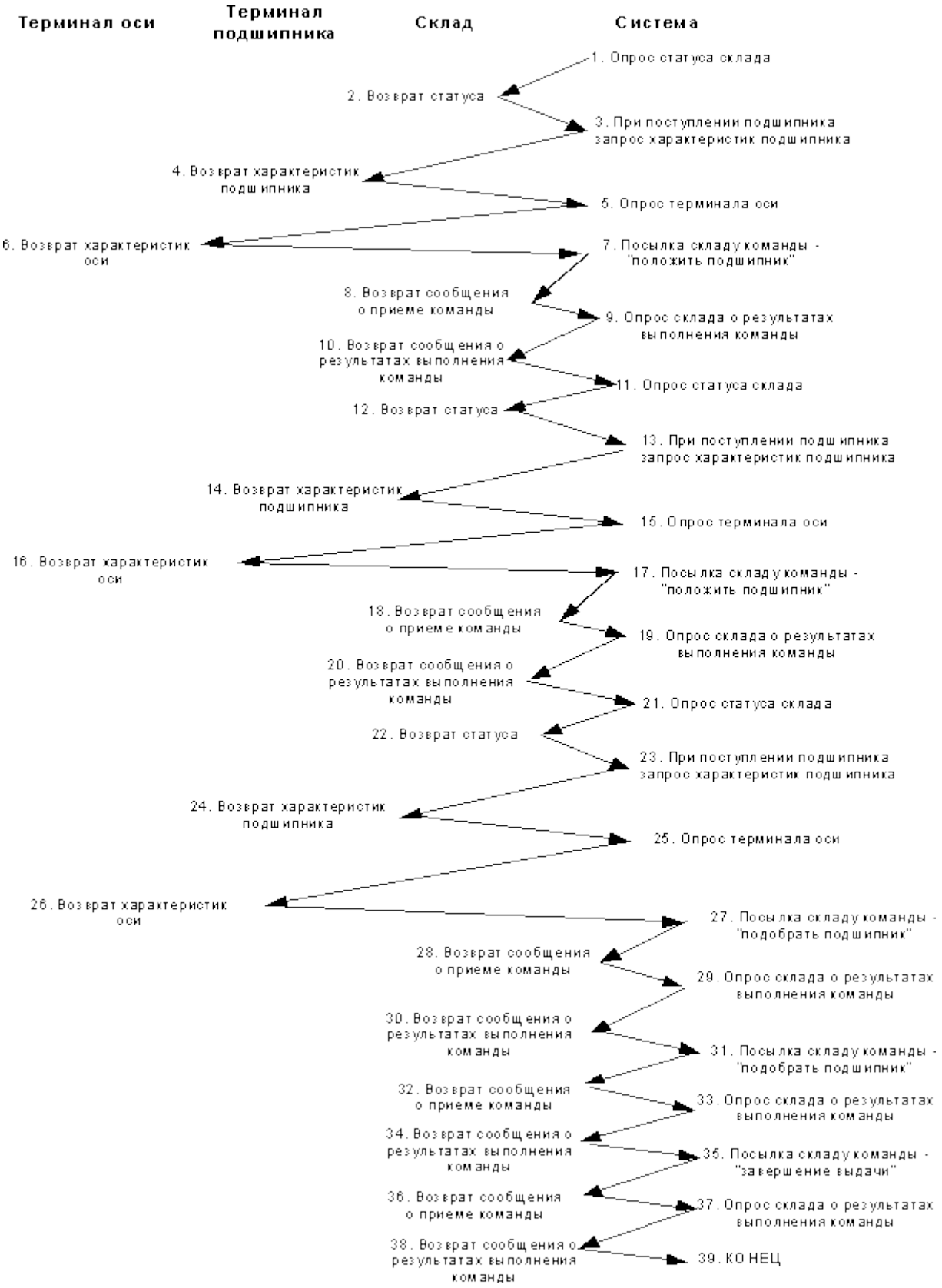


Рис. 4.2. Приклад use case

Список альтернативних шляхів

У покроковому описі випадку використання розглядався оптимістичний сценарій, наприклад при аналізі статусу складу в пунктах 3 і 11 вважалося, що надійшов підшипник. Але необхідно розглянути і можливі альтернативні сценарії:

3, 11 - аналіз статусу складу:

3.a, 11.a - підшипник в маніпуляторі.

3.b, 11.b - склад вільний.

3.c, 11.c - помилкове стан.

При отриманні повідомлення від складу про виконання команди вважалося, що команда виконана без помилки, але тут теж існують альтернативні варіанти:

8, 16, 22, 24, 26 - отримання повідомлення про виконання команди:

8.a, 16.a, 22.a, 24.a, 26.a - ні складу.

8.b, 16.b, 22.b, 24.b, 26.b - ні повідомлення.

8.C, 16.c, 22.c, 24.c, 26.c - команда виконана з помилкою.

Список альтернативних варіантів для розглянутого випадку використання можна продовжити.

Специфікація тестового випадку № 1

Стан оточення (*вхідні дані):*

Статус складу (StoreStat = 32). Прийшов підшипник.

Статус обміну з терміналом підшипника (0 - є підшипник) і його параметри (RollerPar = "0 NewUser Depot1 123456 1 грудня 1 +1").

Статус обміну з терміналом осі (1 - немає осі) і її параметри (AxlePar = "1 NewUser Depot1 123456 1 0 12 12").

Статус команди (CommandStatus = 0). *Команда* успішно прийнята.

Повідомлення від складу (StoreMessage = 1). *Команда* успішно виконана.

Очікувана послідовність подій (*вихідні дані):*

Система запрошувати статус складу (*виклик функції* GetStoreStat) і отримує 32.

Система запитує параметри підшипника (*виклик функції* GetRollerPar) і отримує 0 NewUser Depot1 123456 12 Січень 1 січня.

Система запитує параметри осі (*виклик функції* GetAxlePar) і отримує 1 NewUser Depot1 123456 1 0 12 грудня.

Система додає в *чергу* команд складу на останнє *місце* команду SendR (отримати з приймача в клітинку) (*виклик функції*SendStoreCom) і отримує повідомлення про те, що *команда* успішно прийнята - 0.

Система запрошувати склад про результати виконання команди (*виклик функції* GetStoreMessage) і отримує повідомлення про те, що *команда* успішно виконана - 1.

Система запрошувати статус складу (*виклик функції* GetStoreStat) і отримує 32.

Система запитує параметри підшипника (*виклик функції* GetRollerPar) і отримує 0 NewUser Depot1 123456 12 Січень 1 січня.

Система запитує параметри осі (*виклик функції* GetAxlePar) і отримує 1 NewUser Depot1 123456 1 0 12 грудня.

Система додає в *чергу* команд складу на перший *місце* команду GetR (отримати з приймача в клітинку) (*виклик функції*SendStoreCom) і отримує повідомлення про те, що *команда* успішно прийнята - 0.

Система запрошувати склад про результати виконання команди (*виклик функції* GetStoreMessage) і отримує повідомлення про те, що *команда* успішно виконана - 1.

Змінюємо стан оточення (*вхідні дані):*

Статус обміну з терміналом підшипника (1 - немає підшипника) і його параметри (RollerPar = "1 NewUser Depot1 123456 1 грудня 1 +1").

Статус обміну з терміналом осі (0 - є вісь) і її параметри (AxlePar = "0 NewUser Depot1 123456 1 0 12 12").

Очікувана послідовність подій (*вихідні дані):*

Система запрошувати статус складу (*виклик функції* GetStoreStat) і отримує 32.

Система запитує параметри підшипника (*виклик функції* GetRollerPar) і отримує 1 NewUser Depot1 123456 12 Січень 1 січня.

Система запитує параметри осі (*виклик функції* GetAxlePar) і отримує 0 NewUser Depot1 123456 1 0 12 грудня.

Система додає в *чергу* команд складу на останнє *місце* команду SendR (*виклик функції* SendStoreCom) і отримує повідомлення про те, що *команда* успішно прийнята - 0.

Система запрошувати склад про результати виконання команди (*виклик функції* GetStoreMessage) і отримує повідомлення про те, що *команда* успішно виконана - 1.

Система додає в *чергу* команд складу на останнє *місце* команду SendR (*виклик функції* SendStoreCom) і отримує повідомлення про те, що *команда* успішно прийнята - 0.

Система запрошувати склад про результати виконання команди (*виклик функції* GetStoreMessage) і отримує повідомлення про те, що *команда* успішно виконана - 1.

Система додає в *чергу* команд складу на останнє *місце* команду Term (*виклик функції* SendStoreCom) і отримує повідомлення про те, що *команда* успішно прийнята - 0.

Система запрошувати склад про результати виконання команди (*виклик функції* GetStoreMessage) і отримує повідомлення про те, що *команда* успішно виконана - 1.

У всіх наступних розділах буде детально розглядатися саме цей **тестовий випадок!**

Опис процесу системного тестування

Розглянемо процес системного тестування:

Аналіз. Тестируемая система аналізується (перевіряється) на наявність певних властивостей, яким треба приділити особливу увагу, і визначаються відповідні **тестові випадки.**

Побудова. Вибрані на стадії аналізу **тестові випадки** переводяться на *мова програмування.*

Виконання та аналіз результатів. Проводиться виконання **тестових випадків.** Отримані результати аналізуються, щоб визначити, чи успішно пройшла система випробування на **тестовому наборі.**

Процес запуску тестових випадків і аналізу отриманих результатів має бути докладно описаний у **тестових процедурах.**

Далі ми розглянемо три різних підходи, які використовуються при системному тестуванні:

- Ручне тестування.
- Автоматизація виконання та перевірки результатів тестування за допомогою скриптів.
- Автоматична генерація тестів на основі формального опису.

Ручне тестування

Ключові слова: производительность , запуск , тестировщик , сервер , Ручное те стування , тестова процедура , список

Найбільш поширеним способом розробки тестів є створення тестового коду вручну. Такий спосіб створення тестів є найбільш гнучким, проте *продуктивність* тестувальників при створенні тестового коду порівняння

з продуктивністю розробників при створенні коду продукту, а обсяги тестового коду часто бувають в 1-10 разів більше обсягу самого продукту.

У цьому випадку *запуск* тестів здійснюється вручну. Перевірку, чи пройшла тестована система випробування на заданому**тестовому випадку,** *тестувальник* також здійснює вручну, порівнюючи фактичні результати журналу тесту с **очікуваними результатами,** описаними в специфікації **тестового випадку.**

Функції dll-бібліотеки забезпечують звернення до сервера для отримання інформації про стан елементів комплексу і повертають серверу інформацію про функціонування системи. Значить, для моделювання стану оточення **(вхідних даних)** необхідно створити спеціальний *сервер.*

Крім того, необхідно зберігати одержувану від сервера інформацію про функціонування системи **(вихідні дані)** у журналі (рис. 5.1).

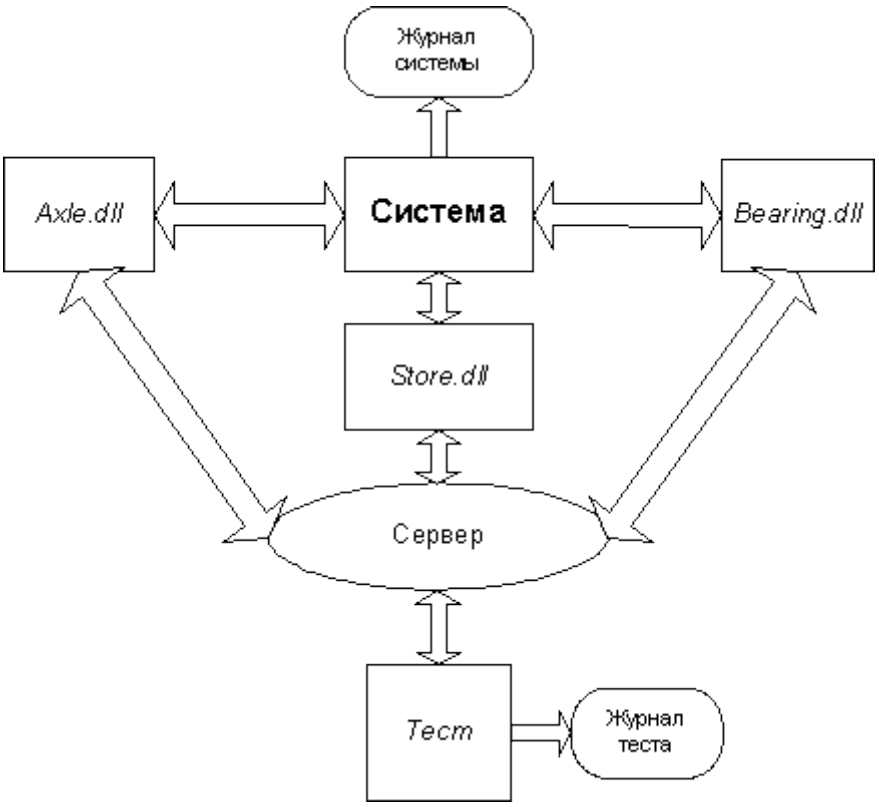


Рис. 5.1. Система і її оточення (ручне тестування)

При розробці тестів був використаний наступний підхід:

- стан оточення задається в тесті (вхідні дані);
- в тесті створюється сервер:
    - сервер за запитом від dll передає інформацію про заданому стані оточення;
    - сервер отримує від dll інформацію про функціонування системи;
- отримувана інформація зберігається в журналі тесту.

Детальний опис тестового випадку № 1

Ознайомлення з цим пунктом корисно випередити вивченням п. 7, що містить опис *ручного тестування.* Тут розглядається та частина тесту на C #, яку вам доведеться написати самостійно при виконанні завдань. Наведений приклад був розроблений у відповідності зі специфікацією тестового випадку N1. Для простоти будемо вважати, що події відбуваються послідовно у суворо заданому порядку. Реально наша система являє собою багатопотокове застосування, тому ми не можемо це гарантувати.

```
class Test1: Test {
override public void start ()
{/ / Задаємо стан оточення (вхідні дані)
StoreStat = "32";  / / Поступив підшипник
RollerPar = "0 NewUser Depot1 123456 12 січня 1 +1";
/ / Статус обміну з терміналом підшипника (0 - є підшипник)
/ / І його параметри
AxlePar = "1 NewUser Depot1 123456 1 0 12 12";
/ / Статус обміну з терміналом осі (1 - немає осі) і її параметри
CommandStatus = "0";  / / Команда успішно прийнята
StoreMessage = "1";  / / Успішно виконана
```

/ / Отримуємо інформацію про функціонування системи

wait ("GetStoreStat"); / / Опитування статусу складу

wait ("GetRollerPar");

/ / Отримання інформації про підшипнику з терміналу підшипника

wait ("GetAxlePar");

/ / Отримання інформації про вісь з терміналу осі

wait ("SendStoreCom");

/ / Додавання в чергу команд складу на перше місце

/ / Команди GetR (отримати з приймача в клітинку)

wait ("GetStoreMessage");

/ / Отримання повідомлення від складу про результати виконання команди

/ / В результаті перший підшипник повинен бути прийнятий

wait ("GetStoreStat"); / / Опитування статусу складу

wait ("GetRollerPar");

/ / Отримання інформації про підшипнику з терміналу підшипника

wait ("GetAxlePar");

/ / Отримання інформації про вісь з терміналу осі

wait ("SendStoreCom");

/ / Додавання в чергу команд складу на перше місце

/ / Команди GetR (отримати з приймача в клітинку)

wait ("GetStoreMessage");

/ / Отримання повідомлення від складу про результати виконання

/ / Команди. У результаті другої підшипник повинен бути прийнятий

/ / Задаємо новий стан оточення (вхідні дані)

RollerPar = "1 NewUser Depot1 123456 12 січня 1 +1";

/ / Статус обміну з терміналом підшипника (1 - немає підшипника)

/ / І його параметри

AxlePar = "0 NewUser Depot1 123456 1 0 12 12";

/ / Статус обміну з терміналом осі (0 - є вісь) і її параметри

/ / Отримуємо інформацію про функціонування системи

wait ("GetStoreStat"); / / Опитування статусу складу

wait ("GetRollerPar");

/ / Отримання інформації про підшипнику з терміналу підшипника

wait ("GetAxlePar");

/ / Отримання інформації про вісь з терміналу осі

wait ("SendStoreCom") ;/ / Додавання в чергу команд складу на

останнє місце / / команди SendR (комірку на вихід)

wait ("GetStoreMessage");

/ / Отримання повідомлення від складу про результати виконання

/ / Команди

/ / В результаті перший підшипник для осі повинен бути виданий

wait ("SendStoreCom");

/ / Додавання в чергу команд складу на останнє місце

/ / Команди SendR (комірку на вихід)

wait ("GetStoreMessage");

/ / Отримання повідомлення від складу про результати виконання

```
/ / Команди.
/ / В результаті другого підшипник для осі повинен бути виданий
wait ("SendStoreCom");
/ / Додавання в чергу команд складу на останнє місце
/ / Команди Term (завершення команд видачі)
wait ("GetStoreMessage");
/ / Отримання повідомлення від складу про результати виконання
/ / Команди
finish ();
}
}
```

Приклад 5.1. Приклад фрагмента тесту (варіант 1)

При розробці тестів не обов'язково чекати кожної події, яке має відбуватися відповідно з випадком використання. Досить викликатиwait для подій, після настання яких треба змінювати стан оточення. У період очікування настання події, заданого в wait, може відбуватися будь-яку кількість інших подій. Всі ці події будуть занесені в журнал. При необхідності чекати не перша, а n-го виклику можна викликати wait з одним і тим же параметром n разів (наприклад, в циклі). При такому підході тест буде набагато коротше, наприклад Наведений вище тест буде виглядати наступним чином:

```
class Test1: Test
{
override public void start ()
{
StoreStat = "32" ;/ / Прийшов підшипник
```

```
RollerPar = "0 NewUser Depot1 123456 12 січня 1 +1" ;/ / його параметри
AxlePar = "1 NewUser Depot1 123456 1 0 12 12" ;/ / ні осі
CommandStatus = "0" ;/ / команда успішно прийнята
StoreMessage = "1" ;/ / команда успішно виконана
wait ("SendStoreCom") ;/ / перший підшипник прийнятий
wait ("SendStoreCom") ;/ / другий підшипник прийнятий
RollerPar = "1 NewUser Depot1 123456 12 січня 1 +1";
/ / Більше немає підшипників
AxlePar = "0 NewUser Depot1 123456 1 0 12 12" ;/ / є вісь
wait ("SendStoreCom") ;/ / видача підшипника
wait ("SendStoreCom") ;/ / видача підшипника
wait ("SendStoreCom") ;/ / завершення видачі
finish ();}
}
```

Приклад 5.2. Приклад фрагмента тесту (варіант 2)

Опис тестових процедур

Тестові процедури - це формальний документ, що містить опис необхідних кроків для виконання тестового набору. У разі ручних тестів *тестові процедури* містять повний опис всіх кроків і перевірок, що дозволяють протестувати продукт і винести вердикт PASS / FAIL. Процедури повинні бути складені таким чином, щоб будь-який інженер, не пов'язаний з даним проектом, був здатний адекватно провести цикл тестування, володіючи тільки самими базовими знаннями про застосовувані інструментарії.

Як запустити тест

Для того щоб запустити тест, потрібно:

У методі Class1.Main створити екземпляр нового класу і викликати його метод start. Так як метод start є віртуальним, то можна звертатися до тестів, використовуючи посилання на тип Test. Це робиться так:

Test t = new <ваш клас-нащадок Test>;

t.start ();

Зібрати і запустити програму.

Перевірка результатів виконання тестів (порівняння з очікуваним результатом)

Після завершення тесту слід переглянути текстовий журнал тесту (.. \ SystemTesting \ ManualTests \ Tests \ bin \ Debug \ log.log), щоб з'ясувати, яка послідовність подій в системі була реально зафіксована (вихідні дані) і порівняти їх з очікуваними результатами, заданими в специфікації тестового випадку № 1.

Як уже згадувалося, крім журналу тесту, створюється ще й відповідний **журнал системи.** В обох журналах міститься інформація про що відбувалися в системі подіях. До того, як вручну порівнювати журнал тесту з очікуваними результатами, заданими в специфікації тестового випадку, ви можете використовувати SystemLogAnimator (п. 14) для візуалізації відповідного журналу системи і отримати наочне (у тому числі ретроспективне) уявлення про те, як функціонувала система. Це важливо при розробці тестових випадків, тому що, не знаючи в деталях, як працює система, можна написати неправильний тест. Необхідно навчитися відрізняти, що ви насправді виявили - помилку в системі або результат роботи неправильного тесту.

Приклад неправильної тесту

Припустимо, що ви не зрозуміли Додаток 2 (FS) у всіх деталях і неправильно склали специфікацію тестового випадку № 1 і тест.Наприклад:

/ / Задаємо стан оточення (вхідні дані)

StoreStat = "32"; / / Поступив підшипник

...

/ / Отримуємо інформацію про функціонування системи

wait ("GetStoreStat"); / / Опитування статусу складу

/ / Замість того, щоб отримати інформацію про підшипнику з

/ / Терміналу підшипника, ми хочемо отримати інформацію

/ / Об осі з терміналу осі

wait ("GetAxlePar");

...

У журналі тесту ми побачимо наступну інформацію:

CALL: GetStoreStat 0

RETURN: 32

CALL: GetRollerPar

RETURN: 0 NewUser Depot1 123456 1 12 1 1

CALL: GetAxlePar

RETURN: 1 NewUser Depot1 123456 1 0 12 12

...

Головне - зуміти розібратися і виправити специфікацію тестового випадку і тест, якщо ви самі допустили помилку.

Завдання 3

Для тестового випадку № 1 необхідно скласти повний *список* всіх можливих альтернативних шляхів (див. підрозділ " *Список*альтернативних шляхів ") та розробити відповідні тести.

Крім того, необхідно:

- вибрати **випадок використання** на підставі дерева рішень (.. \ SystemTesting \ Decision Tree.vsd);
- скласти покроковий опис обраного випадку використання;
- врахувати всі альтернативні шляхи;
- скласти специфікації тестових випадків;
- розробити відповідні **тестові випадки (тести);**
- скласти **тестові звіти** (табл. 2.2).

Автоматизація тестування за допомогою скриптів

Ключові слова: Автоматизації тестування , запуск , сервер , мова скриптів , tcl , додаток

Загальна тенденція останнього часу передбачає максимальну *автоматизацію тестування,* яка дозволяє справлятися з великими обсягами даних і тестів, необхідних для сучасних продуктів.

У цьому випадку *запуск* тестів і перевірка того, що тестована система пройшла випробування на заданому **тестовому випадку,**буде здійснюватися автоматично.

Тут ще раз повторимо, що функції в dll були переписані так, що вони звертаються до сервера для отримання інформації про стан елементів комплексу і повертають серверу інформацію про функціонування системи. Для завдання стану оточення **(вхідних даних)** необхідно звернутися до сервера і передавати йому необхідну інформацію. В даному випадку був розроблений *сервер,* який окрім прийому і передачі інформації ще здійснює перевірку правильності поведінки системи (рис. 6.1). Він являє собою модель тестованої системи. У рамках моделі задані очікувані результати та здійснюється порівняння **вихідних даних** з **очікуваними результатами.** Хоча вважалося, що розроблена модель є коректною, повної і несуперечливої, у вас є реальна можливість знайти помилки і в самій моделі.

При розробці тестів був використаний наступний підхід:

- стан оточення задається в тесті **(вхідні дані);**
- розроблений сервер:
    - передає інформацію про заданому стані оточенні за запитом від dll;
    - отримує від dll інформацію про функціонування системи (вихідні дані);
    - порівнює вихідні дані з очікуваним результатом.
- отримувана інформація зберігається в журналі тіста;
- будується таблиця покриття FS (.. \ SystemTesting \ ScriptTests \ Logs \ summary.html).

Детальний опис тестового випадку № 1

Вивчення цього пункту корисно випередити ознайомленням з п.8, в якому описаний підхід до *автоматизації тестування* за допомогою *мови скриптів.* Тут розглядається тест на *tcl,* подібні тести необхідно буде писати самостійно при виконанні завдань.Наведений приклад був розроблений у відповідності зі специфікацією тестового випадку № 1. Для простоти будемо вважати, що події відбуваються послідовно у суворо заданому порядку. Реально наша система - багатопоточне *додаток,* тому така умова далеко не завжди можна гарантувати.

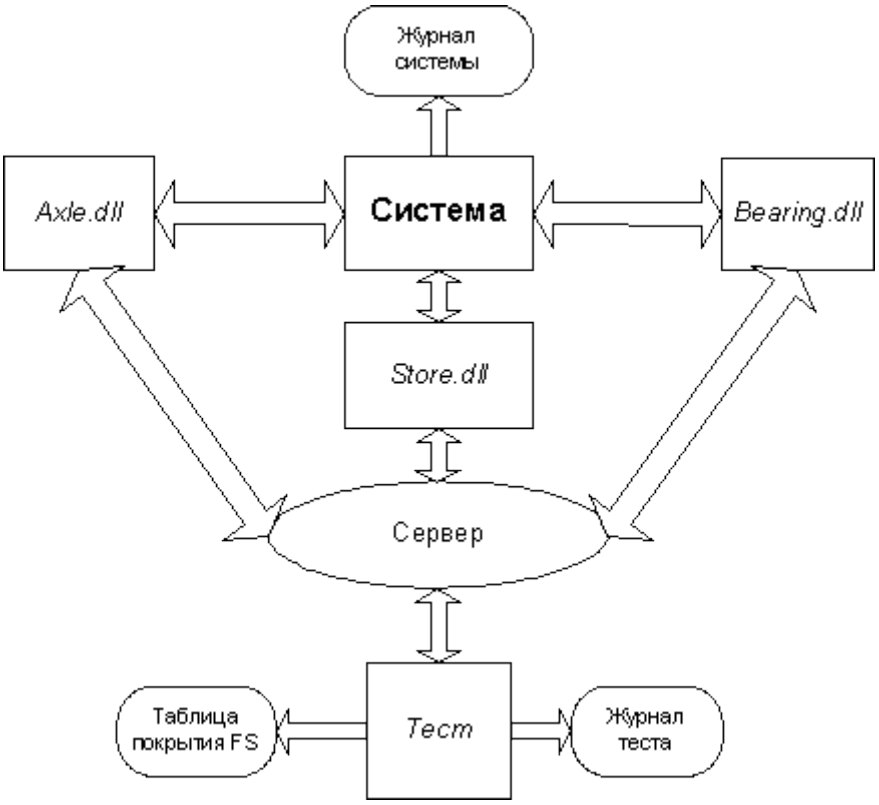


Рис. 6.1. Система і її оточення (скрипти)

source bin \ \ srv.tcl / / запуск сервера

global StoreStat / / статус складу

global RollerPar / / термінал підшипника

global AxlePar / / термінал осі

global CommandStatus / / повертається значення функції

/ / SendStoreCom про результати отримання команди

global StoreMessage

/ / Повідомлення від складу про результати виконання команди

global rollers_found / / 1 (можна підібрати відповідний

/ / Підшипник або 0 (не можна)

global fds / / рядок для графи "Покриття FS" у підсумковій

/ / Таблиці результатів тестування (за замовчуванням - Default)

global last_command / / остання команда тестованої системи

global allowed / / список дозволених команд

set fds "1.a.1; 1.a.4; 2.a; 2.c; 3.a; 4.c"

/ / Покриває задані пункти FS

StartTest Warehousetest0001 / / запуск тесту Timeout 30

/ / Процес тестування буде перерваний через 30 сек

/ / Задаємо стан оточення (вхідні дані)

set StoreStat 32 / / Поступив підшипник

set RollerPar "0 NewUser Depot1 123456 12 Січня 1 +1"

/ / Статус обміну з терміналом підшипника

/ / (0 - є підшипник) і його параметри

set AxlePar "1 NewUser Depot1 123456 1 0 12 12"

/ / Статус обміну з терміналом осі (1 - немає осі)

/ / І її параметри

set CommandStatus 0 / / команда успішно прийнята

set StoreMessage 1 / / команда успішно виконана

set rollers_found 1 / / можна підібрати відповідний підшипник

/ / Отримуємо інформацію про функціонування системи

Wait "GetStoreStat *" 0 1

/ / Необмежений час (0) чекаємо отримання команди ("опитування

/ / Статусу складу ") і якщо команда не отримана, то буде

/ / Зафіксована помилка (1)

```
Wait "GetRollerPar" 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати інформацію про підшипнику з терміналу підшипника")
/ / І якщо команда не отримана, то буде зафіксована помилка (1)
set allowed [list "GetAxlePar"]
/ / Отримання команди "отримати інформацію про вісь з терміналу
/ / Осі "дозволено і не повинно викликати помилку
Wait "SendStoreCom 1 *" 10 січень
/ / Протягом 10 секунд чекаємо отримання команди ("додати в
/ / Чергу команд складу на перше місце команду GetR (1 -
/ / Отримати з приймача в осередок) "і якщо команда за це
/ / Часом не отримана, то буде зафіксована помилка (1)
Wait GetStoreMessage 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати повідомлення від складу про результати виконання
/ / Команди ") і якщо команда не отримана, то буде
/ / Зафіксована помилка (1)
/ / В результаті перший підшипник повинен бути прийнятий
set allowed [list]
Wait "GetStoreStat *" 0 1
/ / Необмежений час (0) чекаємо отримання команди ("опитування
/ / Статусу складу ") і якщо команда не отримана, то буде
/ / Зафіксована помилка (1)
```

```
Wait GetRollerPar 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати інформацію про підшипнику з терміналу
/ / Підшипника ") і якщо команда не отримана, то буде
/ / Зафіксована помилка (1)
set allowed [list "GetAxlePar"]
/ / Отримання команди "отримати інформацію про вісь з терміналу
/ / Осі "дозволено і не повинно викликати помилку
Wait "SendStoreCom 1 *" 10 1
/ / Протягом 10 секунд чекаємо отримання команди ("додати в
/ / Чергу команд складу на перше місце команду GetR
/ / (1 - отримати з приймача в осередок) ") і якщо команда за
/ / Цей час не отримана, то буде зафіксована помилка (1)
Wait "GetStoreMessage" 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати повідомлення від складу про результати виконання
/ / Команди ") і якщо команда не отримана, то буде
/ / Зафіксована помилка (1)
/ / В результаті другого підшипник повинен бути прийнятий
/ / Задаємо новий стан оточення (вхідні дані)
set StoreStat 32 / / Поступив підшипник
set RollerPar {1 NA NA 0 0 0 0 0}
/ / Статус обміну з терміналом підшипника (1 - немає підшипник)
```

/ / І його параметри

set AxlePar "0 NewUser Depot1 123456 1 0 12 12"

/ / Статус обміну з терміналом осі (0 - є вісь) і

/ / Її параметри

/ / Отримуємо інформацію про функціонування системи

Wait "GetStoreStat *" 0 1

/ / Необмежений час (0) чекаємо отримання команди ("опитування

/ / Статусу складу ") і якщо команда не отримана, то буде

/ / Зафіксована помилка (1)

Wait "GetRollerPar" 0 1

/ / Необмежений час (0) чекаємо отримання команди

/ / ("Отримати інформацію про підшипнику з терміналу

/ / Підшипника ") і якщо команда не отримана, то буде

/ / Зафіксована помилка (1)

Wait GetAxlePar 0 1

/ / Необмежений час (0) чекаємо отримання команди

/ / ("Отримати інформацію про осі з терміналу осі") і якщо

/ / Команда не отримана, / / то буде зафіксована помилка (1)

/ / У результаті повинна прийти вісь

Wait "SendStoreCom 2 *" 10 січень

/ / Протягом 10 секунд чекаємо отримання команди ("додати в

/ / Чергу команд складу на останнє місце команду SendR

/ / (2 - осередок на вихід) ") і якщо команда за цей час не

```
/ / Отримана, то буде зафіксована помилка (1)
if {! [string compare $ last_command "SendStoreCom 2 9 9 9 0 0 1"]}
{
/ / Була послана команда видати перший підшипник для осі
Wait "GetStoreMessage" 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати повідомлення від складу про результати виконання
/ / Команди ") і якщо команда не отримана, то буде
/ / Зафіксована помилка (1)
Wait "SendStoreCom 2 9 9 9 0 1 1" 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Додати в чергу команд складу на останнє місце
/ / Команду SendR (2 - осередок на вихід) ") і якщо команда за
/ / Цей час не отримана, то буде зафіксована помилка (1)
/ / Послали команду видати другий підшипник
}
if {! [string compare $ last_command "SendStoreCom 2 9 9 9 0 1 1"]}
{
/ / Якщо була послана команда видати другий підшипник для осі
Wait "GetStoreMessage" 0 1
/ / Необмежений час (0) чекаємо отримання команди
/ / ("Отримати повідомлення від складу про результати виконання
/ / Команди ") і якщо команда не отримана, то буде
```

/ / Зафіксована помилка (1)

Wait "SendStoreCom 2 9 9 9 0 0 1" 0 1

/ / Необмежений час (0) чекаємо отримання команди

/ / ("Додати в чергу команд складу на останнє місце

/ / Команду SendR (2 - осередок на вихід) ") і якщо команда за

/ / Цей час не отримана, то буде зафіксована помилка (1)

/ / Послали команду видати перший підшипник

}

Wait "GetStoreMessage" 0 1

/ / Необмежений час (0) чекаємо отримання команди ("отримати

/ / Повідомлення від складу про результати виконання команди ") і

/ / Якщо команда не отримана, то буде зафіксована помилка (1)

Wait "SendStoreCom 20 *" 0 1

/ / Необмежений час (0) чекаємо отримання команди

/ / ("Додати в / / чергу команд складу на останнє місце

/ / Команду Term (20 - / / завершення команд видачі) ") і якщо

/ / Команда не отримана, то буде зафіксована помилка (1)

Wait "GetStoreMessage" 0 1

/ / Необмежений час (0) чекаємо отримання команди

/ / ("Отримати повідомлення від складу про результати виконання

/ / Команди ") і якщо команда не отримана, то буде

/ / Зафіксована помилка (1)

EndTest

Лістинг 6.1. Тест на tcl / tk

Опис тестових процедур

Як запустити тест

Запустити run.bat. У файлі run.bat запускається tests.bat. Файл tests.bat містить команди запуску тестів і установки необхідного стану бази даних:

```
 osql-H <Host>-S <Server>-d WarehouseTCL-U sa-P sa-i
sql \ ClearDB.sql
/ / Викликається скрипт ClearDB.sql з підкаталогу sql.
/ / Передбачається, що SQL Server виконується на машині
/ / <Host> І називається <Server>.
/ / Ці параметри були налаштовані автоматично при
/ / Установці практикуму.
/ / База даних називається WarehouseTCL.  Якщо назви відрізняються,
/ / Необхідно замінити параметри командного рядка утиліти OSQL:
/ /-H <host> - задає ім'я хоста, на якому виконується SQL Server;
/ /-S <server> - ім'я сервера;
/ /-D <db> - ім'я бази даних;
/ /-U <username> - ім'я користувача;
/ /-P <password> - пароль;
/ /-I <script> - ім'я скрипта SQL.
bin \ launcher tests \ warehousetest0001.tcl
/ / Викликається тест warehousetest0001.tcl
copy log.txt logs \ warehousetest0001.txt
```

/ / Файл log.txt (лог тестованої системи) копіюється у файл

/ / Warehousetest0001.txt. Для виключення частини тестів з набору

/ / Досить закомментировать відповідні рядки (команда

/ / REM).

Перевірка результатів виконання тестів (порівняння з очікуваним результатом)

У цьому випадку запуск тестів і перевірка того, що тестована система пройшла випробування на заданому **тестовому випадку,**здійснюється автоматично і результати представляються у вигляді таблиці; тут, як і в попередньому випадку створюється журнал тесту, а також можна використовувати SystemLogAnimator для візуалізації журналу системи.

Приклад неправильної тесту

Розглянемо той же приклад неправильного тесту, як і у випадку ручного тестування:

При надходженні підшипника замість того, щоб отримати інформацію про підшипнику з терміналу підшипника, ми хочемо отримати інформацію про вісь з терміналу осі.

Переконайтеся, що система функціонує по-іншому.

Завдання 4

Потрібно виконати ті ж завдання, що і для ручного тестування.

Лекція 11: Підтримка процесу тестування при промисловій розробці програмного забезпечення

Анотація: Лекція присвячена промисловим процесам підтримки тестування: управління якістю та управління проектно. Розглядаються особливості цих процесів при розробці сертифицируемого програмного забезпечення. Мета даної лекції: дати уявлення про процеси управління якістю та управління проектно

Ключові слова: життєвий цикл програмного забезпечення , ISO 9001 , конфигурация , целостность , компонент , software , A ND ,equipment , certification , ISO , IEEE , верификация , конфигурационное упр авління , configuration item , прогін тестів , білд ,програмне забезпечення , циклу , ПО , індекс

26.1. Управління якістю

26.1.1. Завдання і цілі управління якістю

Промислова розробка програмного забезпечення як частину промислової розробки складних систем - високотехнологічний процес, в який залучено безліч розрізняються за чисельністю, сфері діяльності та кваліфікації колективів розробників. Одна з основних завдань при спільній роботі великої кількості розробників або їх колективів: забезпечення єдиної схеми роботи, що дозволяє планувати виконання робіт, забезпечувати цілісність і несуперечність різних вузлів системи, а також давати гарантії відповідності системи очікуванням замовника.

Виконання цього завдання полегшується при підпорядкуванні процесу розробки технологічним вимогам, що регламентує різні аспекти розробки: життєвий цикл проекту та розроблюваної системи, вимоги до процесів розробки, впровадження та супроводу системи.

При розробці систем різного типу (інформаційних, вбудованих, систем реального часу, систем безпеки) технологічні вимоги можуть відрізнятися і висвітлювати ті чи інші аспекти процесу розробки. Проте, зазвичай

впровадження технології в розробку переслідує одну основну мету - забезпечення і гарантію якості розроблюваної системи. Це послужило передумовою до створення документів, що визначають вимоги до технології розробки систем - стандартів якості.

Згідно підходу стандартів системи якості: якість - це сукупність характеристик об'єкта, що має відношення до його здатності задовольнити встановлені і передбачувані вимоги споживача. При цьому, що важливо, під об'єктом якості може розумітися як власне продукція (товари або послуги) або процес її виробництва, так і виробник (організація, система чи навіть окремий працівник).

У цих стандартах визначаються критерії якості продукції, специфічні для кожного з етапів життєвого циклу продукції. Основне призначення стандартів якості - визначення вимог до технологічних процесів, в т.ч. процесам розробки, дотримання яких дозволяє гарантувати постійний з точки зору обраних критеріїв рівень якості створюваних систем (Рис 26.1).

Рис. 26.1. Місце стандартів якості у розробці системи

Неминучі відмінності в процесах розробки та експлуатації систем в різних галузях послужили передумовою створення галузевих стандартів - стандартів, які регламентують процеси розробки і сертифікації систем, призначених для вузькоспеціального використання (авіаційні двигуни, системи інформаційної безпеки).

Оскільки область застосування стандартів якості досить широка (від окремої галузі до загальної застосовності), основним об'єктом їх розгляду є не конкретні методики розробки, а загальні технологічні процеси.

Більшість стандартів орієнтовані на процеси, які не залежать безпосередньо від конкретного життєвого циклу системи. Для кожного процесу визначаються цілі і описуються засоби задоволення цих цілей. Для даних життєвого циклу в стандартах представляється опис, який показує, що цілі задоволені.

Основний процес, розглянутий стандартами - випуск продукції. У разі розробки програмних систем цей процес - проект розробки програмного забезпечення, незалежно від того, який *життєвий цикл програмного забезпечення* був вибраний. Процес розробки зазвичай розпадається на більш дрібні підпроцеси.

Підпроцеси можуть бути віднесені до трьох категорій: процес планування програмного забезпечення; процеси розробки програмного забезпечення, які зазвичай включають розробку вимог до програмного забезпечення, проектування програмного забезпечення, кодування і комплексування програмного забезпечення; і процеси забезпечення цілісності, які включають в себе верифікацію програмного забезпечення, гарантію якості програмного забезпечення, управління конфігураціями програмного забезпечення та взаємодія при сертифікації.

26.1.2. Система менеджменту якості по ISO 9000

Сімейство стандартів ISO 9000 - група міжнародних стандартів, що встановлюють правила менеджменту якості при випуску продукції. Вітчизняна група стандартів, відповідна міжнародним ISO 9000, отримала назву ДСТУ ISO 9000 [6]. Під поняття "випуск продукції" потрапляє і розробка програмного забезпечення.

При цьому стандарти ISO 9000 проводять відмінність між вимогами до систем управління якістю і вимоги до продукції. Стандарт не гарантує якість продукції - якість продукції в стандарті прямо не згадується, тим самим він відрізняється від керівних документів з перевірки якості випуску продукції різного роду (в т.ч. і програмних систем).

Вимоги до систем менеджменту якості встановлені в стандарті *ISO 9001* (ГОСТ Р ІСО 9001). Вони є загальними і застосовні до організацій усіх галузей промисловості чи економіки, незалежно від категорії продукції.

Вимоги до продукції можуть бути встановлені споживачами або організацією, виходячи з передбачуваних запитів споживачів або вимог регламентів. Вимоги до продукції і в ряді випадків до пов'язаних з нею процесів можуть міститися, наприклад в технічних умовах, стандартах на продукцію, стандартах на процеси, контрактних угодах і регламентах.

Коротко повторимося, зробивши важливе уточнення - цей стандарт може бути сформульовано таким чином: всі процеси, які можуть істотно вплинути на якість готової продукції, повинні бути задокументовані, за виконання цих правил повинна бути призначена персональна відповідальність, регулярно повинна проводитися перевірка відповідності реальних процесів документованим вимогам . Важливо, що обов'язковою вимогою є встановлення відповідальності за якість процесів.

Отже "система якості" - це сукупність організаційної структури, методик, процесів і ресурсів, необхідних для загального керівництва якістю.

В основі ISO 9000 лежать 8 принципів.

1. Орієнтація на споживача

Організації залежать від своїх споживачів і тому повинні розуміти поточні та майбутні потреби, виконувати їхні вимоги і прагнути перевершити їх очікування.

2. Лідерство керівника

Керівники встановлюють єдність мети та напрямів діяльності організації. Їм слід створювати та підтримувати внутрішнє середовище, в якому працівники можуть бути повністю залучені до виконання завдань організації.

3. Залучення працівників

Працівники на всіх рівнях становлять основу організації, і їхнє повне залучення дає можливість організації з вигодою використовувати їх здібності.

4. Процесний підхід

Бажаний результат досягається ефективніше, коли діяльністю та відповідними ресурсами управляють як процесом.

5. Системний підхід до менеджменту

Виявлення, розуміння та управління взаємопов'язаними процесами як системи сприяють результативності та ефективності організації при досягненні її цілей.

6. Постійне поліпшення

Постійне поліпшення діяльності організації в цілому слід вважати незмінною мета.

7. Прийняття рішень, засноване на фактах

Ефективні рішення грунтуються на аналізі даних та інформації.

8. Взаємовигідні відносини з постачальниками

ISO 9000 визначає наступні основні процеси верхнього рівня:

- Система менеджменту якості
- Відповідальність керівництва
- Управління ресурсами
- Випуск продукції
- Вимірювання, аналіз, поліпшення

Система менеджменту якості є основою основ при досягненні показників якості. Основним завданням системи менеджменту якості є визначення процесів та їх застосування у всій організації, послідовність і взаємодія процесів. Ефективність роботи процесів досягається управлінням необхідними для процесів ресурсами і документами. Оцінка ефективності ведеться за

допомогою моніторингу, вимірювання та аналізу заданих для процесів критеріїв результативності. Всі вимоги системи менеджменту якості фіксуються в стандартах підприємства на відповідні процеси.

Відповідальність керівництва з точки зору ISO 9000 полягає у прийнятті зобов'язань з впровадження та підтримання на підприємстві системи менеджменту якості. До відома співробітників повинна доводитися інформація про необхідність досягнення заданої якості продукції, і також контролюється забезпечення процесів, орієнтованих на випуск продукції, необхідними ресурсами. Основні документи, що розробляються керівництвом - політика та цілі у сфері якості, що визначають поточні та майбутні шляхи розвитку підприємства для досягнення необхідного рівня якості.

Стандарти якості також регламентують вимоги до різних етапах процесу випуску продукції, спрямовані на підвищення її якості. Основна суть цих вимог полягає в забезпеченні максимально прослеживаемого технологічного процесу, задокументованого в стандартах підприємства.

26.1.3. Аудит процесів розробки та верифікації

Аудит є процесом, що дозволяє збирати дані про функціонування системи менеджменту якості для подальшого їх аналізу з метою поліпшення існуючих на підприємстві процесів. Аудити проводяться періодично, згідно з розробленою програмою або позапланово, на підставі аналізу результатів проведених аудитів.

Внутрішні аудити проводяться силами служби якості підприємства, а аудитори призначаються з числа досвідчених співробітників.Мета таких аудитів - поточний контроль системи менеджменту якості підприємства, що дозволяє своєчасно виявити проблеми.

Зовнішні аудити проводять аудитори з аудиторської компанії, мета таких аудитів - підтвердження відповідності системи менеджменту якості підприємства вимогам стандарту якості.

На відміну від процесу верифікації, який перевіряє якість розроблюваної програмної системи, аудит в складі процесу управління якістю спрямований на перевірку технологічних процесів - як розробки, так і верифікації. Так, верифікація відповідає на запитання "чи розроблено програмна система відповідно до вимог?", А аудит менеджменту якості відповідає на запитання "чи відповідали процеси розробки та верифікації встановленим нормам і схемами технологічних процесів, визначених у стандартах проекту та / або підприємства?".

Тобто аудит якості процесів розробки - це теж перевірка відповідності, але в ролі вимог тут виступають стандарти системи менеджменту якості, а в ролі перевіреній системи - процеси.

Результати аудитів використовуються в якості одного з інформаційних потоків для проведення аналізу результатів функціонування та відповідності процесів.

Після проведення аудиту, у разі виявлення невідповідностей (явно суперечать стандартам якості випадків) та / або спостережень (випадків, що не суперечить стандартам якості, але потребують модифікації для підвищення ефективності роботи технологічного процесу), робляться коригувальні та / або запобіжні дії.

26.1.4. Коригувальні дії та корекція процесів

Відповідно до вимог стандарту *ISO 9001*, підприємство, в разі виникнення невідповідностей (тобто не виконання встановлених вимог до продукції, що випускається, до системи менеджменту якості або до її окремих процесам) або передумов до їх появи, має визначити дії, які дозволять не тільки виправити дану проблему і усунути причини її появи, але й запобігти можливість виникнення її в майбутньому за допомогою розробки і впровадження коригувальних та попереджувальних дій.

Основним документів, визначальним проблему, є записка за якістю, що ідентифікує суть проблеми і співробітника, що виявив проблему.

Як правило, для вирішення виявлених і класифікованих проблем застосовуються три типи заходів: корекція, коригувальну дію і запобіжна дія. Термін "корекція" має відношення до ремонту, переробці чи регулюванню і відноситься до усунення наявного невідповідності. Термін "коригувальну дію" відноситься до усунення причини невідповідності.

Термін "запобіжна дія" відноситься до усунення причин потенційного невідповідності, дефекту або іншої небажаної потенційно можливої ситуації з тим, щоб запобігти їх виникненню.

Таким чином, що прийшла в процес менеджменту якості записка за якістю або закривається, або породжує коригуючий / запобіжна дія (КД / ПД), а часом і корекцію. Залежно від рівня значущості, пошук рішень проблеми може проводитися або на рівні процесу менеджменту якості, або на рівні керівництва підприємства, процесу або окремого проекту. Зауважимо, що причин невідповідності може бути декілька, і в ряді випадків єдиним способом усунення є корекція вимог (стандартів).

Слід також зауважити, що виконання заходів щодо усунення причин невідповідностей (КД / ПД) і самих невідповідностей (корекція) по суті є такими ж роботами, як і решта планові заходи.

26.2. Конфігураційне управління

Процеси підтримки цілісності, розглянуті в попередньому розділі, залишаються активними в ході всього *життєвого циклу програмного забезпечення,* до їх числа входить і процес конфігураційного управління.

Основне завдання даного процесу - забезпечення структурної цілісності розроблюваної системи. Всі дані, що входять в проект, розглядаються як єдина *конфігурація,* структурна *цілісність* якої досягається за допомогою контролю всіх вхідних в неї *компонент,*забезпечення їх фізичного збереження, контролю та управління змінами *компонент* системи.

Застосування процесу конфігураційного управління є основною вимогою при розробці систем, до їх надійності (тобто, згідно ГОСТ 13377-75 [22] - властивості об'єкта виконувати задані функції, зберігаючи в часі значення встановлених експлуатаційних показників у заданих межах, що відповідають заданим режимам і умовами використання, технічного обслуговування, ремонту, зберігання і транспортування), до яких пред'являються підвищені вимоги, зокрема, при розробці бортових авіаційних систем, інформаційних систем великого масштабу, систем безпеки (див., наприклад, DO-178B [7]).

Основні завдання і цілі процесів конфігураційного управління (КУ) полягають у наступному [16].

- **Ідентифікація:** присвоєння унікальних імен об'єктам конфігураційного управління (ОКУ). Кожен об'єкт в конфігурації повинен відрізнятися від усіх інших об'єктів. Дана вимога не може бути задоволена створенням пошукових образів об'єктів (пошук документів по внутрішньому вмісту), тому що одне із застосувань ідентифікації - забезпечення трасуванню об'єктів. За допомогою трас між документами створюються логічні посилання, що дозволяють простежити залежності між різними групами проектної документації, покриття вимог і т.п.
- **Управління:** управління випуском продукту і його змінами протягом усього життєвого циклу. У даний набір цілей входить надання інформації, яка необхідна керівництву для контролю змін у даних, що відносяться до продукту, що випускається.
- **Обчислення статусів:** зберігання і створення звітів по станах об'єктів конфігураційного управління і запитів на зміну. Крім обчислення статусів у вигляді звітів, подібних індексам конфігурації, функція обчислення статусів повинна дозволяти обчислювати статус окремо взятого ОКУ і визначати життєві цикли, які є сукупністю статусів і правил їх зміни.
- **Аудит та інспекції:** перевірка завершеності та повноти продукту і підтримку цілісності і несуперечності зв'язків усіх його компонент. Процес

КУ може регламентувати як процедури проведення аудитів та їх фази, так і виключно точки життєвого циклу, в яких проводяться аудити конфігурації.

- **Випуск:** управління збіркою і побудовою остаточної версії продукту. У даному процесі зазвичай використовуються індекси конфігурацій, в яких перераховуються ОКУ, необхідні в процесі складання версії для тієї чи іншої цільової платформи.
- **Управління процесом:** перевірка дотримання організацією правил проведення процедур і моделі життєвого циклу. Управління здійснюється у вигляді постійного контролю за процедурами або в робочому порядку, або у вигляді регулярних аудитів і дозволяє упевнитися в технологічності процесів розробки, що в кінцевому результаті позитивно позначається на якості продукції.
- **Колективна робота:** управління роботою і взаємодією між безліччю користувачів, що працюють над продуктом, в тому числі управління проектом і розподіл завдань між виконавцями. Також в процесі забезпечення колективної роботи повинен проводитися контроль виконання розробниками поставлених перед ними завдань шляхом відстеження статусу документа.

Розглянемо процес управління проектно в рамках документа DO-178B (*Software* Considerations in Airborne Systems *and EquipmentCertification),* що визначає вимоги до процесів розробки авіаційного бортового програмного забезпечення. Вимоги цього стандарту до процесу управління проектно є досить жорсткими і в цілому співставні з вимогами інших стандартів (*ISO* 10007 [23], *IEEE* 1042 [24]).

У стандарті DO-178B визначаються шість основних процесів програмного проекту, з яких три можна віднести до виробничих: планування, розробка та *верифікація,* а три до підтримує: забезпечення якості, взаємодія з

сертифікуючим органом і *конфігураційне управління.* Виробничих процесів присвячені відповідно глави 4, 5 і 6.

Процес конфігураційного управління розглядається в сьомий чолі документа. При цьому деякі його аспекти зачіпаються в четвертому розділі, присвяченій плануванню, а деякі - в главі 11, присвяченій даними процесу розробки.

26.2.1. Завдання процесу конфігураційного управління

Основним завданням процесу конфігураційного управління є забезпечення гарантії того, що організація-розробник має всі необхідні дані для підтвердження факту відповідності виробленого продукту вимогам.

Конфігураційне управління можна визначити як процес, за допомогою якого керівництво проекту має можливість на постійній основі ідентифікувати, встановлювати зв'язки, супроводжувати і управляти різними компонентами проекту. Цей процес гарантує цілісність компонент та простежуваність всіх змін, що виникають в будь-який момент життєвого циклу проекту.

Базовим поняттям процесу є об'єкт (елемент) конфігураційного управління (ОКУ, *Configuration Item).* Під об'єктами конфігураційного управління можуть розумітися всі основні результати діяльності проекту. Такі результати ідентифікуються і контролюються за допомогою процесу конфігураційного управління. Об'єктами конфігураційного управління можуть бути елементи апаратури, програми, документація, процедури та матеріали навчання, засоби обслуговування і т.д. З метою ідентифікації об'єктів конфігураційного управління можуть бути привласнені номери.

Ще один термін, використовуваний в процесі конфігураційного управління - базова конфігурація (Baseline). Під базовою конфігурацією (БК) розуміється об'єкт конфігураційного управління (окремий елемент або сукупність елементів), який пройшов процедуру затвердження і може бути змінений тільки в рамках процедури управління змінами.

Базова конфігурація - це свого роду фотознімок, "заморожена ситуація" вимог, специфікацій або результатів, що знаходяться в розробці. Базова конфігурація може бути представлена документом або набором документів. Створення базовій конфігурації звичайно являє собою фіксацію деякої умови, що виникає при завершенні кожного з основних кроків процесу розробки.

Базова конфігурація може складатися з сукупності однорідних документів: наприклад, сукупність вимог, коди сукупності програмних модулів. Але базова конфігурація може складатися і з сукупності різнорідних за своєю суттю ОКУ: наприклад, вимоги і відповідні програмний код, тест-план, результати *прогону тест-плану.*

Таким чином процес управління проектно покликаний забезпечувати:

- об'єктивність і контрольованість даних проекту;
- доступність і восстанавливаемость даних проекту, включаючи об'єктні коди програм (наприклад, об'єктний код може не зберігатися, але зберігається вихідний код і зафіксована процедура трансляції);
- контрольованість вхідних і вихідних даних процедур проекту, забезпечуючи їх цілісність і повторюваність;
- точки контролю, можливість обчислення статусу конфігурації та управління змінами через управління ОКУ і створення БК;
- фіксацію проблем і відстеження прийнятих по них рішень;
- можливість простеження стану проекту шляхом контролю вихідних даних його життєвого циклу;
- гарантії відповідності виробленої продукції ставляться до ній вимогам;
- обмеження доступу та збереження ОКУ.

Збереження ОКУ передбачає виконання дій з архівування даних, які перебувають під опікою процесу конфігураційного управління, і проведення аудитів конфігурацій. Тобто дані не тільки повинні бути огороджені від

випадкового (несанкціонірованнного) спотворення, а й збережені на випадок виходу з ладу засобів зберігання (пожежі, затоплення, руйнування носіїв тощо).

Конфігураційне управління дозволяє команді розробників програми або проекту точно визначати статус будь компоненти в усі час її життєвого циклу і дає можливість перевоссоздать будь-яку версію в будь-який момент часу. Компонентами можуть бути будь-які комбінації апаратури, програм, обслуговування та навчання.

26.2.2. Процедури процесу конфігураційного управління

Конфігураційне управління побудовано як композиція декількох підпроцесів, що функціонують спільно:

- ідентифікація конфігурацій;
- управління змінами;
- формування базових конфігурацій;
- обчислення статусів конфігурацій;
- підтримка обмежень доступу;
- архівування, аудити / огляди конфігурацій.

Слід особливо зауважити, що процес конфігураційного управління аж ніяк не закінчується із завершенням робіт проекту з виробництва продукції. Процес конфігураційного управління продовжує функціонувати, і, можливо, дуже значний час, забезпечуючи підтримку супроводу програмного продукту.

26.2.2.1. Ідентифікація конфігурацій

Метою процедури ідентифікації є присвоєння кожному ОКУ унікального імені (коду), що забезпечує його впізнання серед інших ОКУ.Слід зауважити, що процедура ідентифікації з очевидністю повинна передувати процедурі простеження (трасування). У тих випадках, коли ідентифікація об'єкта не може бути досягнута шляхом нанесення на неї ідентифікаційного коду (наприклад, для

об'єктного коду програми), вона повинна бути забезпечена непрямим шляхом, наприклад, ідентифікаційним полем, значення якого може бути проконтрольовано допоміжним програмним засобом.

Процедура КУ проекту повинна встановлювати систему ідентифікації для кожної окремо конфігурується компоненти програмного забезпечення. Оскільки сама компонента може при цьому складатися з окремих складових частин, то ідентифікація повинна рекурсивно поширюватися на всі її складові частини до досягнення рівня атомарного ОКУ (тобто файлу).

Ідентифікація ОКУ відбувається шляхом приміщення цих об'єктів в базу даних проекту. У самому простому випадку база даних проекту може бути загальнодоступним мережевим каталогом на сервері, ідентифікація ОКУ і їх версій при цьому повинна проводитися вручну.

Існує ряд систем конфігураційного управління, які являють собою інструменти для забезпечення колективної роботи з базою даних проекту. Ці системи беруть на себе всі основні функції конфігураційного управління, перераховані вище, в т.ч. схоронність ОКУ, автоматичну нумерацію версій, запобігання неавторизованих дій над ОКУ і облік стану ОКУ.

Ідентифікатором ОКУ служить ім'я файлу в сукупності з шляхом всередині бази даних. Файл присвоюється менеджером конфігурацій; він же має право перейменовувати ОКУ.

Щоб виключити можливість появи в базі даних проекту неправильно пойменованих, неправильно розміщених і не підлягають зберіганню в репозиторії об'єктів, операції New Folder, Introduce і Rename можуть виконувати тільки керівник проекту та менеджер конфігурацій.

Складові ОКУ ідентифікуються шляхом складання індексів конфігурацій, в яких перераховуються ОКУ (із зазначенням номера версії), що входять до складу конфігурації даного ОКУ. Індекс конфігурації, в свою чергу, є об'єктом конфігураційного управління і може входити як складова частина в іншій

ОКУ. Таким чином, ОКУ, в загальному випадку, утворюють ієрархію, вершиною якої є конфігурація продукту, що включає в себе всі інші ОКУ.

26.2.2.2. Базові конфігурації та простежуваність

Базові конфігурації зазвичай використовуються як основа для переходу від однієї процедури життєвого циклу проекту до іншої. У той момент, коли процес розробки переходить від одного кроку до іншого, специфічні для цього кроку результати (документи, специфікації або продукти) інспектують, щоб переконатися в їх якості та зв'язках (трасуванню) з попередніми результатами.Специфічні версії об'єктів конфігурації, що належать їм, ідентифікуються. Коли (і якщо) результати (виходи) кроку пройшли процедуру інспекції (тільки після цього), вони можуть бути оголошені базовою конфігурацією і стають готовими до використання на наступному кроці процесу як входу.

Принциповим є те факт, що базова конфігурація може змінюватися тільки через процедуру Управління Змінами. При цьому вимоги DO-178B зобов'язують забезпечити простежуваність "походження" базовій конфігурації. Іншими словами, має бути забезпечене вказівку того, з якої попередньої БК отримана дана і за допомогою якої процедури.

У документі DO-178B застосовується єдиний термін "трасуванню". Він використовується і для позначення посилань від коду програми до вимог, і для вказівки на батьківську базову конфігурацію. Можливо, що з метою більш точної ідентифікації слід в ряді випадків розрізняти поняття "простежуваність" (еволюція БК) і "трасуванню" - зв'язку різнотипних документів (відображення перетворення входу виробничої процедури в її вихід). Зазвичай під трассируемого розуміється можливість ідентифікувати та історію, і поточний стан (статус) кожного об'єкта конфігурації в будь-якій точці життєвого циклу проекту. Необхідною також є і можливість трасування об'єктів конфігурації щодо вимог замовника як первинного входу проекту.

26.2.2.3. Управління змінами

Більшість конфігураційних об'єктів в ході життєвого циклу проекту претерпивают зміни. У процесі фіксації цих змін виникає дерево версій, що представляє варіанти об'єкта за ступенем його "завершеності". Кожна нова гілка і лист такого дерева представляють нову версію ОКУ. Тільки остання коректна версія будь-якого об'єкта повинна поширюватися "за замовчуванням" розробникам у відповідь на їхні запити, застарілі версії можуть бути архівовані. При цьому процедура архівування повинна забезпечувати восстанавливаемость (зберігання або обчислення) будь запитаної версії ОКУ.

Процедура управління змінами визначає оцінку і трасування запитів на зміни, аналіз потенційного впливу змін і прийняття рішень щодо внесення змін в об'єкти конфігурації. Ця процедура повинна забезпечувати запобігання "випадкового" внесення зміни в базову конфігурацію.

Реалізація запитів на зміни повинна включати трасування даної зміни через процедуру фіксації проблеми, вироблення і втілення зміни, контролю його результатів і фіксації відповідних даних життєвого циклу проекту.

Результати процедури внесення змін повинні інспектуватися, що, власне, і призводить до можливості затвердження нової базової конфігурації.

Версії (version) і Редакції (release) - ці терміни іноді взаємозамінні. У даному документі термін "версія" використовується насамперед для посилань на кожне нове прояв об'єкта конфігураційного управління, якому присвоєно унікальний ідентифікаційний номер, і має на увазі наявність ланцюжка ОКУ пов'язаних ставленням простежуваності. Іншими словами, одна версія ОКУ виходить з іншої через процедуру управління змінами.

Редакцією тут називається специфічна версія об'єкта конфігурації, призначена для "зовнішнього" використання. Як правило, це зовнішнє використання - завантаження коду програми в цільової процесор або відвантаження результатів замовнику. Редакціями можуть бути версії об'єктів, що утворюють комплект системи для внутрішнього тестування ("лоад", " *білди* "). Такий об'єкт (базову

конфігурацію) не можна змінити. Вона вже відіслана і починає жити" своїм життям "поза проектом. Можна утворити нову редакцію і переслати її заново.

26.2.2.4. Обчислення статусу конфігурації

Після того, як зміна об'єкту конфігурації санкціоновано, зазвичай виникає деяка тимчасова затримка на час реалізації зміни.Керівництву та учасникам проекту необхідно мати відомості про процес внесення змін, а не тільки про факт його завершення.Процедура обчислення статусу - механізм, який використовується для простеження еволюції кожного об'єкта системи та її поточного стану.

Процедура обчислення статусу повинна забезпечувати можливість реководству проекту на підставі відстеження станів окремих ОКУ судити про стан розробки в цілому. Ця процедура забезпечує проектного менеджера великою кількістю даних про його продукт, включаючи те, як він розробляється і чи всі необхідні властивості дійсно реалізовані. У кінцевому рахунку, процедура забезпечує актуальну та об'єктивну інформацію про статус кожного об'єкта конфігурації в будь-який момент процесу розробки, яка включає дані таких типів даних:

- час виникнення кожної БК і зміни (проблеми);
- час визначення кожного об'єкта конфігурації;
- описова інформація про кожен об'єкт конфігурації;
- статус запитів на зміни (прийнятий, відхилений, очікує виконання, виконано);
- опис статусів;
- описова інформація про кожному запиті на зміну;
- статус зміни;
- описова інформація про кожну зміну.

Складність процесу обчислення статусів збільшується в міру розвитку розробки. Ця складність в основному виражається у швидкому зростанні обсягів даних, які записуються і обробляються.

26.2.2.5. Архівування, аудити та огляди конфігурацій

Як вже говорилося, процедура архівування повинна забезпечувати восстанавливаемость (зберігання або обчислення) будь запитаної версії ОКУ.

Збереження даних передбачає не тільки можливість відновлення шуканої версії ОКУ в усі час дії процесу конфігураційного управління, а й захист даних проекту від несанкціонованого доступу. Іншими словами, зміни об'єкта можуть проводитися тільки тією особою, якій це зміна довірено керівництвом проекту.

Менеджер проекту повинен бути впевнений, що необхідну управління конфігурацією реалізується - іншими словами, всі прийняті зміни реалізовані, а результат являє собою те, що специфіковано в його проектної документації. Для досягнення необхідно високого рівня довіри він повинен планувати регулярні аудити та огляди процесу конфігураційного управління та його даних.

Вимоги для цих аудитів і оглядів зазвичай специфицируются в плані конфігураційного управління, але також вони можуть бути задані в планах проекту або плані гарантії якості, як це здасться відповідним. Відповідальність за нормальну реалізацію аудитів конфігурацій лежить на менеджері конфігурацій, якщо ця посада передбачена в проекті, а якщо ні - на менеджері проекту або менеджері якості.

Аудитам конфігурацій адресуються наступні питання, що відносяться до змінених об'єктів конфігурації.

- Проведено Чи зміни так, як вони специфіковані, і чи проведені відповідні технічні інспекції?
- Витримано чи слідування відповідним стандартам?

- Витримано чи проходження встановленим процедурам управління конфігураціями для запису та видачі звітів?
- Модифіковані Чи всі пов'язані з зміною об'єкти конфігурації?

26.2.2.6. Управління інструментальними засобами

Інструментальні засоби, що використовуються в проекті, повинні зберігатися в репозиторії проекту. Винятком можуть бути інструментальні засоби зовнішнього походження (наприклад, покупні або надані замовником), зберігання яких в репозиторії може виявитися неможливим через їх великого обсягу.

Зберіганню в репозиторії проекту підлягають, як мінімум, для користувача документація, виконуваний код інструментального засобу, а також інші файли, необхідні для роботи програмного засобу, такі, як бібліотеки, бази даних, параметри налаштування і т.п. Якщо використанню інструментального засобу повинна передувати процедура установки або генерації, то зберіганню підлягають усі файли, необхідні для виконання такої процедури.

Крім того, інструментальні засоби власної розробки повинні зберігатися разом з вихідним кодом і проектною документацією, а також з іншими даними, необхідними для відтворення виконуваного коду даного інструментального засобу.

Процедури проекту повинні точно ідентифікувати конфігурацію використовуваних інструментальних засобів. Модифікація цієї конфігурації допускається тільки через управління змінами. Налаштування СКУ повинні запобігати несанкціоноване зміна інструментальних засобів.

26.2.3. Рівні управління даними

Документ DO-178B в третьому розділі сьомої глави виділяє дві категорії управління даними життєвого циклу програмного проекту.Вони відповідно названі CC1 і CC2 категорії управління даними. Вибір прямування тієї чи іншої

категорії визначається додатком А. У спрощеному вигляді відмінність між категоріями можна простежити за наступними відповідностям:

ідентифікація об'єктів (CC1, CC2)

формування базових конфігурацій (CC1)

забезпечення трасуванню (CC1, CC2)

фіксація проблем (CC1)

управління змінами (фіксація даних) (CC1, CC2)

аналіз змін (CC1)

обчислення статусу конфігурації (CC1)

зміни (CC1, CC2)

забезпечення обмежень доступу (CC1, CC2)

контроль завантажувальних модулів (CC1)

зберігання даних (CC1, CC2)

26.3. Управління якістю та конфігураційне управління при розробці сертифицируемого програмного забезпечення.

Стандарт DO-178B забезпечує авіаційне співтовариство керівними вказівками, призначеними для встановлення в конструктивній манері і з прийнятним рівнем достовірності факту того, що *програмне забезпечення* бортових авіаційних систем та обладнання узгоджуються з вимогами льотної придатності. [7 , розділ 1].

Процес конфігураційного управління в даному стандарті розглянуто в розділах 7 (Процес конфігураційного управління при розробці програмного забезпечення), 11.4 (План конфігураційного управління при розробці програмного забезпечення) і 12.1.5 (Міркування про конфігураційному управлінні при розробці програмного забезпечення).

Основні цілі процесу конфігураційного управління, відповідно до вимог даного стандарту, включають в себе наступні положення [7, розділ 7.1].

- Підтримання чітко визначеною і керованої конфігурації програмного забезпечення протягом усього його життєвого циклу.
- Надання можливості повторюваною реплікації виконуваного об'єктного коду, тобто можливості заново згенерувати об'єктний код з метою перевірки або після модифікації.
- Забезпечення контролю входів і виходів процесів життєвого циклу з метою гарантії цілісності та повторюваності дій процесів.
- Визначення чітких моментів часу для проведення інспекцій, обчислення статусів та управління змінами, що досягається за допомогою управління об'єктами конфігураціями і визначення їх базових станів.
- Надання засобів і методик, що дають гарантію того, що всі виявлені проблеми беруться до уваги, а зміни протоколюються, санкціонуються і реалізовуються.
- Надання підтверджень санкціонування змін в програмному забезпеченні за допомогою управління виходами процесів *життєвого циклу програмного забезпечення.*
- Допомога в оцінці відповідності програмного забезпечення вимогам.
- Забезпечення надійного архівування, відновлення та управління об'єктами конфігурації.

В цілому процес конфігураційного управління, що охоплюється стандартом DO-178B, спрямований на підтримку цілісності даних, створюваних в ході всіх стадій життєвого *циклу* продукції. Основна специфіка процесу КУ, регламентованого даними стандартом, полягає в обліку аспектів сертифікації на льотну придатність, яку має проходити всі *програмне забезпечення,* що

використовується в бортових авіаційних системах. Дані процесу КУ використовуються як основні даних, що надаються сертифікує органу.

Сертифікуючим органам надаються індекси конфігурацій - списки унікальним чином ідентифікованих елементів (вихідних текстів, файлів даних, об'єктного і виконуваного коду), що входять до *програмне забезпечення.* Для підтвердження відповідності якості програмного забезпечення заданого рівня критичності бортового *ПО* представляються результати його тестування, проведені відповідно до вимог до даного рівня. Процес конфігураційного управління повинен забезпечувати трасуванню всіх вимог, вихідних текстів, тестів та їх результатів.

Для проведення сертифікації всі елементи конфігурації повинні мати відповідні позначки про готовність до сертифікації, а всі проблеми, що виникають в ході розробки, повинні бути тим чи іншим чином закриті. Даний аспект сертифікації забезпечується наданням інформації про ведення проблем і включенням повідомлень про проблеми і пов'язаних з ними запитів на зміну в *індекс*зміни, що підлягає сертифікації.

Всі розглянуті вище вимоги є загальними вимогам до процесу КУ, але процес розробки і виконання бортового *ПО* має одну особливість - середовищем розробки служать ті або інші комп'ютери загального призначення, а середовищем часу виконання служить відповідне системне *ПО* бортового обчислювача. Стандартом DO-178B висуваються вимоги, що враховують цю особливість: існують вимоги *щодо* забезпечення підтримки процесу завантаження об'єктного коду програми на бортовий обчислювач, який гарантіруюет завантаження виконуваного коду із застосуванням відповідних заходів безпеки, що забезпечують його незмінність і*цілісність* [7 , Розділ 4.2.8].

Таким чином, система конфігураційного управління, яка задовольняє вимогам стандарту DO-178B, повинна забезпечувати ідентифікацію та трасуванню об'єктів конфігураційного управління (ОКУ), підтримку створення базових версій (індексів конфігурацій), управління повідомленнями про проблеми і

викликаних ними змінах, облік стану конфігурації і ОКУ, а також забезпечувати процеси резервного копіювання і відновлення даних і керування процесом завантаження об'єктного коду на бортовий обчислювач.