О .О. Смагіна, С. О. Переяславська

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ
ДЕРЖАВНИЙ ЗАКЛАД
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

О.О. Смагіна, С. О. Переяславська

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

Навчальний посібник до вивчення дисципліни
для студентів спеціальності

121– „Інженерія програмного забезпечення”

Старобільськ

ДЗ „ЛНУ імені Тараса Шевченка”

2021

УДК 004.05(175.8)
С50

Р е ц е н з е н т и :

Величко В. Є. доктор педагогічних наук, кандидат фізико-математичних наук, доцент, в.о. завідувача кафедри методики навчання математики та методики навчання інформатики ДВНЗ „Донбаський державний педагогічний університет”.

Кірєєв І. Ю. – кандидат технічних наук, доцент кафедри інформаційних технологій та систем ДЗ „Луганський національний університет імені Тараса Шевченка”.

С50 **Смагіна О.О.** Якість програмного забезпечення та тестування : навч. посіб. до вивчення дисц. для студ. спец. 121 – „Інженерія програмного забезпечення”/ О. О. Смагіна, С. О. Переяславська; Держ. закл. „Луган. нац. ун-т імені Тараса Шевченка”. – Старобільськ : ДЗ „ЛНУ імені Тараса Шевченка”, 2021. – 286 с.

Навчальний посібник структуровано відповідно до розділів робочої програми курсу „Якість програмного забезпечення та тестування” для спеціальності 121 – „Інженерія програмного забезпечення” кафедри інформаційних технологій та систем ДЗ ЛНУ імені Тараса Шевченка. Посібник охоплює сучасні парадигми та технології забезпечення та контролю якості програмного забезпечення при його розробці. Також розглядаються методи тестування, верифікації і валідації; підходи до створення звітності з проблем при розробці програмного забезпечення; інструментальні засоби контролю якості.

Навчальний посібник призначений для студентів фізико-математичного та технічного профілю, учителів-предметників ліцеїв, коледжів, гімназій, слухачів курсів підвищення кваліфікації, а також для самоосвіти.

УДК 004.05(175.8)

Рекомендовано до друку Вченою радою
Луганського національного університету імені Тараса Шевченка
(протокол № 10 від 26 березня 2021 р.)

ЗМІСТ

ВСТУП

Актуальність курсу „Якість програмного забезпечення та тестування” зумовлена, перш за все тим, що розвиток програмної індустрії у світі обумовлює більш жорсткі вимоги до якості створюваних програмних продуктів. У цьому контексті найбільш актуальним стає питання забезпечення якості розробленого програмного продукту, а саме тестування, яке і є одним із найбільш ефективних способів підвищення якості. Якість програмних продуктів – це сукупність властивостей програмного забезпечення, що забезпечують її спроможність задовольняти встановлені потреби відповідно до призначення. Із технічного погляду тестування полягає у виконанні програми на деякій множині вихідних даних і порівнянні отриманих результатів із заздалегідь відомими.

Навчальна дисципліна є невід’ємною частиною циклу дисциплін, необхідних фахівцям, із вивчення понять, принципів, методологій та технологій тестування програмних продуктів; засвоєння основних понять і визначень із галузі тестування, критеріїв вибору тестів; огляду різновидів тестування; аналізу особливостей процесу й технології тестування; набуття навичок у застосуванні сучасних інформаційних технологій для аналізу та тестування інформаційних систем; створення звітної тестової документації.

Метою даного курсу є розгляд та ознайомлення студентів з існуючими способами контролю якості розробки програмного забезпечення з позицій тестування.

У результаті вивчення курсу студент повинен знати: поняття верифікації, валідації і тестування; прийоми тестування на різних фазах розробки якісного програмного продукту; умови ефективного застосування інструментальних засобів в розробці якісного програмного забезпечення; принципи розробки тестових програм і тестових наборів в програмному проєкті; етапи та елементи розробки проєктної документації для етапу тестування; необхідність та ефективність сумісної роботи проєктної команди, яка складається з розробників і тих, хто тестує розроблюване ПЗ; вміти розробляти документацію на

систему, що тестується: опис вимог до системи, тести, тестові процедури і специфікації розробника; планувати процес тестування; розробляти різні види тестів і тестуючих програм; шукати дефекти системи в процесі тестування, приймати участь в їх виправленні і модернізації додатку, який проходить тестування. За результатами виконання лабораторних робіт студент повинен вміти використовувати засоби для автоматизованого тестування, створювати звіти на основі результатів випробувань.

Навчальний посібник складається з теоретичних відомостей, лабораторних робіт, переліку питань до модульної роботи та рекомендованої літератури.

Критерії оцінювання. При вивченні дисципліни „Якість програмного забезпечення та тестування" поточний контроль знань і навичок студентів проводиться у формі захисту лабораторних робіт, контроль самостійної роботи виконується у формі співбесіди або захисту індивідуальних завдань, в якості підсумкового контролю передбачене модульне тестування. Максимальна кількість балів за результатами модульної роботи – 30 балів, лабораторних робіт – 60 балів, самостійної роботи – 10 балів. Максимальна можлива кількість балів за курс – 100 балів.

Зміст навчального посібника відповідає вимогам освітніх програм і розрахований на студентів ВНЗ, слухачів курсів, а також для самоосвіти.

Навчальний посібник може бути корисним для студентів інших спеціальностей у якості посібника для самостійного опанування технологій тестування програмного забезпечення.

Тема 1. Місце верифікації серед процесів розробки програмного забезпечення

1.1. Поняття верифікації

Верифікація – це процес визначення, чи виконують програмні засоби і їх компоненти вимоги, покладені на них в послідовних етапах життєвого циклу програмної системи, що розробляється.

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 колективній розробці програмних систем. При цьому в задачі верифікації включається контроль результатів одних розробників при передачі їх в якості вихідних даних іншим розробникам.

Для підвищення ефективності використання людських ресурсів при розробці верифікація повинна бути тісно інтегрована з процесами проєктування, розробки і супроводу програмної системи.

Заздалегідь розмежуємо поняття верифікації та налагодження. Обидва ці процеси спрямовані на зменшення помилок в кінцевому програмному продукті, однак налагодження – процес, спрямований на локалізацію і усунення помилок в системі, а верифікація – процес, спрямований на демонстрацію наявності помилок і умов їх виникнення.

Крім того, верифікація, на відміну від налагодження – контрольований і керований процес. Верифікація включає в себе аналіз причин виникнення помилок і наслідків, які спричинить їх виправлення, планування процесів пошуку помилок і їх виправлення, оцінку отриманих результатів. Все це дозволяє говорити про верифікації як про процес забезпечення

заздалегідь заданого рівня якості створюваної програмної системи.

1.2. Життєвий цикл розробки програмного забезпечення

Колективна розробка, на відміну від індивідуальної, вимагає чіткого планування робіт і 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 розроблювану систему і його формалізацію у вигляді вимог верхнього рівня. Завершується життєвий цикл розробки введенням системи в експлуатацію. Однак, 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.

У контексті даного курсу практично не будуть зачіпатися такі етапи життєвого циклу, як системна інтеграція і супровід. Для цілей курсу достатньо обмежитися спрощеним уявленням, що після реалізації коду і доведення його відповідності вимогам розробка ПО завершується.

1.3. 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

і/або замовником. В даний час існує декілька основних моделей життєвого циклу, які можуть бути адаптовані під реальну розробку.

1.3.1. Каскадний життєвий цикл

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

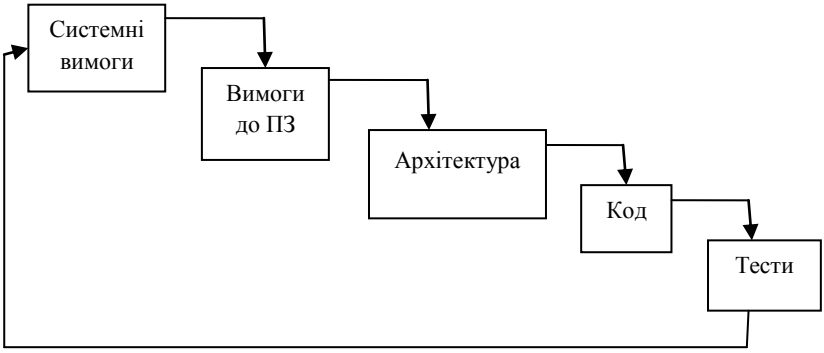


Рис. 1.1. Каскадна модель життєвого циклу

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

На другому етапі за системними вимогами складаються вимоги до програмного забезпечення – тут основна увага приділяється функціональності програмної компоненти, програмним інтерфейсам. Звичайно, всі програмні комплекси виконуються на будь-якої апаратній платформі. Якщо в ході проєкту потрібна 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 за конкретними модулями, інтерфейси між ними, заголовки окремих функцій і т.д.

На четвертому етапі пишеться програмний код, відповідний до 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.

Звичайно, що в разі чималих систем такий підхід себе не виправдовує. Робота на кожному етапі займає чимало часу, а внесення змін до первинних документів або неможливо, або викликає лавиноподібні зміни на всіх інших етапах.

Як правило, використовується модифікація каскадної моделі, яка припускає повернення на будь-який з раніше виконаних етапів. При цьому фактично виникає додаткова процедура прийняття рішення.

Дійсно якщо тести виявили невідповідність реалізації вимогам, то причина може критися: (а) - в неправильному тесті, (б) - в помилку кодування (реалізації), (в) - в невірної архітектурі системи, (г) - в некоректності вимог до програмного забезпечення і т. д. Всі ці випадки вимагають аналізу, для того щоб прийняти рішення про те, на який етап життєвого циклу треба повернутися для усунення виявленої невідповідності.

1.3.2. 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).

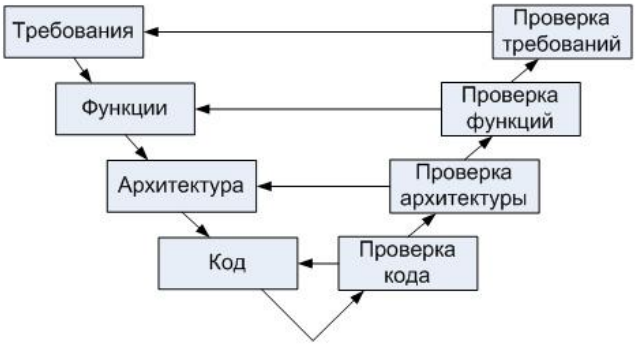


Рис. 1.2.V-подібний життєвий цикл

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.

1.3.3. Спіральний життєвий цикл

Обидва розглянутих типи життєвих циклів припускають, що заздалегідь відомі всі вимоги користувачів або, принаймні, передбачувані користувачі системи настільки кваліфіковані, що можуть висловлювати свої вимоги до майбутньої системи, коли не бачать її перед очима.

Звичайно, 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 черговий повний опис системи. Цим рішенням стала спіральна модель життєвого циклу (Рис. 1.3).

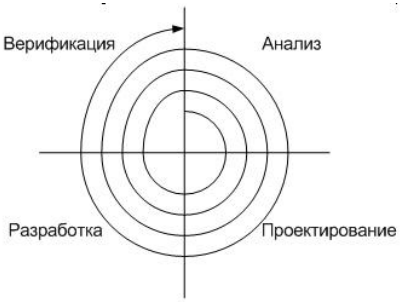


Рис. 1.3.Спіральний життєвий цикл

У спіральної моделі розробка системи відбувається повторюваними етапами – витками спіралі. Кожен виток спіралі – один каскадний або V-подібний життєвий цикл. В кінці кожного витка виходить кінцева версія системи, яка 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 доростаючи до повної.

1.3.4. 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 з 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 у всіх попередніх типах життєвого циклу інтерфейси досить чітко визначаються на самому початку розробки, оскільки заздалегідь відомі всі модулі, то при екстремальному підході інтерфейси проєктуються „на льоту", разом з модулями, що розробляються.

1.3.5. 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 в залежності 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 виробляються у взаємодії з замовником системи під час її розробки.

Таблиця 1.1. Порівняння різних типів життєвого циклу

Тип життєвого циклу	Довжина циклу	Верифікація і 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

Тип життєвого циклу	Довжина циклу	Верифікація і внесення змін	Інтеграція окремих компонент системи
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 порушувався тільки один процес – процес розробки системи. Насправді в будь-якій моделі життєвого циклу можна побачити чотири види процесів:

1. Основний процес розробки
2. Процес верифікації
3. Процес управління розробкою
4. Допоміжні (підтримуючі) процеси

Процес верифікації – процес, спрямований на перевірку коректності системи, що розроблюється та визначення ступеня її відповідності вимогам замовника.

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 проєкту. При класичних схемах досить просто побудувати ієрархічну піраміду підпорядкованість, 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 всього життєвого циклу основного процесу розробки.

Допоміжні (підтримуючі) процеси забезпечують своєчасне створення всього, що може знадобитися розробнику або кінцевому користувачеві. Сюди входить підготовка документації для користувачів, підготовка приймально-здавальних тестів, управління конфігураціями і змінами, взаємодія із замовником і т.д. Взагалі кажучи, допоміжні

процеси можуть існувати протягом всього життєвого циклу розробки, а можуть бути своєрідним поєднанням між процесом розробки та процесом експлуатації.

Основна мета процесу управління конфігураціями – забезпечення цілісності всіх даних, що виникають в процесі колективної розробки. Під цілісністю розуміється, перш за все, ідентифікованість, доступність цих даних в будь-який момент часу і недопущення несанкціонованих змін. Важливим аспектом при цьому стає процес управління змінами даних, тобто планування та затвердження будь-яких змін в проєктну документацію або програмний код, а також визначення області впливу цих змін.

Процес гарантії якості забезпечує проведення перевірок, що гарантують, що процес розробки задовольняє набору певних вимог (стандартів), необхідних для випуску якісної продукції. Фактично він перевіряє, що всі передбачені стандартами розробки процедури виконуються і при виконанні дотримуються декларовані для них правила.

Потрібно особливо відзначити, що процес гарантії якості не гарантує розробку якісної програмної системи. Він гарантує лише, що процеси розробки побудовані і виконуються таким чином, щоб не знижувати якість продукції.

Вимоги, які пред’являються до організації роботи, необхідної для випуску якісної продукції, оформлені у вигляді стандартів якості. Найбільш часто цитована і відомий стандарт якості – серія стандартів ISO 9000. На додаток до них існує стандарт, що містить вимоги до життєвого циклу розробки ПЗ – ISO 12207. За стандартом ISO/IEC 12207 в листопаді 2002 року вийшов стандарт ISO/IEC 15288, присвячений процесам життєвого циклу систем. Широта застосування програмних засобів призвела до того, що програмне забезпечення та процеси його розробки не могли розглядатися у відриві від систем, але тільки як складова частина системи і процесу її створення. У Додатках до стандарту ISO/IEC 12207 були введені мета процесу і його виходи і визначена еталонна модель процесу, що відповідає вимогам стандарту ISO/IEC 15504-2. Міжнародний

стандарт ISO/IEC 12207: 2008, являє собою перероблені і виправлені доповнення до стандарту ISO/IEC 12207 та є першим кроком в стратегії SC7 з гармонізації специфікацій, що має на меті створення повністю інтегрованого набору процесів ЖЦ систем і програмних засобів і керівництва щодо їх застосування.

1.4. Сучасні технології розробки програмного забезпечення

1.4.1. Microsoft Solutions Framework

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

Microsoft Solutions Framework є схемою для прийняття рішень з планування і 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 являє 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 проєктного планування, заснованого на управлінні проєктними групами, а також набір методик для оцінки ризиків.

MSF складається з двох моделей:

- модель проєктної групи;
- модель процесів,

і трьох дисциплін:

- управління проєктами;
- управління ризиками;
- управління підготовкою.

В MSF немає ролі „менеджер проєкту” і ієрархії керівництва, управління розробкою розподілено між керівниками окремих проєктних груп всередині колективу, виконують такі завдання:

- управління програмою;
- розробка;
- тестування;
- управління випуском;
- задоволення споживача;
- управління продуктом.

Життєвий цикл процесів в MSF поєднує водоспадну і спіральну моделі розробки: проєкт 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 з замовником.

При управлінні проєктом чітко ставиться мета, яку необхідно досягти в результаті, і враховуються обмеження, що накладаються на проєкт. 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 середу розробки, в повній мірі підтримує основні ідеї MSF – Microsoft Visual Studio Team Edition. Це перший програмний комплекс, що представляє собою не середовище розробки для індивідуальних членів колективу, а комплексне засіб підтримки колективної роботи.

До складу Visual Studio Team Edition входить спеціальна редакція для тестувальників - Team Edition for Software Testers (Рис. 1.6). Матеріали для лабораторних робіт з даного курсу орієнтовані на це середовище розробки, в той час як лекційні матеріали орієнтовані на виклад загальних принципів і методик тестування.

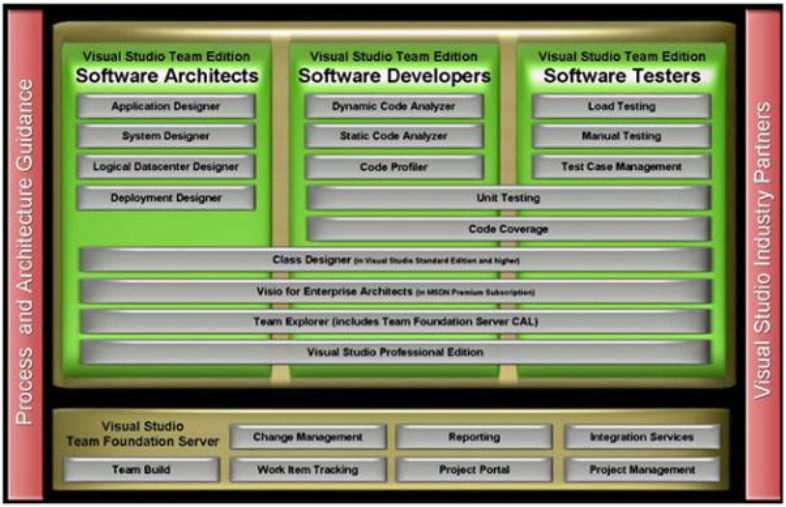


Рис. 1.6.Структура Microsoft Visual Studio Team System

1.4.2. Rational Unified Process

Rational Unified Process – це методологія створення програмного забезпечення, оформлена у вигляді бази знань, що розміщується на Web, яка забезпечена пошуковою системою.

Продукт Rational Unified Process (RUP) розроблений і підтримується Rational Software. Він регулярно оновлюється з метою врахування передового досвіду і поліпшується 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 і надає найкраще з накопиченого досвіду зі створення ПЗ, за допомогою посібників, шаблонів і настанов з користування інструментальними засобами для всіх критично важливих робіт, протягом життєвого циклу створення і

супроводу ПЗ. Забезпечуючи кожному члену групи доступ до тієї ж самої бази знань, незалежно від того, чи розробляє він вимоги, проєктує, виконує тестування або управляє проєктом – RUP гарантує, що всі члени групи використовують спільну мову моделювання та процес, мають узгоджене бачення того, як створювати ПО. В якості мови моделювання в загальній базі знань використовується Unified Modeling Language (UML), який є міжнародним стандартом.

Особливістю RUP є те, що в результаті роботи над проєктом створюються і удосконалюються моделі. Замість створення величезної кількості паперових документів, RUP спирається на розробку і розвиток семантично збагачених моделей, всебічно представляють розроблювану систему. RUP – це керівництво по тому, як ефективно використовувати UML. Стандартна мова моделювання, яка використовується усіма членами групи, робить зрозумілими для всіх опис вимог, проєктування і архітектуру системи.

RUP підтримується інструментальними засобами, які автоматизують багато елементів процесу розробки. Вони використовуються для створення і вдосконалення різних проміжних продуктів на різних етапах процесу створення ПО, наприклад, при візуальному моделюванні, програмуванні, тестуванні і т.д.

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

RUP описує, як ефективно застосовувати комерційно обґрунтовані і практично випробувані підходи до розробки ПЗ

для колективів розробників, де кожен з членів отримує переваги від використання передового досвіду в:

- ітераційній розробці ПО;
- управлінні вимогами;
- використанні компонентної архітектури;
- візуальному моделюванні;
- тестуванні якості ПЗ;
- контролі за змінам в ПЗ.

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

1.4.3. eXtreme Programming

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

XP орієнтована на:

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

Основними принципами XP є розробка невеликими ітераціями на підставі порції вимог замовника (так званих 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.

1.4.4. 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, які працюють із застосуванням продуктів і технологій компанії Rational. Супровід розробки системи і самої системи регламентується методологією RUP, проте дана технологія досить сильно орієнтована на внутрішньо фірмові інструментальні засоби.

Extreme Programming добре підходить для проєктних груп малого розміру і для невеликих систем з часто змінюваними вимогами. Основна проблема XP – супроводжуваність. У разі плинності кадрів в колективі розробників значна частина проєктної інформації може бути втрачена через практично відсутню документацію.

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.	адаптується	будь-які	Зручно (MSF + MOF)
XP	2 - 10 чол.	стандарти відсутні	будь-які	Складно (залежність від конкретних учасників колективу)

Microsoft Solutions Framework є найбільш збалансованою технологією, орієнтованою на проєктні групи малих і середніх розмірів. MSF не накладають ніяких обмежень на інструментарій, що використовується, та містить рекомендації досить загального характеру. Однак, ці рекомендації можуть бути використані для побудови конкретного процесу, що відповідає потребам колективу розробників.

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

Коли проєктна 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 і їх особистими уподобаннями, прийнятої методологією розробки, особливостями проєкту та іншими факторами. Практично в будь-якому колективі розробників можна виділити перераховані нижче ролі. Деякі з них можуть бути зовсім відсутніми, 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 проєкту і фахівцем з сертифікації або впровадження. Зазвичай замовник має право змінювати вимоги до продукту (тільки у взаємодії з менеджерами), читати проєктну і сертифікаційну документацію, що стосується нетехнічні особливості розроблюваної системи.

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

Менеджер програми. Ця роль управляє комунікаціями і взаємовідносинами в проєктній групі, є в деякому роді координатором, розробляє функціональні специфікації і управляє ними, веде графік проєкту і звітує за станом проєкту, ініціює прийняття критичних для ходу проєкту рішень.

Менеджер програми має право змінювати функціональні специфікації верхнього рівня, план-графік проєкту, розподіл

ресурсів за завданнями. Часто на практиці роль менеджера проєкту і менеджера програми виконує одна людина.

Розробник. Розробник приймає технічні рішення, які можуть бути реалізовані і використані, створює продукт, що задовольняє специфікаціям і очікуванням замовника, консультує інші ролі в ході проєкту. Він бере участь в оглядах, реалізує можливості продукту, бере участь у створенні функціональних специфікацій, відстежує та виправляє помилки за прийнятний час. В контексті конкретного проєкту роль розробника може означати, наприклад, інсталяцію програмного забезпечення, налаштування продукту або послуги. Розробник має доступ до всієї проєктної документації, включаючи документацію з тестування, має право на зміну програмного коду системи в рамках своїх службових обов'язків.

Спеціаліст з тестування. Спеціаліст з тестування визначає стратегію тестування, тест-вимоги і тест-плани для кожної з фаз проєкту, виконує тестування системи, збирає і аналізує звіти з 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 по виконанню тестів і збору інформації, розробник тестів – всю іншу роботу.

Спеціаліст з контролю якості. Ця роль належить члену проєктної 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, передбаченим проєктною документацією. Слід розрізняти фахівця з тестування та фахівця з контролю якості. Останній не є членом технічного персоналу проєкту, відповідальним за деталі і техніку роботи. Контроль якості має на увазі в першу чергу контроль самих процесів розробки і перевірку їх відповідності зазначеним в стандартах якості критеріям.

Фахівець з сертифікації. При розробці систем, до надійності яких пред'являються підвищені вимоги, перед введенням системи в експлуатацію потрібно підтвердження з боку уповноваженого органу (зазвичай державного) відповідності її експлуатаційних характеристик заданим критеріям. Така відповідність визначається в ході сертифікації системи. Спеціаліст з сертифікації може або бути представником органів, що сертифікують, включеним до складу колективу розробників, або навпаки – представляти інтереси розробників в органі, що сертифікує. Фахівець з сертифікації приводить документацію на програмну систему у відповідність до вимог органа, що сертифікує або бере участь в процесі створення документації з урахуванням цих вимог. Також фахівець з сертифікації відповідальний за всю взаємодія між колективом розробників і органом, що сертифікує. Важливою особливістю ролі є незалежність фахівця від проєктної групи на всіх етапах створення продукту. Взаємодія фахівця з членами проєктної групи обмежується менеджерами по проєкту і за програмою.

Фахівець з впровадження та супроводу. Бере участь в аналізі особливостей майданчика замовника, на якому планується проводити впровадження розроблюваної системи, виконує весь спектр робіт по встановленню та налагодженню системи, проводить навчання користувачів.

Спеціаліст 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 призначеного для користувача інтерфейсу і архітектури продукту.

Технічний редактор. Особа, яка 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, керівництва користувача).

1.6. Завдання і 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 і 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 і в специфікації вимог до функціональних компонентів низького рівня, які задовольняють вимогам високого рівня;
- специфікації вимог до функціональних компонентів ПЗ, розташовані між компонентами високого і низького рівня, задовольняють вимогам більш високого рівня;
- архітектура ПЗ і вимоги до компонентів низького рівня коректно перероблені в вихідні тексти, що їх задовольняють, програмних та інформаційних модулів;
- вихідні тексти програм і відповідний їм виконуваний код не містять помилок.

Крім того, верифікації на відповідність специфікації вимог на конкретний проєкт програмного засобу підлягають вимоги до технологічного забезпечення життєвого циклу ПЗ, а також вимоги до експлуатаційної і технологічної документації.

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

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

- клас комплексу програм, який 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 логічні і символьні дані для підготовки і відображення рішень.

Визначимо деякі поняття і визначення, пов'язані з процесом тестування як складової частини верифікації. Майерс дає наступні 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 ще невиявленої помилки.

Вдалий тест – тест, який виявляє поки ще невиявлену помилку.

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 несподіваних результатів, яке могло бути викликано дефектами, що містяться в ній.

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

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

1.7. Тестування, верифікація та валідація – відмінності в поняттях

Незважаючи на схожість, терміни „тестування”, „верифікація” і „валідація” означають різні рівні перевірки коректності роботи програмної системи. Щоб уникнути подальшої плутанини, чітко визначимо ці поняття (рис 1.7).

Тестування програмного забезпечення – вид діяльності в процесі розробки, який пов’язаний з виконанням процедур, спрямованих на виявлення (доказ наявності) помилок (невідповідностей, неповноти, двозначностей і т.д.) в поточному визначенні програмної системи, що розробляється. Процес тестування відноситься в першу чергу до перевірки коректності програмної реалізації системи, відповідності реалізації вимогам, тобто тестування – це кероване виконання програми з метою виявлення невідповідностей її поведінки і вимог.

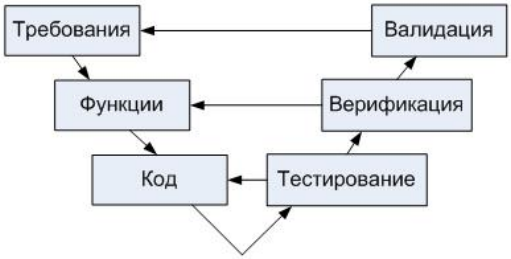


Рис. 1.7.Тестування, верифікація та валідація

Верифікація програмного забезпечення – більш загальне поняття, ніж тестування. Метою верифікації є досягнення гарантії того, що об'єкт, який верифікується, (вимоги або програмний код) відповідає вимогам, реалізований без непередбачених функцій і задовольняє проєктним специфікаціям і стандартам. Процес верифікації включає в себе інспекції, тестування коду, аналіз результатів тестування, формування та аналіз звітів про проблеми. Таким чином, прийнято вважати, що процес тестування є складовою частиною процесу верифікації.

Валідація програмної системи – метою цього процесу є доказ того, що в результаті розробки системи ми досягли тих цілей, які планували досягти завдяки її використанню. Іншими словами, валідація – це перевірка відповідності системи очікуванням замовника.

Якщо подивитися на ці три процесу з точки зору питання, на який вони дають відповідь, то тестування відповідає на питання „Як це зроблено?" або „чи відповідає поведінка розробленої програми вимогам?", верифікація – „Що зроблено?" або „Чи відповідає розроблена система вимог?", а валідація – „Чи зроблено те, що потрібно?"або „Чи відповідає розроблена система очікуванням замовника?".

1.8. 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 1.8).

Результатом етапу розробки вимог до системи є сформульовані вимоги до системи: документи, що описують загальні принципи роботи системи, її взаємодія з „навколишнім середовищем" – користувачами системи, а також, програмними і апаратними засобами, що забезпечують її роботу. Зазвичай паралельно з вимогами до системи створюється план верифікації і визначається стратегія верифікації. Ці документи визначають загальний підхід до того, як буде виконуватися тестування, які методики будуть застосовуватися, які аспекти майбутньої системи повинні бути піддані ретельній перевірці. Ще одне завдання, яке вирішується за допомогою визначення стратегії верифікації – визначення місця різних верифікаційних процесів і їх зв'язків з процесами розробки.

Верифікаційний процес, який працює з системними вимогами – це процес валідації вимог, зіставлення їх реальним очікуванням замовника. Валідація є засобом довести не тільки коректність реалізації системи з точки зору замовника, а й коректність принципів, покладених в основу її розробки.

Рис. 1.8.Процеси і документи при розробці програмних систем

Вимоги до системи є основою для процесу розробки функціональних вимог і архітектури проєкту. В ході цього процесу розробляються загальні вимоги до програмного забезпечення системи, до функцій, які вона повинна виконувати. Функціональні вимоги часто включають в себе визначення моделей поведінки системи в штатних і позаштатних ситуаціях, правила обробки даних і визначення інтерфейсу користувача. Текст вимоги, як правило, включає в себе слова „повинна, повинен" і має структуру типу „В разі, якщо значення температури на датчику АВС досягає 30 і вище градусів Цельсія, система повинна припиняти видачу звукового сигналу". Функціональні вимоги є основою для розробки архітектури системи – опису її структури в термінах підсистем і структурних одиниць мови.

На базі функціональних вимог пишуться тест-вимоги – документи, що містять визначення ключових точок, які повинні бути перевірені для того, щоб переконатися в коректності реалізації функціональних вимог. Часто тест-вимоги починаються словами „Перевірити, що" і містять посилання на відповідні їм функціональні вимоги. Прикладом тест-вимог для наведеного вище функціонального вимоги можуть служити

„Перевірити, що в разі падіння температури на датчику АВС нижче 30 градусів Цельсія система видає попереджуючий звуковий сигнал” і „Перевірити, що в разі, коли значення температури на датчику АВС вище 30 градусів Цельсія , система не видає звуковий сигнал”.

Одна з проблем, яка виникає при написанні тест-вимог – деякі вимоги неможливо протестувати: наприклад, вимога „Інтерфейс користувача повинен бути інтуїтивно зрозумілим” неможливо перевірити без чіткого визначення того, що є інтуїтивно зрозумілим інтерфейсом. Такі неконкретні функціональні вимоги зазвичай згодом видозмінюють.

Архітектурні особливості системи також можуть служити джерелом для створення тест-вимог, які враховують особливості програмної реалізації системи. Прикладом такої вимоги є, наприклад, „Перевірити, що значення температури на датчику АВС не виходить за 255”.

На основі функціональних вимог і архітектури пишеться програмний код системи, для його перевірки на основі тест-вимог готується тест-план – опис послідовності тестових прикладів, які виконують перевірку відповідності реалізації системи вимогам. Кожен тестовий приклад містить конкретний опис значень, що подаються на вхід системи, значень, які очікуються на виході, і опис сценарію виконання тесту.

Залежно від об’єкта тестування тест-план може бути підготовлений або у вигляді програми на будь-якій мові програмування, або у вигляді вхідного файлу даних для інструментарію, який виконує систему, що тестується, і передає їй значення, зазначені в тест-плані, або у вигляді інструкцій для користувача системи, яка описує необхідні дії, які потрібно виконати для перевірки різних функцій системи.

В результаті виконання всіх тестових прикладів збирається статистика про успішність проходження тестування – відсоток тестових прикладів, для яких реальні вихідні значення співпали з очікуваними, так званих пройдених тестів. Непройденние тести є вихідними даними для аналізу причин помилок і подальшого їх виправлення.

На етапі інтеграції здійснюється складання окремих модулів системи в єдине ціле і виконання тестових прикладів, які перевіряють всю функціональність системи.

На останньому етапі здійснюється поставка готової системи замовнику. Перед впровадженням фахівці замовника спільно з розробниками проводять приймально-здавальні випробування – виконують перевірку критичних для користувача функцій відповідно до заздалегідь затвердженою програмою випробувань. При успішному проходженні випробувань система передається замовнику, в іншому випадку відправляється на доопрацювання.

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

1.9.1. Модульне тестування

Модульне тестування призначене 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.

1.9.2. 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 в спільно працюючі 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.

1.9.3. 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.

1.9.4. Тестування навантаження

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 заданому навантаженні) або окремі архітектурні рішення. З точки зору замовника системи тестування навантаження є одним із способів перевірки роботи системи в умовах, наближених до реальних.

1.9.5. 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, які не брали участь в розробці інспектується документа.

1.10. Верифікація програмного забезпечення, що сертифікується

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

Сертифікація ПЗ – процес встановлення і офіційного визнання того, що розробка ПЗ проводилася відповідно до певних вимог.

Заявник – організація, що подає заявку в відповідний орган на отримання сертифіката (відповідності, якості, придатності і т.п.).

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

Наглядовий орган – комісія фахівців, що спостерігають за процесами розробки Заявником інформаційної системи, яка сертифікується, і дають висновок про відповідність даного процесу певним вимогам, який передається на розгляд в органи сертифікації.

Сертифікація може бути спрямована на отримання сертифіката відповідності або сертифіката якості.

У першому випадку результатом сертифікації є визнання відповідності процесів розробки певними критеріями, а функціональності системи – певним вимогам.

У другому випадку результатом є визнання відповідності процесів розробки певними критеріями, що гарантує

відповідний рівень якості продукції, що випускається і його придатності для експлуатації в певних умовах. Прикладом таких стандартів може служити серія міжнародних стандартів якості ISO 9000: 2000 (ДСТУ ISO 9000-2001).

Тестування програмного забезпечення, що сертифікується, 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 відмовобезпеки системи, виявлені в процесі тестування.

Наприклад, згідно з вимогами стандарту DO-178C, для того, щоб задовольнити цілям тестування програмного забезпечення, необхідне наступне:

- тести, в першу чергу, повинні грунтуватися на вимогах до програмного забезпечення;
- тести повинні розроблятися для перевірки правильності функціонування і 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.

Також в цьому стандарті йдеться про тестування, засноване на вимогах. Встановлено, що ця стратегія найбільш ефективна при виявленні помилок. Настанови для вибору тестових прикладів, заснованих на вимогах, включають наступне:

- для досягнення цілей тестування програмного забезпечення повинні бути проведені дві категорії тестів: тести для нормальних ситуацій і тести для ненормальних (не відображені у вимогах) ситуацій;

- повинні бути розроблені спеціальні тестові приклади для вимог на програмне забезпечення та джерел помилок, властивих процесу розробки програмного забезпечення.

Метою тестів для нормальних ситуацій є демонстрація здатності програмного забезпечення давати відгук на нормальні входи і умови відповідно до вимог.

Метою тестів для ненормальних ситуацій є демонстрація здатності програмного забезпечення адекватно реагувати на ненормальні входи і умови, іншими словами, це не повинно викликати відмову системи.

Категорії відмовних ситуацій для системи встановлюються шляхом визначення небезпеки відмовної ситуації, наприклад, для літака і тих, хто в ньому знаходиться. Будь-яка помилка в програмному забезпеченні може викликати відмову, який внесе свій вклад в відмовну ситуацію. Таким чином, рівень цілісності програмного забезпечення, необхідний для безпечної експлуатації, пов’язаний з відмовними ситуаціями для системи.

Існує 5 рівнів відмовних ситуацій від несуттєвої до критично небезпечної. Згідно з цими рівнями вводиться поняття рівня критичності програмного забезпечення. Від рівня критичності залежить склад документації, що надається в сертифікуючий орган, а значить і глибина процесів розробки і верифікації системи. Наприклад, кількість типів документів і обсяг робіт по розробці системи, необхідних для сертифікації за найнижчим рівнем критичності DO-178C можуть відрізнятися на один-два порядки від кількості і обсягів, необхідних для сертифікації за найвищим рівнем. Конкретні вимоги визначає стандарт, за яким планується проводити сертифікацію.

Тема 2. Тестування програмного коду (методи і оточення)

2.1. Завдання і цілі тестування програмного коду

Тестування програмного коду – процес виконання програмного коду, спрямований на виявлення існуючих в ньому дефектів. Під дефектом тут розуміється участок програмного коду, виконання якого за певних умов призводить до несподіваної поведінки системи (тобто поведінки, що не відповідає вимогам). Несподівана поведінку системи може призводити до збоїв в її роботі та відмов, в цьому випадку говорять про суттєві дефекти програмного коду. Деякі дефекти викликають незначні проблеми, що не порушують процес функціонування системи, але ускладнюють роботу з нею. У цьому випадку говорять про середні або малозначні дефекти.

Завдання тестування при такому підході – визначення умов, при яких виявляються дефекти системи, і протоколювання цих умов. До завдань тестування зазвичай не входить виявлення конкретних дефектних ділянок програмного коду і ніколи не входить виправлення дефектів – це завдання налагодження, яке 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 в програмному коді системи. Однак, в поєднанні з процесами верифікації та валідації, спрямованими на усунення суперечливості та неповноти проєктної документації (зокрема – вимог до системи), грамотно організоване тестування дає гарантію того, що система задовольняє вимогам і веде себе відповідно до них у всіх передбачених ситуаціях.

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 і галузевих стандартів якості. 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, що мають певну вимогами функціональність і інтерфейси. Після цього окремо тестується кожен модуль – виконується модульне тестування. Потім відбувається збірка окремих модулів в більш великі 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 модульному тестування, особливості інтеграційного і системного тестування будуть розглянуті пізніше.

В 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 усіма необхідними даними, використовується тестове оточення.

Завдання тестового оточення – створити середовище виконання для модуля, емулювати всі зовнішні інтерфейси, до яких звертається модуль.

Типова процедура тестування полягає в підготовці і 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 тестованому модулі, або в тестовому оточенні, або в описі тесту.

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

При тестуванні часто буває необхідно враховувати не тільки вимоги до системи, але і структуру програмного коду модуля, що тестується. В цьому випадку тести складаються таким чином, щоб детектувати типові помилки програмістів, викликані неправильною інтерпретацією вимог. Застосовуються перевірки граничних умов, перевірки класів еквівалентності. Відсутність в системі можливостей, не заданих вимог, гарантовано різними оцінками покриття програмного коду тестами, тобто оцінками того, який відсоток тих чи інших мовних конструкцій виконаний в результаті виконання всіх тестових прикладів.

2.2. Методи тестування

2.2.1. Чорний ящик

Основна ідея в тестуванні системи як чорного ящика полягає в тому, що всі матеріали, які доступні тестувальників, – 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 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 в разі подачі невірних вхідних значень. В ідеальній ситуації всі варіанти критичних ситуацій повинні бути описані в вимогах на систему і тестувальникам залишається тільки вигадувати конкретні перевірки цих вимог. Однак 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 – зміну 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 з огляду на їх неповноту – 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 чорного ящика називають також тестуванням за вимогами, тому що це єдине джерело інформації для побудови тест-плану.

2.2.2. 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 внутрішньої структури – бачить її програмний код.

Доступність програмного коду розширює можливості тестувальника тим, що він може бачити відповідність вимог ділянкам програмного коду і визначати тим самим, чи на весь 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.

2.2.3. Тестування моделей

Тестування моделей знаходиться трохи осторонь від класичних методів верифікації програмного забезпечення. Причина перш за все в тому, що об’єкт тестування – не система, як така, а її модель, спроєктована формальними засобами. Якщо не брати до уваги питання перевірки коректності і 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.

Однак тестування моделей не отримало 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.

2.2.4. 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 модуля 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.

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

2.3. Тестове оточення

Основний обсяг тестування практично будь-якої складної системи зазвичай виконується в автоматичному режимі. Крім того, тестована 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 деяке середовище, яка забезпечить запуск і виконання тестового модуля, 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. 2.1).

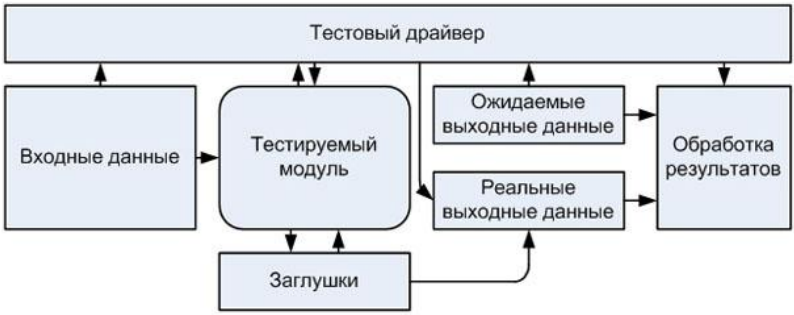


Рис. 2.1.Узагальнена схема середовища тестування

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 тестовий 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 тестового модуля і адекватно моделює поведінку системи.

2.3.1. Драйвери і заглушки

Тестове оточення для програмного коду на структурних мовах програмування складається з двох компонентів – драйвера, який забезпечує запуск і виконання тестового модуля, 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 неправильне 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 тестовий модуль 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 ПЗ значення і передавати їх у драйвер.

Для тестування програмного коду, написаного на процедурній мові 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 управління при його виклику.

Функції-заглушки можуть поміщатися в той же файл вихідного коду, що і основний текст драйвера. Імена і параметри заглушок повинні збігатися з іменами і параметрами функцій, що «заглушаються», реальної системи. Ця вимога важлива не стільки з точки зору коректної збірки системи (при складанні тестового драйвера та тестового ПЗ 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 тестованому 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, що зчитується 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 досить велику ділянку оперативної пам'яті, в якій і буде проводитися 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 трасувальні повідомлення і переміщує передане значення в пам'ять, емулює ППЗУ (функція NV_Write), або повертає за посиланням значення, яке зберігається в пам'яті, що емулює ППЗУ (функція NV_Read).

Схема взаємодії тестованого ПЗ (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. 2.2 і рис. 2.3.

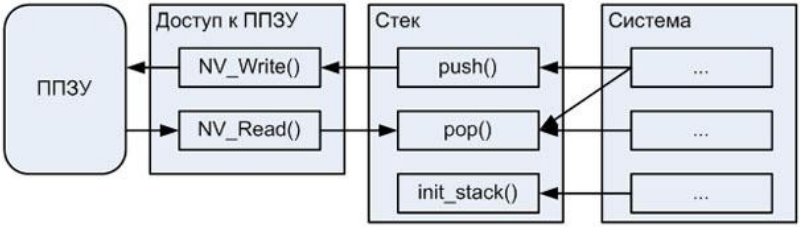


Рис. 2.2.Схема взаємодії частин реальної системи

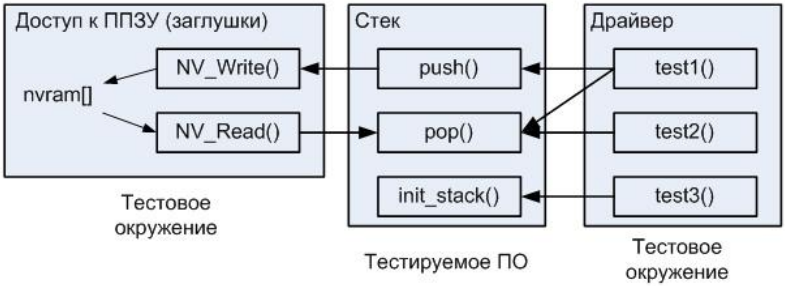


Рис. 2.3.Схема взаємодії тестового оточення та тестового ПЗ

2.3.2. 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 тестованим об'єктом є функція, то в об'єктно-орієнтованому ПЗ мінімальним об'єктом є клас. При застосуванні принципу інкапсуляції всі внутрішні дані класу і деяка частина його методів недоступна ззовні. В цьому випадку тестувальник позбавлений можливості звертатися в 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 так званий 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 в більшості методологій і середовищ розробки. Його класичне назва – unit testing (тестування модулів.

2. Аналогічно до попереднього 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 методів 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 в 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 можна створювати заглушки, тому що 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.

2.3.3. 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 проєктування програмних систем, яка послідовно пройшла наступні етапи.

1. Монолітні програми, що містять в 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 до програми 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 міжпроцесну взаємодію. Такі програми утворюють програмний комплекс, призначений для вирішення загального завдання. 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, що розділяються між різними процесами. Для вирішення особливо складних завдань процеси можуть бути запущені на різних фізичних комп'ютерах і взаємодіяти через мережу. Перевага використання взаємодії між процесами полягає в ще більшій універсальності – взаємодіючі процеси можуть бути замінені незалежно один від одного при збереженні інтерфейсу взаємодії. Інша перевага полягає в тому, що обчислювальне навантаження розподіляється між процесами. Це дозволяє операційній системі управляти пріоритетами виконання окремих частин програмного комплексу, виділяючи більшу або меншу кількість ресурсів ресурсоємким процесам.

При виконанні багатьох процесів, які вирішують 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.

Для моделювання кінцевих автоматів використовуються stateflow або SDL-діаграмми, акцент в яких робиться відповідно на умовах переходу між станами і пересилаються даними.

Так, на рис 2.4 показана схема процесу прийому/передачі даних. Закругленими прямокутниками вказані стани процесу, тонкими стрілками – переходи між станами, великими стрілками – пересилаються дані. Перебуваючи в стані «Старт», процес посилає до зовнішнього світу (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 стартовий стан. У разі надходження невірних даних (наприклад, неправильного формату або з невірною контрольною сумою) процес переходить в стан «Помилка», вийти з якого можливо тільки завершенням і перезапуском процесу.

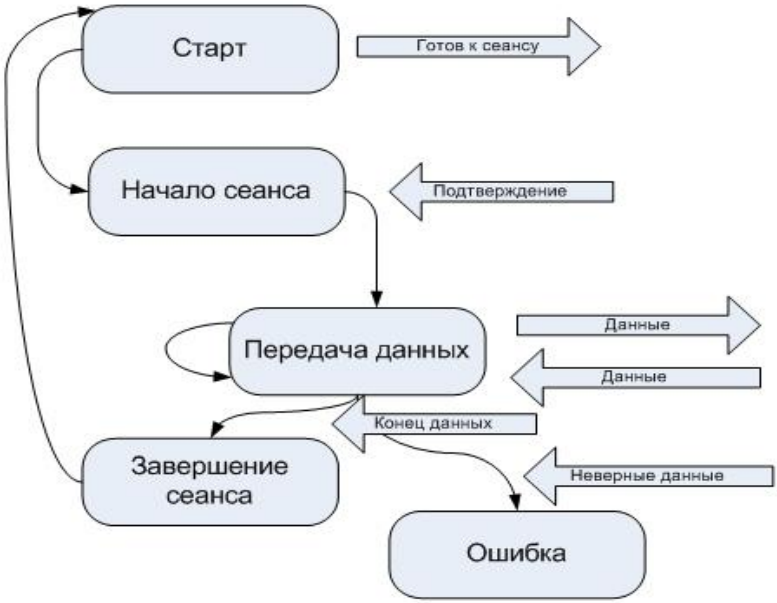


Рис. 2.4. Приклад кінцевого автомату процесу прийому-передачі даних

Тестове оточення для такого процесу також має 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 дані, що приймаються відповідно до вимог, формати даних, що передаються коректні, а також що процес під час своєї роботи дійсно проходить всі стани кінцевого автомата, що моделює його поведінку.

Тема 3. Тестування програмного коду (тестові приклади)

3.1. Тестові приклади

Тестове оточення забезпечує процес тестування необхідної інфраструктури і підтримує її. Безпосередньо для тестування крім тестового оточення необхідно визначити перевірочні завдання, які буде виконувати система або її частина. Такі перевірочні завдання називають тестовими прикладами.

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.

3.1.1. 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 впливи.

Наприклад, функціональні вимоги на програмний модуль, який розраховує і перевіряє 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. Змінна 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 змінна Empty повинна бути встановлена значення TRUE.
2. Підрахунок контрольної суми запису
а. 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 з

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.

Якщо 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.

Значення, що повертається процедурою, має дорівнювати TRUE, якщо підраховане значення дорівнює нулю.

При ненульовому значенні підрахованої контрольної суми повинно повертатися значення FALSE.

b. Установка значення змінної Empty

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

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

Приклад 3.1.

Початковий етап роботи тестувальника полягає в формуванні тест-вимог, відповідних функціональним вимогам. Основна мета тест-вимог – визначити, яка функціональність системи повинна бути протестована. У найпростішому випадку одній функціональній вимозі відповідає одна тест-вимога. 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 ініціалізації 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 записи.

d. Перевірити, що значення змінної Empty встановлюється при кожному виклику функції Set_CRC в залежності від значень полів записи, 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 контрольна сума перевіряється записи правильна, і FALSE – в іншому 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 в залежності від значень полів запису, як визначено в секції 3b функціональних вимог.

Приклад 3.2.

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

3.1.2. 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, тобто перевірити роботу основних алгоритмів. Так, для функції обчислення контрольної суми допустимими вхідними даними буде довільний 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 досить одного тестового прикладу. Але функціональні вимоги можуть визначати різні групи допустимих даних, які можуть об’єднуватися в класи еквівалентності. В цьому випадку

необхідно визначати як мінімум один тестовий приклад для одного класу еквівалентності.

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 нульової 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 приклади з мінімальними і максимальними значеннями полів у 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 в разі, 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, які 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 поведінку під час передачі їй даних, не передбачених вимогами – занадто довгих або занадто коротких рядків, невірних символів, чисел за межами обчислюваного діапазону і т.п. Невірні дані, як і допустимі, також можна розділяти на різні класи еквівалентності. Прикладом невірних даних для функції Set_CRC може служити запис з іншою структурою, переданий в функцію через приведення типів. Якщо розрахунок контрольної суми використовує імена полів запису, то контрольна сума може виявитися обчисленою невірно або може статися 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. В першу чергу, ці дефекти можуть проявлятися в тому, що не всі внутрішні дані системи після реініціалізації прийдуть в початковий стан. В результаті може статися 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 нештатне навантаження, явно не передбачене вимогами. Наприклад, є 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 є така можливість 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 стани середовища виконання.

Позаштатні стани 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 нештатний 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);
```

3.1.2.1. Граничні умови

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

- 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 в функцію

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

яка 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.

3.1.3. Перевірка робастності (виходу за межі діапазону)

Робастність системи – це ступінь її чутливості до факторів, які враховані на етапах її проєктування, наприклад, до неточності основного алгоритму, що приводить до помилок округлення при обчисленнях, збоїв у зовнішньому середовищі або до даних, значення яких знаходяться поза допустимим діапазоном. Найчастіше під робастністю 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 робастності до тестових прикладів, розглянутих у попередньому розділі, додаються ще два тестових приклади:

- Мінімальне значення - 1
- Максимальне значення + 1,

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

Таким чином, якщо зобразити допустимий інтервал як на рис 3.1, то можна бачити, що для тестування інтервальних значень досить 7 тестових прикладів – п’яти допустимих і двох на робастність.

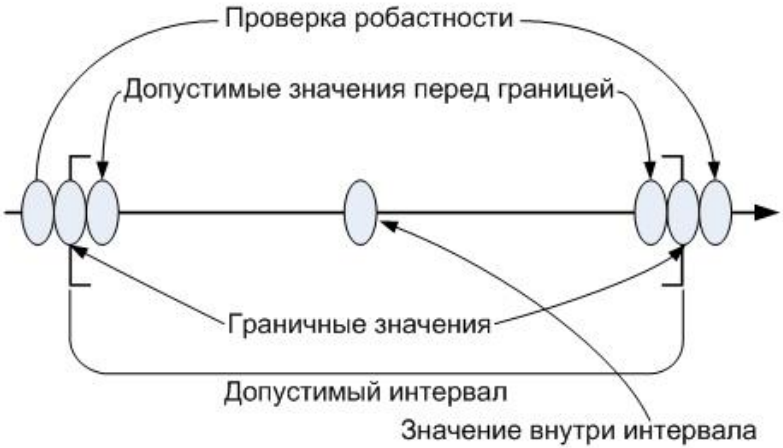


Рис. 3.1.Рекомендовані перевірочні значення

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

3.1.4. 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 робастності 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 наступними 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.

– Перевірити, що в разі присутності в імені файлу символів, які не є буквами латинського алфавіту і цифрами, система виводить повідомлення про помилку.

– Перевірити, що в тому випадку, коли довжина імені файлу перевищує 11 символів, система видає повідомлення про помилку

– Перевірити, що система не розрізняє регістр символів імені при відкритті файлу.

– 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:

4. Ім’я, яке складається з цифр і букв змішаного регістра
5. Ім’я, яке складається з цифр і букв нижнього регістру
6. Ім’я, яке складається з цифр і букв верхнього регістру
7. Ім’я, яке складається тільки з цифр
8. Ім’я, яке складається тільки з букв
9. Ім’я, яке включає знаки пунктуації

10. Ім’я, яке включає керуючі символи.

Ці класи еквівалентності ілюструють, що перевірки на границях 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.

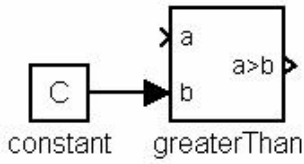
3.1.5. Тестування операцій порівняння чисел

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.

В таблиці 3.1 наведені тестові набори для блоків, що реалізують операції порівняння, в разі, коли на один з входів блоку подається константа.

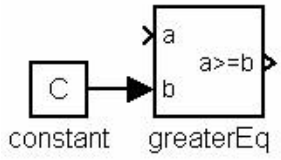
Таблиця 3.1. Блоки порівняння і визначені для них тестові набори

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



№ набору	**1**	**2**	**3***	**4**	**5**
Вхід a	b -	b +	b	min	max

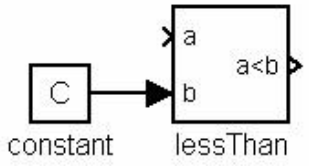
greaterEq блок.
Реалізує операцію порівняння a> = b (b - константа, на вході a може бути змінна числового типу)



№ набору	**1**	**2**	**3***	**4**	**5**
Вхід a	b -	b +	b	min	max

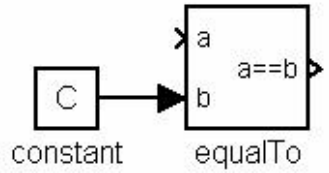
	d	d			
Вихід	F	T	F	F	T

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	T	F	F	T	F

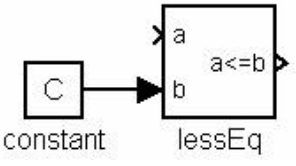
equalTo блок.
Реалізує операцію порівняння a = b (b - константа, на вході a може бути змінна числового типу)



№ набору	**1**	**2**	**3**	**4**
Вхід a	≠b	b	min	max
Вихід	F	T	F	F

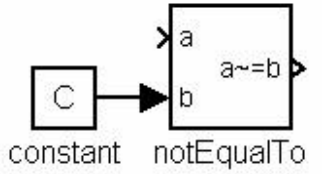
	d	d			
Вихід	F	T	T	F	T

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	T	T	F

notEqualTo блок.
Реалізує операцію порівняння a ≠ b (b - константа, на вході аможет бути змінна числового типу)



№ набору	**1**	**2**	**3**	**4**
Вхід a	≠b	b	min	max
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 дорівнює 1;

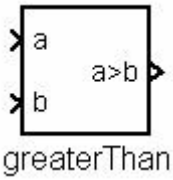
– min – мінімальне значення змінної на вході a;

– max – максимальне значення змінної на вході a.

В таблиці 3.2 наведені тестові набори для блоків, що реалізують операції порівняння, в разі, коли на обидва входи блоку подаються змінні.

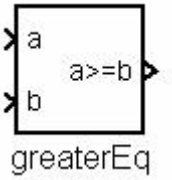
Таблиця 3.2. Блоки порівняння і визначені для них тестові набори (продовження)

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)

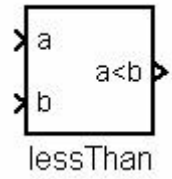


№ набору	1	2	3*	4	5
Вхід a	val	val	val	min	max
Вхід b	val + d2	val - d2	val	max	min
Вихід	F	T	T	F	T

lessThan блок.
Реалізує операцію порівняння

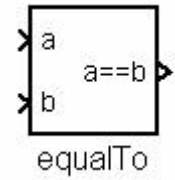
lessEq блок.
Реалізує операцію порівняння

a <b (a, b - змінні числового типу)



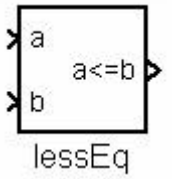
№ набору	1	2	3*	4	5
Вхід a	val	val	val	min	max
Вхід b	val + d2	val - d2	val	max	min
Вихід	T	F	F	T	F

equalTo блок.
Реалізує операцію порівняння a = b (a, b - змінні будь-якого типу)



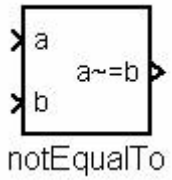
№ набору	1	2	3	4
Вхід a	val1	val	min	max
Вхід b	val2	val	max	min
Вихід	F	T	F	F

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	T	F	T	T	F

notEqualTo блок.
Реалізує операцію порівняння a ≠ b (a, b - змінні будь-якого типу)



№ набору	1	2	3	4
Вхід a	val1	val	min	max
Вхід b	val2	val	max	min
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 – змінна цілого типу, то d2 дорівнює 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.

Тема 4. Тестування програмного коду (покриття)

4.1. Тест-плани

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 опис 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 частину тестових прикладів, об'єднаних за певною ознакою.

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

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

Єдина схема ідентифікації та трасування тестових прикладів

Оскільки тестові приклади пишуться на підставі функціональних або тест-вимог, при тестуванні необхідно упевнитися, що для кожної вимоги існує хоча б один тестовий приклад. Це досягається введенням єдиної схеми ідентифікації тестових прикладів (наприклад – наскрізну нумерацію) 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 в тестах.

Внесення змін до тестових прикладів

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 необхідне створення нових тестових прикладів, які перевіряють 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 повною реініціалізацією 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, досягнутий під час попереднього тесту. Такі пов'язані тестові приклади повинні бути окремо позначені для того, щоб зберегти коректний порядок їх слідування.

4.1.1. 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 в цьому тест-плані має унікальний номер і посилання на тест-вимогу, на основі якої 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 в кожному тестовому прикладі обов'язково перераховані всі вхідні значення і очікувані вихідні значення, а також сценарій, що описує послідовність дій, які необхідно виконати тестовому 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 непорожньому 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 тесту:

1. Встановлення значення поля CRC в 12345
2. Встановлення значень полів A-F 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, заданого в специфікації вимог.

Вхідні дані: CRC = 0, A-D заповнені байтами 01010101b

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

Сценарій тесту:

1. Встановлення значення поля CRC в 0
2. Заповнення байт полів A-D 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 A-F запису при обчисленні поля CRC (підрахунку контрольної суми).

Вхідні дані: CRC = 0, A-D заповнені байтами 01010101b

Очікувані вихідні дані: A-D заповнені байтами 01010101b,

Сценарій тесту:

1. Встановлення значення поля CRC в 0
2. Заповнення байт полів AD байтами 01010101b
3. Виклик функції Set_CRC
4. Перевірка значень байт полів A-D 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.

4.2. Оцінка якості тестованого коду

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

Кожен непройденний тест вказує на потенційний дефект в тестованій 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 тестових оточенням збирається статистика виконання, яка, як правило, записується в файл звіту про виконання тестів. Існує кілька ступенів подробиць статистики виконання тестів:

– Виведення кількості пройдених і кількість не пройдених тестових прикладів, а також їх загальної кількості.

наприклад,
180 test cases passed
20 test cases failed
200 test cases total

– 1 + виведення ідентифікаторів, що не пройшли тестових прикладів. Дозволяє локалізувати тестові приклади, які потенційно 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 + виведення очікуваних і реальних вихідних даних, що не збіглися. 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 і розбіжності і 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 ступеню повноти тестування системи – рівні покриття програмного коду тестами.

4.3. Покриття програмного коду

4.3.1. 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 обсягу 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, що називається відсотком покритих тест-вимог.

Покриття вимог дозволяє оцінити ступінь повноти системи тестів по відношенню до функціональності системи, але не дозволяє оцінити повноту по відношенню до 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 певна 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 невикористаний код. Таким чином, відсутність покриття будь-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.

4.3.2. Рівні покриття

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 ускладнений аналіз покриття деяких керуючих структур.

Наприклад, для повного покриття всіх рядків наступної ділянки програмного коду на мові С досить 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, в разі 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 логічних операторів || 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 фрагменту програмного коду:

```
if (condition)
 functionA ();
else
 functionB ();
```

якщо function A () містить 99 операторів, а function B () - один оператор, то єдиного тестового прикладу, який встановлює condition в true, буде досить для досягнення необхідного рівня покриття. При цьому аналогічний тестовий приклад, який встановлює значення condition в false, дасть занадто низький рівень покриття.

За гілками умовних операторів (Decision Coverage)

Для забезпечення повного покриття за цим методом 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 за гілками потрібно як мінімум два тестових приклади.

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 за гілками ділянки програмного коду

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

необхідні два тестових приклади:

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 за гілками 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 цієї ділянки коду буде повним. Для перевірки виклику функції function1() необхідно додати ще один тестовий приклад (який, проте, не покращує ступеня покриття за гілками):

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

За компонентами логічних умов

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

Покриття за умовами (Condition Coverage)

Для забезпечення повного покриття за цим методом кожна компонента логічної умови в результаті виконання тестових прикладів повинна вживати всіх можливих значень, 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 приклади:

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

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

При цьому значення логічної умови братиме значення тільки true, таким чином, при повному покритті за умовами не досягатиме покриття за гілками.

Покриття за гілками / умовами (Condition / Decision Coverage)

Даний метод поєднує вимоги попередніх двох методів – для забезпечення повного покриття необхідно, щоб як логічна умова, так і кожна його компонента вжила всіх можливих значень.

Для покриття розглянутого вище фрагменту з умовою condition1 | condition2 буде потрібно 2 тестових приклади:

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

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

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 в разі n компонент буде потрібно 2n тестових прикладів, кожен з яких перевіряє один набір значень. Тести, необхідні для повного покриття за цим методом, дають повну таблицю істинності для логічного виразу.

Незважаючи на очевидну повноту системи тестів, що забезпечує цей рівень покриття, даний метод рідко застосовується на практиці в 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.

4.3.3. 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 за гілками / умовами (Modified Condition /Decision Coverageабо MC / DC).

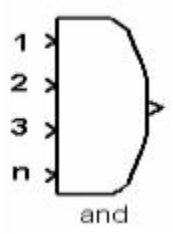
Для забезпечення повного покриття за цим методом необхідно виконання наступних умов:

- кожна логічна умова повинна приймати всіх можливих значення;
- кожна компонента логічної умови повинна хоча б один раз приймати всіх можливих значень;
- має бути показано незалежний вплив кожної з компонент на значення логічної умови, тобто вплив при фіксованих значеннях інших компонент.

Покриття за цією метрикою вимагає досить великої кількості тестів для того, щоб перевірити кожну умову, яка може вплинути на результат виразу, однак ця кількість значно менше, ніж вимагається для методу покриття за всіма умовами. В таблиці 4.1 наведені приклади тестових наборів, необхідних для тестування логічних блоків по MC / DC. Так, наприклад, для блоку OR досить n + 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, що показує незалежний вплив входів на значення виходу.

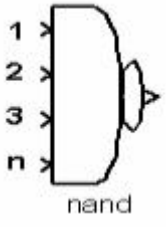
Таблиця 4.1. Логічні блоки і визначені для них тестові набори

AND блок.
Реалізує логічну функцію И для двох або більше входів



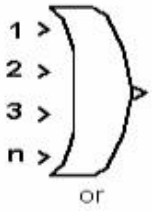
№ набору	1	2	3	4	…	n + 1
Вхід 1	T	F	T	T	…	T
Вхід 2	T	T	F	T	…	T
Вхід 3	T	T	T	F	…	T
…	…	…	…	…	…	…
Вхід n	T	T	T	T	…	F
Вихід	T	F	F	F	…	F

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

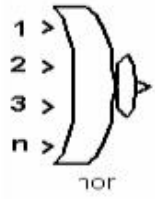


№ набора	1	2	3	4	…	n + 1
Вхід 1	T	F	T	T	…	T
Вхід 2	T	T	F	T	…	T
Вхід 3	T	T	T	F	…	T
…	…	…	…	…	…	…
Вхід n	T	T	T	T	…	F
Вихід	F	T	T	T	…	T

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



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



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

4.3.4. 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 по його усуненню. 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 належить до розряду А. Для рівня А необхідно перевірити об'єктний код, згенерований компілятором, і з'ясувати, трасується він в вихідний текст чи ні. Якщо об'єктний код не трасується в вихідний текст, повинні бути проведені перевірки об'єктного коду на предмет правильності генерації послідовності команд. Прикладом об'єктного коду, який безпосередньо не трасується в 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, яка не виконувалася в ході тестування. Для вирішення цієї обставини можуть знадобитися додаткові дії в процесі перевірки програмного забезпечення. Ця невиконувана частина коду може бути результатом:

– недоліків у формуванні тестових прикладів або тестових процедур, заснованих на вимогах. За таких обставин має 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: В цьому випадку повинні бути модифіковані вимоги на програмне забезпечення, розроблені і виконані додаткові тестові приклади і тестові процедури;

– надмірності умови. Логіка роботи такої умови повинна бути переглянута. Наприклад, в умови if (A && B ||! B) принципово неможливо перевірити, що частина умовиA && B буде дорівнювати False в разі, коли A = True і B = False, так як друга частина умови (!B)буде дорівнювати 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 switch може приймати певні значення, які він описує, і, як наслідок, гілка default ніколи не буде виконана.

Тема 5. Повторюваність тестування

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

Тестування програмної системи – не разовий захід, а постійний процес, активний протягом всього 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 викликала 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, спричинені виправленням старих.

У загальному випадку повторне виконання тестів може завершитися одним з 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, які раніше виконувалися успішно, завершується з негативним результатом. Причини цього можуть бути такі:
   – коректна 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 залишався непоміченим.

Перші дві причини помітні тільки за допомогою аналізу змін в функціональних вимогах і тест-вимог, а також поточного стану тест-планів і 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 в самому початку або під час виконання певного тестового прикладу.

Дана проблема найчастіше пов'язана зі зміною зовнішнього оточення частини системи, що тестується, яке моделює тестове оточення. Через такі зміни можуть змінюватися 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 потрібна 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 неможливе. Наприклад, коли потрібно тривалий час для виконання всіх тестів, а час, відведений на процес тестування, обмежений. 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 проєкту.

Процес, що включає в себе повторне виконання тестів, називають регресійний тестуванням. Регресійне тестування включає в себе такі 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 зміненої функціональності;
– розробка тестових прикладів і тестового оточення з використанням методик, що полегшують модифікацію при змінах в тестованій системі;
– розробка тестових прикладів, структура яких повністю виключає їх взаємний вплив за 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.

5.2. Передумови для виконання тесту, настройка тестового оточення

Вхідні дані в кожному тестовому прикладі явно задають початковий стан тестованої системи і режими її роботи при виконанні тестового сценарію.

Однак неявний вплив на виконання тесту здійснює 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 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-плані, проте необхідний 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 настройками за замовчуванням після включення (режим FACTORY_SETTINGS) і з останніми збереженими настройками після перезавантаження (режим COLD_START). При цьому при старті в режимі FACTORY_SETTINGS значення за замовчуванням присвоюються всім налаштуванням 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 налаштування встановлюються в значення за замовчуванням.

2. Перевірити, що після перезавантаження системи налаштування встановлюються в значення, які були збережені.

Необхідні як мінімум три тестових приклади з наступними сценаріями:

Тестовий приклад 1.

1. Включити систему в режимі FACTORY_SETTINGS.

2. Перевірити, що налаштування мають значення за замовчуванням.

(В реальному тест-плані тут повинні 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 налаштування мають значення за замовчуванням.

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 3.

1. Включити систему в режимі FACTORY_SETTINGS.

2. Змінити значення параметрів системи (в реальному тест-плані тут повинні бути встановлені конкретні значення змінних).

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

4. Перевірити, що налаштування мають останні введені значення

(В реальному тест-плані тут повинні 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 чимало часу, то сумарний час виконання трьох тестових прикладів буде ще більше. Якщо загальна кількість подібних тестових прикладів велика (десятки і сотні), то при такому виконанні тестів буде нераціонально витрачатися час на виконання тестових прикладів – час на ініціалізацію системи в кожному тестовому прикладі буде перевищувати сумарний час виконання «корисних» етапів сценаріїв тестових прикладів.

Для економії часу можна форматувати 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 налаштування мають значення за замовчуванням

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 2.

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

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 3.

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

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

При такій структурі тестових прикладів важлива послідовність їх виконання. Перший тестовий приклад ініціалізує систему, що тестується, і приводить її в необхідний початковий стан (запускає її в режимі FACTORY_SETTINGS), Другий і третій приклади, вважаючи, що система вже ініціалізована, перевіряють тільки її роботу при перезавантаженні.

В ході розробки системи вимоги і програмний код можуть змінитися таким чином, що при регресійному тестуванні може бути прийнято рішення про виконані тестові завдання 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 стандартні заводські при старті системи, а сама система запускається в позаштатному режимі – перезавантажується без включення.

Щоб діагностувати такі ситуації, до складу передумов 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_Система_Стартовала = FALSE

Тестовий приклад 1

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

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 2.

1. Перевірити, що прапор Флаг_Система_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення

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

3. Перевірити, що налаштування мають значення за замовчуванням

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 3.

1. Перевірити, що прапор Флаг_Система_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення

2. Змінити значення параметрів системи (в реальному тест-плані тут повинні бути встановлені конкретні значення змінних)

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

4. Перевірити, що налаштування мають останні введені значення

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

При такому підході для виконання тестових прикладів спочатку повинні бути проведені 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 до 40. Перший тестовий приклад при цьому ініціалізує систему, а решта працюють з системою, що вже стартувала. Якщо просто виконувати тестові приклади 25-40, то

їх виконання виявиться неможливим – вони не ініціалізують систему. Розумним виходом з цієї ситуації є виконання тестових прикладів 1, 25-40.

5.3. Залежність між тестовими прикладами

Для полегшення проведення регресійного тестування (і тестування взагалі) тестові приклади часто розбивають на групи. Кожна група містить набір тестових прикладів, які перевіряють 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 форматувати значення змінних або стан системи, необхідний для виконання всієї групи. Наприклад, якщо система працює в двох режимах – нормальному і 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, наступним чином:

Початкові установки тестового оточення.

Встановити значення прапора Флаг_Система_Стартовала = FALSE

Налаштування за замовчуванням для групи:

Встановити сервісний режим роботи системи

Налаштування за замовчуванням для тестового прикладу:

Обнулити показники 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_Система_Стартовала = TRUE

3. Перевірити, що налаштування мають значення за замовчуванням (в реальному тест-плані тут повинні бути перевірки конкретних значень змінних)

Група 2: Тестування перезавантаження системи (режим COLD_START)

Тестовий приклад 2.

1. Перевірити, що прапор Флаг_Система_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення

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

3. Перевірити, що налаштування мають значення за замовчуванням

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

Тестовий приклад 3.

1. Перевірити, що прапор Флаг_Система_Стартовала = TRUE, інакше перервати тестування з видачею діагностичного повідомлення

2. Змінити значення параметрів системи

(В реальному тест-плані тут повинні бути перевірки конкретних значень змінних).

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

4. Перевірити, що налаштування мають останні введені значення

(В реальному тест-плані тут повинні 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 та, 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 тестову залежність.

Приклад передбаченої тестової залежності було розглянуто в попередньому розділі – коректність виконання тестів визначалася порядком їх виконання. Така тестова залежність вимагає документування та супровід, 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 передумов.

Перший спосіб зручний при порівняно невеликій 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.

Тема 6: Документація, що супроводжує процес верифікації та тестування (тест-вимоги)

В ході роботи над проєктом зі створення будь-якої складної програмної системи створюється велика кількість проєктної документації. Основне її призначення: координація спільних дій великої кількості розробників протягом більш-менш тривалих проміжків часу – в процесі первісної розробки системи, в процесі виконання робіт з її модифікації, в процесі супроводу. Структурний склад проєктної документації в більшості проєктів практично однаковий – це вимоги до системи різного рівня (системні, функціональні та структурні), опис її архітектури, програмний код, тести і документи, які супроводжують процес впровадження (керівництва з установки, налаштування, інструкція користувача).

Оскільки верифікація програмної системи (в оптимальному випадку) виконується протягом всього життєвого циклу розробки досить великим колективом розробників, при тестуванні створюється тестова документація. Основне її призначення, крім синхронізації дій тестувальників різних рівнів, – забезпечення гарантій того, що тестування виконується відповідно до обраних критеріїв оцінки якості, а також того, що всі аспекти поведінки системи протестовані. Також тестова документація використовується при внесенні змін до системи для перевірки того, що як стара, так і нова функціональність працює коректно (рис 6.1).

Перед початком верифікації менеджером тестування (test manager) створюється документ, званий планом верифікації (або планом тестування, але це не те ж саме, що тест-план). План тестування – організаційний документ, що містить вимоги до того, як має виконуватися тестування в даному конкретному проєкті. У ньому визначаються загальні підходи до узгодження процесів розробки і верифікації, визначаються методики проведення верифікації, склад тестової документації та її взаємозв’язок з документацією розробників, терміни різних

етапів верифікації, різні ролі і кваліфікація тестувальників, необхідні для виконання всіх робіт з тестування, вимоги до інструментів тестування, а також оцінюються ризики і наводяться шляхи для їх подолання.

В даному документі також визначаються вимоги власне до тестової документації – тест-вимогам, тест-планів, звітів про виконання тестування.

Згідно з цими вимогами по системним і функціональним вимогам розробниками тестів (test procedure developers) створюються тест-вимоги – документи, в яких докладно описано те, які аспекти поведінки системи повинні бути протестовані. На підставі опису архітектури створюються низькорівневі тест-вимоги, де описуються аспекти поведінки конкретної програмної реалізації системи, які необхідно протестувати.

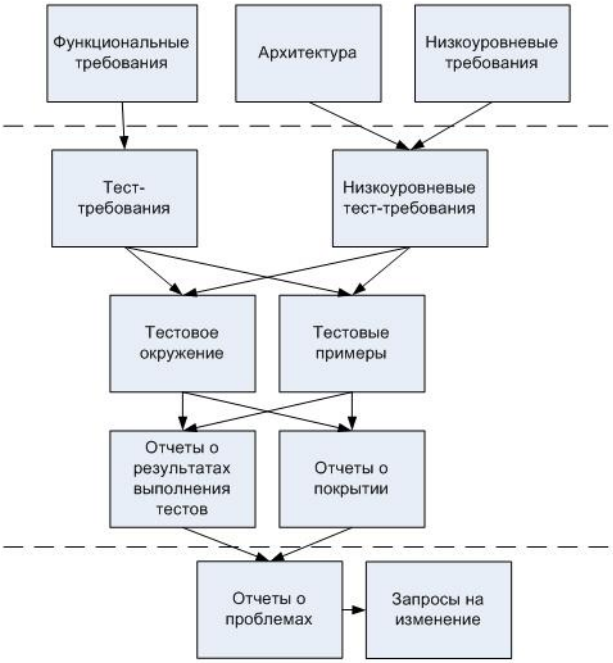


Рис. 6.1. Документація, що супроводжує процес верифікації

На підставі тест-вимог розробниками тестів (test developers) створюються тест-плани – документи, які містять докладний покроковий опис того, як повинні бути протестовані тест-вимоги.

На підставі тест-вимог і проєктної документації розробників також створюється тестове оточення, необхідне для коректного виконання тестів на тестових стендах – драйвери, заглушки, конфігураційні файли і т.п.

За результатами виконання тестів тестувальниками (testers) створюються звіти про виконання тестування (вони можуть створюватися або автоматично, або вручну), які містять інформацію про те, які невідповідності вимогам були виявлені в результаті тестування, а також звіти про покриття, що містять інформацію про те, яка частка програмного коду системи була задіяна в результаті виконання тестування.

За наявності невідповідностей створюються звіти про проблеми – документи, які направляються на аналіз до групи розробників з метою визначення причини виникнення невідповідності.

Зміни в систему вносяться тільки після всебічного вивчення цих звітів і локалізації проблем, що викликали невідповідність вимогам. Для того, щоб процес змін не вийшов з під контролю і будь-яка зміна була запротокольована, створюється запит на зміну системи. Після завершення всіх робіт за запитом на зміну процес тестування повторюється до тих пір, поки не буде досягнутий прийнятний рівень якості програмної системи. Формати різних тестових документів описані в стандартах IEEE 1012 і IEEE 829.

Слід особливо відзначити, що всі документи повинні мати унікальні ідентифікатори і зберігатися в єдиній базі документів проєкту. Це дозволить зберегти керованість процесом тестування і підтримувати необхідну якість розроблюваної системи. Немає нічого гіршого за ситуацію, коли знайдена проблема не була виправлена через те, що звіт про неї був загублений і не потрапив до розробника.

6.1. Стратегія і плани верифікації

Перший документ, який входить до складу технологічної документації процесу верифікації – стратегія тестування. Стратегія верифікації визначає загальні підходи і методики верифікації, необхідні рівні верифікації проєктної документації та програмного коду, технології та інструментальні засоби.

Інший, не менш важливий документ, який створюється перед початком процесу верифікації – план верифікації. Цей план містить послідовний опис всіх етапів верифікації, процедур на кожному етапі і зв'язків з етапами розробки.

Для кожного етапу визначається:

- типи вхідних і вихідних документів;
- загальна процедура верифікації;
- ролі і відповідальності;
- формати і угоди по ідентифікації вихідних документів;
- критерії оцінки результативності етапу.

Іноді план верифікації поділяється на окремі документи, що описують більш детально кожен з етапів, наприклад:

- план верифікації системних вимог;
- план верифікації архітектури;
- план тестування програмного коду;
- план тестування модулів і їх інтеграції;
- план системного тестування;
- план навантажувального тестування;
- план напівнатурних випробувань;
- план приймально-здавальних випробувань.

Згідно з розділом 4 стандарту IEEE 829 основне завдання плану тестування як документа – визначення меж тестування, підходу до тестування, необхідних для тестування ресурсів, плану-графіка тестування. План тестування визначає тестовані елементи і функції системи, завдання, які вирішуються в ході тестування, співробітників, відповідальних за тестування, і ризики, пов'язані з цим планом. Така форма плану тестування є достатньо повною і включає в себе не тільки технічні аспекти, пов'язані власне з описом тестових прикладів, а й організаційні, пов'язані із загальним управлінням процесом тестування. На

практиці обсяги технічних і організаційних розділів планів тестування можуть досить сильно варіюватися. Однак, мінімально необхідні елементи, які рекомендується включати в кожен план тестування – це:

– *ідентифікатор плану тестування і номер його версії*, який дозволяє однозначно знаходити потрібний план тестування і його останню актуальну версію в базі даних проєкту;

– *загальний опис тест-плану*;

– *трасування на інші документи* – стандарти, плани тестування, тест-вимоги, результати виконання тестів;

– *визначення тестованих областей системи* – зазначення частин проєктної документації, вихідних текстів, виконуваного коду, що піддаються верифікації із зазначенням типу верифікації (автоматизовані тести, формальні інспекції, тестування на моделях, напівнатурні випробування і т.п.);

– *визначення підходів до тестування* – визначення загальних методик, яких слід дотримуватися при тестуванні системи. Незважаючи на те, що більшість тестів можуть досить сильно відрізнятися, загальні методи і підходи до їх побудови можуть бути єдиними;

– *критерій успішності / неуспішності проходження тестів (pass / fail criteria)* – в даному розділі описується те, яким чином визначається успішність проходження тестів для різних частин системи;

– *тестові документи* – як правило, план тестування містить в якості додатків всі тестові документи нижчих рівнів – тест-вимоги, тест-плани, результати виконання тестів, дані про збір покриття. У разі, якщо включати ці документи до складу плану тестування видається недоцільним (наприклад, в разі їх значного обсягу), рекомендується поміщати посилання на ці документи;

– *вимоги до середовища тестування* – даний розділ описує вимоги до апаратних і програмних засобів, необхідних для проведення тестування. У разі програмно-апаратних засобів програмна система зазвичай працює на

спеціальному апаратному забезпеченні, а інструментальні засоби для тестування – на звичайних РС загального призначення. Для виконання тестування в таких умовах потрібно або використання емуляторів, або програмно-апаратний комплекс для сполучення спеціального апаратного забезпечення з РС. Крім того, як правило, до складу програмних засобів тестування входять крос-засоби розробки. У разі, якщо тестується система загального призначення, в даному розділі просто перераховуються вимоги до обладнання, необхідного для тестування, які, як правило, трохи вище, ніж вимоги до обладнання, достатнього для роботи системи;

– *людські ресурси та рівень їх підготовки* – в даному розділі наводиться склад групи тестування, необхідний для успішного завершення тестування в поставлені терміни, а також наводиться необхідні знання для різних ролей в групі;

– *план-графік тестування* – містить терміни всіх фаз тестування;

– *ризики* – містить список ризиків, які можуть перешкодити завершити тестування в строк або з необхідним рівнем якості. Як правило, для кожного ризику оцінюється ймовірність його виникнення, а також наводяться спільні шляхи, за допомогою яких можна уникнути виникнення ризику або ліквідувати його наслідки.

Стратегія і плани тестування дещо відрізняються від іншої документації, яка стосується процесу тестування: в цих документах досить багато уваги приділяється тому, як повинен бути організований процес тестування, а не тому, як тестувати саму систему.

6.2. Тест-вимоги

Тест-вимоги – основний документ для тестувальника, який визначає функціональність системи з точки зору того, що повинно бути перевірено, щоб упевнитися в її коректному функціонуванні, а також на підставі якого зовнішнього ефекту можна переконатися, що функція, що перевіряється, реалізована правильно.

Існує два підходи до написання тест-вимог – функціональний і структурний. Тест-вимоги, написані в рамках функціонального підходу, ґрунтуються на системних вимогах і вимогах до програмного забезпечення системи.

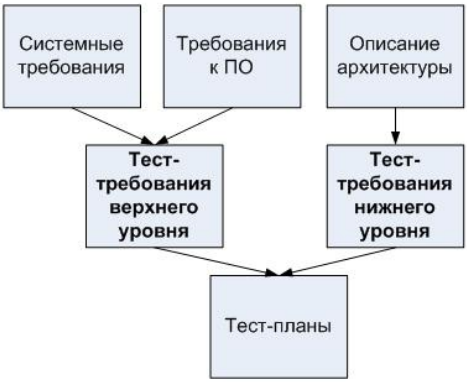


Рис. 6.2. Місце тест-вимог серед проєктної документації

Тест-вимоги, написані в рамках структурного підходу, пишуться на підставі опису архітектури системи і беруть до уваги будову вихідних текстів системи. Через таку відмінності функціональний і структурний підходи часто називають підходами чорного і білого ящиків. Структурні тест-вимоги важливі в тому випадку, коли до надійності системи пред'являються підвищені вимоги, тобто коли важливо перевірити не тільки, наскільки коректно система в цілому відпрацьовує сценарії своєї роботи (коректні і некоректні з точки зору користувача), але і як в різних нестандартних ситуаціях будуть вести себе окремі її компоненти.

На практиці майже завжди застосовуються обидва підходи до розробки тест-вимог. В результаті до складу документації проєкту включаються тест-вимоги верхнього рівня і тест-вимоги нижнього рівня, за якими складаються тест-плани (рис 6.2).

Як вже говорилося вище, тест-вимоги містять опис вимог щодо перевірки всіх основних функцій системи. Тест-вимоги повинні бути достатніми для побудови тест-плану

перевірки реалізації завдання без знайомства з її програмними текстами, тобто тест-вимоги повинні мати властивість ізоляції від внутрішньої структури системи.

Як правило, структура тест-вимог відповідає структурі розділу функціональних вимог на систему. Завдання кожного вимоги – визначення того, що треба перевірити. Техніка виконання кожної такої перевірки – завдання тест-плану. Звичайний формат опису окремого вимоги наступний:

Перевірити, що при <опис зовнішнього впливу> [відбувається] <Опис реакції програми>.

Тест-вимоги, написані в рамках функціонального підходу, зазвичай поділяють на такі групи:

- функції контролю вхідних даних;
- функції обробки помилок (введення, обчислень);
- функції отримання основного результату;
- функції обробки особливих ситуацій;
- функції оформлення і виведення результатів.

Конкретизація програмної реалізації може зажадати уточнень або розширень реакцій на різні ситуації, що виникають при вирішенні задачі. В цьому випадку рекомендується оформити додаткові тест-вимоги низького рівня для структурної перевірки системи.

Сукупність тест-вимог повинна володіти деякими важливими властивостями: повнота, верифікованість та несуперечливість.

Як правило, одній системній або функціональній вимозі відповідає мінімум одна тест-вимога. Якщо сукупність перевірок, що задаються тест-вимогами, покриває всю функціональність системи, визначену в системних вимогах і вимогах до програмного забезпечення, то говорять про повноту тест-вимог. При змінах вимог до системи для підтримки повноти повинні змінюватися і тест-вимоги.

Як системні вимоги і вимоги до ПЗ, так і тест-вимоги повинні мати властивість верифікованості. Тобто для кожної вимоги повинна існувати можливість визначити чіткий критерій перевірки – виконується ця вимога в реалізованій системі чи ні.

Прикладом вимоги, що не верифікується, може бути така «вимога»:

Система повинна мати інтуїтивно зрозумілий інтерфейс.

Очевидна «тест-вимога» буде виглядати як «Перевірити, що система має інтуїтивно зрозумілий користувальницький інтерфейс».

Без чіткого визначення критеріїв інтуїтивної зрозумілості перевірити таку вимогу за допомогою написання тестових прикладів не представляється можливим. Однак, якщо супроводити таку вимогу кількісними або якісними характеристиками інтуїтивно зрозумілого інтерфейсу – написання тестових прикладів за вимогами стає можливим. Так, серед критеріїв інтуїтивної зрозумілість можуть бути наступні: глибина вкладеності меню не більше трьох, наявність спливаючих підказок на кожному елементі управління кожної екранної форми і т.п.

При великій кількості тест-вимог і частих їх змін може виникнути ситуація, в якій різні вимоги перестають бути узгодженими. У цьому випадку такі вимоги мають взаємовиключні один одного критерії перевірки. Тобто, наприклад, в простому випадку, одна тест-вимога на призначений для користувача інтерфейс може декларувати необхідність перевірки того, що ви самі ввели пароль з довжиною не більше 16 символів, а тест-вимога до бази даних системи, що допустимий розмір пароля, що зберігається в БД – від 4 до 12 символів. У цьому випадку ці дві вимоги є суперечливими. Для того, щоб усунути цю суперечність, потрібно проводити аналіз системних і функціональних вимог з подальшою модифікацією тест-вимог. Тест-вимоги за якими складаються тест-плани для тестування системи, зазвичай мають властивість несуперечності, оскільки суперечності зазвичай усуваються на рівні верифікації проєктної документації. Однак суперечності можуть бути виявлені і пізніше, в результаті спроби створити адекватні тестові приклади.

Тема 7. Документація, що супроводжує процес верифікації та тестування (тест-плани)

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 7.1).

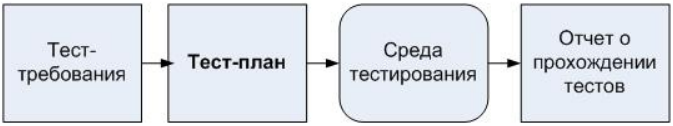


Рис. 7.1. Місце тест-планів серед проєктної документації

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 вимоги, які перевіряються цим пунктом;
- конкретний 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 формалізованій формі і служити вхідним документом для тестового оснащення, 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-плану.

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

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

Іншою формою подання тест-планів є таблиця. Ця форма найбільш часто використовується при чітко і формально визначених 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 подання тестових прикладів є визначення прикладів у вигляді кінцевого автомата. Така форма подання використовується при тестуванні протоколів

зв’язку або програмних модулів, взаємодія яких із зовнішнім світом здійснюється за допомогою обміну повідомленнями по заздалегідь заданому інтерфейсу. Модуль при цьому може бути представлений як кінцевий автомат з набором станів, а тест-план буде складатися 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.

7.1. 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 та його мета;
- посилання на частину системи, що тестується;
- посилання на використовувану проєктну документацію, зокрема тест-вимоги;
- перерахування дій сценарію;
- очікувана реакція системи на кожен пункт сценарію.

Мається на увазі, що дії сценарію повинні бути описані таким чином, щоб їх могла 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 очікуваній чи ні.

Так, невдалою очікуваною реакцією при ручному тестуванні був би 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 записи за замовчуванням при максимальному захисті системи передача не відбувається.

Тест-вимоги: 8.5.8.1, 8.5.8.2

Передумови для тесту: 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	Повинно з'явитися запрошення терміналу TRANSFER
2	Запустити процес передачі даних за допомогою введення команди SEND DATA	Повинно з'явитися запрошення *DATA TRANSFER* INITIATED і наступними двома рядками Enter your credentials ... Login:
3	Ввести ім'я облікового запису default	Повинен 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 зі звітами про проведення тестування, додаючи в таблицю опису сценарію третю і четверту колонки – «Реальний результат» і «Відповідає», в які заносяться реальна реакція системи і вказівка на збіг / розбіжність результатів відповідно. В кінці 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, суміщеного зі 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 представляють собою саме покроковий опис дій, які необхідно виконати тестовому оточенню для проведення тестування. Можлива і ближча до природної мови форма підготовки тестових прикладів. Наприклад, при тестуванні логічної функції з рівнем покриття MC/DC і 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)
```

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

7.2. Таблиці

Як вже говорилося вище, табличне представлення тестів зручно при чітко формалізованих вхідних і вихідних потоках даних системи. Наприклад, в 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 комірках 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 і супроводу великої кількості однотипних тестів. Середовище тестування, яке застосовує табличний 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 має бути передано в даному тестовому прикладі, відзначається приміщенням спеціальної мітки (наприклад, символу X) На перетині значення і тестового прикладу в таблиці:

```
INPUTS:                 | a b c d e f |
------------------------+-------------+
Power_On_Mode           |
```

COLD | X X X
WARM | X X X
Configuration_Store_Id |
0xFFFD | X X X X X X
IR_Access_Mode |
1 | X X X X
0 | X
0xFFFF | X
Reset_Mode |
0 | X X X X X X
Reset_Source |
0 | X X
1 | X
2 | X X X

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

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 з встановленими вхідними даними.

7.3. Кінцеві автомати

Форма подготововкі тест-планів у вигляді опису кінцевих автоматів зручна при тестуванні програмних модулів або систем, поведінка яких також може бути описана у вигляді кінцевого автомата. В цьому випадку процес тестування є обмін повідомленнями між двома кінцевими автоматами, що змінюють свій стан в 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 тестуючого кінцевого автомата і визначення сценаріїв переходу між станами тестових прикладів.

Розглянемо такий тест-план на наступному прикладі. Нехай тестовий 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 в стан «Прийом даних» після отримання повідомлення «Початок даних». Він може переходити з цього стану в нього ж по отриманню кожного наступного правильного блоку даних, в стан «Помилка» після отримання 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 7.2.

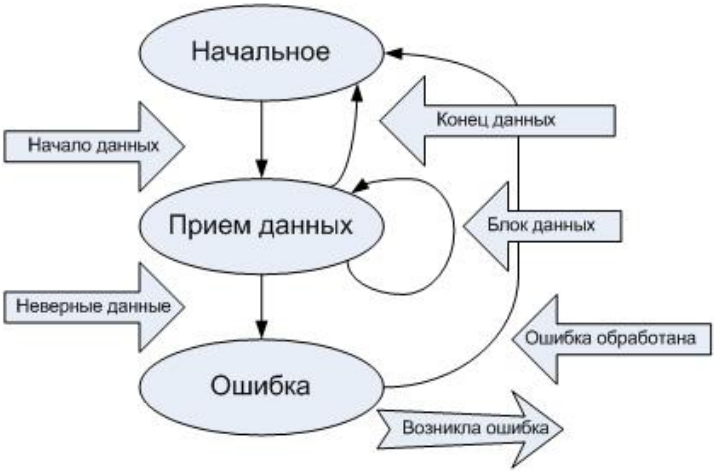


Рис. 7.2. Структурна схема тестованого кінцевого автомата

Тестуючий кінцевий автомат повинен вміти посилати повідомлення, що сприймаються тестованим автоматом і

сприймати повідомлення, що їм посилають. При цьому метою тестування буде проведення тестованого автомата по всім станам усіма можливими способами. Один з можливих варіантів побудови тестуючого 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 матиме три стани: «Початковий», «Передача даних» і «Обробка помилки». При переході з початкового стану в стан «Передача даних» він передає повідомлення «Початок даних», в стані «Передача даних» він буде передавати 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 надішле повідомлення «Кінець даних». Структурна схема такого автомата показана на рис 7.3.

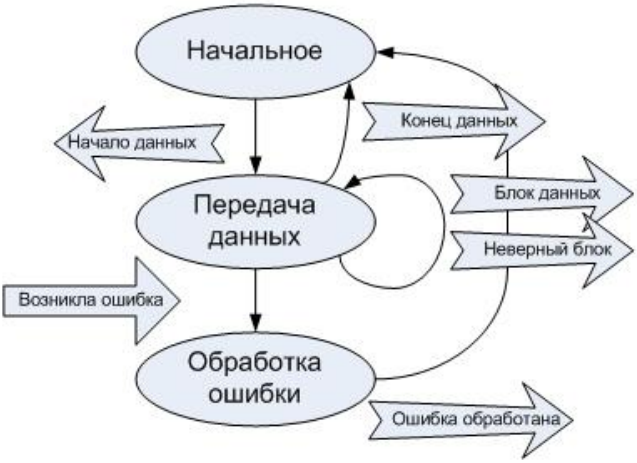


Рис. 7.3. Структурна схема тестуючого кінцевого автомата

Нижче 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)

В розділі STATES DEFINITON визначені стани тестуючого 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 в функцію повинні 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, що обробляє 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.

7.4. Генератори тестів

У деяких випадках для спрощення процедури тестування використовуються спеціальні інструментальні засоби, що автоматично генерують тестові приклади. Ці системи різняться за використовуваними методами генерації тестових прикладів, а одержувані тестові приклади розрізняються за областями застосування.

Розрізняють такі способи генерації тестових прикладів:

- за формалізованими вимогами;
- випадковим чином;
- з програмного коду.

Перший метод генерації тестових прикладів прийнятний для тестування системи як "чорного ящика", але вимагає, щоб тест-вимоги (або системні / функціональні вимоги) були підготовлені на спеціальній 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 в виділенні додаткового часу на переклад всіх вимог в 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. В цьому випадку не може йти й мови про систематизоване тестування та гарантії якості системи. Такий підхід може застосовуватися тільки при необхідності перевірити поведінку системи в 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 кожну логічну умову 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 то, 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 складне, ніж ручне написання тестів для перевірки вимог. Зазвичай рекомендується спочатку написати всі тести за вимогами, а потім, в разі необхідності, скористатися генератором тестів з програмного коду.

7.5. Звіти про проходження тестів

Звіти про проходження тестів – основне (а іноді єдине) джерело для висновку про відповідність протестованої системи вимогам. Після виконання всіх тестів, описаних в тест-плані, середовище тестування створює звіт про те, наскільки успішно система виконала ці тести. Такий звіт як мінімум містить інформацію про кожний виконаний тестовий приклад (його ідентифікатор) і результат його виконання – успіх або невдача.

За результатами аналізу звітів про проходження тестів можуть бути виявлені або дефекти в самій системі, чи неправильно складені або суперечливі вимоги. В обох випадках результати аналізу є основою для створення запитів на зміну вимог і / або коду системи. Після коректного виправлення

дефектів при регресійному тестуванні неуспішно виконані тестові приклади повинні виконатися успішно (рис 7.4).

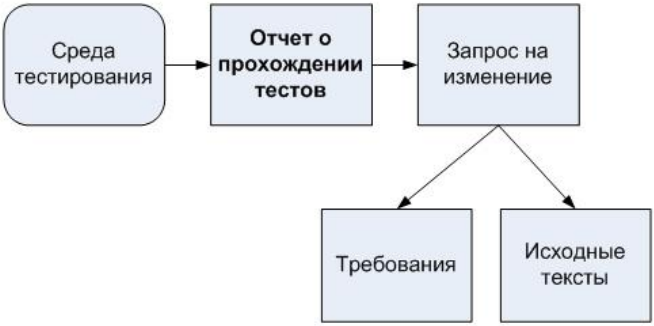


Рис. 7.4. Генерація звіту про проходження тестів і зміни за результатами його аналізу

Звіти про проходження тестів можуть служити основою для відстеження стану проєкту: якщо з плином часу кількість виявлених 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 в систему кількість дефектів неминуче зростає. Таким чином, ідеальний графік залежності кількості дефектів від часу схожий на синусоїду з амплітудою, що зменшується на кожному напівперіоді.

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. 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 системи, що розробляється 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 проєкту або тестованої системи
2. Загальний ідентифікатор групи тестових прикладів, включених до звіту
3. Ідентифікатор модуля, що тестується або групи модулів і номера 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 стенда, на якому виконувався тест
7. Імена та прізвища автора тестів і / або особи, що виконував тести.

Нижче показані два приклади таких заголовків частин звіту, що створюються 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: 7/29/16; 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.
 ---   ---------    ----------   ----------------------     --------


;  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 – завершився він успішно або в результаті його виконання були виявлені які-небудь невідповідності з очікуваним результатом. У деяких проєктах ця частина звіту може бути представлена в одній з двох форм – повній або короткій. Повна форма містить всю інформацію про тестовий приклад, 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 очікуваних і реальних вихідних значень тестового прикладу, що не співпали
- 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 короткій і повній 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.

Зазвичай 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 виконані тестові завдання 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 неуспішним виконання 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)
```

7.6. Автоматичне і ручне тестування

Деякі тестові приклади не можуть бути виконані в автоматичному режимі і тому вимагають ручної роботи тестувальника з їх виконання. Результати виконання ручних тестових прикладів можуть заноситися в той же самий документ, що і результати виконання автоматичних тестових прикладів. Особливо часто це робиться в разі, якщо і автоматичні, і ручні тести перевіряють одну і ту ж функціональну частину тестованої системи. В цьому випадку при генерації звіту про проходження тестів для ручних тестів генерується форма, в яку тестувальник заносить дані про результати проведеного ним ручного тестування. Саме 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

Тема 8. Документація, що супроводжує процес верифікації та тестування (звіти)

8.1. Звіти про покриття програмного коду

Ступінь покриття програмного коду тестами – важливий кількісний показник, що дозволяє оцінити якість системи тестів, а в деяких випадках і якість програмної системи, що тестується. 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 8.1). Формат звітів про покриття зазвичай єдиний всередині проєкту або декількох проєктів 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.

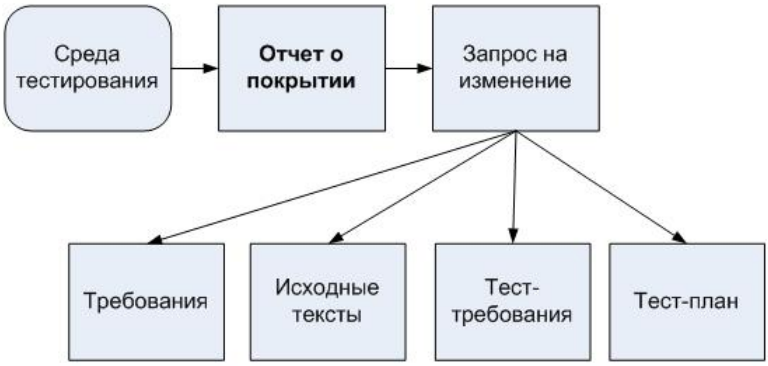


Рис. 8.1.Генерація звіту про покриття та зміни за результатами його аналізу

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 в разі нештатної роботи системи. В 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 описана в вимогах).

Типовий звіт про покриття являє собою список структурних елементів програмного коду, що покривається (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 елементів, що покриваються, 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 – загальний 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 проєкту, або вибірково для визначених функцій.

У разі, якщо обсяг 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 застосовується для покриття гілок і логічних умов, але може застосовуватися для покриття по рядках.

Приклад такого звіту наведено нижче:

Зеленим кольором позначені виконані в результаті тестування ділянки методу, червоним – невиконані.

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

8.2. Звіти про проблеми

Кожна невідповідність до вимог, знайдена тестувальником, має бути задокументована у вигляді звіту про проблему. Ймовірність виявлення і виправлення помилки, що викликала цю невідповідність, залежить від того, наскільки якісно вона задокументована. Звіти про проблеми можуть надходити не тільки від тестувальників, але і від фахівців технічної підтримки або користувачів, проте їх загальна мета – вказати на наявність проблеми в системі, яка повинна 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 до звіту 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 невідтворюваності проблеми, тобто проблеми, у якої точно не відомий спосіб її викликати. Знаходження такого способу – одна з найбільш нетривіальних завдань в 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. В цьому випадку мінімізується ймовірність непорозуміння суті проблеми.

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

Структура звіту про проблему в цілому мало відрізняється в різних проєктах, зміни зазвичай стосуються тільки порядку і імен проходження полів. Деякі поля можуть відображати специфіку даного конкретного проєкту, проте зазвичай ці поля такі:

– Об’єкт, у якому знайдена проблема. Тут міститься максимально повна інформація; для документації це назва документа, розділ, автор, версія. Для вихідних текстів це ім’я модуля, ім’я функції / методу або номера рядків, версія.

– Випуск і версія системи. Визначає місце, звідки було взято об’єкт з виявленої проблемою. Зазвичай потрібна окрема ідентифікація версії системи (а не тільки версії вихідних текстів), оскільки може виникнути плутанина з повторно виявленими проблемами. В цьому випадку, якщо проблема вже була колись виявлена розробником і потім знову з’явилася з-за того, що в систему потрапила не найостанніша версія програмного модуля, розробник може вирішити, що йому прийшов старий звіт і проблеми насправді не існує.

– Тип звіту
    1) Помилка кодування – код не відповідає вимогам.
    2) Помилка проєктування – тестувальник не згоден з проєктною документацією.
    3) Пропозиція – у тестувальника виникла ідея, як можна вдосконалити код.
    4) Розбіжність з документацією – поведінка ПО не відповідає керівництву користувача або проєктній документації або взагалі ніде не описана. При цьому у тестувальника немає підстав оголошувати, де саме знаходиться помилка.
    5) Взаємодія з апаратурою – невірна діагностика, помилка в інтерфейсі з пристроєм.
    6) Питання – тестувальник не впевнений, що це проблема, і йому потрібна додаткова інформація.

– Ступінь важливості. Суворого критерію визначення ступеня важливості не існує, і зазвичай це поле кодують від 1 (незначно) до 10 (фатально). Однак способів обґрунтованої оцінки немає – дуже складно визначити, наскільки фатальною може виявитися, наприклад, помилка в один символ в керівництві користувача.

– Суть проблеми. Коротке (не більше 2 рядків) визначення проблеми. Навіть якщо дві проблеми дуже схожі, їх опис повинен відрізнятися.

– 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.

Звичайний вигляд звіту про проблему, що відповідає даній структурі, такий:

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 рішення: _____ Дата формування рішення __ __ __

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

.....

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

.....

План перевірок, які 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 адекватне проблемі, тобто не усуває її)

8.3. Трасувальні таблиці

На кожному етапі життєвого циклу розробки програмної системи створюється різного роду проєктна документація. Як

правило, документація кожного наступного етапу створюється на базі документації попереднього етапу. Для спрощення навігації по різним документам і в т.ч. спрощення верифікації документації та самої системи часто використовуються перехресні посилання між розділами документів.

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

[ANCHOR: код]

Тут код записується у вигляді AAA_RR_NNNNNN, де AAA – тип документа (SYS, ORD і т.п.),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 обчислюваного атрибуту повинна бути визначена роль, від якої проводиться обчислення. [ANCHOR: SYS_02_000084].

Якщо виникає необхідність послатися на вимогу з того ж самого документа або з будь-якого іншого, то у посиланні вказується код якоря для відповідної вимоги. Так, наприклад, якщо посилання записується в тексті і має такий вигляд:

[REF: код],

де код має той же формат, що і для якоря, посилання на вимоги матиме такий вигляд:

Для доступу до назви ролі у формулі розрахунку значення обчислюваного атрибута повинна використовуватися мнемоніка

[RoleName]. [ANCHOR: SRD_02_000058] [REF: SYS_02_000084].

Система якорів і посилань використовує ті ж самі ідеї, що і 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:

- звичайне гіперпосилання;
- посилання вимог нижнього рівня на вимоги верхнього рівня;
- посилання на різні варіанти однієї і тієї ж вимоги, призначеної для різних варіантів системи (наприклад, для різних платформ).

В цьому випадку можна вказувати тип посилання поруч з кодом якоря, на який вона посилається. Однак, часто застосовується і інший метод організації типізованих посилань між документами – трасувальні таблиці.

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

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

8.3.1. Можливі форми трасувань таблиць

Одна з форм трасувань таблиць має на 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 стовпчику перераховані якоря вимог до програмного забезпечення, в першому рядку – якоря системних вимог. На перетині вказані типи посилань: cross-ref – звичайна інформаційне перехресне посилання; based on – вимога до ПЗ, заснована на даній системній вимозі; variant – може використовуватися або одна, або інше вимога до ПЗ в залежності від варіанту системи.

Також часто використовується більш проста форма трасувальних таблиць. У першому стовпці перераховуються якоря або номера розділів документа, який посилається на інші. У рядку містяться назви документів, на які йде посилання, а в осередках – якоря або номера розділів, на які йде посилання.

Тема 9. Модульне тестування

9.1. 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.

9.2. Завдання і цілі модульного тестування

Кожна складна програмна система складається з окремих частин – модулів, що виконують ту або 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 назву 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, які описують конкретні тестові приклади.

Основна мета модульного тестування – впевнитись у відповідності до вимог кожного окремого модуля системи перед тим, як буде проведена його інтеграція до складу системи.

При цьому в ході модульного тестування вирішуються такі основні завдання:

1. Пошук і документування невідповідностей вимогам
2. Підтримка розробки та рефакторінга низькорівневої архітектури системи і міжмодульної взаємодії
3. Підтримка рефакторингу модулів
4. Підтримка усунення дефектів і відладка.

Перше завдання – класична задача 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 властиве «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 і нелогічні або заплутані механізми роботи з модулем.

Третє завдання, пов'язана з підтримкою процесу зміни системи. Досить часто в ході розробки потрібно проводити рефакторинг модулів або їх груп – оптимізацію або повну переробку програмного коду з метою підвищення його супроводжуваності, 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.

Останнє, четверте, завдання пов'язане зі зворотним зв'язком, який отримують розробники від тестувальників у вигляді звітів про проблеми. Докладні 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, зазвичай невеликі за розміром, модульне тестування вважається

найбільш простим (хоча і досить трудомістким) етапом тестування системи. Однак, незважаючи на зовнішню простоту, з модульним тестуванням пов'язані дві проблеми.

Перша з них пов'язана з тим, що не існує єдиних принципів визначення того, що в точності є окремим модулем.

Друга полягає в розбіжностях в трактуванні самого поняття модульного тестування – розуміється чи під ним відокремлений тестування модуля, робота якого підтримується тільки тестовим оточенням, або 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 інтеграційне тестування.

9.3. Поняття модуля і його меж. 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 в списку завдань проєкту (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 тестовий модуль приймається або програмний модуль (одиниця компіляції) в разі, якщо система розробляється на процедурній мові програмування, або клас, якщо система розробляється на об'єктно-орієнтованій мові.

У разі систем, написаних на процедурних мовах, процес тестування модуля відбувається так: 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, перш за все викликаних інкапсуляцією даних і методів в класах.

У разі об'єктно-орієнтованих систем дрібніший поділ класів і використання окремих методів як тестованих модулів недоцільно, оскільки для тестування кожного методу буде потрібна розробка тестового оточення, який можна порівняти за складністю з уже 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. В ході такого тестування перевіряється взаємодія методів всередині 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 виявляються недоступними для відповідних публічних методів;

– дефектів успадкування, при наявності яких схема успадкування блокує важливі дані або методи від класів-нащадків;

– дефектів поліморфізму, при яких поліморфна поведінка класу виявляється поширена не на всі можливі класи;

– дефектів інстанціювання, при яких у новостворених об'єктах класу не встановлюються коректні значення за замовчуванням параметрів і внутрішніх даних класу.

Однак, вибір класу в якості модуля, що тестується має і ряд пов'язаних проблем.

- **Визначення ступеня повноти тестування класу**. У тому випадку, якщо в якості модуля, що тестується обраний 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 його – видалити 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 більш високому, ніж окремі класи, і встановлення адекватності тестових сценаріїв вимогам залишається на відкуп тестувальників.

- **Протоколювання станів об'єктів і 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 в кожен 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 інкапсуляцію 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 низькорівневу проєктну 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 станами здійснювалися переходи і чи всі можливі реакції перевірялися.

- **Тестування змін.** Модульні тести – потужний інструмент перевірки коректності змін, внесених до початкового коду при рефакторінгу. Однак, в результаті рефакторінга 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 класі 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 потрібна зміна драйвера і, можливо, заглушок. Але тільки модульного тестування в даному випадку недостатньо.

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 зазначити 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.

9.4. Підходи до проєктування тестового оточення

Незалежно від того, яка мінімальна одиниця вихідних кодів системи прийнята за мінімальний тестовий модуль, існує ще одна відмінність в 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 проводиться 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, який може викликати маскування дефектів (ефект парного числа помилок). Основний недолік даного методу – підвищена трудомісткість написання драйверів і заглушок.

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, які не моделюють внутрішній стан системи.

Проте, при використанні цього методу зростає складність написання тестових прикладів – для приведення системи в потрібний стан системи заглушок, як правило, потрібно тільки встановити значення тестових змінних, а для приведення в потрібний стан частини реальної системи необхідно виконати сценарій, що призводить до такого стану. Кожен тестовий приклад в цьому випадку повинен містити такий сценарій.

Крім того, при цьому підході не завжди вдається локалізувати помилки, приховані всередині модуля, які можуть проявитися при інтеграції наступних модулів.

9.5. Організація модульного тестування

Модульне тестування, з точки зору тестувальника, – це комплекс робіт з виявлення дефектів в 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 проєкту, 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 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 та календарне планування
    2. Етап визначення властивостей, які підлягають тестуванню
    3. Етап уточнення основного плану, сформованого на етапі (1)
- Фаза отримання набору тестів
    4. Етап розробки набору тестів
    5. Етап реалізації уточненого плану
- Фаза вимірювань модуля, що тестується
    6. Етап виконання тестових процедур
    7. Етап визначення достатності тестування

8. Етап оцінки результатів тестування і модуля, що тестується.

Під час етапу планування основних підходів в якості вхідних даних використовується загальний план проєкту (модульне тестування, як частина проєктних 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 і формати запису даних по проведеному тестуванні. Описується зовнішній інтерфейс тестованих модулів і їх інформаційне оточення;

– визначення вимог до повноти тестування – визначається необхідний ступінь покриття програмного коду різних ділянок модуля, що тестується, визначається підходи до класів еквівалентності (чи потрібно тестування за межами діапазону);

– визначення вимог до завершення тестування – визначаються умови, перевірка яких дозволяє стверджувати, що тестування модуля завершено, і умови, при 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 необхідна для проведення кількість ресурсів і час їх зайнятості;

– визначення загального плану-графіка робіт на підставі загального плану проєкту 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 по визначенню 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 модуля, що тестується – якщо тестований модуль може бути представлений у вигляді кінцевого автомата з певним набором станів, то кожний стан має бути ідентифіковано, а також повинні бути виділені всі вимоги, що ставляться 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 і т.п .;

– вибір елементів, що піддаються тестуванню в разі, коли не може застосовуватися повне тестування, необхідно вибрати елементи модуля, що тестуються, 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 і додаткові вимоги до ресурсів, складається детальний план-графік робіт.

За завершенням цих етапів фаза планування вважається закінченою і починається фаза розробки тестів. При цьому процес розробки тестів підпорядковується 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) .

В ході цього етапу повинні бути вирішені такі 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) – в разі досить докладних функціональних вимог і чітко прописаної концепції розробки тестів явні тестові процедури можуть і не розроблятися. Однак, при наявності необхідних ресурсів, розробка тест-вимог дозволить більш чітко інтерпретувати функціональні вимоги, що піддаються

тестуванню, – 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 (в разі необхідності) – деякі тестові приклади ґрунтуються 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).

На наступному етапі проводиться реалізація тестів (наприклад, у вигляді тестового оточення і формалізованих описів тестових прикладів). В 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, і збір і аналіз результатів тестування.

Збору підлягає наступна інформація:

- результат виконання кожного тестового прикладу (пройшов / не пройшов);
- інформація про інформаційне оточенні системи в разі, якщо тест не пройшов;

– інформація про ресурси, які потрібні були для виконання тестового прикладу.

За результатами аналізу цієї інформації складаються запити на зміну проєктної документації, програмного коду модуля, що тестується або тестового оточення.

Етапи розробки (доопрацювання), реалізації та виконання тестів тривають до тих пір, поки не буде досягнуто критерію завершення модульного тестування. Прикладом такого критерію може служити відсутність тестових прикладів, що не пройшли, при 75% покритті рядків вихідного коду.

Після припинення тестування виконуються роботи по оцінці 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 проєкту, яка зберігає всі дані, використовувані і створювані в процесі розробки програмної системи.

Тема 10. Інтеграційне тестування

Результатом тестування і верифікації окремих модулів, що складають програмну систему, має бути висновок про те, що ці модулі не суперечать один одному і відповідають вимогам. Однак окремі модулі рідко функціонують самі по собі, тому наступне завдання після тестування окремих модулів – тестування коректності взаємодії декількох модулів, об'єднаних в єдине ціле. Таке тестування називають інтеграційним. Його мета – переконатися 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 тестованій реалізації системи. З іншого боку, результати виконання інтеграційних тестів – один з основних джерел інформації для процесу поліпшення і 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 модуля, один з яких накопичує повідомлення протоколу про прийняті файли, 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 в прямому порядку, а модуль виведення 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 взаємодію модулів за допомогою інтеграційних тестів. Ключовим моментом тут є те, що в зворотному хронологічному

порядку повідомлення виводить система в цілому, тобто, перевіривши модуль виведення і виявивши, що він виводить повідомлення в прямому порядку, ми не зможемо гарантувати, що ми виявили дефект.

В результаті проведення інтеграційного тестування та усунення всіх виявлених дефектів виходить узгоджена і цілісна архітектура програмної системи, тобто можна вважати, що інтеграційне тестування – це тестування архітектури і низькорівневих функціональних вимог.

Інтеграційне тестування, як правило, являє собою ітеративний процес, при якому перевіряється функціональної все більш і більш збільшується в розмірах сукупності модулів.

10.1. Організація інтеграційного тестування

Інтеграційне тестування проводиться вже по завершенні модульного тестування для всіх інтегрованих модулів. Однак це далеко не завжди так. Існує кілька методів проведення інтеграційного тестування:

- висхідне тестування;
- монолітне тестування;
- низхідне тестування.

Всі ці методики грунтуються на знаннях про архітектуру системи, яка часто зображується у вигляді структурних діаграм або діаграм викликів функцій. Кожен вузол на такій діаграмі є програмний модуль, а стрілки між ними є 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 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. При такому підході область пошуку проблем у

тестувальника досить вузька, і тому набагато вище ймовірність правильно ідентифікувати дефект.

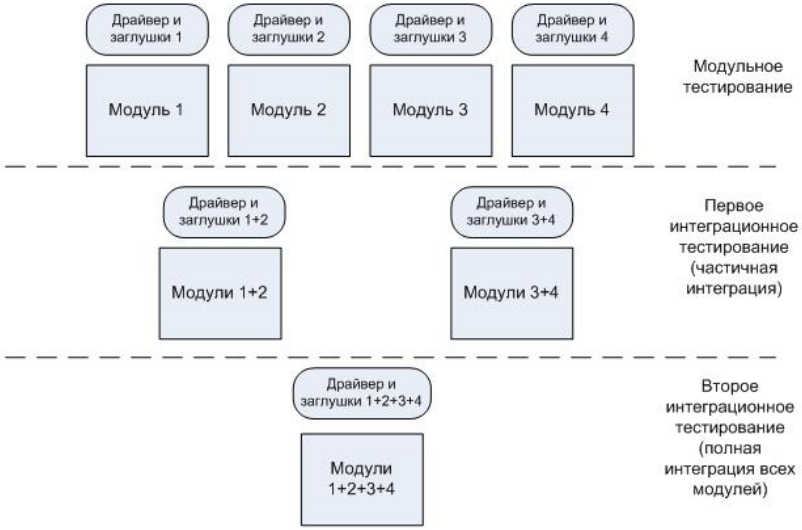


Рис. 10.1. Розробка драйверів і заглушок при висхідному інтеграційному тестуванні

Однак, у висхідного методу тестування є істотний недолік – необхідність в розробці драйвера і заглушок для модульного тестування перед проведенням інтеграційного тестування і необхідність в розробці драйвера і заглушок при інтеграційному тестуванні частини модулів системи (Рис 10.1)

З одного боку драйвери і заглушки – потужний інструмент тестування, з іншого – їх розробка потребує значних ресурсів, особливо при зміні складу інтегрованих 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. В першу чергу це пов’язано з тим, що при інтеграції модулів відпадає необхідність в 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. 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.

Монолітне тестування має ряд 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 в системі, що викликав цю проблему, буде усувати 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, що супроводжує процес тестування) перетворюється на недолік. Кожна внесене зміна вимагає повторення всіх тестів.

Низхідне тестування передбачає, що процес інтеграційного тестування рухається слідом за розробкою. Спочатку тестують тільки самий верхній керуючий рівень

системи, без модулів нижчого рівня. Потім поступово з більш високорівневими модулями інтегруються більш низькорівневі. В результаті застосування такого методу відпадає необхідність в драйверах (роль драйвера виконує більш високорівневий модуль системи), проте зберігається потреба в заглушках (Рис 10.2).

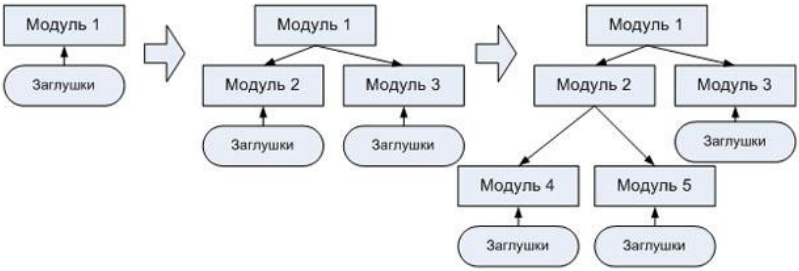


Рис. 10.2. Поступова інтеграція модулів при низхідному методі тестування

У різних фахівців в області тестування різні думки з приводу того, який з методів більш зручний при реальному тестуванні програмних систем. Йордан доводить, що низхідне тестування найбільш прийнятно в реальних ситуаціях, а Майерс вважає, що кожен з підходів має свої переваги і недоліки, але в цілому висхідний метод краще.

У літературі часто згадується метод інтеграційного тестування об'єктно-орієнтованих програмних систем, який заснований на виділенні кластерів класів, що мають разом деяку замкнуту і закінчену функціональність. За своєю суттю такий підхід не є новим типом інтеграційного тестування, просто змінюється мінімальний елемент, який отримують в результаті інтеграції. При інтеграції модулів на процедурних мовах програмування можна інтегрувати будь-яку кількість модулів за умови розробки заглушок. При інтеграції класів в кластери існує досить нечітке обмеження на закінченість функціональності кластера. Однак, навіть в 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. У літературі 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), вся послідовності викликів повинна бути виконана хоча б один раз (всі шляхи на структурній діаграмі повинні бути покриті).

На практиці найчастіше в різних частинах проєкту застосовуються всі розглянуті методи в сукупності. Кожен модуль тестують у міру готовності окремо, а потім включають в уже готову композицію. Для одних частин тестування виходить низхідним, для інших – висхідним. У зв'язку з цим представляється корисним розглянути ще один тип класифікації типів інтеграційного тестування – класифікацію за часом інтеграції.

В рамках цієї класифікації виділяють:

- тестування з пізньою інтеграцією;
- тестування з постійною інтеграцією;
- тестування з регулярною або пошаровою інтеграцією.

Тестування з пізньої інтеграцією – практично повний аналог монолітного тестування. Інтеграційне тестування при такій схемі відкладається на якомога більш пізні терміни проєкту. Цей підхід виправданий в тому випадку, якщо система є конгломератом слабко пов'язаних між собою модулів, які взаємодіють за яким-небудь стандартним інтерфейсом, визначеним поза проєктом (наприклад, в разі, якщо система складається з окремих Web-сервісів). Схематично тестування з пізньою інтеграцією може бути зображено у вигляді ланцюжка R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, 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 має на увазі, що, як тільки розробляється новий модуль системи, він відразу ж інтегрується з усією іншою системою. При цьому тести для цього модуля перевіряють як 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 нижча, 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 R-C-I-R-C-I-R-C-I, в якій фаза тестування модуля навмисно опущена і замінена на тестування інтеграції.

При тестуванні з регулярною або пошаровою 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, напрям проходу по ієрархії в цьому підході не задано.

Таблиця 10.1. Основні характеристики різних видів інтеграційного тестування

Властивість	Вид інтеграції: Висхідне	Низхідне	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

У таблиці 10.1 наведені основні характеристики розглянутих видів інтеграційного тестування. Час інтеграції характеризує момент, коли проводиться перше інтеграційне тестування і всі наступні. Частота інтеграції – наскільки часто при розробці виконується інтеграція. Необхідність в драйверах і заглушках визначена в останніх двох рядках таблиці.

10.2. Планування інтеграційного тестування

Процес організації та планування інтеграційного тестування багато в чому схожий з процесом організації модульного тестування. Однак інтеграційне тестування має ряд 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 призначеного для користувача інтерфейсу; потім інтерфейсні компоненти і компоненти роботи з БД інтегруються разом).

Складається інтеграційний тест-план, наприклад, кластерного типу, в якому для кожного кластера з інтегрованих модулів визначається наступне:

- кластери, від яких залежить даний кластер;
- кластери, які повинні бути протестовані до тестування даного кластера;
- опис функціональності тестованого кластера;
- список модулів в кластері;
- опис тестових прикладів для перевірки кластера.

Планування інтеграційного тестування повинно бути синхронізовано із загальним планом проєкту, 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 проєкту і враховувати витрати ресурсів на тестування інтеграції з уже працюючими частинами системи.

Тема 11. Системне тестування

11.1. Завдання і цілі системного тестування

Після завершення інтеграційного тестування всі модулі системи є узгодженими за інтерфейсами і функціональності. Починаючи з цього моменту можна переходити до тестування системи в цілому як єдиного об'єкта тестування , тобто 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 до них.

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 неправильною взаємодією з іншими системами, апаратним забезпеченням, невірним розподілом пам'яті, відсутністю коректного звільнення ресурсів і т.п.

Після завершення системного тестування розробка переходить в 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 вимагає значних ресурсів, для його проведення часто виділяють окремий колектив тестувальників, а найчастіше системне тестування виконується організацією, яка не пов'язана з колективом розробників і тестувальників, які виконували роботи на попередніх етапах тестування.

Системне тестування проводиться в кілька фаз, на кожній з яких перевіряється один з аспектів поведінки системи,

тобто проводиться один з типів системного тестування. Всі ці фази можуть проводитися одночасно або послідовно.

11.2. Види системного тестування

Прийнято виділяти такі види системного тестування:

- функціональне тестування;
- тестування продуктивності;
- навантажувальний або стресове тестування;
- тестування конфігурації;
- тестування безпеки;
- тестування надійності та відновлення після збоїв;
- перевірки зручності.

В 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 і 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 досвідчений 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. В 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 за всіма можливими переходами 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 в програмному коді системи і 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 запитів користувачів. 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, призначених для 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 нестачі системних ресурсів. У цьому випадку за результатами тестування проводиться доробка системи, змінюються алгоритми виділення і 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 непридатною для тестування. Навпаки, вимога «Час відгуку на запит користувача не повинен перевищувати 2 секунди» може бути протестована.

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 і т.п.). Також важливі швидкісні характеристики тестованої системи (кількість оброблених в одиницю часу запитів, часові інтервали між початком обробки кожного наступного запиту, рівномірність часу відгуку в різні моменти часу і т.п.).

Для проведення тестування продуктивності потрібна наявність генератора запитів, який подає на вхід системи потік даних, типових для сеансу роботи з нею. Тестове оточення повинно 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, однак його основне завдання – не визначити продуктивність системи, а оцінити продуктивність і стійкість системи в разі, 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, при 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 важливе при тестуванні web-систем і систем з відкритим доступом, рівень навантаження на які часто дуже складно прогнозувати.

Тестування конфігурації. Більшість програмних систем масового призначення призначено 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.

В ході тестування конфігурації перевіряється, що програмна система коректно працює на всьому підтримуваному апаратному забезпеченні та спільно з іншими програмними системами. Необхідно також перевіряти, що система продовжує стабільно працювати при зміні 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 закінчення паперу в принтері), так і позаштатні (збій живлення).

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. В ході цього тестування перевіряється, що інформація не губиться, не пошкоджується, її неможливо підмінити, а також до неї неможливо дістати несанкціонований доступ, в тому числі за допомогою використання вразливостей в самій програмній системі.

Незважаючи на те, що сертифікація – процес, наступний за верифікацією, вимоги цих стандартів можуть бути використані і при тестуванні системи. Виділяються такі групи

властивостей програмної системи, що підлягають перевірці (деякі групи властивостей укрупнені для скорочення списку):

– розмежування і контроль доступу – запобігання доступу до «чужої» інформації;

– очищення і захист пам'яті – запобігання доступу до залишкової інформації після видалення об'єктів з пам'яті;

– маркування і захист інформації, що передається в 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 проєктування та архітектури – система повинна бути спроєктована таким чином, щоб гарантувати захищеність інформації з певним рівнем впевненості;

– тестування – всі функції з забезпечення безпеки повинні бути протестовані у всіх режимах;

– цілісність і відновлення засобів захисту – система повинна мати засоби контролю коректності всіх правил розмежування доступу і системи безпеки в цілому, а також засоби їх відновлення при збої;

– документація розробника, адміністратора і користувача – всі засоби системи по забезпеченню безпеки повинні бути описані у відповідних інструкціях.

При розробці та верифікації програмної системи, яка буде піддаватися подальшій сертифікації, роботи по 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, яка перервала її роботу. При тестуванні відновлення після збоїв

імітуються збої обладнання або навколишнього програмного забезпечення або збої програмної системи, викликані зовнішніми факторами. 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 не функціональних вимог – вимоги до зручності використання призначеного для користувача інтерфейсу системи.

В результаті виконання всіх розглянутих вище видів тестування робиться висновок про функціональність і властивості системи, після чого вузькі місця системи допрацьовуються до реалізації необхідної функціональності або до досягнення системою необхідних властивостей.

При розробці масового 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 замовлення фазу альфа- і бета-тестування замінюють приймально-здавальні випробування. Під час цих випробувань замовник упевняється, що система працює відповідно до його потреб (як зафіксованих в технічному завданні на систему, так і не зафіксованих). Замовник може проводити такі випробування самостійно, виконуючи заздалегідь підготовлені тести системи, або проводити їх спільно з представниками колективу розробників. В цьому випадку тестові приклади також готуються розробниками, наприклад, на основі тестових прикладів, що використовувалися на етапі системного тестування.

Завершуються приймально-здавальні випробування або підписанням акту 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 акту система надходить в експлуатацію замовнику.

Таким чином, після проведення системного тестування та приймально-здавальних випробувань проводяться сертифікаційні випробування. Сертифікація програмного забезпечення – процес встановлення і офіційного визнання того, що розробка ПЗ проводилася відповідно до певних вимог. У процесі сертифікації відбувається взаємодія заявника, органа, що сертифікує, і наглядового органу.

Заявник – це організація, що подає заявку в відповідний сертифікуючий орган на отримання сертифіката (відповідності, якості, придатності і т.п.) вироба.

Сертифікуючий орган – організація, яка розглядає заявку заявника про проведення сертифікації ПЗ і або самостійно, або шляхом формування спеціальної комісії проводить набір процедур, спрямованих на проведення процесу сертифікації ПЗ заявника.

Наглядовий орган – комісія фахівців, що спостерігають за процесами розробки заявником інформаційної системи, що сертифікується, і дають висновок про відповідність даного процесу певним вимогам, який передається на розгляд в сертифікуючий орган.

Основний об’єкт перевірки в ході сертифікаційних випробувань – чи задовольняє процес розробки програмної системи регламенту і рекомендацій стандарту, на відповідність яким проводиться сертифікація. Така відповідність визначається за допомогою аналізу життєвого циклу системи і документів, що сертифікуються, що створюються на ключових етапах. Весь процес аналізу і ті властивості системи, які піддаються сертифікації, описується в плані сертифікаційних випробувань, який затверджується спільно заявником і органом, що сертифікує.

Лабораторна робота 1. Специфікація вимог

Мета:

- розглянути загальні підходи до тестування, вивчити архітектуру програмного комплексу «Калькулятор»;
- набути практичних навичок в області тестування і верифікації програмного забезпечення.

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *теми 1*.

Протягом усього курсу всі лабораторні роботи виконуються на одному наскрізному прикладі – програмному продукті «Калькулятор».

Припустимо, що ми є частиною колективу розробників, яка приймає замовлення на розробку програмної системи «Калькулятор» (надалі просто «Система»). Припустимо також, що інша частина колективу вже сформувала функціональні вимоги, архітектуру і написала програмний код системи. Таким чином, наша задача – це ділянка життєвого циклу системи з тестування та перевірки вимог.

Система «Калькулятор». Загальний опис

Основна мета Системи – обчислення математичних виразів з коректною структурою. Формально ця пропозиція розкрита в матеріалах до лабораторних робіт, а тут дамо деякі коментарі. У найпростішому випадку, будемо вважати коректними наступні вирази:

1

1 + 1

(1 + 1)

(1 + 1) * 2 і т.д., тобто, вирази, коректні в математичному сенсі.

Однак, серед обчислюваних калькулятором виразів є і інші приклади, ніж просто коректні математичні вирази. Це

пов’язано з деякими математичними операціями і дробовими числами, коректність обробки яких складно протестувати.

Вимоги до системи

Система повинна виконувати свою основну мету двома способами: за допомогою графічного інтерфейсу і за допомогою командного рядка.

Архітектура

В архітектурі системи виділено 3 модуля. Кожен з модулів займається певним завданням. Відповідно, Система – це взаємодія цих 3-х модулів. Розбиття Системи на модулі випливає з різної функціональності цих модулів. Розглянемо їх.

1. Модуль математичних функцій.Так як Система буде мати справу з математикою, нам буде потрібно подібний модуль. У нього включені такі функції, як додавання, множення і ін.
2. Модуль аналізу і обчислення виразів – це модуль, який займається головним завданням Системи. Аналіз і компіляція виразів – ось основні функції цього модуля. Безпосередні обчислення цей модуль не проводить, а лише викликає функції з математичного модуля.
3. Модуль графічного інтерфейсу забезпечує управління системою в графічній формі. Основні функції цього модуля – введення і виведення даних.

Взаємодія модулів показано на рис.1.1.

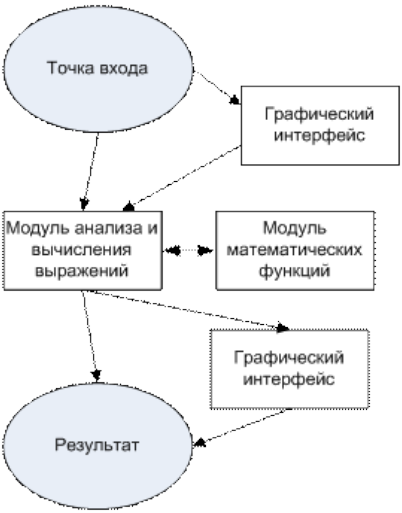


Рис. 1.1. Взаємодія модулів системи

Як видно з рисунка, передати дані в Систему можна двома способами: або через графічний модуль, або через командний рядок (останнє неявно простежується на рисунку). У будь-якому випадку, після передачі виразу Системі починає роботу модуль аналізу і обчислень, який по мірі необхідності використовує модуль математики для обчислення арифметичних функцій. Після закінчення роботи модуля аналізу і обчислень на вихід передається результат.

Програмний код

Весь програмний код Системи розбитий на модулі відповідно до архітектури. Це дозволить нам тестувати кожен модуль окремо, про що ми і будемо говорити далі.

Тестування системи

Наше основне завдання – протестувати Систему.

Якщо в загальному випадку розглядати життєвий цикл системи (наприклад, V-подібний), то наше завдання лежить десь праворуч (рис.1.2).

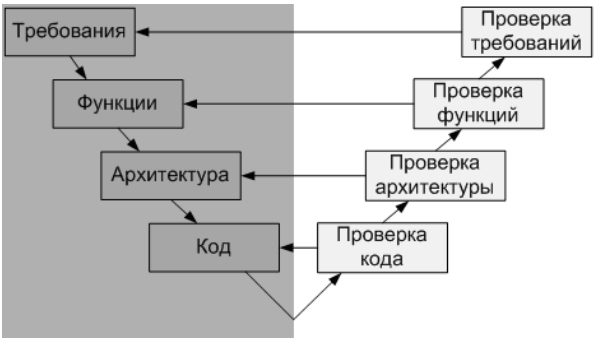


Рис. 1.2. Узагальнений V-подібний життєвий цикл розробки і верифікації програмних систем

Що стосується нашої системі, життєвий цикл можна уявити згідно з рис.1.3. Спочатку необхідно перевірити код, потім архітектуру і вимоги.

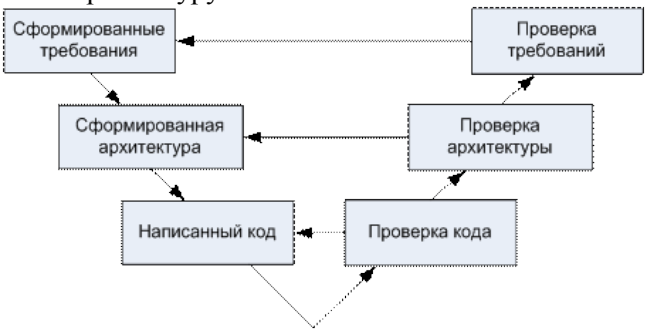


Рис. 1.3. V-подібний життєвий цикл розробки і верифікації системи «Калькулятор»

Перевірка програмного коду

На цьому етапі необхідно перевірити коректність роботи написаного коду. Для цього пропонується проводити тестування кожного модуля окремо. Тобто, ми будемо тестувати модулі окремо, підміняючи використовувані методи інших модулів «заглушками».

Наприклад, при тестуванні модуля аналізу і обчислень виразів модуль, який відповідає за обчислення простих

математичний функцій, можна замінити на модуль, що містить стандартні методи області Math. Так ми будемо точно знати, що всі помилки, виявлені при тестуванні, не мають відношення до нашої заглушки. Таким чином, замінивши всі модулі, крім тестованого, заглушками, ми зможемо стверджувати, що всі помилки, виявлені при тестуванні, будуть відноситися до тестованого модуля.

Більш того, заглушки дають нам додаткову перевагу в тестуванні. Ми можемо написати заглушки, які повертають користувачеві додаткову інформацію під час тестування. Наприклад, нам необхідно дізнатися значення певної змінної під час виконання програми. Для цього ми можемо написати заглушку, яка буде записувати значення цієї змінної в лог-файл або в консоль.

Тестування конкретних модулів.

GUI. На прикладі цього модуля можна було б дізнатися, які підходи існують для тестування сучасного графічного інтерфейсу. В рамках даного курсу цей вид тестування розглядатися не буде.

Математичні функції. Цей модуль ми будемо досліджувати як «чорний ящик» і з'ясовувати, чи дійсно реалізовані в ньому математичні функції працюють коректно.

Обчислення виразів. При тестуванні цього модуля нам належить перевірити коректність алгоритмів аналізу і компіляції математичних виразів.

При тестуванні буде використовуватися наступна послідовність дій. Спочатку ми познайомимося з методами ручного тестування в середовищі розробки при ручному тестуванні модуля аналізу та обчислення виразів. Потім ми перейдемо до модульного тестування. Після цього ми дізнаємося, що таке покриття і як вони використовуються в процесі тестування.

Перевірка архітектури

Після перевірки кожного модуля окремо ми проведемо інтеграційне тестування. На цьому етапі перевіряється, як модулі взаємодіють один з одним. За умови, що всі модулі

протестовані і помилок в них не виявлено, всі помилки на цьому етапі будуть відноситися саме до взаємодії модулів між собою.

Перевірка вимог

Після проходження всіх етапів тестування необхідно провести перевірку вимог Системи в цілому, тобто провести системне тестування. Але в рамках даного курсу цей вид тестування розглядатися не буде.

Прикладна програма. Специфікація на програму «Калькулятор. Базова версія»

Дана специфікація вимог далеко не повна, зокрема, не повна специфікація призначеного для користувача інтерфейсу, функціональних вимог. Студентам передбачається доповнити специфікацію самостійно.

1. Загальний опис

Калькулятор складається з трьох модулів – «Графічний інтерфейс», «Модуль, що аналізує і обчислює введений вираз» (AnalaizerClass.dll) і «Модуль, який реалізує математичні функції» (CalcClass.dll). Після того, як користувач введе обчислюваний вираз одним з двох способів, управління передається аналізуючому модулю, який форматує вираз, виділяючи числа і оператори, перевіряє коректність скобочної структури, а також виявляє невірні з точки зору математики конструкції (наприклад, 3 + * + 3), переводить вираз в зворотній користувацький запис, після чого обчислює вирази, використовуючи математичні функції з модуля CalcClass.

2. Опис інтерфейсу.

1. Вхідні дані

- Параметри виклику (формат командного рядка)

calc.exe [expression]

expression – математичний вираз, який задовольняє вимозі 3.2

- Стан інформаційного оточення.

В папці з програмою також знаходяться файли CalcClass.dll, AnalaizerClass.dll

2. Вихідні дані.

- Коди повернення програми.

Число і 0 на новому рядку – результат обчислень виразу.

Error: <повідомлення про помилку> і код помилки на новому рядку – повідомлення про помилку в разі невідповідності вхідного виразу вимогам 3.2.

- Стан інформаційного оточення після завершення програми.

В папці з програмою також знаходяться файли CalcClass.dll, AnalaizerClass.dll

- Повідомлення про помилки, що видаються програмою (коди помилок).

Error 01 at <i> – Неправильна дужкова структура, помилка на <i> символі

Error 02 at <i> – Невідомий оператор на <i> символі

Error 03 – Невірна синтаксична конструкція вхідного виразу

Error 04 at <i> – Два поспіль оператора на <i> символі

Error 05 – Незакінчений вираз

Error 06 – Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647

Error 07 – Занадто довгий вираз. Максмальная довжина – 65536 символів.

Error 08 – Сумарна кількість чисел і операторів перевищує 30

Error 09 – Помилка поділу на 0

3. Опис файлів, що входять в пакет калькулятора.

CalcClass.dll – бібліотека, в якій реалізовані всі необхідні математичні функції.

AnalaizerClass.dll – модуль, в якому реалізований синтаксичний аналіз виразу, а також його обчислення.

calc.exe – графічна оболонка, головний модуль.

4. Інтерфейс користувача.

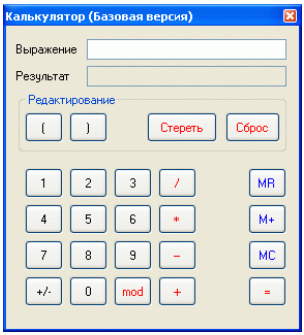


Рис. 1.4. Інтерфейс користувача системи «Калькулятор»

Клавіші "1" "2" "3" "4" "5" "6" "7" "8" "9" "0" "/" "*" "-" "+" "mod" "(" ") " – вводять відповідний символ в поле виразу. Клавіша «Скидання» очищає поле «Вираз», клавіша «Стерти» видаляє останній введений символ. Клавіша «=» починає виконання обчислень. "MR", "M +" і "MC" управляють пам'яттю калькулятора, "+/-" – тригер унарного плюса, унарного мінуса.

3. Опис архітектури

Як вже зазначалося вище, в архітектурі системи виділено 3 модуля. Кожен з модулів займається певним завданням. Відповідно, Система – це взаємодія цих 3-х модулів. Розглянемо їх докладніше.

1. Модуль математичних операцій (CalcClass.dll)

Модуль містить всі математичні функції, які використовуються в програмі.

```
/// <summary>
/// Функція складання числа a і b
/// </ summary>
/// <param name = "a"> доданок </ param>
/// <param name = "b"> доданок </ param>
/// <returns> сума </ returns>
public static int Add (long a, long b)

/// <summary>
/// функція віднімання чисел a і b
/// </ summary>
```

```
/// <param name = "a"> зменшуване </ param>
/// <param name = "b"> від'ємник </ param>
/// <returns> різниця </ returns>
public static int Sub (long a, long b)


/// <summary>
/// функція множення чисел a і b
/// </ summary>
/// <param name = "a"> множник </ param>
/// <param name = "b"> множник </ param>
/// <returns> твір </ returns>
public static int Mult (long a, long b)

/// <summary>
/// функція знаходження частного
/// </ summary>
/// <param name = "a"> ділене </ param>
/// <param name = "b"> дільник </ param>
/// <returns> приватне </ returns>
public static int Div (long a, long b)

/// <summary>
/// функція ділення по модулю
/// </ summary>
/// <param name = "a"> ділене </ param>
/// <param name = "b"> дільник </ param>
/// <returns> залишок </ returns>
public static int Mod (long a, long b)

/// <summary>
/// унарний плюс
/// </ summary>
/// <param name = "a"> </ param>
/// <returns> </ returns>
public static int ABS (long a)
```

```
/// <summary>
/// унарний мінус
/// </ summary>
/// <param name = "a"> </ param>
/// <returns> </ returns>
public static int IABS (long a)

Використовується також глобальна змінна:
/// <summary>
/// Останнє повідомлення про помилку
/// Поле і властивість для нього
/// </ summary>
private static string _lastError = "";

public static string lastError
```

Лістинг 1.1. Модуль математичних операцій

2. Модуль аналізу і обчислення виразів

Складається з наступних методів і властивостей:

```
/// <summary>
/// позиція вираження, на якій виявлена синтаксична помилка (у випадку знаходження на рівні виконання – не визначається)
/// </ summary>
private static int erposition = 0;
/// <summary>
/// Вхідний вираз
/// </ summary>
public static string expression = "";

/// <summary>
/// Показує, чи є необхідність у виведенні повідомлень про помилки.
У разі консольного запуску програми це значення – false.
/// </ summary>
```

public static bool ShowMessage = true;

/// <summary>

/// Перевірка коректності скобочной структури вхідного виразу

/// </ summary>

/// <returns> true - якщо все нормально, false - якщо є помилка </ returns>

/// метод біжить по вхідному виразу, символ за символом аналізуючи його і зважаючи на кількість скобочек. У разі виникнення помилки повертає false, а в erposition записує позицію, на якій виникла помилка.

public static bool CheckCurrency ()

/// <summary>

/// Форматує вхідний вираз, виставляючи між операторами пробіли і видаляючи зайві, а також відловлює непізнані оператори, стежить за кінцем рядка

/// також знаходить помилки на кінці рядка

/// </ summary>

/// <returns> кінцевий рядок або повідомлення про помилку, що починаються зі спец. символу & </ returns>

public static string Format ()

/// <summary>

/// Створює масив, в якому розташовуються оператори і символи, представлені в зворотньому польському записі (безскобочной)

/// На цьому ж етапі знаходяться майже всі інші помилки (див. код). По суті це компіляція.

/// </ summary>

/// <returns> масив в зворотньому польському записі

</ returns>

public static System.Collections.ArrayList CreateStack ()

/// <summary>

/// Обчислення зворотнього польського запису

/// </ summary>
/// <returns> результат обчислень або повідомлення про помилку </ returns>
public static string RunEstimate ()

/// <summary>
/// Метод, який організовує обчислення. По черзі запускає
CheckCorrncy, Format, CreateStack і RunEstimate
/// </ summary>
/// <returns> </ returns>
public static string Estimate ()

Лістинг 1.2. Модуль аналізу і обчислення виразів

3. Модуль графічного інтерфейсу забезпечує управління системою в графічній формі. Основні функції цього модуля – введення і виведення даних.

Взаємодія модулів показана на рисунку:

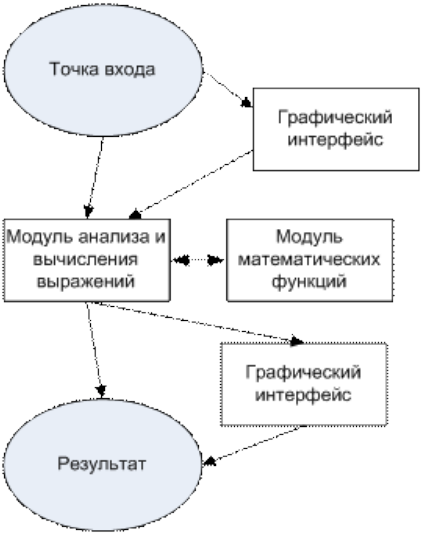


Рис. 1.5. Взаємодія модулів системи «Калькулятор»

4. Функціональні вимоги

1. Вимоги до програми

- Калькулятор повинен виконувати такі арифметичні операції: додавання, віднімання, множення, знаходження частного, знаходження залишку.
- Калькулятор повинен підтримувати роботу з цілими числами в межах від -2147483648 до 2147483647 (надалі MININT і MAXINT). У разі виходу за ці межі має видаватися повідомлення про помилку Error 06.
- Калькулятор повинен мати пам’ять на одне ціле число, а також можливість виводити це число на екран, скидати його значення на 0 і додавати до нього будь-яке інше число, введене в поле введення.
- При натисканні на клавішу M+ до числа, записаного в пам’ять, додається число, записане в поле «Результат». При цьому на складання накладаються обмеження з 3.2.1.
- Якщо в поле «Результат» записаний код помилки, то при натисканні на клавішу M + має видаватися повідомлення «Неможливо перетворити у число».
- При натисканні на кнопку MC число в пам’яті обнуляється.
- При натисканні на кнопку MR число з пам’яті приписується в кінець виразу в рядок «Вираз».
- Калькулятор повинен надавати можливість користувачеві працювати з операціями унарного плюса і унарного мінуса.
- Якщо між натисканнями на кнопку <+/-> проходить менш 3 секунд, то введений оператор змінюється на протилежний.
- Якщо між натисканнями на кнопку <+/-> проходить більше 3 секунд, то для вираження дописується знак "-".
- Калькулятор повинен мати графічний інтерфейс, який містить кнопки з цифрами і арифметичними операціями, кнопкою рівності, кнопками роботи з пам’яттю, кнопками редагування дужок і кнопками скидання, перемикачем унарного

мінуса / унарного плюса, текстовими полями для введення виразу і виведення результату.

- При натисканні на клавішу <Enter> калькулятор повинен проводити обчислення виразу.
- При натисканні на клавішу <ESC> програма повинна припиняти свою роботу.
- У разі невірно побудованого обчислюваного виразу або невідповідності його вимогам 3.2 в текстове вікно результату повинно виводитися відповідні повідомлення.

2. Арифметичні операції

- Додавання.
  - Для чисел, кожне з яких менше або дорівнює MAXINT і більше або дорівнює MININT, функція підсумовування повинна повертати правильну суму з точки зору математики.
  - Для чисел, сума яких більше ніж MAXINT і менше ніж MININT, а також в разі, якщо будь-який з доданків більше ніж MAXINT або менше ніж MININT, програма повинна видавати помилку Error 06.
- Віднімання.
  - Для чисел, кожне з яких менше або дорівнює MAXINT і більше або дорівнює MININT, функція віднімання повинна повертати правильну різниця з точки зору математики.
  - Для чисел, різниця яких більше ніж MAXINT і менше ніж MININT, а також в разі, якщо будь-який з чисел більше ніж MAXINT або менше ніж MININT, програма повинна видавати помилку Error 06.
- Множення
  - Для чисел, добуток яких менше або дорівнює MAXINT і більше або дорівнює MININT, функція множення повинна повертати правильний добуток з точки зору математики.
  - Для чисел, добуток яких більше ніж MAXINT і менше ніж MININT, а також, в разі якщо будь-

який з множників більше ніж MAXINT або менше ніж MININT, програма повинна видавати помилку Error 06.

- Знаходження частного
  - Для чисел, менших або рівних MAXINT і більших або рівних MININT, приватна яких менше або дорівнює MAXINT і більше або дорівнює MININT і дільник НЕ дорівнює 0, функція розподілу повинна повертати правильне приватне з точки зору математики.
  - Для чисел, приватне яких більше ніж MAXINT і менше ніж MININT, а також в разі, якщо будь-який з чисел більше ніж MAXINT або менше ніж MININT, і для подільника, що не рівного 0, програма повинна видавати помилку Error 06.
  - Якщо дільник дорівнює 0, програма повинна видавати помилку Error 09
- Ділення із залишком
  - Для чисел, менших або рівних MAXINT і більших або рівних MININT, залишок яких менше або дорівнює MAXINT і більше або дорівнює MININT і дільник НЕ дорівнює 0, функція розподілу повинна повертати правильний залишок з точки зору математики.
  - Для чисел, залишок яких більше ніж MAXINT і менше ніж MININT, а також в разі, якщо будь-який з чисел більше ніж MAXINT або менше ніж MININT, і для подільника, що не рівного 0, програма повинна видавати помилку Error 06
  - Якщо дільник дорівнює 0, програма повинна видавати помилку Error 09
- Унарний плюс \ мінус.
  - Для чисел, менших або рівних MAXINT і більших або равних MININT, операція унарного плюса / мінуса повинна повертати число відповідного знака.
  - Для чисел більших MAXINT або менших MININT функція повинна видавати помилку Error 06.
- Додаткові вимоги до вхідного виразу

- Максимальна сумарна кількість операторів і чисел - 30.
- Максимальна глибина вкладеності скобочной структури - 3.
- В якості унарного мінуса використовується символ «m», в якості унарного плюса - «p».
- Для операції знаходження приватного – «/», для знаходження залишку – «mod».
- Між операторами, дужками і числами може бути будь-яка кількість пробілів.
- Дозволяється використовувати лише дужки виду «(" і ")»
- Максимальна довжина виразу – 65535 символів.

Завдання

Вивчити специфікацію вимог і виявити наявні недоліки. Обгрунтувати їх та офортити у вигляді звіта з лабораторної роботи.

Лабораторна робота 2. Тестові приклади. Класи еквівалентності. Ручне тестування в Visual Studio

Мета:

- розглянути підходи до тестування системи;
- розглянути поняття класів еквівалентності, граничних умов;
- провести огляд можливостей Visual Studio по ручному тестуванню і опису тестових прикладів (Manual Testing).

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *тем 2-3.*

2.2. Тестові приклади

2.2.1. Розробка тестових прикладів

Безпосередньо для тестування програмного забезпечення необхідно визначити перевірочні завдання, що виконуються системою або її окремою частиною. Такі завдання називаються тестовими прикладами.

Можна виділити два підходи до створення тестових прикладів – виходячи з функціональних вимог (або з будь-якої іншої документації, що описує систему) і виходячи з коду. При тестуванні функціональності програми застосовується підхід «чорного ящика», тобто для кожної вимоги до системи формуються тест-вимоги, які, як правило, деталізують функціональні вимоги так, що на одну функціональну вимогу може припадати кілька тест-вимог. Самі тест-вимоги визначають, що повинно бути протестовано, але не визначають, як. Конкретні значення задаються в тестових прикладах. Таким чином, одній тест-вимозі може відповідати відразу кілька тестових прикладів.

Кожен тестовий приклад складається з набору вхідних значень і набору очікуваних вихідних значень. Розглянемо специфікацію з попередньої лабораторної.

Почнемо ми з тестування окремих складових програми (в даному випадку – модуль математика) на допустимі дані і, зокрема, на допустимі граничні дані.

Розглянемо приклад. Для цього візьмемо вимогу 4.2.4.1.

Вимога 4.2.4.1: Для чисел, менших або рівних MAXINT і більших або рівних MININT, частна яких менше або дорівнює MAXINT і більше або дорівнює MININT і дільник НЕ дорівнює 0, функція розподілу повинна повертати правильне частне з точки зору математики.

Функція, яку будемо тестувати:

```
/// <summary>
 /// приватне
 /// </ summary>
 /// <param name = "a"> ділене </ param>
 /// <param name = "b"> дільник </ param>
 /// <returns> приватне </ returns>
 public static long Div (long a, long b)
```

В принципі, тестування такої функції легко автоматизується за допомогою Unit Testing, так як у неї рівень доступу public, тобто до неї можна звернутися з будь-якого класу. До того ж вона є статичною, що дозволить викликати її, не створюючи екземпляр класу CalcClass. Однак в цій лабораторній роботі ми розглянемо ручне тестування.

Перш за все за цією функціональною вимогою складемо тест-вимоги. На перший погляд, очевидно, що питання для перевірки звучить так: «Перевірити, що для чисел, менших або рівних MAXINT і більших або рівних MININT, приватна яких менше або дорівнює MAXINT і більше або дорівнює MININT і дільник НЕ дорівнює 0, функція розподілу повертає правильне приватне з точки зору математики». Однак, це не зовсім так. Фраза «менших або рівних» відразу ж наводить на думку про перевірку двох випадків: 1) хоча б одне з чисел строго одно MAXINT і 2) всі числа менше, ніж MAXINT.

Зауваження. Варто зауважити, що зараз ми пишемо дуже докладні тест-вимоги, які практично відразу можна відобразити в тестові приклади. Така ситуація спостерігається,

наприклад, в проєктах, в яких тест-вимоги відсутні, а тестові приклади пишуться відразу на підставі функціональних вимог.

Тест-вимоги

1. Перевірити, що для чисел, менших MAXINT і більших 0, функція розподілу повертає правильне приватне з точки зору математики.
2. Перевірити, що для діленого, меншого MAXINT і більшого 0, і дільника, меншого 0 і більшого MININT, функція розподілу повертає правильне приватне з точки зору математики.
3. Перевірити, що для діленого, меншого 0 і більшого ніж MININT, і дільника, більшого 0 і меншого MAXINT, функція розподілу повертає правильне приватне з точки зору математики.
4. Перевірити, що для чисел, менших 0 і більших MININT, функція розподілу повертає правильне приватне з точки зору математики.
5. Перевірити, що для діленого, рівного 0, і дільника, меншого MAXINT і більшого 0, функція розподілу повертає правильне приватне з точки зору математики.
6. Перевірити, що для діленого, рівного 0, і дільника, більшого MININT і меншого 0, функція розподілу повертає правильне приватне з точки зору математики.
7. Перевірити, що для діленого, рівного MAXINT, і дільника, меншого MAXINT і більшого MININT, функція розподілу повертає правильне приватне з точки зору математики.
8. Перевірити, що для діленого, рівного MININT, і дільника, меншого MAXINT і більшого MININT, функція розподілу повертає правильне приватне з точки зору математики.
9. Перевірити, що для подільника, рівного MAXINT, і ділимо, меншого MAXINT і більшого MININT, функція розподілу повертає правильне приватне з точки зору математики.

10. Перевірити, що для подільника, рівного MININT, і ділимого, меншого MAXINT і більшого MININT, функція розподілу повертає правильне приватне з точки зору математики.

11. Перевірити, що для подільника, рівного MAXINT, і ділимо, рівного MININT, функція розподілу повертає правильне приватне з точки зору математики.

12. Перевірити, що для подільника, рівного MAXINT, і ділимо, рівного MAXINT, функція розподілу повертає правильне приватне з точки зору математики.

13. Перевірити, що для подільника, рівного MININT, і ділимо, рівного MININT, функція розподілу повертає правильне приватне з точки зору математики.

14. Перевірити, що для подільника, рівного MININT, і ділимо, рівного MAXINT, функція розподілу повертає правильне приватне з точки зору математики.

Складемо тестові приклади і запишемо їх у вигляді таблиці (табл. 2.1).

Таблиця 2.1. Тестові приклади для вимоги 4.2.4.1

№	Вхідні значення: ділене, дільник	Очікуваний результат	Номер тест-вимоги	Примітки
1)	43 / 21	2	1)	Найчастіший випадок - коректні вхідні дані
2)	87/-56	-1	2)	-//-
3)	-9/2	-4	3)	-//-
4)	-4321/-50	86	4)	-//-
5)	0/1234567890	0	5)	Часто помилки

				виявляються при нульових значеннях змінних
6)	0/-1098765432	0	6)	-//-
7)	2147483647/95	22605091	7) и 1)	Перевірка граничних умов
8)	-2147483648/9	-238609294	8) и 2)	-//-
9)	99/2147483647	0	9) и 1)	-//-
10)	-87/-2147483648	0	10) и 4)	-//-
11)	-2147483648/2147483647	-1	11)	Помилка
12)	2147483647/2147483647	1	12)	-//-
13)	-2147483648/-2147483648	1	13)	Помилка
14)	2147483647/-2147483648	0	14)	Помилка

Зауважимо, що, ми не можемо точно сказати, де сталася помилка. Це добре видно з рис. 2.1.

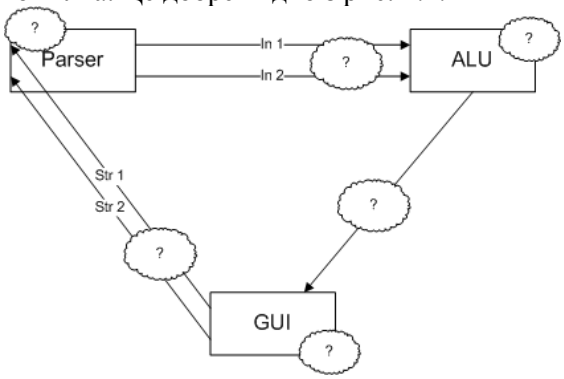


Рис. 2.1. Позиціонування помилки

Частини рисунку, помічені питанням, – це ті частини програми, в яких могла бути помилка. Це самі обчислення, перетворення типів, помилка GUI або передачі параметрів (можливо, з некоректним перетворенням типів). Про виявлені помилки складається звіт, який віддається розробникам. При цьому виправлення помилки в обов'язки тестувальника не входить, так як він займається саме тестуванням, а не відладкою додатка.

У першому тесті ми ввели два числа і отримали вірний результат. Всі інші тести точно такі ж. Але перебрати їх всі не вийде, так як всіляких комбінацій - 429496736 * 429496735 = 184467445805156960. Очевидно, що більшість вхідних значень приведуть до одного і того ж результату, і немає сенсу перевіряти їх все. Якщо програма пройде перший тест, то вона, швидше за все, пройде і інші.

Якщо від двох тестових прикладів очікується отримати один і той же результат, значить, вони належать одному класу. Такі безлічі прикладів називаються класами еквівалентності. Класи еквівалентності – це, в першу чергу, спосіб зменшення необхідного числа тестових прикладів. 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.

2.2.2. Класи еквівалентності

Розглянемо ще один приклад.

Вимога 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.

Функція, яку будемо тестувати:

```
/// <summary>
 /// Додавання
 /// </ summary>
```

```
/// <param name = "a"> доданок </ param>
/// <param name = "b"> доданок </ param>
/// <returns> сума </ returns>
public static long Add (long a, long b)
```

У порівнянні з попередньою вимогою у цієї явно є недоліки. Тут нічого не говориться про обмеження на суму. Можна легко підібрати два таких числа, які будуть задовольняти заявленій вимозі, а їх сума буде виходити за межі int. Швидше за все, це помилка проєктування програми. Тоді необхідно виправити специфікацію і повідомити про це іншим учасникам розробки, перш за все її упорядника.

Після виправлень функціональна вимога 4.2.1.1. буде виглядати так:

Вимога 4.2.1.1: Для чисел, менших або рівних MAXINT і більших або рівних MININT, сума яких менше або дорівнює MAXINT і більше або дорівнює MININT, функція підсумовування повинна повертати правильну суму з точки зору математики.

Тестування на допустимі дані нічим не буде відрізнятися від тестування функції розподілу. Складемо класи еквівалентності.

У найпростішому випадку будь-який з доданків ділиться на 3 класу еквівалентності: MININT, MAXINT і проміжне значення. Якщо підходити більш серйозно, то можна виділити 7 допустимих класів еквівалентності: MININT, MININT + 1, має негативного граничне число, 0, додатне не граничне число, MAXINT-1, MAXINT.

З огляду на те, що у нас два ідентичних вхідних параметра, для повного розгляду всіх класів еквівалентності необхідно скласти і перевірити 7 * 7 = 49 тестових прикладів, що все одно набагато менше, ніж повний перебір.

При цьому, як показав тест з розподілом, помилка може проявитися лише в декількох з цих прикладів, які не сильно відрізняються від інших граничних класів еквівалентності.

Деякі класи еквівалентності не задовольняють вимозі 4.2.1.1, так як виводять суму за допустимі межі. Поведінка методу на таких вхідних даних описана в вимозі 4.2.1.2.

На рис. 2.2 показано можливе виділення класів еквівалентності (кольорами зображені області коректних і некоректних значень, а кружками – самі класи еквівалентності):

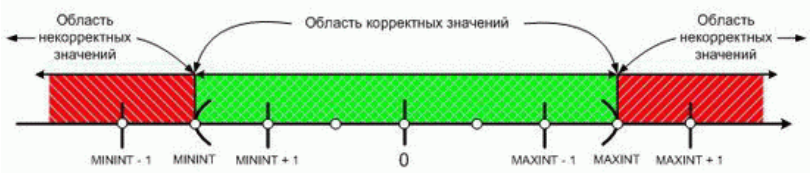


Рис. 2.2. Класи еквівалентності

Іноді зручніше скласти класи еквівалентності за вихідним параметром (в даному випадку їх буде 7) і вже по ним підбирати вхідні дані і складати тестові приклади.

Основний спосіб пошуку дефектів – передача системі даних, не передбачених вимогами: занадто довгих або занадто коротких рядків, невірних символів, чисел за межами обчислюваного діапазону і т.п. Невірні дані, як і допустимі, також можна розділяти на різні класи еквівалентності. Як простий приклад знову розглянемо функцію додавання.

Зауваження. Як вже зазначалося вище, тест-вимоги складені дуже докладно і, фактично, відповідають тестовим прикладам. Тому поведінку методу на некоректних даних описано в специфікації, хоча подібна ситуація в житті рідко зустрічається.

У вимозі 4.2.1.2. Для чисел, сума яких більше ніж MAXINT і менше ніж MININT, а також в разі, якщо будь-який з доданків більше ніж MAXINT або менше ніж MININT, програма повинна видавати помилку Error 06.

Інтерфейс методу Add не дозволяє нам дізнатися про помилку, що сталася під час виконання методу. Середовище .NET надає потужний засіб для знаходження і обробки помилок (і не тільки) під час виконання програми – Exception (Виняток). Саме з використанням винятків і будуть працювати методи

класу CalcClass. Зараз скористаємося іншим методом, а саме створимо в класі математичних функцій глобальну статичну змінну типу string lastError. У неї будемо записувати коди помилок, які сталися під час роботи програми, а в самі методи вставимо код, що виводить на екран повідомлення про помилку.

Складемо тест-вимоги.

1. Якщо одна з складових більше, ніж MAXINT, то функція повинна видати повідомлення «Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647» і записати в змінну lastError «Error 06».

2. Якщо одна зі складових менше, ніж MININT, то функція повинна видати повідомлення «Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647» і записати в змінну lastError «Error 06».

3. Якщо сума доданків більше, ніж MAXINT, то функція повинна видати повідомлення «Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647» і записати в змінну lastError «Error 06».

4. Якщо сума доданків менше, ніж MININT, то функція повинна видати повідомлення «Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647» і записати в змінну lastError «Error 06».

При складанні тестових прикладів цим тест-вимогам буде відповідати більш чотирьох прикладів, так як необхідно перевірити випадки, коли одне число більше MAXINT, а інше задовольняє вимогам або більше MAXINT і так далі. У той же час деякі тестові приклади можуть покривати відразу кілька тест-вимог. Так, приклад «Якщо перший доданок більше MAXINT, а другий доданок менше MININT, при цьому сума чисел більше MAXINT, то функція повертає повідомлення «Занадто мале або занадто велике значення числа для int. Числа повинні бути в межах від -2147483648 до 2147483647» і записує

в змінну lastError «Error 06», перевіряє відразу перше, друге і третє тест-вимога.

Для неприпустимих даних також можна скласти класи еквівалентності, причому як по вхідним, так і по вихідних параметрах, і по ним підібрати тестові приклади. У нашому випадку на кожну змінну можна виділити 4 класи: багато менше MININT, MININT-1, MAXINT + 1, багато більше MAXINT. Таким чином, треба перевірити 16 тестових прикладів.

Насправді, глянувши на будь-яку з розглянутих функцій, в загальному випадку виділяють 4 основні класи еквівалентності (рис.2.3).

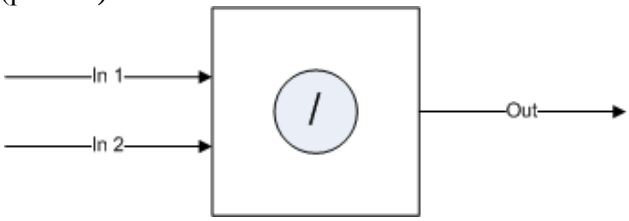


Рис. 2.3. Структурна схема функції розподілу

1. Обидва вхідних параметра належать до допустимої області, і вихідне значення належить до допустимої області.
2. Перший вхідний параметр належить до допустимої області, другий не належить до допустимої області.
3. Перший вхідний параметр не належить до допустимої області, другий належить до допустимої області.
4. Обидва вхідних параметри належать до допустимої області, а значення функції не належить до допустимої області.

2.3. Можливості Visual Studio з ручного тестування і опису тестових прикладів (Manual Testing)

Деякі тестові приклади не можуть бути виконані в автоматичному режимі, занадто складні для автоматизації їх

виконання або їх автоматичне виконання зажадає занадто багато часу, і тому вони вимагають ручної роботи тестувальника. Visual Studio має інструмент для роботи з ручними тестами.

Ручне тестування в Visual Studio представляю собою сценарій виконання тесту.

Розглянемо процес створення ручного тесту.

Спочатку створимо новий тестовий проєкт. Для цього зайдемо в File-> New-> Project ... (Можна також натиснути Ctrl + Shift + N або натиснути на іконку  на панелі Standart).

У діалоговому вікні New Project виберемо тип проєкту Test-> Test Project. В поле Name задамо ім'я нашого проєкту (наприклад, ManualTestProject). натиснемо ОК. (Рис. 2.4).

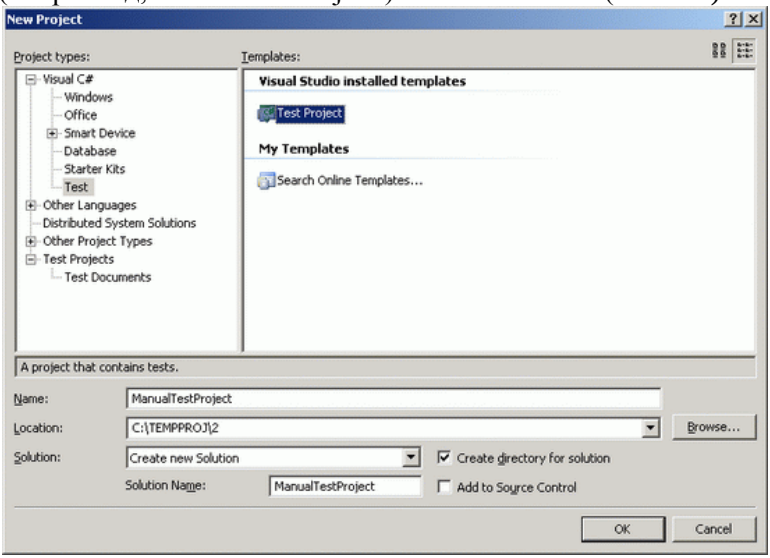


Рис. 2.4. Діалогове вікно New Project

Новий тестовий проєкт створений.

Тепер подивимося на вікно Solution Explorer (рис.2.5). Створений тестовий проєкт містить три файли, пов'язаних з тестуванням:

AuthoringTest.txt	**Примітки про створення тестів, що включають інструкції по додаванню додаткових тестів до проєкту.**
UnitTest1.cs	Порожня структура unit test класу, куди поміщаються додаткові тести.
ManualTest1.mht	Шаблон у форматі Word, який заповнюється інструкціями при ручному тестуванні.

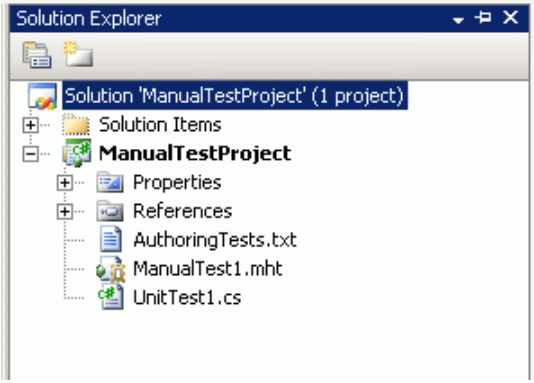


Рис. 2.5. Вікно Solution Explorer

Файл UnitTest1.cs нам не знадобиться для ручного тестування, тому його можна видалити з проєкту. Для додавання в проєкт шаблону для ручного тестування потрібно в меню Test вибрати New Test. У діалоговому вікні Add New Test вибрати Manual Test (text format). В поле Test Name потрібно ввести назву тесту, наприклад ManualTest1.mtx. Ні в якому разі не можна міняти дозвіл цього файлу. В полі Add to Test Project виберемо створений нами раніше ManualTestProject. Натиснемо OK. (Рис. 2.6) У наш тестовий проєкт буде додано файл з ручним тестом ManualTest1.mtx.

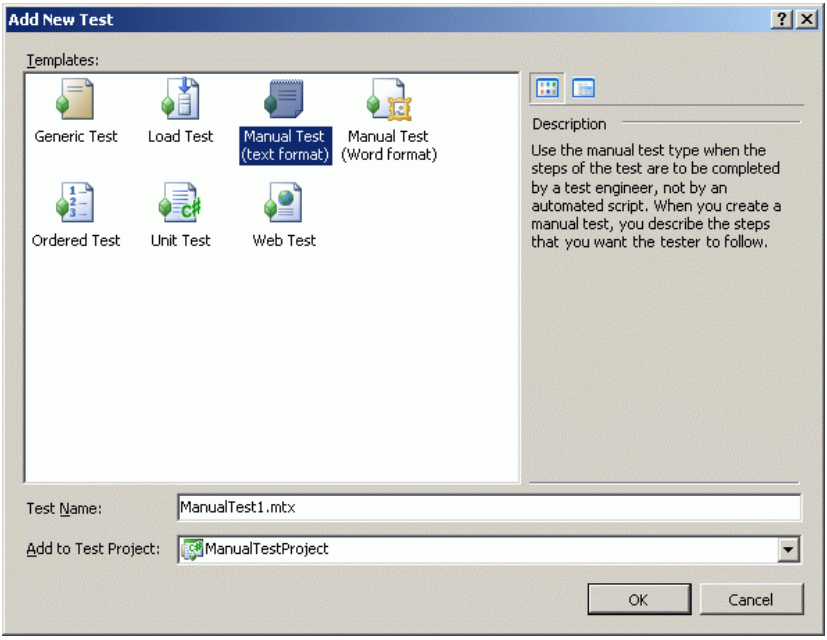


Рис. 2.6. Діалогове вікно Add New Test

Тепер переконаємося, що ручний тест доданий і готовий до виконання. В меню Test натиснемо на пункт Windows і в підменю виберемо Test View. Відкриється вікно Test View, в якому видно тест MyManualTest (Рис. 2.7).

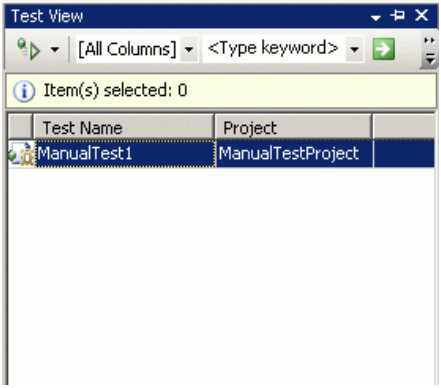


Рис. 2.7. Вікно Test View

Новий ручний тест доданий, і тепер все готово до його редагування. необхідно відкрити шаблон тесту (наприклад, зробивши подвійний клік мишкою по ManualTest1.mht (ManualTest1.mtx) в Solution Explorer). Далі, дотримуючись інструкції, вводимо інформацію про тест в шаблон. шаблон включає в себе назву тесту (Test Title), опис тесту (Test Details), функціональність, яку треба перевірити (Test Target), дії, які необхідно зробити для проведення тесту (Test Steps), і опис історії змін тесту (Revision History). Після завершення редагування необхідно зберегти шаблон.

Наступний етап – виконання тесту тестувальником.

У вікні Test View натиснемо правою кнопкою миші по створеному нами ручному тесту (ManualTest1) І виберемо Run Selection (Або натиснемо в вікні Test View на кнопку ).

З’явиться діалогове вікно, яке попереджає про те, що тест буде виконаний, коли всі ручні тести будуть пройдені. Через деякий час з’явиться діалогове вікно, Що повідомляє нам про те, що всі ручні тести готові до виконання. Відкриється вікно Test Results, в якому наш тест буде позначений як Pending (Виконується), і вікно MyManualTest [Running], що починає виконання тесту. Дотримуючись сценарію тесту і залишаючи свої коментарі у верхній частині вікна, тестувальник виконує тест, після чого вказує, пройдений тест чи ні (Pass / Fail), і

натискає Apply у верхній частині екрану. У вікні Test Results відобразяться зміни, тобто замість Pending буде Passed або Failed (В залежності від того, що ви вказали у вікні MyManualTest [Running] після виконання ручного тесту).

Завдання

Скласти тест-вимоги і провести ручне тестування таких методів:

```
1. Знаходження залишку
2.        /// <summary>
3.         /// Розподіл по модулю
4.         /// </ summary>
5.         /// <param name = "a"> ділене </ param>
6.         /// <param name = "b"> дільник </ param>
7.         /// <returns> залишок </ returns>
 public static int Mod (long a, long b)

8. Унарний плюс
9.        /// <summary>
10.        /// унарний плюс
11.        /// </ summary>
12.        /// <param name = "a"> </ param>
13.        /// <returns> </ returns>
 public static int ABS (long a)

14.           Унарний мінус
15.       /// <summary>
16.        /// унарний мінус
17.        /// </ summary>
18.        /// <param name = "a"> </ param>
19.        /// <returns> </ returns>
 public static int IABS (long a)

20.           Віднімання
21.       /// <summary>
```

```
22.     /// віднімання
23.     /// </ summary>
24.     /// <param name = "a"> уменьшаемое </
param>
25.     /// <param name = "b"> від'ємник </ param>
26.     /// <returns> різниця </ returns>
        public static int Sub (long a, long b)
```

За результатами ручного тестування заповнити звіт про проблему (зразок в архіві).

Лабораторна робота 3. Тестове оточення

Мета:

- розглянути питання проєктування тестового оточення;
- провести огляд понять тестових класів і тестових проектів.

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *тем 4-6.*

Основний обсяг тестування практично будь-якої складної системи зазвичай виконується в автоматичному режимі. Крім того, тестована 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 деяке середовище, яке забезпечить запуск і виконання тестового модуля, передасть йому вхідні дані, збере реальні вихідні дані, отримані в результаті роботи системи на заданих вхідних даних (рис. 3.1). Після цього середовище має порівняти реальні вихідні дані з очікуваними і на підставі даного порівняння зробити висновок про відповідність поведінки модуля заданому.

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 тестовий модуль.

Оскільки тестове оточення саме є програмою (причому, часто реалізоване неправильною мовою програмування, на якому написана система), воно теж має бути протестовано. Метою тестування тестового оточення є доказ того, що тестове

оточення ніяким чином не спотворює виконання тестового модуля і адекватно моделює поведінку системи.

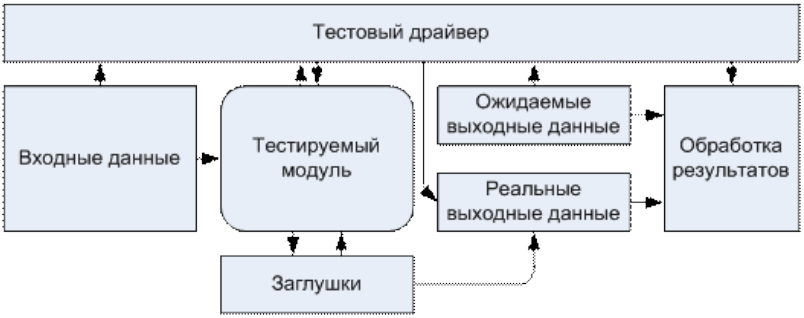


Рис. 3.1. Узагальнена схема середовища тестування

Тестове оточення для програмного коду на структурних мовах програмування складається з двох компонентів – драйвера, який забезпечує запуск і виконання тестового модуля, 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 неправильне 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 тестовий модуль вхідних значень і прийом результатів
3. 2 + виведення 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 і виводити повідомлення про те, що заглушка була викликана
2. 1 + виводити повідомлення зі значеннями параметрів, переданих у функцію

3. 2 + повертати значення, заздалегідь задане у вхідних параметрах тесту

4. 3 + виводити значення, заздалегідь задане у вхідних параметрах тесту

5. 3 + приймати від тестованого ПЗ значення і передавати їх в драйвер

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 тестованим об'єктом є функція, то в об'єктно-орієнтованому ПЗ мінімальним об'єктом є клас. При застосуванні принципу інкапсуляції всі внутрішні дані класу і деяка частина його методів недоступна ззовні. В цьому випадку тестувальник позбавлений можливості звертатися в 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 такзваний 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. Саме такий підхід до тестування прийнятий зараз в більшості методологій і середовищ розробки. Його класичне назва – unit testing (тестування модулів).

2. Аналогічно до попереднього 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 аналогічна оточенню для тестування структурних програм.

4. Використовуються спеціальні засоби доступу до закритих даних і методів 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 в цьому випадку не можна гарантувати того, 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 можна створювати заглушки, тому що 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.

На прикладі системи «Калькулятор»

Тести, виконані у минулій лабораторній роботі, як правило, проводяться не вручну. Для цілей тестування пишуть спеціальну програму – тестовий драйвер, який і проводить тестування. Більш того, такі програми часто пишуться на іншій мові, ніж програма, що тестується, або створюються автоматично, за допомогою спеціальних утиліт.

В цій лабораторній роботі ми самі напишемо простий тестовий драйвер на C# для тестування функцій «Калькулятор», використовуючи специфікацію другої лабораторної роботи.

Код програми змінений для спрощення компіляції окремих модулів. Так, виключена робота зі змінною Program.res, а клас CalcClass оголошений як public.

Для початку розглянемо функцію розподілу. Тест-вимоги до неї ми вже склали. Для простоти будемо користуватися лише чотирма загальними тест вимогами.

1. Обидва вхідних параметра належать допустимої області, і вихідне значення належить допустимої області.
2. Перший вхідний параметр належить допустимої області, другий не належить допустимої області
3. Перший вхідний параметр не належить допустимої області, другий належить допустимої області
4. Обидва вхідних параметрів належать допустимої області, а значення функції не належить допустимої області.

Складемо програму. Код програми наведено у додатку А.

Кожен тестовий приклад знаходиться всередині блоку try-catch для того, щоб перехопити сгенероване виключення всередині методів Add ().

При цьому файл CalcClass.dll, в якому і буде реалізовано всі математичні методи, необхідно додати в References проєкту.

Проведемо тестування і отримаємо наступний результат:

Test Case 1
Вхідні дані: a = 78508, b = -304
Очікуваний результат: res = 78204 && error = ""
Код помилки:
Одержаний результат: res = 78204 error =
Тест пройдений

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 пройдений

Test Case 3
Вхідні дані: a = 3000000000, b = -2850800078
Очікуваний результат: res = 0 && error = "Error 06"
Код помилки: Error 06
Одержаний результат: res = 0 error = Error 06
Тест пройдений

Test Case 4
Вхідні дані: a = 2000000000, b = 2000000000
Очікуваний результат: res = 0 && error = "Error 06"
Код помилки: Error 06
Одержаний результат: res = 0 error = Error 06
Тест пройдений

Точно такий же результат ми б отримали і при ручному тестуванні, якби виявлені помилки були виправлені. Зауважимо, що при такому підході до тестування нам вдається локалізувати помилки. Якщо щось працює не так, як треба, то можна з упевненістю стверджувати, що помилка міститься саме в функції розподілу, в той час, як в лабораторній роботі 2 ми не могли сказати, де саме вона сталася.

Завдання

В архіві знаходяться .dll файли, які потрібно протестувати методом «чорного ящика», і приклад тестового драйвера.

Скласти тест-план і провести модульне тестування наступних методів:

1. Знаходження залишку.
2. /// <summary>
3. /// Розподіл по модулю
4. /// </ summary>

5. /// <param name = "a"> ділене </ param>
6. /// <param name = "b"> дільник </ param>
7. /// <returns> залишок </ returns>
public static int Mod (long a, long b)
8. Унарний плюс.
9. /// <summary>
10. /// унарний плюс
11. /// </ summary>
12. /// <param name = "a"> </ param>
13. /// <returns> </ returns>
public static int ABS (long a)
14. Унарний мінус.
15. /// <summary>
16. /// унарний мінус
17. /// </ summary>
18. /// <param name = "a"> </ param>
19. /// <returns> </ returns>
public static int IABS (long a)
20. Віднімання.
21. /// <summary>
22. /// віднімання
23. /// </ summary>
24. /// <param name = "a"> уменьшаемое </ param>
25. /// <param name = "b"> від'ємник </ param>
26. /// <returns> різниця </ returns>
public static int Sub (long a, long b)
27. Множення.
28. /// <summary>
29. /// множення
30. /// </ summary>
31. /// <param name = "a"> множник </ param>
32. /// <param name = "b"> множник </ param>
33. /// <returns> твір </ returns>
public static int Mult (long a, long b)
34. Розподіл.

```
35.       /// <summary>
36.        /// приватне
37.        /// </ summary>
38.        /// <param name = "a"> ділене </ param>
39.        /// <param name = "b"> дільник </ param>
40.        /// <returns> приватне </ returns>
 public static int Div (long a, long b)
```

Лабораторна робота 4. Модульне тестування

Мета:

- розглянути питання модульного тестування, його завдання і цілі;
- розглянути тестування класів, проєктування тестового оточення.

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *теми 9.*

Кожна складна програмна система складається з окремих частин – модулів, що виконують ту або іншу функцію в складі системи. Для того, щоб упевнитися в коректній роботі системи в цілому, необхідно спочатку протестувати кожен модуль системи окремо. У разі виникнення проблем це дозволить простіше виявити модулі, що викликали проблему, і усунути відповідні дефекти в них. Таке тестування модулів окремо отримало назву 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, які описують конкретні тестові приклади.

Основна мета модульного тестування – впевнитись у відповідності до вимог кожного окремого модуля системи перед тим, як буде проведена його інтеграція до складу системи.

При цьому в ході модульного тестування вирішуються чотири основні завдання.

1. *Пошук і документування невідповідностей вимогам* – це класична задача тестування, що включає в себе не тільки розробку тестового оточення і тестових прикладів, а й виконання тестів, протоколювання результатів виконання, складання звітів про проблеми.
2. *Підтримка розробки та рефакторінга низькорівневої архітектури системи і міжмодульної*

взаємодії – це завдання більше властиве «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 самого модуля. Однак, навіть при класичній схемі тестування модульні тести можуть виявити проблеми в дизайні системи і нелогічні або заплутані механізми роботи з модулем.

3. *Підтримка рефакторинга модулів* – це завдання пов'язане з підтримкою процесу зміни системи. Досить часто в ході розробки потрібно проводити рефакторинг модулів або їх груп – оптимізацію або повну переробку програмного коду з метою підвищення його супроводжуваності, швидкості роботи або надійності. Модульні тести при цьому є потужним інструментом для перевірки того, що новий варіант програмного коду працює в точності так само, як і старий.

4. *Підтримка усунення дефектів і налагодження* – це завдання пов'язана зі зворотним зв'язком, яке отримують розробники від тестувальників у вигляді звітів про проблеми. Докладні звіти про проблеми, складені на етапі модульного тестування, дозволяють локалізувати і усунути багато дефектів в програмній системі на ранніх стадіях її розробки або розробки її нової функціональності.

В силу того, що модулі, що піддаються тестуванню, зазвичай невеликі за розміром, модульне тестування вважається найбільш простим (хоча і досить трудомістким) етапом тестування системи. Однак, незважаючи на зовнішню простоту, з модульним тестуванням пов'язано дві проблеми.

1. Не існує єдиних принципів визначення того, що в точності є окремим модулем.

2. Відмінності в трактуванні самого поняття модульного тестування: чи розуміється під ним відокремлене тестування модуля, робота якого підтримується тільки тестовим оточенням, або 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 інтеграційне тестування.

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 в списку завдань проєкту (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 тестовий модуль приймається або програмний модуль (одиниця компіляції) в разі, якщо система розробляється на процедурній мові програмування, або клас, якщо система розробляється на об'єктно-орієнтованій мові.

У разі систем, написаних на процедурних мовах, процес тестування модуля відбувається так, як це було розглянуто в темах 2-4 – для кожного модуля розробляється тестовий драйвер, що викликає функції модуля і збирає результати їх роботи, і набір заглушок, які імітують поведінку функцій, що містяться в інших модулях і не потрапляють під тестування даного модуля. При тестуванні об'єктно-орієнтованих систем існує ряд особливостей, перш за все викликаних інкапсуляцією даних і методів в класах.

У разі об'єктно-орієнтованих систем дрібніший поділ класів і використання окремих методів як тестованих модулів недоцільно в зв'язку з тим, що для тестування кожного методу

буде потрібна розробка тестового оточення, яке можна порівняти за складністю 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. В ході такого тестування перевіряється взаємодія методів всередині класу і правильність доступу методів до внутрішніх даних класу. При такому тестуванні можливе виявлення не тільки стандартних дефектів, пов'язаних з виходами за межі діапазону або невірно реалізованими вимогами, а також виявлення специфічних дефектів об'єктно-орієнтованого програмного забезпечення:

- дефектів інкапсуляції, в результаті яких, наприклад, приховані дані класу виявляються недоступними за допомогою відповідних публічних методів;
- дефектів успадкування, при наявності яких схема успадкування блокує важливі дані або методи від класів-нащадків;
- дефектів поліморфізму, при яких поліморфна поведінка класу виявляється поширеною не на всі можливі класи;
- дефектів інсталіції, при яких у новостворених об'єктах класу не встановлюються коректні значення за замовчуванням параметрів і внутрішніх даних класу.

Підходи до проєктування тестового оточення

Незалежно від того, яка мінімальна одиниця вихідних кодів системи вибирається за мінімальний тестовий модуль, існує ще одна відмінність в 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. Тут для кожного знову розроблюваного модуля створюється тестовий драйвер і заглушки, за допомогою яких виконується набір тестів. Тільки після усунення всіх дефектів в автономному режимі проводиться 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, який може викликати маскування дефектів (ефект парного числа помилок). Основний недолік даного методу – підвищена трудомісткість написання драйверів і заглушок.

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, які не моделюють внутрішній стан системи.

Проте, при використанні цього методу зростає складність написання тестових прикладів, а саме для приведення в потрібний стан системи заглушок, як правило, потрібно тільки встановити значення тестових змінних, а для приведення в потрібний стан частини реальної системи необхідно виконати цілий сценарій. Кожен тестовий приклад в цьому випадку повинен містити такий сценарій.

Крім того, при цьому підході не завжди вдається локалізувати помилки, приховані всередині модуля, які можуть проявитися при інтеграції наступних модулів.

На прикладі «Калькулятор»

Розглянутий раніше в лабораторній роботі приклад простий насамперед за рахунок того, що нам не доводиться створювати тестового оточення. Щоб побачити весь описаний механізм в дії, протестуємо метод RunEstimate класу AnalaizerClass. Цей метод використовує методи з класу CalcClass, в надійності яких ми не впевнені. Замінимо ці методи заглушкою, що складається виключно з функцій стандартного класу Math. Для цього скористаємося файлом My.dll і додамо його в проєкт.

На лабораторній роботі ми не будемо складати тест-вимоги (це буде завдання для самостійної роботи). Продемонструємо, як створити тестове оточення і запустити метод. Перевіряємо операцію складання на прикладі 2 + 2, тобто в стеці до початку виконання самої операції (тобто після компіляції) знаходяться наступні елементи: "2 ","2 ","+ " (див. дод. Б).

Насправді даний підхід дозволяє виявити безліч недоліків програми, які іншими тестами не виявляються. Можна спробувати проекспериментувати з «Калькулятором» і переконатися, що він працює коректно. Однак, якщо в тестований метод подати на вхід не "2 ","2 ","+ ", а "2 ","2 ","+ ","+ ", То програма закінчить роботу з виключенням. Це говорить про те, що метод RunEstimate написаний не коректно. Можна, наприклад, було б приховати, тобто зробити доступ private, всім методам AnalaizerClass, крім Estimate (Це було б більш правильно, але для простоти тестування вони зроблені public. Варто відзначити, що Visual Studio має також механізми для тестування подібних методів.). Тим самим не дозволяється іншим виконувати «потенційно небезпечні» методи і передавати їм некоректні значення. Однак це не є достатнім механізмом захисту програми. Необхідно провести більш якісну валідацію використовуваних методами параметрів.

До проблеми створення тестового оточення можна підійти з двох сторін – або відкомпілювати код, з заздалегідь підключеними dll файлами до проєкту, або скористатися областю CodeDom і компілювати в процесі виконання. Це особливо зручно, якщо потрібно міняти тестове оточення в процесі роботи.

Завдання

У «Матеріалах для виконання лабораторної роботи» є .dll файли, які потрібно протестувати методом «чорного ящика» і приклад тестового драйвера.

Скласти тест-план і провести модульне тестування наступних методів:

1. /// <summary>

/// Перевірка коректності скобочной структури вхідного виразу

/// </ summary>

/// <returns> true - якщо все нормально,

false - якщо є помилка </ returns>

/// метод біжить по вхідному виразу, символ за символом аналізуючи його і перераховуючи кількість дужок.

У разі виникнення

/// помилки повертає false, а в erposition записує позицію, на якій виникла помилка.

public static bool CheckCurrency ()

2. /// <summary>

/// Форматує вхідний вираз, виставляючи між операторами пробіли і видаляючи зайві, а також відловлює непізнані оператори, стежить за кінцем рядка

/// а також відловлює помилки на кінці рядка

/// </ summary>

/// <returns> кінцеву рядок або повідомлення про помилку,

починаються зі спец. символу & </ returns>

public static string Format ()

3. /// <summary>

/// Створює масив, в якому розташовуються оператори і

символи, представлені в зворотньому польському записі (безскобочной)

/// На цьому ж етапі відловлюються майже всі інші помилки (див код). По суті це компіляція.

/// </ summary>

/// <returns> масив зворотнього польського запису </ returns>

public static System.Collections.ArrayList CreateStack ()

4. /// <summary>

/// Обчислення зворотнього польського запису

```
        /// </ summary>
        /// <returns> результат обчислень або повідомлення про помилку </ returns>
        public static string RunEstimate ()
```

Лабораторна робота 5. Автоматизація модульного тестування

Мета:

- розглянути питання тестового оточення, тестових класів, тестових проєктів;
- розглянути можливості Visual Studio з тестування модулів (Unit Testing).

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *теми 9.*

У попередніх лабораторних роботах ми виконували частину роботи вручну. Але при написанні тестів тестувальник також може помилитися, через що в програмі можуть залишитися різні помилки. У разі, якщо програмісти ведуть розробку за методикою екстремального програмування (XP), наслідуючи практику написання тестів перед кодом (test driven development, TDD), кількість тестів, які потрібно написати, стає за обсягом навіть більшим, ніж сам код системи. Однак очевидно, що більшу частину роботи з розробки тестів окремих методів (модульне тестування, unit testing) можна автоматизувати. У Visual Studio розроблені спеціальні засоби для автоматизації модульного тестування. Саме про них і піде мова далі.

Для створення тесту натискаємо правою кнопкою миші на методі Add () і вибираючи пункт меню Create Unit Tests ... (рис. 5.1). З'явиться діалогове вікно, що дозволяє створити тести в іншому проєкті (рис. 5.2). За замовчуванням, створюваний проєкт – новий проєкт на Visual Basic, але також доступні тестові проєкти на C# і C++. Вибираємо Visual C # і натискаємо кнопку OK, перед тим вводимо ім'я проєктуBaseCalculator.Test.

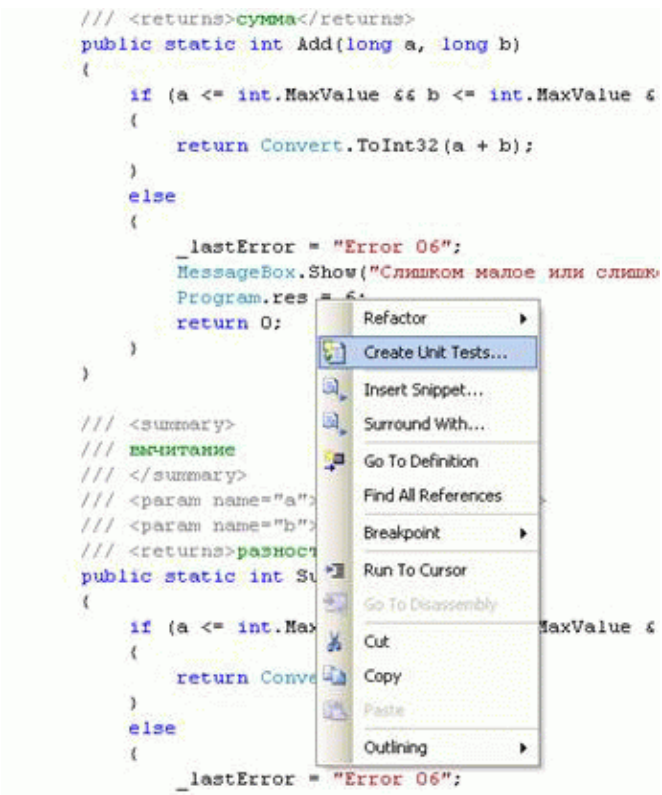


Рис. 5.1. Пункт контекстного меню «Create Unit Tests ...» в методі Add ()

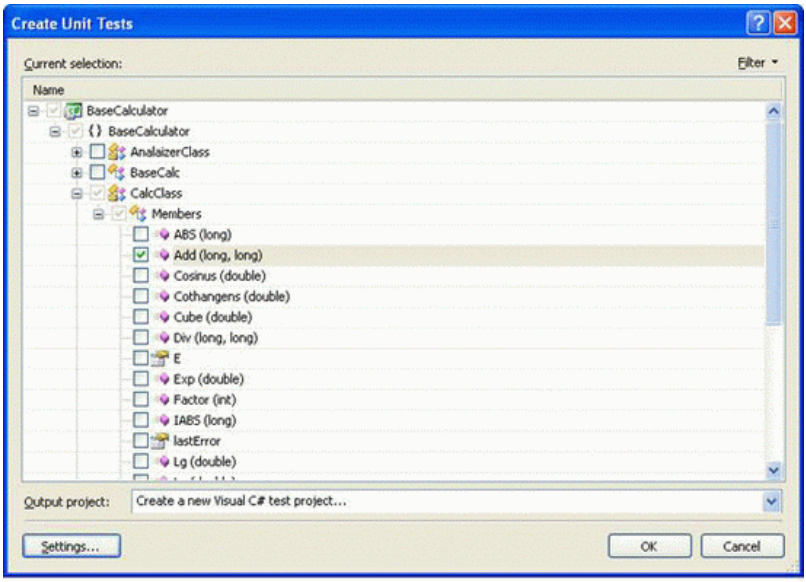


Рис. 5.2. Діалогове вікно «Create Unit tests»

Створений тестовий проєкт містить чотири файли, пов'язаних з тестуванням.

Ім'я файлу	**Примітка**
AuthoringTest.txt	Примітки про створення тестів, що включають інструкції з додавання додаткових тестів до проєкту
CalcClassTest.cs	Включає в себе згенерований тест для тестування методу Add () поряд з методами для тестової ініціалізації та очищення
ManualTest1.mht	Шаблон, який заповнюється інструкціями при ручному тестуванні
UnitTest1.cs	Порожня структура unit test класу, куди поміщаються додаткові тести

Так як ручне тестування ми вже провели, і файл для тестів у нас вже є, то ми видалимоManualTest1.mht iUnitTest1.cs.

В розділ References при генерації тестового проєкту додається посилання на Microsoft.VisualStudio.-QualityTools.UnitTestFramework і проєкт BaseCalculator, який і буде тестуватися. Перше – збірка, яку використовує «движок» модульного тестування при виконанні тестів. Другим є посилання на ту збірку, яку ми тестуємо.

За замовчуванням, згенерований тест-метод – це шаблон з наступною реалізацією:

```
/// <summary>
/// A test for Add (long, long)
/// </ summary>
[DeploymentItem ( "BaseCalculator.exe")]
[TestMethod ()]
public void AddTest ()
{
long a = 0; // TODO: Initialize to an appropriate value

long b = 0; // TODO: Initialize to an appropriate value

 int expected = 0;
 int actual;

 actual = BaseCalculator.Test.
 BaseCalculator_CalcClassAccessor.Add (a, b);

 Assert.AreEqual (expected, actual,
 "BaseCalculator.CalcClass.Add did not return
 the expected value. ");
 Assert.Inconclusive ( "Verify the correctness of this test
method.");
 }
```

Перш за все, зазначимо, що згенерований код позначений атрибутом TestMethod типу TestMethodAttribute, а сам клас позначений атрибутом TestClassAttribute, які оголошені в Microsoft.VisualStudio.QualityTools.UnitTesting.Framework. За допомогою технології Reflection «движок» модульного

тестування знаходить всі тестові класи в проєкті, помічені відповідним атрибутом, а всередині всі необхідні для тестування методи.

На початку тесту оголошується значення всіх необхідних змінних, а також очікуване вихідне значення. Потім відбувається виклик потрібного методу, якому передаються необхідні параметри. У нашому випадку це actual = BaseCalculator.Test.BaseCalculator_CalcClassAccessor.Add (a, b);

Потім йде виклик двох методів класу Assert. Перш за все розглянемо другий метод.

Assert.Inconclusive ("Verify the correctness of this test method.");

Наявність цього методу в тесті говорить про те, що реалізація тесту ще не закінчена. Реалізуємо наш метод:

```
/// <summary>
/// A test for Add (long, long)
/// </ summary>
[DeploymentItem ( "BaseCalculator.exe")]
[TestMethod ()]
public void AddTest ()
{
 long a = 150;

 long b = 350;

 int expected = 500;
 int actual;

 actual = BaseCalculator.
 Test.BaseCalculator_CalcClassAccessor.Add (a, b);

 Assert.AreEqual (expected, actual,
 "BaseCalculator.CalcClass.
 Add did not return the expected value. ");
}
```

Створення тестів

Щоб запустити всі тести в рамках проєкту, необхідно просто запустити тестовий проєкт. Один з можливих способів зробити це – натиснути правою кнопкою миші на проекті BaseCalculator.Test в Solution explorer і вибрати Set as StartUp Project. Потім використовуємо пункти меню Debug-> Start (F5) або Debug-> Start Without Debugging (Ctrl + F5), щоб почати запуск тестів.

У вікні Test Results буде показаний список з усіма тестами проєкту. У момент початку виконання тесту в нашому проєкті містилося два тести: один повністю реалізований тест AddTest, другий – незакінчений AddTest1. У момент запуску обидва тести будуть в стані «незакінчений» (Pending), але як тільки тести буде виконано, з'являться результати виконання Passed і Failed, які ми і очікували (Рис. 5.3).

Test run failed Results: 7/9 passed; Item(s) checked: 2

Result	Test Name	Project	Error Message
Passed	ABSTest2	Test_Lab_5	
Passed	ABSTest	Test_Lab_5	
Passed	ABSTest5	Test_Lab_5	
Passed	ABSTest8	Test_Lab_5	
Passed	ABSTest6	Test_Lab_5	
Passed	ABSTest1	Test_Lab_5	
Passed	ABSTest7	Test_Lab_5	
Failed	ABSTest4	Test_Lab_5	Assert.AreEqual failed. Expecte
Failed	ABSTest3	Test_Lab_5	Assert.AreEqual failed. Expecte

Рис. 5.3. Вікно Test Results після виконання всіх тестів

Щоб подивитися додаткові деталі про тест, ми можемо двічі клікнути на ньому у вікні Test Results і відкрити вікно AddTest [Result]. У ньому можна дізнатися інформацію про швидкість виконання тесту, його результати, помилку, що виникла та інше.

Крім того, ми можемо натиснути правою кнопкою миші на окремих тестах і вибирати пункт меню Open Test, щоб переміститися на код тесту.

Обробка винятків

У лабораторній роботі 4 ми виявили, що метод RunEstimate () класу AnalaizerClass не досить добре перевіряє об'єкти, з якими він працює. Якщо ініціалізувати список opz значенням {2,2, +, +}, то виконання методу RunEstimate () призводить до генерації виключення. Дійсно, реалізуємо тест:

```
/// <summary>
/// A test for RunEstimate ()
/// </ summary>
[DeploymentItem ( "BaseCalculator.exe")]
[TestMethod ()]
public void RunEstimateTest ()
{
 string expected = null;
 string actual;
 // Підготовка тестового оточення

BaseCalculator.Test.BaseCalculator_CalcClassAccessor._lastError =
"";

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz =
      new System.Collections.ArrayList ();

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Ad
d ( "2");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Ad
d ( "2");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Ad
d ( "+");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Ad
d ( "+");
```

```
            actual =
BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.RunEs
timate ();

            Assert.AreEqual (expected, actual,
            "BaseCalculator.AnalaizerClass.RunEstimate did not return
the expected value.");
        }
```

Для роботи цього тесту необхідно створити початкове тестове оточення, при цьому значення _lastError необхідно очистити, так як воно буде «зіпсоване» тестом AddTest1 ().

Незважаючи на те, що явних блоків try-catch немає, згенерований виняток не приведе до припинення роботи тестів, а буде коректно оброблено. У цьому можна переконатися у вікні RunEstimateTest [Result].

Припустимо тепер, що при невірних вхідних параметрах метод RunEstimate () дійсно повинен генерувати виняток, який буде перехоплюватися в іншому місці. Створимо ще один тест:

```
        /// <summary>
        /// A test for RunEstimate ()
        /// </ summary>
        [DeploymentItem ( "BaseCalculator.exe")]
        [TestMethod ()]
        [ExpectedException (typeof
(ArgumentOutOfRangeException),
            "Була оброблена невірна синтаксична конструкція")]
        public void RunEstimateTest1 ()
        {

BaseCalculator.Test.BaseCalculator_CalcClassAccessor._lastError =
"";

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz =
            new System.Collections.ArrayList ();
```

```
BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Add ( "2");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Add ( "2");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Add ( "+");

BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.opz.Add ( "+");


BaseCalculator.Test.BaseCalculator_AnalaizerClassAccessor.RunEstimate ();
        }
```

Відзначимо, що, знову ж таки, немає блоку try-catch з явним тестом на ArgumentOutOfRangeException. Замість цього тест включає додатковий атрибут, ExpectedException, який приймає тип параметра, і довільне повідомлення про помилку. Коли тести виконуються, середовище буде явно стежити за тим, щоб виняток ArgumentException був створений, і якщо метод не буде генерувати такий виняток, то тест буде провалено.

Завдання

У матеріалах для лабораторної роботи представлені вихідні файли модулів для тестування методом «білого ящика» засобами Visual Studio, приклад тестового драйвера.

Скласти тест-план і провести модульне тестування (Засобами Visual Studio) наступних методів (номер завдання оберіть відповідно до Вашого варіанту за списком у журналі):

1. public static int Mod (long a, long b)
   public static bool CheckCurrency ()

2. public static int ABS (long a)

public static string Format ()

3. public static int IABS (long a)
public static string Format ()

4. public static int Sub (long a, long b)
public static System.Collections.ArrayList CreateStack ()

5. public static int Mult (long a, long b)
public static System.Collections.ArrayList CreateStack ()

6. public static int Div (long a, long b)
public static bool CheckCurrency ()

7. public static int ABS (long a)
public static string Format ()

8. public static int IABS (long a)
public static string Format ()

9. public static int Sub (long a, long b)
public static System.Collections.ArrayList
CreateStack ()

10. public static int Mult (long a, long b)
public static System.Collections.ArrayList CreateStack ()

Лабораторна робота 6. Покриття програмного коду

Мета:

- розглянути питання покриття програмного коду;
- розглянути методи перевірки покриття, методи поліпшення покриття, покриття за методом MC / DC;
- розглянути процес збору покриття і генерації звітів про покриття в Visual Studio.

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *теми 4*.

У ході виконання цієї лабораторної роботи познайомимося з однією з оцінок якості системи тестів, а саме 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 тестованої 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, звані відсотком покритих тест-вимог. Покриття вимог дозволяє оцінити ступінь повноти системи тестів по відношенню до функціональності системи, але не дозволяє оцінити повноту по відношенню до 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 певна ділянка 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 невикористаний код. Таким чином, відсутність покриття будь-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.

Рівні покриття

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, а саме керуючі вхідні змінні. Зміна значень цих змінних буде впливати на те, які рядки коду будуть виконуватися в різних тестових прикладах.

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, досить 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. В результаті всі рядки коду виявляються виконаними. Легко побачити, що, незважаючи на повне покриття по рядках, цей набір тестових прикладів не перевіряє всієї функціональності (навіть коли ми не бачимо вимог, логічно припустити, що в них має описуватися поведінка 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 логічних операторів || 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, який встановлює condition в true, буде досить для досягнення необхідного рівня покриття. При цьому аналогічний тестовий приклад, який встановлює значення condition в false, дасть занадто низький рівень покриття.

За гілками умовних операторів (Decision Coverage)

Для забезпечення повного покриття за цим методом 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 за гілками потрібно як мінімум два тестових приклади.

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 за гілками потрібно вже три тестових приклади. Це пов’язано з тим, що перший умовний оператор if має неявну гілка – порожню гілку else. Для забезпечення покриття за гілками необхідно покривати і порожні гілки.

	1	**2**	**3**
prev	оператор	оператор	операнд
ShowMessage	true	false	true

Перші два тестових приклади аналогічні попередньому випадку, третій призначений для покриття неявної гілки. При цьому треба зауважити, що значення змінної ShowMessage не має ніякого значення для покриття – ділянка коду, який використовує цю змінну, просто не виконується.

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

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

повне покриття за гілками 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 цієї ділянки коду буде повним. Для перевірки виклику методу Method () необхідно додати ще один тестовий приклад (який, проте, не покращує ступеня покриття за гілками):

	1	**2**	**3**
condition1	true	false	true
condition2	true	true/false	false

За компонентами логічних умов

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

Покриття за умовами (Condition Coverage)

Для забезпечення повного покриття за цим методом кожна компонента логічної умови в результаті виконання тестових прикладів повинна вживати всіх можливих значення, але при цьому не потрібно, щоб сама логічна умова приймала всі можливі значення. Так, наприклад, при тестуванні наступного фрагмента

```
if (condition1 || condition2)
 MethodA ();
else
 MethodB ();
```

для покриття за умовами потрібно два тестових приклади:

	1	**2**
condition1	true	False
condition2	false	True

При цьому значення логічної умови приматиме значення тільки true; таким чином, при повному покритті за умовами не буде досягатися покриття за гілками.

Покриття за гілками / умовами (Condition / Decision Coverage)

Даний метод поєднує вимоги попередніх двох методів: для забезпечення повного покриття необхідно, щоб як логічна умова, так і кожна її компонента вжила всіх можливих значень.

Для покриття розглянутого вище фрагмента з умовою (Condition1 || condition2) буде потрібно 2 тестових приклади:

	1	**2**
condition1	true	false
condition2	true	false

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

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 в разі n компонент потрібно 2n тестових прикладів, кожен з яких перевіряє один набір значень. Тести, необхідні для повного покриття за цим методом, дають повну таблицю істинності для логічного виразу.

Незважаючи на очевидну повноту системи тестів, що забезпечує цей рівень покриття, даний метод рідко застосовується на практиці в 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

потрібна різна кількість тестових прикладів. Для першого випадку для повного покриття потрібно 6 тестів, для другого – 11.

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 за гілками / умовами (Modified Condition / Decision Coverage або MC / DC).

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

- кожна логічна умова повинна вживати всіх можливих значень;
- кожна компонента логічної умови повинна хоча б один раз вживати всіх можливих значення;
- має бути показаний незалежний вплив кожної з компонент на значення логічної умови, тобто вплив при фіксованих значеннях інших компонент.

Покриття за цією метрикою вимагає досить великої кількості тестів для того, щоб перевірити кожну умову, яка може вплинути на результат виразу, однак ця кількість значно менша, ніж вимагається для методу покриття за всіма умовами.

Приклад 1. Розглянемо фрагмент коду, який ми використовували як приклад для покриття за рядками і за гілками. Для покриття цієї ділянки коду за методом MC / DC введемо умовні позначення. Позначимо перевірку prev == "оператор" як A, перевірку prev == "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 однаковості.

З урахуванням зроблених позначень фрагмент коду може бути записаний так:

```
if (A || B)
{
 if (C)
 {
 ...
 }
 else
 {
 ...
 }
}
```

Для тестування першої умови за MC / DC треба показати незалежність результату (тобто функції A || B) від кожного аргументу. Відповідно, для цього використовуються три тестових приклади:

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 в MC / DC) в залежності від умови C необхідно, щоб в тестових прикладах C приймало значення як true, так і false.

Підсумкова таблиця тестових прикладів для покриття за 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 за 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 ());
}
```

Початковий умовний вираз в операторі if можна записати як (A & B) || C. Цей вираз залежить від 3 змінних, тобто може бути розглянуто як булева функція з трьома

аргументами. Згідно з методом MC / DC для тестування функції з трьома входами досить 4 тестових приклади: один базовий і три показують незалежний вплив кожного входу на вихід.

Почнемо побудова набору тестів з самої 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 приклади:

1. A & B = 0,C = 0 (Базовий приклад)
2. A & B = 0,C = 1 (Незалежний вплив C на вихід)
3. A & B = 1,C = 0 (Незалежний 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 за 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 C 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 приклади 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, "p")	«M» (m, але не p)	«M» (m, але не p)

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, за необхідності, додаткові перевірки, спрямовані на з'ясування причини недостатнього покриття, а також визначення необхідних дій по його усуненню. 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 належить до розряду А. Для рівня А необхідно перевірити об'єктний код, згенерований компілятором, щоб встановити, трасується він в оригінальний текст чи ні. Якщо об'єктний код не

трасується в оригінальний текст, повинні бути проведені перевірки 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.

Звіти про покриття програмного коду

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 проєкту або декількох проєктів і часто залежить від особливостей інструментальних засобів тестування.

У звіті про покриття в стандартизованій 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, а саме документи, де описуються об'єкти розробки, які необхідно змінити, і причини цих змін.

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 в разі нештатної роботи системи. В 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 описана в вимогах).

Можливі форми звітів про покриття

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

1. назва функції або методу;
2. тип покриття (за рядками, за гілками, MC / DC або інший);
3. кількість елементів, що покриваються, 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 – загальний 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 проєкту, або вибірково для визначених функцій або методів.

У разі, якщо обсяг 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 застосовується для покриття гілок і логічних умов, але може використовуватися для покриття за рядками.

Можливості Visual Studio з побудови покриття коду

Щоб побачити, яка частина коду вашого проєкту фактично тестується, використовуйте такий інструмент для тестувальників в Visual Studio, як Покриття коду. Цей інструмент показує відсоток коду, який був виконаний, і «розфарбовує» його, показуючи, які рядки коду були виконані, а які ні.

В попередніх лабораторних роботах ми познайомилися з можливостями Visual Studio 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 в лабораторній роботі 2.

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. 6.1).

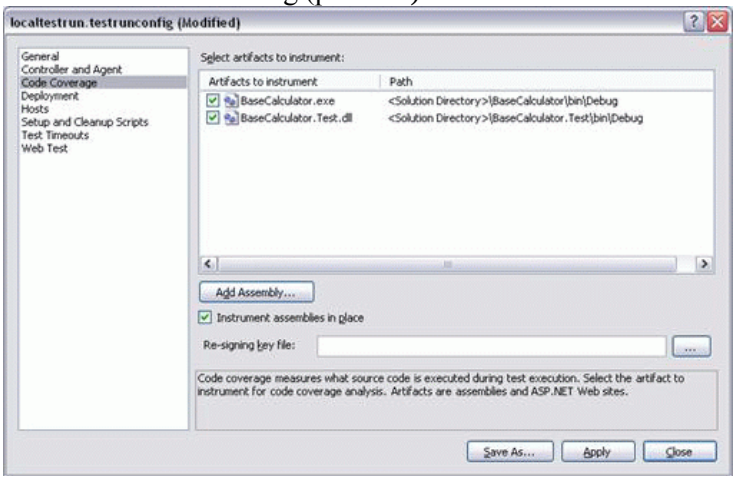


Рис. 6.1. Діалоговове вікно "localtestrun.testrunconfig"

Вибираємо Code Coverage.

В полі Select artifacts to instrument відзначаємо пункти BaseCalculator.exe і BaseCalculator.Test.dll

Натисніть Apply, потім закрийте діалогове вікно. Ми налаштували файл конфігурації.

Далі в меню Test виберіть Select Active Test Run Configuration. Підменю показує всі зміни 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 в лабораторній роботі 2 тести.

Після виконання всіх тестів у вікні Test Results в меню Test виберемо Windows і в підменю, що розкрилося, натиснемо на Code Coverage Results. Відкриється вікно Code Coverage Results.

Рядки в класі CalcClass представляють його методи. Колонки в вікні Code Coverage Results показують статистику покриття для окремих методів, для класів, і для всього простору імен.

Щоб подивитися, яка саме частина коду була покрита, клікнемо два рази на рядок з методом Add.

Відкриється файл вихідного тексту CalcClass.cs на методі Add. У цьому файлі ми бачимо «забарвлений» код. Лінії, пофарбовані в блакитний колір, були виконані в процесі виконання тестів, а лінії, пофарбовані в червоний, не були виконані. (Рис.6.2)

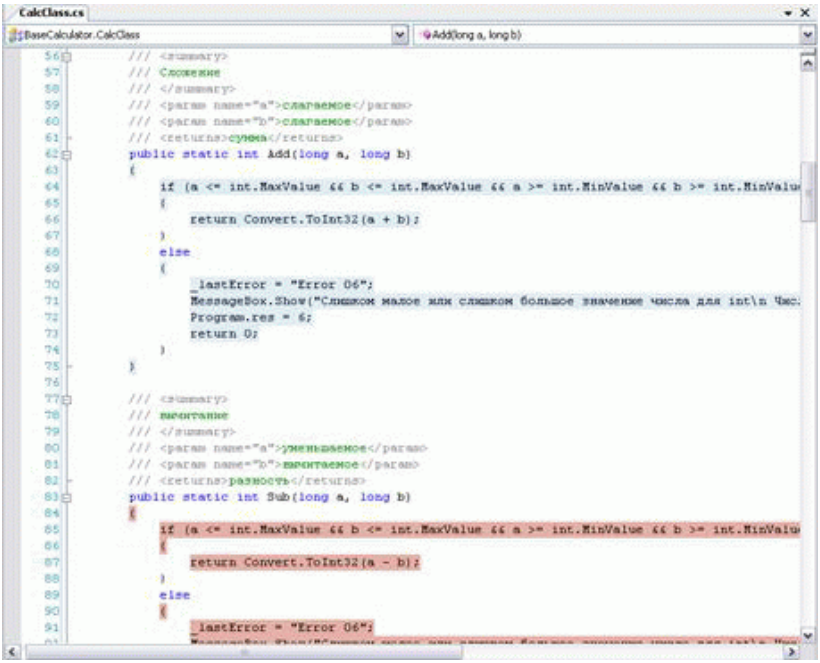


Рис. 6.2. «Фарбування» покриття коду

Прокручуючи файл, ви можете побачити в ньому покриття для інших методів.Точно так само 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).

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 місце розташування файлу.

Завдання

Для виконання лабораторної роботи Вам знадобляться модульні тести, які були отримані в ході виконання лабораторної роботи 2.

Необхідно доопрацювати модульні тести, домігшись максимального покриття коду за MC \ DC.

Лабораторна робота 7. Повторюваність тестування, залежності тестових прикладів

Мета:

- розглянути питання залежності тестових прикладів;
- розглянути ініціалізацію тестового оточення перед виконанням;
- виконати послідовність тестів.

Як вже було сказано в попередніх темах, тестування програмної системи – не разовий захід, а постійний процес, активний протягом всього 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 викликала 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 кожної 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, спричинені виправленням старих.

У загальному випадку повторне виконання тестів може завершитися одним з 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, які раніше виконувалися успішно, завершується з негативним результатом. Причини цього можуть бути такі:

- коректна зміна функціональності тестованої системи, в результаті якої тестовий приклад перестав відповідати вимогам;
- некоректна зміна функціональності системи, в результаті якої 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 в функціональних вимогах і тест-вимогах, а також поточного стану тест-планів і 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 в самому початку або під час виконання певного тестового прикладу.

Дана проблема найчастіше пов'язана зі зміною зовнішнього оточення частини системи, що тестується, яке моделює тестове оточення. Через такі зміни можуть змінюватися 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 потрібна 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 проєкту.

Процес, що включає в себе повторне виконання всіх тестів, називають регресійний тестуванням. Регресійне тестування включає в себе такі 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, що перевіряють функціональність системи або проведення аналізу, що дозволяє виявити тести, які повинні бути повторно виконані для тестування зміненої функціональності.
- Розробка тестових прикладів і тестового оточення з використанням методик, що полегшують модифікацію при змінах в тестованій системі.
- Розробка тестових прикладів, структура яких повністю виключає їх взаємний вплив за залишковими даними.

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

Вхідні дані в кожному тестовому прикладі явно задають початковий стан тестованої системи і режими її роботи при виконанні тестового сценарію.

Однак неявний 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 тестованої системи 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-плані, проте необхідний 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 йдуть кілька десятків тестових прикладів, а при регресійному тестуванні потрібно виконати, наприклад, тестові приклади з номерами від 25 до 40. Перший тестовий приклад при цьому ініціалізує систему, а решта працюють з вже системою, що вже стартувала. Якщо просто виконувати тестові приклади 25-40, то їх виконання виявиться неможливим, вони не ініціалізують систему. Розумним виходом з цієї ситуації є виконання тестових прикладів 1, 25-40.

Для полегшення проведення регресійного тестування (і тестування взагалі) тестові приклади часто розбивають на групи. Кожна група містить набір тестових прикладів, які перевіряють окрему замкнуту частину 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 встановлення початкового стану тестового оточення для виконання тестів: так, перед виконанням групи тестів можна форматувати значення змінних або стан системи, необхідний для виконання всієї групи. Наприклад, якщо система працює в двох режимах – нормальному і 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 налаштування 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, а саме 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 тривалий час виконання таких тестів).

У разі, коли в наборі тестових прикладів тести не є автономними, говорять про тестову залежності. Тестова залежність буває двох видів: передбачена структурою тестових прикладів і паразитна.

Приклад передбаченої тестової залежності було розглянуто в попередньому розділі – коректність виконання тестів визначалася порядком їх виконання. Така тестова залежність вимагає документування та супроводження, як і сам опис тестових прикладів.

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 в динамічній пам'яті при програмуванні.

Розглянемо повторюваність тестування на прикладі нашої системи «Калькулятор».

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 вона не дорівнює порожньому рядку, видає повідомлення про помилку і перериває роботу. Однак у властивості lastError немає аксессора set, і значить, ніякий зовнішній модуль не може поміняти його значення. Напрошується питання: а як же скидається це значення? Проведемо три тести поспіль на методі складання (див. дод. В).

Результат:

Test Case 1

Вхідні дані: a = 78508, b = -304

Очікуваний результат: res = 78204 && error = ""
Код помилки:
Одержаний результат: res = 78204 error =
Тест пройдений

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 пройдений

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 пройдений

Як видно, незважаючи на те, що третій тест операції додавання повинен бути виконаний, він не проходить, хоча за першим тесту видно, що додавання працює правильно, а значення lastError точно таке ж, що і в другому тесті. Це може свідчити, наприклад, про те, що під час виклику методу Add на початку своєї роботи не очищається поле _lastError. Проведемо тестування всіх функцій (див. дод. Г).

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 пройдений

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 пройдений

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 пройдений

Test Case 5 - перевіряємо добуток 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 пройдений

Ми бачимо, що жоден з методів не очищає поле _lastError. Це може бути або помилкою проєктування, 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 наступним 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.

Розглянутий приклад є досить простим, і помилка буде легко виявлена при тестуванні. У лабораторній роботі 4, 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 метод System.Reflection.Assembly.LoadFile, який відразу підключить необхідну збірку по шляху бібліотеки.

Далі ми можемо проводити скільки завгодно тестів методів класуAnalaizerClass.

Але якщо нам знадобиться поміняти заглушку класу CalcClass (наприклад, на ту, яка виводить на екран якісь додаткові відомості), то ми отримаємо помилку access denied, тому що збірка вже завантажена і використовується процесом, і перекомпілювати її не вийде.

Це інша проблема регресійного тестування. Тут вже помилка не в тестованій програмі, а в самій побудові тестів і тестового оточення. Як було сказано раніше, 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.

Впорядковані тести (ordered tests) в Visual Studio

Упорядкований тест містить в 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.

Розглянемо спочатку створення упорядкованого тесту.

Для початку відкриємо BaseCalculator, з яким ми працювали в лабораторній роботі 2 і лекції 4. Він вже містить тести.

Додамо в тестовий проєкт упорядкований тест. Для цього в меню Test виберемо New Test. У діалоговому вікні Add New Test вибираємо Ordered Test. В 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 проєкт натискаємо OK. (Рис.7.1)

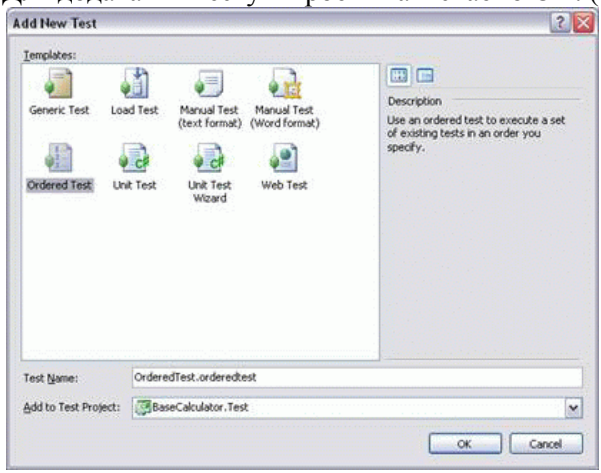


Рис. 7.1. Діалогове вікно Add New Test

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. 7.2).

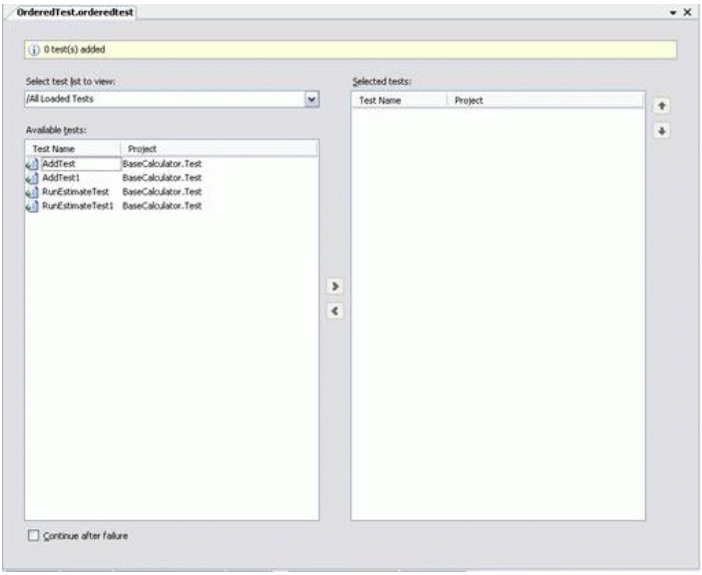


Рис. 7.2. Вікно редагування упорядкованого тесту

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.

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.

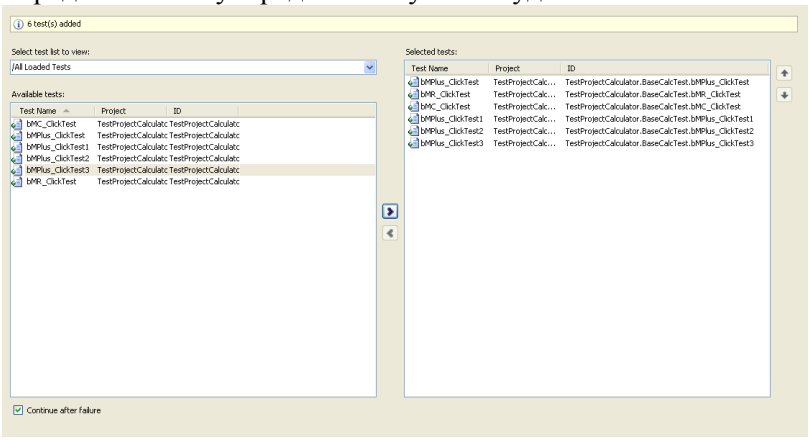


Рис. 7.3. Вікно редагування упорядкованого тесту

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 в вікні Test View на кнопку).

Відкриється вікно Test Results, в якому після виконання упорядкованого тесту відобразяться результати його виконання Passed або Failed (Рис.7.4).

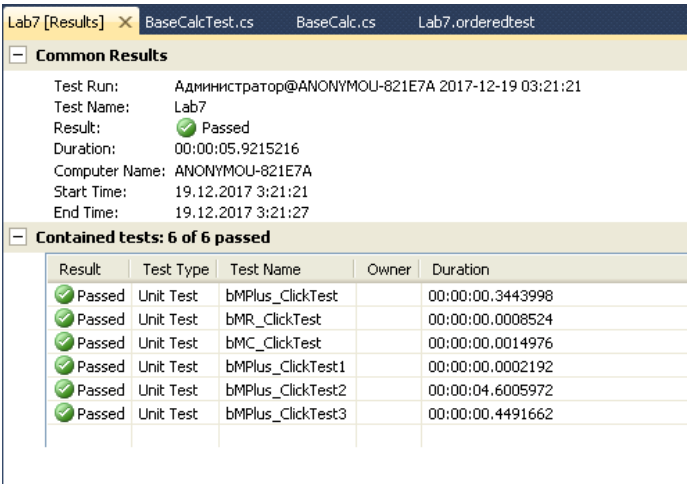


Рис. 7.4. Вікно Test Results

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 місце розташування файлу.

Завдання

У матеріалах для лабораторної роботи знаходяться вихідні тексти програми BaseCalculatorNew для тестування методом «білого ящика» засобами Visual Studio і приклад тестового драйвера.

Написати тести для методів роботи з пам'яттю калькулятора, використовуючи впорядковані тести (ordered Tests) і розташувавши тести в такому порядку, щоб якомога рідше проводити підготовку тестового оточення. Обгрунтувати свій вибір.

Лабораторна робота 8. Інтеграційне тестування

Мета:

- розглянути питання тестування міжмодульних інтерфейсів;
- розглянути питання тестування інформаційного обміну між модулями.

Для успішного виконання даної лабораторної роботи необхідно засвоїти теоретичний матеріал *теми 10.*

Результатом тестування і верифікації окремих модулів, що складають програмну систему, є висновок про те, що ці модулі є внутрішньо несуперечливі і відповідають вимогам. Однак окремі модулі рідко функціонують самі по собі, тому наступне завдання після тестування окремих модулів – тестування коректності взаємодії декількох модулів, об'єднаних в єдине ціле. Таке тестування називають інтеграційним. Його мета – переконатися в коректності спільної роботи компонент системи.

Інтеграційне тестування називають ще тестуванням архітектури системи. З одного боку, ця 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 тестованій реалізації системи. З іншого боку, результати виконання інтеграційних тестів – один з основних джерел інформації для процесу поліпшення і уточнення архітектури системи, міжмодульних і межкомпонентних інтерфейсів. Тобто з цієї точки зору інтеграційні тести перевіряють коректність взаємодії компонент системи.

В результаті проведення інтеграційного тестування та усунення всіх виявлених дефектів виходить узгоджена і цілісна архітектура програмної системи, тобто можна вважати, що

інтеграційне тестування – це тестування архітектури і низькорівневих функціональних вимог.

Інтеграційне тестування, як правило, являє собою ітеративний процес, при якому функціональність, що перевіряється, все більш і більш збільшується в розмірах сукупності модулів.

Як правило, інтеграційне тестування проводиться вже по завершенні модульного тестування для всіх інтегрованих модулів. Однак це далеко не завжди так. Існує кілька методів проведення інтеграційного тестування:

- висхідне тестування;
- монолітне тестування;
- низхідне тестування.

На практиці найчастіше в різних частинах проєкту застосовуються всі розглянуті методи в сукупності. Кожен модуль тестують у міру готовності окремо, а потім включають в уже готову композицію. Для одних частин тестування виходить низхідним, для інших – висхідним. У зв'язку з цим представляється корисним розглянути ще один тип класифікації типів інтеграційного тестування – класифікацію за частотою інтеграції:

- тестування з пізньою інтеграцією;
- тестування з постійною інтеграцією;
- тестування з регулярною або пошаровою інтеграцією.

Тестування з пізньою інтеграцією – практично повний аналог монолітного тестування. Інтеграційне тестування при такій схемі відкладається на якомога більш пізні терміни проєкту. 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, визначеного поза проєктом (наприклад, в разі, якщо система складається з окремих Web-сервісів).

Схематично тестування з пізньої інтеграцією може бути зображено у вигляді ланцюжка R-C-V-R-C-V-R-C-V-I-R-C-V-R-C-V-I, 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 має на 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. 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.

Схематично тестування з постійною інтеграцією може бути зображено у вигляді ланцюжка R-C-I-R-C-I-R-C-I, в якій фаза тестування модуля навмисно опущена і замінена на тестування інтеграції.

При тестуванні з регулярною або пошаровою інтеграцією інтеграційному тестуванню підлягають сильно пов’язані між собою групи модулів, 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 низхідного або висхідного тестування, напрямок проходу по ієрархії в цьому підході не задано.

Таблиця 8.1. Основні характеристики різних видів інтеграційного тестування

	Висхідне	Низхідне	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	так	так
Чи потрібні заглушки	так	так	немає	немає	немає	так

Таблиця 8.1 представляє основні характеристики розглянутих вище видів інтеграційного тестування. Час інтеграції характеризує момент часу, коли проводиться перше інтеграційне тестування і всі наступні, частота інтеграції – наскільки часто при розробці виконується інтеграція. Необхідність в драйверах і заглушках визначена в останніх двох рядках таблиці.

Як уже зазначалося, у VisualStudio під unit-testing мається на увазі саме інтеграційне тестування, а конкретно тестування з постійною інтеграцією. У лабораторії роботі 5 ми вже протестували метод RunEstimate (), при інтеграції класів AnalaizerClass iCalcClass. 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 прикладу на кожен метод (див. дод. Д).

Запустивши тести, можна переконатися в коректній (для даних тестових прикладів) роботі методів.

Як вже зазначалося в лабораторії роботі 4, ці методи мають багато недоліками, і деякі з них генерують виключення на некоректних вхідних даних. Це, в принципі, логічно, так як запуск методу RunEstimate () має на увазі, що вхідний вираз вже оброблено в методах CheckCurrency (), 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. Основну ж увагу варто приділити методу Estimate (), який по черзі запускає всі перераховані вище методи і має рівень достуку 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.

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 учасники (див. дод. Е).

Таким чином, перевіривши методи Main () і Estimate (), а також деякі методи візуального інтерфейсу, можна переконатися у відповідності системи вимогам або, навпаки, виявити помилки в міжмодульній взаємодії.

Завдання

У матеріалах для лабораторної роботи видані вихідні тексти програми BaseCalculatorNew для тестування методом «білого ящика» засобами VisualStudio і приклад тестового драйвера.

Скласти тест-план і провести інтеграційне тестування (засобами VisualStudio) методів Main () і Estimate ().

ПИТАННЯ ДО МОДУЛЬНОЇ РОБОТИ

1. Основні поняття тестування: відладка, тестування, фази тестування, дефект, верифікація, валідація.
2. Функції тестування. Планування випробувань.
3. Критерії вибору тестів, проєктування тестів та реалізація.
Види специфікацій ПЗ.
4. Основні характеристики специфікацій.
5. Моделі життєвого циклу розробки ПЗ.
6. Програмні та експлуатаційні документи.
7. Основні функції та ролі учасників розробки ПЗ.
8. Види тест - планів.
9. Основні компоненти тест – планів.
10. Основні розділи тест - планів.
11. Види тестування.
12. Етапи розробки тестової документації.
13. Атрибути тест - кейса.
14. Види тестування за ступенем ізольованості компонент.
15. Атрибути звіту про помилки.
16. Види та типи тестування. Їх характеристики, використання та застосування, переваги і недоліки.
17. Особливості тестування веб - додатків.
18. Огляд інструментів тестування веб - сайтів.
19. Українські та міжнародні стандарти ПЗ.
20. Особливості тестування об'єктно-орієнтованого програмного забезпечення.
21. Тестування класів. Побудова тестових випадків. Тестування взаємодії і функціонування компонент. Тестування ієрархій класів.
22. Автоматизація процесу тестування. Огляд інструментів автоматизації тестування.
23. Особливості документування тестових процедур для ручних і автоматизованих тестів.
24. Поняття та процес швидкого тестування. Компоненти швидкого тестування.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Бейзер Б. Тестирование чёрного ящика. Технология функционального тестирования / Б. Бейзер. – Санкт-Петербург, 2004. – 221 с.

2. Бек К. Экстремальное программирование: планирование / К. Бек. – Санкт-Петербург : Питер, 2003. – 143 с.

3. Канер С. Тестирование программного обеспечения/ Канер С., Фолк Дж.. Нгуен Енг// - К.: ДиаСофт. – 2000. – 544с.

4. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Л.Криспин, Д.Грегори // Вильямс. – 2016. – 464с.

5. Майерс Г. Искусство тестирования программ / Г. Майерс, Т. Баджетт, К. Сандлер // Вильямс. – 2012. – 272с.

6. Ошероув Р. Искусство автономного тестирования с примерами на C#. 2-е издание/ Р. Ошеуров. – М.: ДМК Пресс. – 2014. – 360с.

7. Плаксин М.А. Тестирование и отладка программ для професионалов будущих и настояних / М.А. Плаксин. – М.: БИНОМ. – 2015. – 170с.

8. Роман Савин. Тестирование Дот ком или Пособие по жесткому обращению с багами в интернет-стартапах. – Дело. – 2007. – 312с.

9. Хамбл Д. Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения нових версій программ / Д.Хамбл, Д. Фарли // Вильямс. – 2016. – 432с.

10. Черников Б. В. Оценка качества програмного обеспечения. Практикум /Б.В. Черников, Б. Е. Поклонов // М.: Издательский дом «Форум». – 2012. – 216с.

11. Gojko Adzic. Fifty Quick Ideas to Improve Your Tests / Gojko Adzic, David Evans, Tom Roden// Neuri Consulting LLP. – 2015. – 198p.

12.Jonathan Rasmusson. The Way of the Web Tester. A Beginner’s Guide to Automating Tests / J. Rasmusson// Pragmatic Bookshelf. – 2016. – 258p.

```
private void buttonStartDel_Click (object sender, EventArgs e)
 {
 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 + = "перехоплений 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 + = "перехоплений виняток:" +
ex.ToString () + "\ nТест не пройдений. \ n";
}
try
{
richTextBox1.Text + = "Test Case 3 \ n";
richTextBox1.Text + = "Вхідні дані: a = 3000000000, b = -
2850800078 \ n ";
richTextBox1.Text + = "Очікуваний результат: res = 0 && error =
\ "Error 06 \" \ n ";
int res = CalcClass.Add (3000000000, -2850800078);
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 + = "перехоплений виняток:" +
ex.ToString () + "\ nТест не пройдений. \ n";
}
try
{
richTextBox1.Text + = "Test Case 4 \ n";
richTextBox1.Text + = "Вхідні дані: a = 2000000000, b =
2000000000 \ n ";
richTextBox1.Text + = "Очікуваний результат: res = 0 && error =
\ "Error 06 \" \ n ";
int res = CalcClass.Add (2000000000, 2000000000);
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 + = "перехоплений виняток:" +
ex.ToString () + "\ nТест не пройдений. \ n";
}
}
```

```
private void buttonStart_Click (object sender, EventArgs e)
 {
 // створюємо провайдер для генерування і компіляції коду на C#
 System.CodeDom.Compiler.CodeDomProvider prov =
System.CodeDom.Compiler.CodeDomProvider.CreateProvider
("CSharp");
 // створюємо параметри компіляції
 System.CodeDom.Compiler.CompilerParameters cmpparam = new
System.CodeDom.Compiler.CompilerParameters ();
 // результат компіляції - бібліотека
 cmpparam.GenerateExecutable = false;
 // не включаємо інформацію відладчика
 cmpparam.IncludeDebugInformation = false;
 // підключаємо 2-е стандартні бібліотеки та бібліотеки
CalcClass.dll
 cmpparam.ReferencedAssemblies.Add (Application.StartupPath +
"\\ CalcClass.dll");
 cmpparam.ReferencedAssemblies.Add ( "System.dll");
 cmpparam.ReferencedAssemblies.Add
("System.Windows.Forms.dll");
 // ім’я вихідний збірки - My.dll
 cmpparam.OutputAssembly = "My.dll";
 // компілюємо клас AnalaizerClass із заданими параметрами
 System.CodeDom.Compiler.CompilerResults res =
prov.CompileAssemblyFromFile                    (cmpparam,
Application.StartupPath + "\\ AnalaizerClass.cs");
 // Виводимо результат компіляції на екран
 if (res.Errors.Count! = 0)
 {
 richTextBox1.Text + = res.Errors [0] .ToString ();
 }
 else
 {
```

```
 // завантажуємо тільки що скомпільовану збірку (тут тонкий момент: якщо ми прото завантажимо збірку з файлу, то він буде заблокований,
 // acces denied, тому спочатку читаємо його в потік і лише потім підключаємо)
 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);
 Type [] types = assm.GetTypes ();
 Type analaizer = types [0];
 // знаходимо метод CheckCurrency (він єдиний)
 System.Reflection.MethodInfo addinfo =
analaizer.GetMethod ( "RunEstimate");
 System.Reflection.FieldInfo fieldopz =
analaizer.GetField ( "opz");
 System.Collections.ArrayList ar = new
System.Collections.ArrayList ();
  ar.Add ( "2");
 ar.Add ( "2");
 ar.Add ( "+");
 fieldopz.SetValue (null, ar);
 richTextBox1.Text + = addinfo.Invoke (null, null) .ToString ();
 asmBytes = null;
 }
 prov.Dispose ();
}
```

```
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 + = "перехоплений 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 + = "перехоплений 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 + = "перехоплений виняток:" +
ex.ToString () + "\ n Тест 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 + = "перехоплений виняток:" +
 ex.ToString () + "\ n Тест 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 + = "перехоплений 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 + = "перехоплений виняток:" +
 ex.ToString () + "\ nТест не пройдений. \ n";
 }

 try
 {
 richTextBox1.Text + = "Test Case 5 - перевіряємо добуток 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 + = "перехоплений виняток:" +
 ex.ToString () + "\ n Тест не пройдений. \ n";
 }
```

```
/// <summary>
 /// A test for RunEstimate ()
 /// Перевіряємо, що, якщо в стеці знаходиться коректний вираз,
представлений зворотнім польським записом, то
 /// метод RunEstimate правильно порахує цей 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.RunE
stimate ();

 Assert.AreEqual (expected, actual,
 "BaseCalculator.AnalaizerClass.RunEstimate did not return the
expected value.");
 }

 /// <summary>
 /// A test for CheckCurrency ()
```

```
 /// Перевіряє, що, якщо в вираженні порушена дужкова
структура, то метод повертає false
 /// </ summary>
 [DeploymentItem ( "BaseCalculator.exe")]
 [TestMethod ()]
 public void CheckCurrencyTest ()
 {
 bool expected = false;
 bool actual;


TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.expre
ssion =
 "((5 + 3) * 2 (3 * 10)) - 2) + ((5 + 3)";
 actual =
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Chec
kCurrency ();

 Assert.AreEqual (expected, actual,
 "BaseCalculator.AnalaizerClass.CheckCurrency did not return the
expected value.");
 Assert.AreEqual (18,

TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.erpos
ition,
 "BaseCalculator.AnalaizerClass.CheckCurrency did not return the
expected value.");
 }

 /// <summary>
 /// A test for CreateStack ()
 /// Перевіряє, що якщо відформатований вираз дорівнює
 /// "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)",
 /// то стек містить наступні 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.expre
ssion =
 "((5 + 3) * 2 - (3 * 10)) - 2 + (5 + 3)";


actual =
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Creat
e
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, так
само
 /// ((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.expre
ssion =
 "((5 + 3) * 2 (3 * 10)) - 2 + (5 + 3)";
 actual =
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Esti
mate ();

 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 виду:
 /// "((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.expre
ssion =
 "((5 + 3) * 2 (3 * 10)) - 2 + (5 + 3)";
 actual =
TestProjectCalculator.BaseCalculator_AnalaizerClassAccessor.Form
at ();

 Assert.AreEqual (expected, actual,
 "BaseCalculator.AnalaizerClass.Format did not return the expected
value.");

 }
```

```
/// <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 в разі, якщо будь-який з доданків більше ніж
MAXINT або менше ніж MININT,
 /// програма повинна видавати помилку Error 06 (див. 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.");
 }
```

Навчально-методичне видання

СМАГІНА Ольга Олександрівна
ПЕРЕЯСЛАВСЬКА Світлана Олександрівна

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

Навчальний посібник
для студентів спеціальності
121 – „Інженерія програмного забезпечення”

Редактор – Смагіна О.О.
Комп’ютерний макет – Смагіна О.О., Переяславська С.О.

Здано до склад. 26.02.2021 р. Підп. до друку 26.03.2021 р.
Формат 60x84 1/16. Папір офсет. Гарнітура TimesNewRoman.
Друк ризографічний. Ум. друк. арк. 9,58. Наклад 50 прим. Зам. № 19.

Видавець і виготовлювач
Видавництво Державного закладу
„Луганський національний університет імені Тараса Шевченка”
пл. Гоголя, 1, м. Старобільськ, 92703. Тел./факс: (06461) 2-26-70.
e-mail: mail@luguniv.edu.ua
Свідоцтво суб’єкта видавничої справи ДК № 3459 від 09.04.2009 р.

286