Ю.С. Манжос

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

2012

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
Національний аерокосмічний університет ім. М.Є. Жуковського
«Харківський авіаційний інститут»

Ю.С. Манжос

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

Навчальний посібник

Харків «ХАІ» 2012

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
Національний аерокосмічний університет ім. М.Є. Жуковського
«Харківський авіаційний інститут»

Ю.С. Манжос

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

Навчальний посібник

Харків «ХАІ» 2012

УДК 004.05(075.8)
М23

Рецензенти: д-р техн. наук, проф. Є.А. Фролов,
д-р техн. наук, проф. С.Ю. Шабанов-Кушнаренко

Манжос, Ю.С.
М23 Якість програмного забезпечення та тестування [Текст] : навч. посіб. / Ю.С. Манжос. – Х. : Нац. аерокосм. ун-т ім. М.Є. Жуковського «Харк. авіац. ін-т», 2012. – 198 с.

Розглянуто основні характеристики якості, а також процеси життєвого циклу програмного забезпечення та їх документування. Визначено фактори, що впливають на якість програмного забезпечення. Показано особливості життєвого циклу програмних систем як широкого, так і критичного застосування. Велику увагу приділено методам забезпечення якості і особливостям тестування широкого кола програмних систем.

Для студентів комп'ютерних спеціальностей. Буде корисним аспірантам і здобувачам наукових ступенів.

Іл. 19. Табл. 6. Бібліогр.: 23 назви

УДК 004.05(075.8)

© Манжос Ю.С., 2012
© Національний аерокосмічний університет ім. М.Є. Жуковського «Харківський авіаційний інститут», 2012

ПЕРЕЛІК СКОРОЧЕНЬ

АЕС	–	атомна електростанція
ДПЗ	–	дефект програмного забезпечення
ДСТУ	–	Державний стандарт України
ЖЦ	–	життєвий цикл
ЕАК	–	елемент апаратної конфігурації
ЕОМ	–	електронно-обчислювальна машина
ЕКПЗ	–	елемент конфігурації програмного зебезпечення
ІЗ	–	інструментальний засіб
ІКС	–	інформаційно-керувальна система
МЕК	–	міжнародний електротехнічний комітет
ПЗ	–	програмне забезпечення
ПП	–	прикладна програма
ПС	–	програмна система
ПТК	–	програмно-технічний комплекс
РКК	–	ракетно-космічний комплекс
СА	–	статичний аналіз
ТУ	–	технічні умови
ФБ	–	функціональна безпека

1. ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Класифікація проблем, які виникають під час експлуатації програмних систем

Безпека держави визначається безпекою процесів в економіці, техніці, екології, державному керуванні та інших галузях людської діяльності. Це потребує застосування програмно-технічних комплексів (ПТК), з допомогою яких керують у реальному часі складними технологічними об'єктами та гарантують безпеку продукції, процесів виробництва, унеможливлюючи відмови і збитки, що пов'язані із заподіянням шкоди життю або здоров'ю громадян, майну, середовищу. Інформаційно-керувальні системи (ІКС) мають обмежений термін для виконання у реальному часі важливих або критичних з огляду на ресурсні витрати задач. Невиконання може мати катастрофічні наслідки, тому велику частку ІКС відносять до систем критичного застосування.

Типові представники – військові системи: літаки, кораблі, танки, тактичні й стратегічні ракети; космічні системи; командні, керувальні, комунікаційні та інтелектуальні системи. До 80 % функцій ІКС реалізовано програмно, тому їх відносять до класу систем з інтенсивним використанням програмного забезпечення (ПЗ). Це обумовлює тенденцію експоненціального зростання обсягів ПЗ (рис. 1), який сягнув десяти мільйонів інструкцій для літака F-18 та ста мільйонів – для ІКС атомних електростанцій (АЕС).

Разом з цим зростає й «вага» дефектів програмного забезпечення (ДПЗ) як джерел відмов, оскільки із зростанням складності збільшується й кількість залишкових ДПЗ, що зменшує потенційну безпеку функціонування технічних систем. За деяким оцінюванням ДПЗ призводять до 70 % відмов ІКС у різних галузях енергетики, у ракетно-космічних комплексах (РКК), банківській діяльності, медицині, та до 60 млрд дол. за рік у загальному обсязі збитків економіки США. Тенденція має динаміку наростання в часі. Так, у 70-ті роки відмови апаратури і ПЗ становили 75–85 % і 15–25 % причин відмов ІКС відповідно 2000 року частка відмов ПЗ збільшилася до 60–70 %.

Ще яскравіше тенденція виявляється в РКК. За останні сорок років кожна п'ята аварія в РКК пов'язана з відмовою ІКС, а шість із семи відмов обумовлені залишковими ДПЗ [1]. Наприклад, аварія у вересні 1999 р. апарата Mars Climate Orbiter в атмосфері Марса сталася через неузгодженість фізичних одиниць, використаних різними групами розробників ПЗ, аварія Ariane-5 1996 року через некоректне повторне використання ПЗ і єдиний ДПЗ, які призвели до знищення обладнання

вартістю понад 500 млн доларів. Є приклади інших аварій (Марс-І 1976 р., Аріан-3 1997 р., Аполон-13 1978 р.) та великих фінансових і матеріальних втрат через ДПЗ, що в найважчих випадках вимірювалися життям і здоров'ям людей.

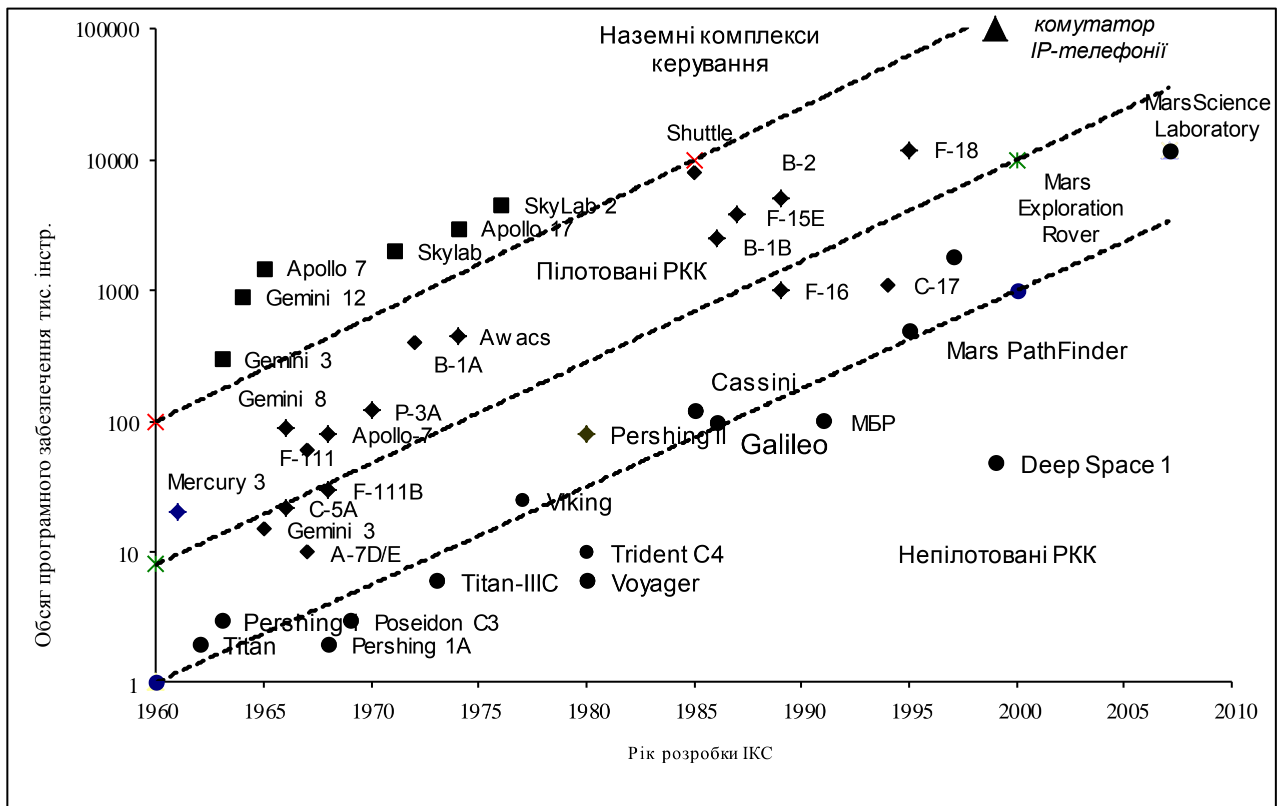

Рис. 1. Зростання обсягів ПЗ ІКС

Існування сучасного суспільства неможливе без використання складних технічних систем, ефективне функціонування яких, у свою чергу, неможливе без комплексної автоматизації виробничих процесів і вдосконалення керування шляхом широкого застосування прогресивних інформаційних технологій. Автоматизація процесів у різних галузях потребує розроблення не тільки ПЗ для ІКС, але й інструментальних засобів (ІЗ) для їх створення, тому що кожні десять років обсяг ПЗ збільшується у 10 разів. Характерна тенденція сучасності – збільшення відносної частки критичних функцій ІКС, реалізованих програмно, тому програмні системи (ПС) стають причиною великих втрат через залишкові ДПЗ, що є факторами ризику виникнення аварійних ситуацій. Тому вартість розробки сучасних авіаційних ІКС сягає 30 % від вартості розробки літака, а трудомісткість – сотен і тисяч людино-років. Так, супроводження ПЗ Shuttle обсягом 400 тис. інструкцій, 40 % з яких з 1980 р. по 1990 р. було модифіковано, коштувало 100 млн дол./рік.

Міжнародні стандарти визначають чотири рівні безпеки для технічних комплексів критичного використання:

A – відмова може мати інтенсивність межах $10^{-9}\dots10^{-8}$ год$^{-1}$ і стати причиною загибелі великої кількості людей;

B – відмова може мати інтенсивність межах $10^{-8}\dots10^{-7}$ год$^{-1}$ і призвести до загибелі декількох людей;

C – відмова може мати інтенсивність в межах $10^{-7}...10^{-6}$ год$^{-1}$ і може являти собою серйозну загрозу здоров'ю і життю декількох чоловік;

D – відмова може мати інтенсивність в межах $10^{-6}\dots10^{-5}$ год$^{-1}$ і призводити лише до незначної шкоди без ризику для людей.

Відмови ІКС становлять близько 20 % від всіх відмов устаткування. Серед відмов комп'ютерних систем частка відмов ПЗ за різним оцінюванням для систем, що обслуговуються, становить близько 30 %, для автономних систем – близько 90 %.

Надійність і безпека ІКС визначаються якістю проектування на етапах, що передують розробленню ПЗ [2]. Помилки цих етапів спричиняють складні ДПЗ. Однак ПЗ не тільки акумулює недоліки попередніх етапів, аде й дає змогу виявити й усунути ДПЗ. Трудомісткість при цьому зростає на порядок. Кваліфікація персоналу мало впливає на кількість ДПЗ, а кожні 1000 рядків нетестованого й некоментованого коду мають близько 50 ДПЗ. Ретельне тестування дає змогу досягти 10, а додаткові заходи – одного ДПЗ на 1000 рядків коду.

Безпека ІКС істотно залежить від ступеня відповідності ПЗ і процесів його розроблення регулювальним вимогам національних і міжнародних стандартів, тому якість і безпека ПЗ – важливі складові нормативного регулювання під час розроблення й експлуатації.

Щоб технічні системи експлуатувалися безпечно, потрібно провести незалежну верифікацію. Великий внесок у розроблення методів верифікації зробили Б.У. Боем, Р. Глас, Б.М. Конорев, В.В. Липаєв, М. Липов, Г. Майєрс, Е. Нельсон, Т. Тейєр, В.В. Шураков та ін.

Розглянемо зовнішні ефекти, що виявляються через наявність ДПЗ та розрізняються насамперед за ступенем серйозності наслідків від вияву дефекту і часу знаходження системи в непрацездатному стані.

Перший вияв дефекту – збій в роботі системи. Збої мають невелику тривалість в часі і їх можна усунути без тривалих процедур відновлення. Зазвичай збій спричиняє короткочасне спотворення даних користувача без припинення роботи всієї системи в цілому. Наслідки збою можуть бути істотними, особливо якщо дані є дуже важливими, проте безперебійна робота системи не порушується.

Відмова – більш серйозний вияв дефекту в системі, при якому вся система або її частина виходить із працездатного стану, тобто з такого, в якому всі аспекти функціонування системи відповідають певним вимогам. У разі відмови системи для її повернення до нормального функціонування потрібне втручання оператора. Для програмних систем причиною відмови

може бути прихований дефект, що виявляється тільки протягом великого проміжку часу (переповнення внутрішнього лічильника часу, даних тощо).

Аварія – відмова системи, при якій система виходить з ладу таким чином, що відновлення її працездатного стану або є неможливим, або забирає багато часу. Можна уникнути виникнення аварійних ситуацій у програмних системах з допомогою повного дублювання системи як за виконуваним програмним кодом, так і за даними.

Збої і відмови є причиною таких ситуацій, у яких працездатний стан системи порушується тимчасово. Аварії є причиною таких ситуацій, у яких працездатний стан системи порушується назавжди або на тривалий термін.

1.1.1. Збої програмного забезпечення

Можна виділити три види збоїв, що призводять до відмовних ситуацій:

– збої в системному ПЗ, які виникають при нештатному використанні системних засобів – операційної системи, системи керування базами даних тощо. Зазвичай наслідки цих збоїв є найбільш важкими. У деяких випадках можлива повна втрата як даних системи, так і даних про стан системи на момент збою – дампів. Такі випадки є найбільш складними для діагностики й виправлення;

– збої в прикладній програмі (ПП), що виникають при недостатній якості тестування прикладної системи або нештатному її використанні. Зазвичай зібрати інформацію про такі збої можна засобами самої системи. У критичних випадках, наприклад при повному краху ПП, можна зібрати необхідні дані про його інформаційне оточення засобами операційної системи або операційного середовища;

– збої – наслідки некоректного використання – виникають при неправильних (непередбачених) діях користувача під час роботи із системою. Вияви збоїв, найбільш складних для аналізу й усунення, можуть полягати не у відмовах системи, а в неправильних або неочевидних з погляду користувача системних діях. При цьому не відбувається автоматична розсилка інформації розробникам, оскільки єдина інформація, на яку доводиться спиратися, – зворотний зв'язок від користувачів. Усунення причин цих збоїв може вестися у декількох напрямах. Слід зазначити, що доробка інструкції користувача є не завжди ефективною, оскільки уважно читає інструкцію лише невелика кількість користувачів, а залучення до розроблення фахівця з предметної області, що автоматизується, або фахівця з ергономіки дасть можливість зробити інтерфейс системи більш зручним і зрозумілим.

Для характеризації збоїв важливими є такі ознаки:

– точка виникнення збою – рядок або оператор програмного коду, що

спричинив відмовну ситуацію; такий оператор може знаходитися в кодах як системних бібліотек, так і програм користувача; зовсім не обов'язково, що збій спричинено саме цим оператором, але з допомогою аналізу оточення виклику і програмних текстів зазвичай можна знайти причину відмови;

– інформаційне оточення системи в момент збою – стан системи в момент збою; до інформаційного оточення в цьому випадку відносять дані, що можуть допомогти при аналізуванні причини збою і для його усунення, наприклад стан стека, значення змінних оточення тощо; такий набір параметрів дає змогу простежити хід виконання програми, який призвів до її збою, і оцінити невідповідності в даних, які могли призвести до збою;

– наявність і тип повідомлення про збій – повідомлення про збій може бути створено автоматичним модулем оповіщення про збої і містити наведену вище інформацію або створюватися користувачем уручну; якщо виходити з припущення, що автоматично створювані повідомлення посилаються розробнику завжди у разі збоїв, які не призводять до повного краху системи і викликані нештатними ситуаціями в роботі системи з огляду на операційне середовище або системи часу виконання, то ця інформація також допомагає оцінити тип збою.

1.1.2. Відмови програмного забезпечення

Відмови спричиняють тривале порушення функціонування системи або переводять її у стан, при якому подальша експлуатація системи є неприпустимою або недоцільною, а відновлення її працездатного стану – неможливим або недоцільним. Надалі будемо називати відмовою стан системи, при якому відновлення її працездатності є можливим.

Відмови класифікують таким чином.

За часовими характеристиками:

– раптова відмова – відмова, що спричинена різким стрибкоподібним зміненням одного з параметрів системи або даних, які оброблюються системою; такі ситуації можуть моделюватися при тестуванні навантаження шляхом різкого збільшення рівня навантаження на систему (наприклад, кількості користувачів, що підключилися одночасно) з подальшою швидкою стабілізацією навантаження;

– поступова відмова – така відмова, що спричинена поступовим зміненням одного з параметрів системи або даних, які оброблюються системою; така відмова може спричинитися, наприклад, при переповненні внутрішнього буфера, що містить інформацію про стан системи в кожний момент часу;

– мерехтлива відмова – це збій одного і того ж характеру, що виникає багато разів через невизначені проміжки часу, а потім самоусувається; оскільки в цьому випадку йдеться вже про дефект системи, що

систематично виявляється, то можна говорити саме про відмову, а не про серію збоїв;

– деградаційна відмова – така відмова, що обумовлена природним зношенням обладнання, на якому функціонує програмна система, навіть при дотриманні всіх норм і правил проектування, експлуатації й супроводження системи; ці відмови спричинені неконструктивними дефектами системи, проте для їх попередження у системі має бути передбачено модуль моніторингу, який повідомляє про перевищення ступеня зношення частин системи; якщо планується тривала експлуатація системи, то відсутність вимог і реалізації такого модуля мають бути виявлені під час верифікації.

За причинами:

– ресурсна відмова – це відмова, унаслідок якої система досягає ресурсної межі, тобто спричинена насамперед браком ресурсів (наприклад, дискового простору) для роботи системи; ситуації, що спричиняють такі відмови, мають моделюватися під час тестування навантаження;

– конструктивна відмова – спричинена порушенням процесу проектування і розроблення ПС або неправильним проектуванням; процес верифікації і тестування передусім спрямовано на виявлення дефектів, які спричинили конструктивні відмови;

– виробнича відмова – пов'язана з порушенням процесу виробництва або супроводу ПС; виробничі відмови можуть виникати у разі неправильного виконання профілактичних робіт під час супроводу ПС, наприклад, можуть бути загублені файли настройки системи, унаслідок чого ПС переходить в режим роботи за замовчуванням, який є несумісним з поточними настройками обладнання; попередження таких відмов полягає перш за все в коректному складанні експлуатаційної і супровідної документації, яку має бути верифіковано;

– експлуатаційна відмова – спричинена порушенням правил експлуатації через людський чинник; тому основні способи виявлення таких відмов – проведення тестування ПС на зручність використання, верифікація експлуатаційної документації, уведення в ПС захисних механізмів, що блокують потенційні помилки оператора.

За способом виявлення:

– явна відмова – така, що виявляється штатними засобами контролю стану системи відразу після виникнення;

– прихована відмова – така, що не виявляється штатними засобами контролю стану ПС або виявляється ними через деякий час після виникнення відмови; така відмова може стати причиною для однієї або декількох залежних відмов.

За зв'язком з іншими відмовами:

– незалежна відмова – це відмова, виникнення якої не обумовлено іншими відмовами;

– залежна відмова – така відмова, виникнення якої спричинено іншими відмовами.

Процес верифікації не гарантує відсутності в системі всіх дефектів, що можуть спричинити збої, відмови або аварії, йдеться тільки про певний рівень імовірності відсутності цих дефектів. Тому для створення більш надійних ПС окрім верифікації використовуються різні методи розроблення стійкого коду.

При цьому внаслідок верифікації із ПС усуваються дефекти, які можна виявити під час аналізування вимог та/або коду, а методи розроблення стійкого коду дають додаткову гарантію того, що система збереже працездатність у випадках, не передбачених вимогами. Проте не слід розцінювати ці методи як заміненя верифікації або грамотного проектування.

1.2. Стандартизація процесів забезпечення якості

Збільшення конкуренції серед організацій-розробників ПЗ, підвищення вимог кінцевого користувача до якості й надійності програмних засобів обумовило важливість питань стандартизації.

Для підтримки конкурентоспроможності розробники ПЗ мають застосовувати все більш ефективні, більш рентабельні методи, технології, інструментальні засоби, що сприяють постійному підвищенню якості й задоволенню вимог споживачів.

Вимоги споживачів часто включають до технічних умов (ТУ) або неформалізованих вимог, описаних на деякій вербальній мові. Проте ТУ і неформалізовані вимоги самі по собі не гарантують їх задоволення в кінцевому продукті, оскільки існує проблема розроблення прийнятних вимог до ПЗ, а також ряд інших проблем, що виникають під час розроблення кінцевого продукту. Це привело до розроблення стандартів, настанов та інших документів, що належать до систем якості й доповнюють релевантні вимоги до ПЗ. У міжнародних стандартах серії ISO 9000 вперше створено загальну основу для стандартів на системи якості, що застосовують у різних галузях, і встановлено, які саме елементи мають належати до системи якості, але не те, яким чином конкретна організація має реалізувати ці елементи. Уведення одноманітних систем якості не є метою цих стандартів. Потреби різних організацій є різними. На проект і реалізацію системи якості обов'язково впливають конкретні цілі, продукція і процеси, а також специфічні методи певної організації.

Міжнародні стандарти серії ISO 9000 базуються на розумінні того факту, що будь-яке виробництво працює з допомогою мережі процесів. Кожний процес має вхідні чинники, а виходом є результати процесу –

продукція. Кожна організація існує для того, щоб виконувати якусь діяльність, що додає вартості. Під час отримання кінцевого продукту мають бути виконані численні операції, що містять організацію, проектування, керування технологічними процесами, маркетинг, навчання, керування людськими ресурсами, стратегічне планування, поставку, технічне обслуговування тощо. Беручи до уваги складну структуру більшості організацій, важливо виділити основні процеси, а також спростити й ранжувати процеси залежно від цілей адміністративного керування якістю.

Будь-яка організація має визначити й встановити свою мережу процесів і інтерфейсів та керувати нею. Організація створює, удосконалює і забезпечує постійний рівень якості своєї продукції з допомогою мережі процесів. Це концептуальна основа стандартів серії ISO 9000.

У стандарті ISO 2382–1 наведено визначення ПЗ як інтелектуального продукту, що складається з програм, процедур, правил і будь-якої іншої документації, яка пов'язана з ним. Таким чином, документація є невід'ємною частиною ПЗ і їй, а також процесу її формування має надаватися пильна увага.

Стандарт ISO/IEC 15504, що є доповненням до інших міжнародних стандартів та інших моделей для оцінювання можливості й ефективності організацій і процесів, як і серія ISO 9000, забезпечує упевненість в керуванні якості постачальника, надаючи користувачам структуру для незалежного оцінювання можливості потенційних постачальників у задоволенні їхніх потреб. Оцінювання процесу забезпечує користувачів здатністю оцінити можливості процесу простим і порівнянним способом, а не використовувати характеристики якості, що базуються на ISO 9001. Крім того, структура, яку описано в ISO/IEC 15504, дає можливість регулювати область оцінювання для покриття специфічних найважливіших процесів. Стандартизація – найбільш перспективний напрям розвитку передових інформаційних технологій в проектуванні, виробництві й менеджменті, і будь-які зусилля в цьому напрямку мають всіляко схвалюватися.

Стандартизація процесу розроблення й експлуатації ПЗ сприяє контролю, оцінюванню й регламентації праці всіх учасників цього процесу, спонукає до дисципліни.

Перш ніж розглядати стандарти, що регламентують аспекти якості ПЗ, необхідно спочатку обговорити загальні питання, що стосуються якості будь-якого виду продукції. До загальних питань належать визначення і термінологія в цій області, основні концепції якості, значення документації в забезпеченні якості продукції, а також вибір і застосування міжнародних стандартів якості. Безумовно, основними стандартами є міжнародні стандарти серії ISO, які розроблено Міжнародною організацією зі стандартизації.

1.3. Загальні положення стандартів серії ISO

На сьогодні серія містить усі міжнародні стандарти з номерами від 9000 до 9004, від 10001 до 10020, а також ISO 8402:

ISO 9000-1-94. Стандарти з адміністративного керування якістю і забезпечення якості. Частина 1. Настанови щодо вибору із застосування.

ISO 9000-2-93. Стандарти з адміністративного керування якістю і забезпечення якості. Частина 2. Загальні настанови із застосування ISO 9001, ISO 9002, ISO 9003.

ISO 9000-3-91. Стандарти з адміністративного керування якістю і забезпечення якості. Частина 3. Настанови із застосування ISO 9001 під час розроблення, постачання та технічного обслуговування ПЗ.

ISO 9000-4-93. Стандарти з адміністративного керування якістю і забезпечення якості. Частина 4. Настанови з адміністративного керування програмою загальної надійності.

ISO 9001-94, ISO 9002-94. Системи якості. Модель забезпечення якості під час проектування, розроблення, виробництва, монтажу та обслуговування.

ISO 9003-94. Системи якості. Модель забезпечення якості під час контролю готової продукції та на заключних випробуваннях.

У стандартах ISO 9004 деталізовано адміністративне керування якістю та наведено елементи системи якості, а його частини містять такі вказівки: загальні, щодо послуг, оброблюваних матеріалів, підвищення якості.

У стандартах ISO 10011-1-90, ISO 10011-2-91, ISO 10011-3-91 деталізовано вимоги системи якості, а в його частинах – вимоги до загальних перевірок, критеріїв кваліфікації експертів-аудиторів систем якості, адміністративного керування програмами перевірок.

У стандарті ISO 10012-1-92 визначено вимоги щодо забезпечення якості вимірювального устаткування, а в його першій частині наведено системи метрологічного забезпечення вимірювального устаткування.

У стандарті ISO 10013 визначено настанови з якості, а також положення щодо розроблення.

У стандарті ISO 8402-94 розміщено словник термінів з керування й забезпечення якості.

У стандартах серії ISO 9000 встановлено, які саме елементи має бути включено до системи якості, тоді як організація сама має реалізувати їх з урахуванням конкретних цілей, продукції і процесів, а також специфічних методів, що використовуються цією організацією.

Крім того, керівні положення й вимоги стандартів серії ISO 9000 визначено в термінах цілей системи якості, яких треба досягти, і не запропоновано способів досягнення цих цілей, залишаючи право вибору

керівництву організації. У стандартах цієї серії відрізняються вимоги до систем якості від вимог замовника до продукції. Вимоги до систем якості є додатковими відносно технічних вимог до продукції. Наприклад, в ISO 12207 встановлено життєвий цикл (ЖЦ) розроблення ПЗ, а процеси й моделі якості, що відповідають процесу забезпечення якості (2.3 за ISO 12207) встановлено в стандартах серії ISO 9000.

У стандарті ISO 9000-1 визначено чотири загальні категорії продукції та охоплено всі види продукції, що поставляються будь-якою організацією:

– технічні засоби;

– ПЗ;

– оброблювані матеріали;

– послуги.

Вимоги до систем якості, встановлені в міжнародних стандартах серії ISO 9000, можна застосовувати до всіх чотирьох загальних категорій продукції, але термінологія і деякі положення й аспекти систем адміністративного керування якістю можуть бути різними. Це видно з назв стандартів ISO 9004-2 і ISO 9004-3. Слід зазначити, що будь-яка організація пропонує продукцію щонайменше двох категорій. Наприклад, організація, що розробляє ПЗ, додатково надає своїм замовникам послуги із супроводження розробленого ПЗ.

Метою керівних положень міжнародних стандартів серії ISO 9000 є задоволення вимог з урахуванням чотирьох аспектів, що є ключовими щодо якості продукції.

Перший аспект – якість завдяки визначенню і модернізації продукції з метою її відповідності вимогам і можливостям ринку.

Другий аспект – якість завдяки відповідності характеристик продукції вимогам і можливостям ринку. Іншими словами, якість завдяки конструкції – це ті властивості, що впливають на безперебійність роботи виробу в змінних умовах виробництва й застосування.

Третій аспект – якість завдяки підтримці постійної відповідності конструкції, реалізації характеристик, закладених до проекту.

Четвертий аспект – якість завдяки технічному обслуговуванню продукції під час її експлуатації для збереження бажаних характеристик.

Міжнародні стандарти серії ISO 9000 базуються на розумінні того, що будь-яка робота виконується з допомогою процесів. Сам процес є (або має бути) перетворенням, що додає вартість. У кожному процесі беруть участь люди та/або інші ресурси. Виходом може бути, наприклад, програма, банківська послуга, готовий (або проміжний) виріб будь-якої основної категорії продукції. Крім того, існує можливість виміряти на вході, на будь-яких стадіях процесу, а також на виході різні характеристики процесів.

Як показано на рис. 2, входи і виходи можуть бути декількох типів: такі, що пов'язані з продукцією, наприклад сировина, готовий виріб; такі,

що пов'язані з інформацією, наприклад вимоги до продукції, інформаційні характеристики.

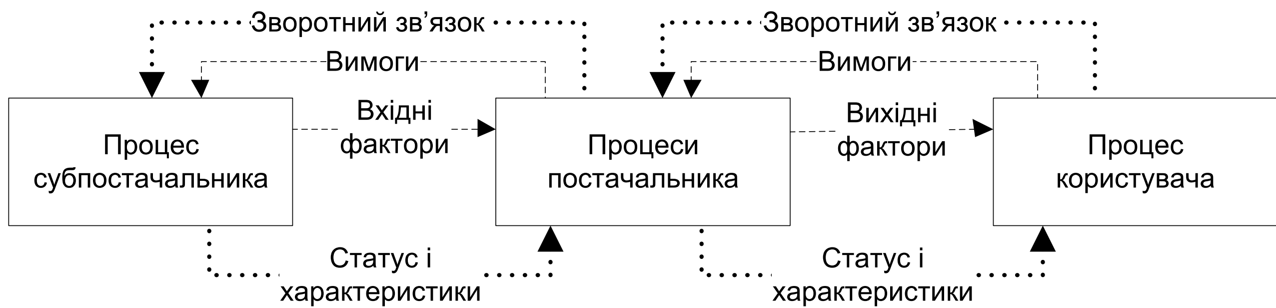

Рис. 2. Приклад взаємозв'язку процесів

Адміністративне керування якістю здійснюється шляхом керування процесами в організації:

– керування структурою та функціонуванням самого процесу;

– керування якістю продукції або інформації в межах структури.

Беручи до уваги складну структуру більшості організацій, важливо виділити основні процеси, а також спростити й ранжувати процеси залежно від цілей адміністративного керування якістю. Прикладом складної мережі процесів може бути організація, що розробляє ПЗ згідно з ISO/IEC 12207 і DO-178.

Будь-яка організація має визначити і встановити свою мережу процесів і їхніх інтерфейсів та керувати нею, створюючи, удосконалюючи й забезпечуючи постійний рівень якості своєї продукції. Це – концептуальна основа стандартів серії ISO 9000. Процеси і їхні інтерфейси мають бути об'єктами аналізу і постійного вдосконалення для забезпечення якості продукції, що виробляється.

Для оцінювання систем якості будь-якої організації у стандарті ISO 9000-1 рекомендовано вирішити три важливих питання щодо кожного оцінюваного процесу:

– чи визначено ці процеси і чи документовано їхні процедури;

– чи застосовуються ці процеси повною мірою і чи виконуються вони згідно з документацією;

– чи є ефективними ці процеси в досягненні очікуваних результатів.

Результатом оцінювання є сукупність відповідей, що пов'язані з підходом, застосуванням і результатом. Оцінювання системи якості може різнитися за охоплюваними областями і містити різні види діяльності.

Одним із найважливіших видів такої систематичної діяльності є оцінювання статусу і адекватності системи якості, що проводиться керівництвом організації згідно із стандартами ISO 9001, 9002, 9003. Висновки, зроблені під час оцінювання системи якості, мають підвищити її

ефективність і економічність. Джерелом інформації для таких висновків є також результати внутрішніх і зовнішніх перевірок системи якості.

Внутрішні перевірки якості, що проводить сама організація, забезпечують інформацію для ефективного аналізу з боку керівництва та дій, що коригують, застерігають і вдосконалюють.

Зовнішні перевірки, що проводяться замовниками продукції і незалежними органами, забезпечують відповідно довіру замовника до постачальника і отримання сертифіката, забезпечивши тим самим довіру до багатьох потенційних споживачів продукції організації.

Стандарт ISO 9000 призначено для застосування в таких випадках:

1. Як керівні настанови з адміністративного керування якістю. Система якості має підвищувати ефективність, щоб вимоги щодо якості продукції виконувалися економічно і оптимально.

2. В умовах укладення контракту між споживачем і розробником. Споживач визначає конкретну модель забезпечення якості.

3. Під час затвердження або реєстрації системи якості іншою стороною. Це та ситуація, коли система якості оцінюється замовником. Постачальник може отримати офіційне визнання відповідності його продукції стандарту.

4. Під час сертифікації або реєстрації системи якості третьою стороною. У цьому випадку систему якості оцінює орган із сертифікації, і організація погоджується підтримувати таку систему якості для всіх споживачів своєї продукції.

Постачальник може вибрати будь-який з двох способів використання стандартів серії ISO 9000: умотивований керівництвом або умотивований зацікавленою особою (найбільш поширеним уважається другий спосіб).

У випадку використання способу, умотивованого зацікавленою особою, постачальник вводить систему якості як відповідь на безпосередні вимоги споживачів. Система якості має відповідати вимогам стандартів ISO 9001, 9002, 9003. Керівництво організації відіграє провідну роль за цим способом, але рушійною силою є зовнішня зацікавлена особа (споживач).

У випадку використання способу, умотивованого керівництвом, саме керівництво організації визначає майбутні потреби і тенденції ринку. Інструкцією із первинного встановлення системи якості є стандарт ISO 9004. Постачальник може застосувати стандарти ISO 9001 – ISO 9003 як модель забезпечення якості для демонстрації адекватності системи якості з метою отримання сертифікату. Система якості, що реалізована за цим способом, є більш ємною і плідною, ніж така, яку реалізовано за першим способом.

У стандартах серії ISO 9000 приділено значну увагу підготовці й використанню документації як виду діяльності, що додає вартості. Відповідна документація має велике значення в таких випадках:

– досягненні необхідної якості продукції;

– оцінювання систем якості;
– підвищення якості;
– збереження досягнутого рівня якості.

При внутрішніх і зовнішніх перевірках наявність документації на процедури є свідченням про те, що процеси визначено, процедури затверджено й вони знаходяться під контролем. Тільки за таких обставин перевірки гарантують певне оцінювання адекватності застосування й виконання мережі процесів організації.

Документація має велике значення в підвищенні якості продукції. Якщо процедури задокументовано, їх застосовують і виконують, то існує можливість визначити, як вони виконуються.

Далі більш докладно розглянемо стандарт ISO 9001, за яким визначається модель забезпечення якості під час проектування, розроблення, виробництва, монтажу та обслуговування всіх видів продукції.

1.4. Застосування ISO 9001 при розробленні програмних систем

Останнім часом збільшилася кількість програмних продуктів, які застосовують у різних галузях господарства і відповідно посилилося значення керування якістю програмних продуктів. Однім із шляхів створення системи керування якістю є розроблення керівних положень щодо забезпечення якості ПЗ.

Вимоги до загальної системи якості визначено у стандарті ISO 9001. Процес розроблення й обслуговування ПЗ відрізняється від такого ж процесу для більшості інших типів промислової продукції. Тому необхідно розробляти додаткові керівні положення щодо системи якості там, де задіяно програмну продукцію, беручи до уваги сучасний рівень розвитку інформаційних технологій.

Природа розвитку ПЗ є такою, що деякі види діяльності пов'язані лише з окремими фазами процесу розроблення, тоді як інші можуть належати до всього процесу. Далі буде розглянуто ці відмінності, а також керівні положення, що сприяють застосуванню стандарту ISO 9001 в організаціях, які розробляють, поставляють і обслуговують програмну продукцію.

Керівні положення призначено для опису пропонованих засобів керування і методів розроблення ПЗ, яке має відповідає вимогам покупця. Це досягається насамперед запобіганням невідповідності продукції на всіх стадіях: від розроблення і до технічного обслуговування.

Постачальник визначає і документально оформляє свою політику, цілі й зобов'язання в області якості, а також забезпечує розуміння цієї політики, її здійснення і впровадження на всіх рівнях організації.

Відповідальність, повноваження і взаємодія всього персоналу, який керує, виконує і перевіряє роботу, що впливає на якість, мають бути чітко визначеними. Особливо це стосується тих, кому необхідна організаційна свобода і повноваження для такого:

– проведення заходів, направлених на попередження випадків невідповідності продукції;

– виявлення і реєстрація будь-яких проблем в області якості продукції;

– ініціація, вироблення рекомендацій або забезпечення виконання рішень в установленому порядку;

– перевірка виконання рішень;

– контроль за подальшим обробленням невідповідної продукції, її поставкою або монтажем доти, доки виявлені дефекти або незадовільні умови не буде усунено.

Постачальник визначає вимоги до внутрішньої перевірки, забезпечує необхідні засоби й призначає спеціально підготовлений персонал для її проведення. Перевірка має складатися з контролю, випробування й регулювання процесів проектування, виробництва, монтажу й обслуговування продукції. Аналізування проекту і перевірка системи якості й процесів мають виконуватися персоналом, що не залежить від безпосередніх розробників.

Постачальник призначає представника керівництва, який незалежно від інших обов'язків буде мати певні повноваження і відповідати за виконання і дотримання вимог стандарту ISO 9001.

Система якості, що задовольняє вимогам ISO 9001, має періодично аналізуватися керівництвом постачальника, щоб гарантувати постійну придатність і ефективність системи. Слід вести протоколи таких аналізувань.

Покупець має співробітничати з постачальником, щоб своєчасно забезпечити його всією необхідною інформацією і вирішувати проблеми, що виникають.

Покупець призначає представника, який буде відповідати за зв'язок з постачальником з питань контракту. Цей представник повинен мати повноваження для вирішення питань, пов'язаних з контрактом (але не обмежуватися ними):

– визначати вимоги покупця до постачальника;

– відповідати на питання постачальника;

– приймати пропозицію постачальника;

– укладати угоду з постачальником;

– гарантувати дотримання організацією, що представляє покупця, угод, укладених з постачальником;

– визначати критерії процедури приймання;

– приймати рішення щодо тих елементів ПЗ, які визнано непридатними для використання.

Регулярний спільний аналіз, що проводиться покупцем і постачальником, має плануватися, щоб охопити таке коло питань:

– відповідність ПЗ технічним завданням, які узгоджено з покупцем;

– результати контролю;

– результати приймальних випробувань.

Результати такого аналізу необхідно узгодити й зареєструвати.

Постачальник розробляє і документально оформлює систему якості, яка має бути єдиним процесом, що перетинає весь ЖЦ продукції. Це буде гарантією, що якість формується під час розроблення, а не досягається випадково наприкінці цього процесу. При цьому головні зусилля треба спрямовувати на попередження виникнення дефектів, а не на їх виправлення.

Постачальник повинен гарантувати ефективну реалізацію документально оформленої системи якості. Усі елементи, вимоги й положення системи якості має бути чітко подано документально.

Постачальник зобов'язаний підготувати і документально оформити план якості, щоб виконати заходи щодо забезпечення якості для кожної розробки ПЗ на базі системи якості та забезпечити її розуміння й дотримання зацікавленими організаціями.

Постачальник розробляє закінчену систему планових внутрішніх перевірок якості, щоб упевнитися у відповідності діяльності щодо забезпечення якості запланованими заходами, і визначає ефективність системи якості. Перевірки мають плануватися, виходячи із статусу й важливості різних видів діяльності відповідно до документально оформлених процедур.

Результати перевірок слід оформляти документально й доводити до відома керівного персоналу, що відповідає за ділянку роботи, яка перевірялася, і повинен вживати своєчасні заходи щодо усунення виявлених під час перевірки недоліків.

Постачальник розробляє, документально оформлює і виконує процедури, що забезпечують таке:

– виявлення причин невідповідності продукції і проведення коригувальних дій, що запобігають повторенню дефектів;

– аналізування всіх процесів, робочих операцій, порушень вимог контрактів, зареєстрованих даних щодо якості, звітів про використання й рекламацій користувачів з метою виявлення й усунення потенційних причин невідповідності продукції;

– проведення профілактичних заходів для вирішення проблем на рівні, що відповідає реальному ризику;

– здійснення контролю для упевнення в дійсній реалізації і ефективності коригувальних дій;

– упровадження змін в процедурах, які викликано коригувальними діями, та їх реєстрація.

1.5. Система якості

Програмний проект має здійснюватися відповідно до моделі ЖЦ. Дії, пов'язані із забезпеченням якості, мають плануватися і проводитися з урахуванням особливостей вибраної моделі ЖЦ.

Постачальник має встановлювати і виконувати процедури, що забезпечують проведення аналізу контракту і координацію цієї діяльності.

Кожний контракт постачальник має вивчити, щоб гарантувати таке:

– область дії контракту, а також вимоги визначено й оформлено документально;

– імовірні випадковості або ризики ідентифіковано;

– інформацію, що є власністю фірми, достатньо захищено;

– будь-які вимоги, які різняться від тих, що містяться в заявці на контракт, було виконано;

– постачальник має можливість виконати контрактні зобов'язання;

– відповідальність постачальника відносно підрядних робіт є певною;

– термінологію узгоджено розробником і покупцем;

– покупець має можливість виконати контрактні зобов'язання.

Для розроблення ПЗ постачальник повинен мати повний та однозначний набір функціональних вимог, що відповідають усім потребам покупця, наприклад: експлуатаційній якості, безпеці, надійності.

Ці вимоги мають бути сформульовані достатньо точно, щоб оцінювати продукцію під час приймання-здавання.

У плані розроблення має бути встановлено впорядкований процес або методологію перетворення вимог покупця на програмний продукт. У ньому може міститися схема розподілу робіт за фазами й ідентифікація:

– фаз розроблення, які необхідно виконати;

– необхідних витрат для кожної фази;

– необхідних результатів за кожною фазою;

– процедур перевірки, які необхідно провести на кожній фазі;

– аналізу потенційних проблем, пов'язаних з фазами розроблення і з виконанням установлених вимог.

У плані розроблення має бути визначено, як управляти проектом, і вміщено ідентифікацію:

– графіка розроблення продукції, виконання контракту й пов'язаних з ним поставок;

– контролю за ходом виконання робіт;

– організаційної відповідальності, ресурсів і розподілу робіт;

– організаційних і технічних інтерфейсів між різними групами.

План розроблення має містити методи, що забезпечують правильність виконання всіх робіт, а також правила, практичні методи й накопичений досвід з розроблення, засоби й технічні прийоми, які використовуються під час розроблення або керування конфігурацією.

Аналізування робіт слід планувати, проводити й документально оформляти, щоб забезпечити вирішення спірних питань, які стосуються розподілу ресурсів, та гарантувати ефективне виконання планів проекту.

Необхідні витрати щодо кожної фази треба визначити й документально оформити. Кожну вимогу слід визначити так, щоб її виконання можна було перевірити. Питання про неповні, двозначні або суперечливі вимоги мають вирішувати особи, які відповідають за розроблення цих вимог.

Результати щодо кожної фази розроблення треба визначити й документально оформити. Їх слід перевірити, щоб вони задовольняли такі умови:

– відповідати вимогам, установленим для кожної фази;

– містити критерії їх приймання або посилання на них для переходу до подальшої фази;

– відповідати прийнятій практиці й накопиченому досвіду з розроблення незалежно від того, чи наведено їх у вхідній інформації;

– ідентифікувати ті характеристики продукції, які є найбільш важливими для її безпеки й ефективного функціонування;

– відповідати чинним нормативним вимогам.

Постачальник має скласти план перевірки всіх результатів розроблення наприкінці кожної фази. Під час перевірки має бути встановлено, що результати розроблення відповідають вимогам, які визначено на початку фази. Ці перевірки необхідно проводити, ґрунтуючись на виконанні таких заходів щодо контролю розроблення:

– аналізування через установлені інтервали під час кожної фази ЖЦ;

– порівняння нового проекту з апробованим аналогічним проектом, якщо такий є;

– проведення випробувань і демонстраційних показів.

Результати перевірок і подальших дій, необхідних для гарантії того, що встановлені вимоги виконано, слід занести до протоколу й перевірити після того, як завершаться відповідні дії.

Постачальник має підготувати план якості як частину робіт з планування розроблення. План якості має коригуватися під час виконання робіт, а пункти, що стосуються кожної фази, – повністю визначатися до початку цієї фази.

План якості треба офіційно розглянути й узгодити зі всіма організаціями, які зацікавлені в його реалізації.

Документ, що описує план якості, може бути як самостійним документом, так і частиною іншого документу, або складатися з кількох документів, включаючи план розроблення.

У плані якості має бути визначено або дано посилання на таке:

– цільові кількісні показники якості;

– задані критерії з витрат і результати кожної фази ЖЦ;

– ідентифікація видів діяльності, пов'язаної з випробуваннями, перевірками і оцінюванням, що мають проводитися;

– докладне планування випробувань, перевірок та оцінювань, у тому числі розроблення графіків, розподіл ресурсів і призначення уповноважених;

– конкретний розподіл відповідальності за заходи щодо забезпечення якості, такі, як аналізування й випробування; керування конфігурацією і контроль за змінами; контроль дефектів і виконання коригуючих дій.

Проектування і реалізація – це ті види діяльності, які трансформують вимоги користувача у програмний продукт. Через складність цієї продукції вся діяльність має здійснюватися в строго встановленому порядку, щоб розробляти продукцію відповідно до завдання, а при забезпеченні якості не слід надмірно покладатися на дії, пов'язані з випробуванням і перевіркою.

На додаток до вимог, які є загальними для всіх фаз ЖЦ, необхідно взяти до уваги такі аспекти, що є властивими діяльності з проектування:

– ідентифікація конструктивних розробок – на додаток до вимог, які стосуються вихідних даних і очікуваних результатів, слід розглянути такі аспекти, як правила проектування й визначення внутрішнього інтерфейсу;

– методологія проектування – має бути використано методологію системного проектування;

– використання минулого досвіду в проектуванні – використовуючи досвід, набутий під час минулого проектування, постачальник має уникати повторень одних і тих самих або аналогічних проблем.

Продукцію має бути спроектовано так, щоб без перешкод проводити випробування, технічне обслуговування й супровід.

На додаток до вимог, що є загальними для всіх видів діяльності, пов'язаних з розробленням, необхідно розглянути такі аспекти для кожного виду діяльності з реалізації проекту:

– встановлення правил програмування, мови програмування, злагоджених правил найменування, кодування й відповідного роз'яснення;

– використання постачальником відповідних методів і засобів реалізації для виконання вимог покупця.

Постачальник має проаналізувати процеси ЖЦ для гарантування виконання вимог і коректного застосування наведених вище методів. Процес проектування або реалізації не можна продовжувати, доки наслідки всіх виявлених недоліків не буде видалено або не стане відомим

ступінь ризику у разі продовження робіт іншими методами. Необхідно складати протоколи таких аналізувань.

Проведення випробування є необхідним на різних рівнях, починаючи від окремого елемента ПЗ і закінчуючи готовою продукцією. Існує декілька різних підходів до випробувань. У деяких випадках оцінювання, випробування на місці й приймальні випробування можуть бути одним і тим самим видом діяльності. Документ, що описує план випробувань, може бути як самостійним документом, так і частиною іншого документа, або складеним з декількох документів.

1.6. Функціональний склад колективу розробників

Майже в будь-якому програмному проекті можна виділити перелічені нижче функції. Деяких із них може не бути зовсім, при цьому окремі люди можуть виконувати відразу декілька функцій, проте загальний склад змінюється мало.

Замовник (заявник) – представник організації, що замовила розроблену систему; він обмежений у своїх діях і спілкується тільки з менеджерами проекту й фахівцем із сертифікації. Замовник має право змінювати вимоги до продукту (тільки у взаємодії з менеджерами), читати проектну й сертифікаційну документацію, що стосується нетехнічних особливостей розробленої системи.

Менеджер проекту забезпечує комунікаційний канал між замовником і проектною групою, керує очікуваннями замовника, розробляє і підтримує бізнес-контекст проекту. Його робота не пов'язана прямо з продажем продукту і сфокусована на продукті, його задача – визначити й забезпечити вимоги замовника. Він має право змінювати вимоги до продукту й фінальну документацію на продукт.

Менеджер програми керує комунікаціями і взаєминами в проектній групі, є деякою мірою координатором, розробляє функціональні специфікації й керує ними, веде графік проекту й складає звіт за станом проекту, ініціює ухвалення критичних для ходу проекту рішень. Має право змінювати функціональні специфікації верхнього рівня, план-графік проекту, розподіл ресурсів по задачах. Часто функції менеджера проекту й менеджера програми виконує одна людина.

Розробник приймає технічні рішення, які можуть бути реалізовані й використані, створює продукт, що задовольняє специфікаціям і очікуванням замовника, консультує інших під час розроблення. Бере участь в оглядах, реалізує вимоги до продукту, розробляє функціональні специфікації, відстежує і виправляє помилки за прийнятний за проектним графіком час. Має доступ до всієї проектної документації, у тому числі до документації з тестування, має право на змінення програмного коду системи в межах своїх службових обов'язків.

Фахівець із тестування визначає стратегію тестування, вимоги до тестів і тестові плани для кожної фази проекту, виконує тестування системи, збирає й аналізує звіти про проходження тестування. Вимоги до тестів мають покривати вимоги до системи, функціональні специфікації, вимоги до надійності та вантажоспроможності. У реальності функції фахівця з тестування часто виконують дві людини – тестувальник і розробник тестів.

Фахівець з контролю якості здійснює взаємодію з розробником, менеджером програми і фахівцями з безпеки й сертифікації з метою відстеження якості продукту, його відповідності стандартам і специфікаціям, які передбачено в проектній документації. Він не відповідає за деталі і технологічний рівень процесів ЖЦ. Під контролем якості мається на увазі насамперед контроль самих процесів розроблення й перевірка їх відповідності певним критеріям стандартів якості.

Фахівець із сертифікації. При розробленні систем, до надійності яких ставляться підвищені вимоги, перед введенням їх до експлуатації потрібно підтвердження відповідності їхніх експлуатаційних характеристик заданим критеріям з боку уповноваженого органу (зазвичай державного). Така відповідність визначається під час сертифікації системи. Фахівець із сертифікації може бути або представником сертифікуючих органів, якого включено до складу колективу розробників, або представником інтересів розробників в сертифікуючому органі. Він приводить документацію у відповідність до вимог сертифікуючого органу або бере участь в процесі створення документації з урахуванням цих вимог, відповідає за взаємодію між колективом розробників і сертифікуючим органом. Важливою особливістю фахівця із сертифікації є його незалежність від проектної групи на всіх етапах створення продукту, він взаємодіє лише з проектними і програмними менеджерами.

Фахівець із впровадження й супроводу бере участь в аналізуванні особливостей апаратури замовника, на якій передбачається впроваджувати розроблювану систему, виконує всі роботи з установлення й настроювання системи, проводить навчання користувачів.

Фахівець з безпеки відповідає за безпеку створюваного продукту. Його робота починається з участі в написанні вимог до продукту й закінчується фінальною стадією сертифікації продукту.

Інструктор відповідає за зменшення витрат на подальший супровід продукту, забезпечення максимальної ефективності роботи користувача. Важливо, що йдеться про продуктивність користувача, а не системи. Для забезпечення оптимального функціонування інструктор збирає статистику щодо дій користувачів і підвищує ефективність, у тому числі з використанням різних аудіовізуальних засобів, а також бере участь у всіх обговореннях інтерфейсу, призначеного для користувача, і архітектури продукту.

Технічний письменник несе обов'язки з підготовки документації до розробленого продукту, фінального опису функціональних можливостей і бере участь в написанні супровідних документів (системи допомоги, інструкції користувача).

1.7. Моделі життєвого циклу програмного забезпечення

Існує тенденція збільшення обсягів робіт, пов'язаних з розробленням ПЗ, порівняно з роботами, пов'язаними з розробленням апаратних засобів. Таким чином, має сенс більш детально розглянути технологічний процес розроблення програмних засобів, а саме ЖЦ ПЗ, що реалізує спеціальне математичне, інформаційне, програмне, лінгвістичне забезпечення ЕОМ і є основою діяльності зі створення й використання програмного забезпечення. У загальному випадку розрізняють поняття ЖЦ ПЗ і технологічного процесу його розроблення. Більш чітко відмінності між цими поняттями видно у програмних засобах. ЖЦ є моделлю створення й використання ПЗ, що відображає його різні стани, починаючи з моменту виникнення необхідності в певному ПЗ і закінчуючи моментом його повного виходу зі вживання.

Існує декілька моделей ЖЦ (рис. 3). Традиційно виділяють такі його етапи:

– стратегічне планування;
– аналіз вимог;
– проектування (попереднє й детальне);
– кодування (програмування);
– тестування й відлагодження;
– експлуатація і супровід.

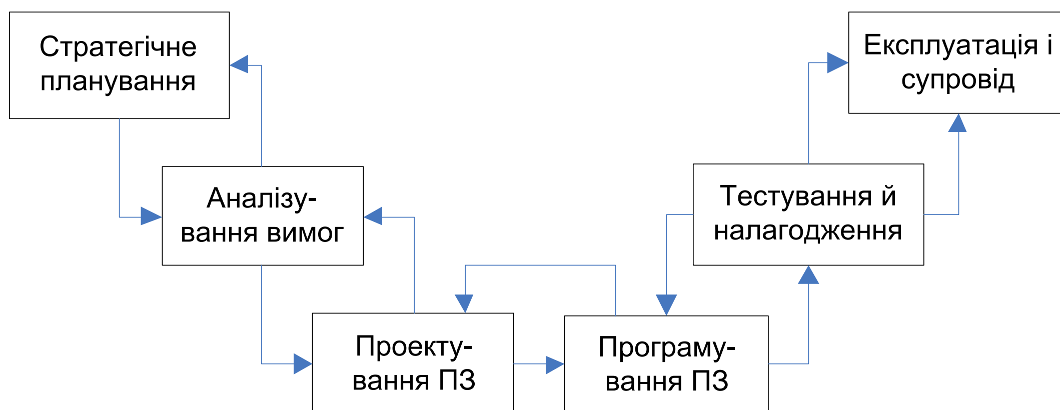

Рис. 3. Модель ЖЦ ПЗ

Кожному етапу відповідають певний результат і набір документації, що є початковими даними для наступного етапу. Після закінчення кожного

з етапів проводиться верифікація документів і рішень з метою перевірки їх відповідності первинним вимогам замовника.

Унаслідок еволюційного розвитку теорії проектування ПЗ і його ускладнення, історично склалися кілька моделей ЖЦ:

Каскадна модель ЖЦ, що давала змогу перейти на наступні етапи ЖЦ тільки після повного закінчення робіт на попередніх етапах. З розвитком обчислювальної техніки складність та обсяги ПЗ істотно збільшилися. У зв'язку з цим виникли проблеми з його розробленням. Продумати всі кроки розроблення нового ПЗ, накреслити етапи проектування й передбачити всі варіанти функціонування при відлагодженні ПЗ стало не під силу одному розробнику. Каскадна модель ЖЦ стала істотно стримувати темпи створення складних програмних систем. Процес відлагодження затягувався, і при цьому не було гарантій безпомилкової роботи програм.

Поетапна модель з проміжним контролем замінила каскадну модель, що жорстко регламентує послідовність етапів і критерії переходів між ними. Це – ітераційна модель розроблення ПЗ із зворотними зв'язками між етапами. Перевірки й коригування проводяться на кожному етапі, що дає змогу істотно зменшити трудомісткість відлагодження порівняно з каскадною моделлю. Ітераційність моделі виявляється в обробленні помилок, які виявлено проміжним контролем. Якщо на якомусь із етапів під час проміжної перевірки виявлено помилку, допущену на більш ранній стадії розроблення, то цей етап треба повторити. При цьому аналізуються причини помилки й коригуться, якщо є потреба, вхідні дані етапу або перелік робіт, що проводяться. Аналогічна ситуація виникає, якщо під час розроблення замовник ставить нові вимоги або змінюються які-небудь умови функціонування.

Спіральна модель підтримує ітерації поетапної моделі, але особлива увага приділяється початковим етапам проектування: аналізуванню вимог, проектуванню специфікацій, попередньому й детальному проектуванню. Кожний виток спіралі відповідає певному етапу моделі створення фрагмента або версії ПЗ, при цьому уточнюють цілі й вимоги до ПЗ, оцінюють якість розробленого фрагмента або версії ПЗ та планують роботи наступного витка. Таким чином, заглиблюються й конкретизуються всі деталі ПЗ, унаслідок чого отримують варіант, який задовольняє всім вимогам замовника.

Кількість, склад і послідовність етапів ЖЦ для кожного конкретного продукту визначається на ранніх стадіях планування. При цьому враховуються особливості експлуатації, наявність деяких обмежень, кількість і кваліфікація персоналу розробників і експлуатаційників, а також безліч інших чинників.

Як було зазначено вище, життєві цикли систем, процесів їх розроблення, експлуатації й супроводу регламентовано в стандартах. При

цьому стандарти, що розроблюються міжнародними організаціями в тій або іншій області діяльності, мають рекомендаційний характер і не мають сили закону. Керівні принципи, які визначено в стандартах, мають офіційну силу тоді, коли їх прийнято урядом країни. Сьогодні загальновизнаними міжнародними лідерами зі стандартизації розроблення ПЗ є такі організації: Американський національний інститут зі стандартизації – ANSI; Міжнародна організація зі стандартизації – ISO; Міністерство оборони США – DOD; Інститут інженерів з електроніки й радіотехніки – IEEE; Європейське космічна агенція – ESA.

Далі розглянемо, які конкретні ЖЦ розроблення ПЗ створено в нашій країні й за кордоном, а також якими стандартами їх регламентовано.

Життєвий цикл ПЗ – сукупність ітераційних процедур, пов'язаних з послідовним зміненням стану ПЗ від формування початкових вимог до нього й до закінчення його експлуатації кінцевим користувачем. У контексті цього курсу майже не будуть розглядуватися такі етапи ЖЦ, як системна інтеграція й супровід. Для цілей курсу достатньо обмежитися спрощеним уявленням, що після реалізації коду і доказу його відповідності вимогам розроблення ПЗ завершується. Будь-який етап ЖЦ має певні початок і закінчення. Склад і послідовність етапів ЖЦ визначають розробники та/або замовники. Сьогодні існує декілька основних моделей ЖЦ, які можна пристосувати під реальну розробку.

Каскадна модель життєвого циклу

Каскадний ЖЦ (іноді називають водоспадним) базується на поступовому збільшенні ступеня деталізації опису всієї розроблюваної ПС. Кожне збільшення визначає перехід до наступного стану розроблення.

На першому етапі складають концептуальну модель системи, описують загальні принципи її побудови, правила взаємодії з навколишнім світом – визначають системні вимоги. На другому етапі за системними вимогами складають вимоги до ПЗ – тут основну увагу приділяють функціональності програмної компоненти, програмним інтерфейсам. Зазвичай всі програмні комплекси виконуються на якій-небудь апаратній платформі. Якщо під час проекту потрібно також розробити апаратну компоненту, паралельно з вимогами до ПЗ готуються вимоги до апаратного забезпечення. На третьому етапі на основі вимог до ПЗ складають детальну специфікацію архітектури системи, її окремих модулів і міжмодульних інтерфейсів, заголовків окремих функцій тощо. На четвертому етапі пишуть програмний код, що відповідає детальній специфікації. На п'ятому етапі виконують тестування – перевірку відповідності програмного коду вимогам, поставленим на попередніх етапах.

Особливість каскадного ЖЦ полягає в тому, що перехід до наступного етапу відбувається тільки тоді, коли повністю завершено всі роботи попереднього етапу, тобто спочатку готують усі вимоги до системи, потім за ними готують усі вимоги до ПЗ, розробляють архітектуру системи і так далі до тестування.

Звичайно, що у разі великих систем такий підхід себе не виправдовує. Робота на кожному етапі займає значний час, а внесення змін в первинні документи або є неможливим, або спричиняє лавиноподібні зміни на всіх інших етапах.

Зазвичай використовується модифікація каскадної моделі, що допускає повернення на будь-який з раніше виконаних етапів. При цьому фактично виникає додаткова процедура прийняття рішення.

Дійсно, якщо тести не відповідають вимогам, то причина може бути в неправильному тесті, помилці кодування (реалізації), неправильній архітектурі системи, некоректності вимог до ПЗ тощо. Усі ці випадки потребують аналізування для прийняття рішення про те, на який етап ЖЦ треба повернутися для усунення знайденої невідповідності.

V-подібний життєвий цикл

Модель V-подібного ЖЦ може розглядатися як подальший розвиток класичної каскадної моделі, що має процеси двох видів – основні процеси розроблення, які є аналогічними процесам каскадної моделі, і процеси верифікації, що є ланцюгом зворотного зв'язку відносно основних процесів (рис. 4).

Наприкінці кожного етапу ЖЦ розроблення, а часто і під час виконання етапу здійснюється перевірка взаємної коректності вимог різних рівнів. Ця модель дає змогу більш оперативно перевіряти коректність розроблення, проте, як і в каскадній моделі, передбачається, що на кожному етапі розробляються документи, у яких описано функціонування системи.

Спіральний життєвий цикл

В обидвах розглянутих типах ЖЦ припускається, що наперед відомими є всі вимоги користувачів або, принаймні, передбачувані користувачі ПЗ є кваліфікованими настільки, що можуть ставити свої вимоги до майбутньої системи, не маючи її перед очима.

Звичайно, така картина є досить утопічною, тому поступово виникла модель ЖЦ, яка виправляє основний недолік V-подібного ЖЦ – припущення про те, що на кожному етапі розробляється повний опис системи. Ця модель отримала назву спіральної моделі ЖЦ (рис. 5).

План приймально-
здавальних випробувань

Визначення й
уточнення вимог
користувача

Змінення вимог
користувача

Приймально-
здавальні
випробування

Вимоги
користувача

Уточнення
вимог
користувача

План системного
тестування

Протестований
програмний продукт

Визначення й
уточнення вимог
до системи

Уточнення вимог
до системи

Системне
тестування

Вимоги до
системи

Уточнення
вимог
до системи

Плани інтеграції і
тестування

Складений
програмний продукт

Архітектурне
проектування

Уточнення
архітектурного
проекту

Інтеграційне
тестування

Вимоги до
архітектури

Уточнення
архітектурного
проекту

План модульного
тестування

Протестовані
програмні модулі

Детальне
проектування

Уточнення
детального
проекту

Модульне
тестування

Уточнення
детального
проекту

Програмні
модулі

Детальний
проект

Кодування

Рис. 4. V-подібний життєвий цикл програмного забезпечення

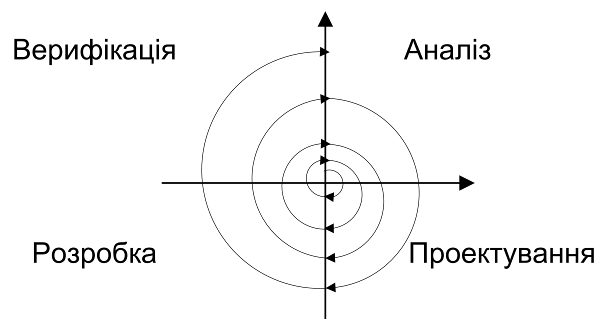

Рис. 5. Спіральний життєвий цикл

У спіральній моделі розроблення системи відбувається етапами, що повторюються, – витками спіралі. Кожний виток спіралі – один каскадний або V-подібний ЖЦ. Наприкінці кожного витка виходить закінчена версія системи, що реалізує деякий набір функцій. Потім її пред'являють користувачу, на наступний виток переносять всю документацію, яку розроблено на попередньому витку, і процес повторюється.

Таким чином, ПЗ розробляється поступово, проходячи постійні узгодження із замовником. На кожному витку спіралі функціональність системи розширюється, поступово наближуючись до повної.

Екстремальне програмування

Реалії останніх років показали, що для систем, вимоги до яких змінюються досить часто, тривалість оберту спірального ЖЦ необхідно ще більше зменшити. У зв'язку з цим зараз стали досить популярними швидкі ЖЦ розроблення, наприклад ЖЦ в методології eXtreme Programming (XP).

Основна ідея ЖЦ екстремального підходу – максимальне укорочення тривалості одного етапу ЖЦ і взаємодія із замовником. На кожному етапі відбувається реалізація і тестування однієї функції системи, після чого ПЗ відразу передається замовнику для перевірки або до експлуатації.

Основна проблема цього підходу – інтерфейси між окремими модулями системи. Якщо в усіх попередніх типах ЖЦ інтерфейси досить чітко визначаються на самому початку розроблення (оскільки наперед відомими є всі модулі), то при екстремальному підході інтерфейси проектуються разом з модулями, що розробляються.

Порівняння різних моделей життєвого циклу

Особливості розглянутих вище ЖЦ зведено в табл. 1. Різні типи ЖЦ застосовуються залежно від частоти внесення змінень в систему, термінів розроблення та її складності. ЖЦ з більш короткими фазами є більш придатними для розроблення систем, вимоги до яких ще не є усталеними і виробляються у взаємодії із замовником системи під час розроблення.

Кожна з моделей ЖЦ має чотири групи процесів:

– основні процеси розроблення ПЗ;

– процеси верифікації;

– процеси керування розробленням;

– допоміжні процеси.

Таблиця 1

Порівняння різних типів життєвого циклу

Тип ЖЦ	Довжина циклу	Частота верифікації і внесення змін	Особливості інтеграції ПЗ
Каскадний	Довга. На всіх етапах розроблення	Рідко. Наприкінці розроблення всієї системи	Інтерфейси модулів визначені й незмінні
V-подібний		Середньо. Наприкінці кожного етапу ЖЦ ПЗ	Інтерфейси модулів змінюються нечасто
Спіральний	Середня. Розроблення однієї версії системи	Середньо. Наприкінці кожного етапу розроблення версії системи	Інтерфейси модулів змінюються періодично, але рідко у межах версії
XP	Короткий. Розроблення однієї історії ПЗ	Дуже часто. Наприкінці розроблення кожної історії	Інтерфейси часто змінюються

Процес верифікації – процес, під час якого перевіряється коректність системи, що розробляється, і визначається ступінь її відповідності вимогам замовника.

Процес керування розробленням залежить від типу ЖЦ основного процесу розроблення. Чим коротше один етап ЖЦ, тим активнішим буде керування і тим більше задач у менеджера проекту. При класичних схемах кожний менеджер відповідає за розроблення певної частини системи. В XP-підході немає жорсткого розподілу системи на частини, і менеджер має охоплювати всі історії. При цьому процес керування буде активним протягом всього ЖЦ основного процесу розроблення.

Допоміжні процеси забезпечують своєчасне створіння всього, що може знадобитися розробнику або кінцевому користувачу. Ці процеси містять підготовку призначеної для користувача документації, приймально-здавальних тестів, керування конфігураціями й змінами, взаємодію із замовником тощо. Допоміжні процеси можуть існувати протягом всього ЖЦ або бути своєрідними стикувальними ланками між процесами розроблення й експлуатації.

Надалі особливу увагу буде приділено найбільш важливим підтримним процессам – керування конфігураціями й забезпечення гарантії якості.

Основна мета процесу керування конфігураціями – забезпечення цілісності всіх даних, які отримують під час колективного розроблення. Під цілісністю розуміється перш за все ідентифіковність, доступність цих даних у будь-який момент часу і недопущення несанкціонованих змін. Важливим аспектом при цьому є процес керування зміненням даних, тобто планування і затвердження будь-яких змінень в проектній документації або програмному коді, а також визначення області впливу цих змінень.

Для забезпечення гарантії якості проводять перевірки, після яких можна гарантувати, що процес розроблення задовольняє певним вимогам (стандартам), необхідним для випуску якісної продукції. Фактично перевіряється, що всі передбачені стандартами процедури виконуються і при виконанні дотримуються декларованих для них правил.

Потрібно особливо зазначити, що процес гарантії якості не є запорукою розроблення якісного ПЗ, а тільки того, що процеси розроблення побудовано й виконано так, щоб не знижувати якість продукції.

Вимоги, що ставляться до організації роботи, яка є необхідною для випуску якісної продукції, оформлено у вигляді стандартів якості ISO 9000 та стандарту, що містить вимоги до ЖЦ розроблення ПЗ, – ISO(ДСТУ) 12207, у якому пропонуються загальні рекомендації з організації життєвого циклу.

1.8. Життєвий цикл розроблення програмного забезпечення з підвищеними вимогами до безпеки

Програмне забезпечення, яке використовується в бортових системах літальних апаратів, повинно виконувати свої функції та відповідати вимогам авіаційної і космічної техніки. Ці вимоги є більш жорсткими за якістю і надійністю, оскільки гарантують безпеку пасажирів і впливають на характеристики літального апарата.

Важливе значення в ЖЦ має етап оцінювання безпеки, коли визначаються вимоги щодо безпеки до апаратних засобів ЕОМ і ПЗ, для усунення й обмеження впливу пов'язаних з ними помилок. Вимоги, пов'язані з безпекою, є частиною вимог до системи і враховуються на всіх етапах ЖЦ ПЗ. До цих вимог належать:

– опис системи й апаратних засобів ЕОМ;

– вимоги сертифікації;

– вимоги системи до ПЗ, у тому числі функціональні й експлуатаційні вимоги, а також такі, що пов'язані з безпекою;

– рівні ПЗ, стани відмов, їхні категорії;

– стратегії безпеки й проект розроблення.

Під час оцінювання безпеки системи визначається вплив процесів проектування і впровадження ПЗ на безпеку системи шляхом використання інформації з етапів ЖЦ, що містить межу поширення несправності, вимоги ПЗ і його архітектуру, а також джерела помилок, які можна знайти й усунути завдяки ПЗ або з допомогою інструментів і методів, що використовуються при розробленні ПЗ.

Існують такі етапи ЖЦ ПЗ:

– планування розроблення, коли визначаються й координуються дії щодо розроблення;

– розроблення, коли створюється ПЗ, що містить роботи з визначення вимог, проектування, кодування, отримання коду;

– інтегрування, коли забезпечується коригування, перевірка й визначення функціональної повноти ПЗ та яке містить верифікацію, контроль за конфігурацією ПЗ, оцінювання якості й перевірку взаємодії етапів.

У проекті ПЗ визначається один або більше ЖЦ шляхом вибору дій на кожному етапі, призначення послідовності цих дій і тих, хто відповідає за них.

Для розроблення конкретного ПЗ послідовність етапів визначається на основі аналізу таких властивостей, як функціональні можливості, складність, розмір, стійкість, використання раніше отриманих результатів і апаратна підтримка. Звичайна послідовність виконання етапів ЖЦ ПЗ містить визначення вимог, проектування, кодування й отримання коду.

Етапи ЖЦ можуть мати ітераційний характер, тобто повторюватися кілька разів. Кількість етапів та ітерацій варіюється через доробки функцій ПЗ, виникнення нових вимог або нового апаратного забезпечення, результатів виконання попередніх етапів, які зазнали змін, а також інших особливостей. Майже всі етапи ЖЦ ПЗ об'єднуються з етапами інтегрування і верифікації.

Для переходу з одного етапу на інший необхідно передбачити певні критерії, які залежать від запланованої послідовності кроків розроблення й інтеграції, а також від рівня ПЗ (рівень ПЗ визначається на основі аналізу важливості в наборі потенційних відмов системи).

Якщо критерій переходу до подальшого етапу задоволено, то немає необхідності, щоб кожний вхід цього етапу було сформовано до початку його виконання. Головний принцип такий: якщо під час виконання етапу проводяться дії над окремими входами, то їх послідовність треба перевірити з метою визначення того, що виходи попередніх кроків етапів розроблення й інтеграції ще мають силу.

Розглянемо детальніше цілі й дії кожного етапу ЖЦ.

На етапі планування розроблення ПЗ створюються плани й вибираються стандарти, за якими буде здійснюватися розроблення й інтеграція. Визначаються засоби для створення ПЗ, яке буде задовольняти вимоги, що ставляться до нього, і забезпечувати достатній рівень надійності. На цьому етапі проводиться:

– визначення дій на етапах розроблення й інтеграції, у яких будуть визначатися вимоги до системи і ПЗ;

– визначення ЖЦ ПЗ, у тому числі взаємодію між етапами, механізму взаємного впливу етапів, критеріїв оцінювання ПЗ при переході від одного етапу до іншого;

– визначення середовища ЖЦ, тобто методів і інструментальних засобів, що використовуються на кожному етапі;

– формування додаткових зауважень до ПЗ;

– розгляд стандартів розроблення ПЗ, співвідношення їх з системною метою безпеки;

– розроблення плану створення ПЗ;

– дороблення й перевірка плану створення ПЗ.

Ефективність планування – визначальний чинник розроблення ПЗ. Основні керівні принципи цього етапу такі.

1. План розроблення ПЗ має бути складено в такий момент ЖЦ, щоб забезпечити своєчасне керування конкретними діями на етапах розроблення й інтеграції.

2. Стандарти розроблення ПЗ, що використовуються в проекті, мають бути точно визначеними й чіткими.

3. Вибрані методи й інструментальні засоби мають бути такими, щоб забезпечувати запобігання помилкам на етапі розроблення або зводити їх до мінімуму.

4. Плани розроблення ПЗ мають забезпечити координацію між рештою етапів з метою узгодження їх стратегій під час розроблення ПЗ.

5. Етап планування розроблення ПЗ має містити засоби перевірки плану на етапі реалізації проекту.

6. На завершальній стадії етапу планування план і стандарти розроблення ПЗ слід проаналізувати стосовно повноти й несуперечності.

Інші етапи ЖЦ можна розпочати до закінчення етапу планування за умови, що є плани й процедури для дій на цих етапах.

План розроблення містить такі компоненти:

– визначення середовища і план програмних аспектів сертифікації, що є основним засобом упровадження методів розроблення, що пропонуються службами сертифікації;

– власне план розроблення, за яким визначається ЖЦ і середовище розроблення;

– план верифікації ПЗ, де визначаються засоби, з допомогою яких задовольняються цілі етапу верифікації;

– план керування конфігурацією ПЗ, де визначаються засоби, з допомогою яких задовольняються цілі етапу оцінювання якості.

Обгрунтовуються методи, інструментальні засоби, процедури, мови програмування, апаратне забезпечення, що буде використовуватися під час розроблення, верифікації, контролю й випуску цих етапів ЖЦ і ПЗ. Прикладом того, що вибір середовища корисно впливає на ПЗ, може бути суворе дотримання стандартів, визначення помилок, запобігання помилкам виконання, методів пом'якшення наслідків збоїв у ПЗ. Середовище ЖЦ є потенційним джерелом помилок, що спричиняють

виникнення збоїв ПЗ. На склад середовища впливають вимоги щодо безпеки, які визначаються на етапі оцінювання безпеки системи.

Мета методів запобігання помилкам – недопущення помилок під час етапу розроблення. Основний принцип при виборі вимог до розроблення ПЗ, методів його проектування й програмування полягає в обмеженні випадків виникнення помилок і застосування таких методів верифікації, при яких буде гарантуватися встановлення факту помилки.

Метою застосування методів пом'якшення наслідків збоїв є занесення характеристик безпеки до проекту ПЗ для гарантії того, що ПЗ буде коректно реагувати на помилки вхідних даних і запобігати помилкам вихідних даних і помилкам керування. Необхідність у застосуванні методів запобігання помилкам і пом'якшення наслідків збоїв визначається системними вимогами й етапом оцінювання безпеки системи.

Середовище розроблення ПЗ – визначальний чинник при розробленні високоякісного і надійного ПЗ, який може несприятливо впливати на процес розроблення ПЗ. Наприклад, компілятор може заносити помилки під час компіляції, завантажувач – давати збої при виявленні помилок розподілу пам'яті. Керівні принципи для вибору методів та інструментальних засобів середовища розроблення ПЗ є такими:

1. Під час етапу планування середовище розроблення треба вибирати, виходячи з мінімуму потенційного ризику для кінцевого ПЗ.

2. Спеціалізовані інструментальні засоби або інструменти і компоненти середовища мають використовуватися за умови гарантії того, що помилки, внесені одним компонентом, виявлялися б іншим. У цьому випадку середовище буде прийнятним.

3. Дії на етапі верифікації і стандарти розроблення мають визначатися так, щоб звести до мінімуму потенційні помилки, які може внести середовище розроблення ПЗ.

4. Якщо визначено необов'язкові характеристики інструментальних засобів середовища для використання в проекті, то вплив цих характеристик слід перевірити і специфікувати. Це особливо важливо там, де інструмент безпосередньо створює частину ПЗ (у цьому значенні компілятор, імовірно, є найбільш важливим інструментом).

Призначенням стандартів розроблення ПЗ є визначення правил і обмежень для етапу розроблення:

а) стандарт вимог до ПЗ призначено для визначення методики, правил і засобів, що використовуються при розробленні вимог високого рівня, і має у своєму складі:

– методи, що використовуються при розробленні вимог до ПЗ (структурні, об'єктно-орієнтовані й ін.);

– нотації, що використовуються для вираження цих вимог (діаграми потоків даних, специфікації формальних мов та ін.);

– обмеження на використання засобів розроблення вимог;

б) стандарти проектування ПЗ, які дають змогу визначити методики, правила і засоби, що використовуються для створення архітектури ПЗ і вимог низького рівня, мають у своєму складі:

– методи опису процесів проектування, що використовуються;

– угоди щодо використання програмних імен;

– умови, щодо накладаються на дозволені методи проектування (наприклад, використання переривань і архітектури, що керується подіями, динамічне керування задачами тощо);

– обмеження на використання засобів проектування;

– обмеження на використання керувальних структур і програмних конструкцій (наприклад, на виключення, рекурсію, динамічні об'єкти, посилання на дані під різними іменами тощо);

– обмеження щодо складності (наприклад, максимальна кількість вкладених викликів процедур або умовних структур, використання безумовних галужень тощо);

в) стандарти кодування ПЗ, які дають змогу визначити мову програмування, методи, правила і засоби, що використовуються для кодування ПЗ, містять у своєму складі:

– мови програмування та(або) їх підмножини (для мови необхідно навести дані, які несуперечливо визначають синтаксис, можливості керування даними й сторонні ефекти мови);

– стандарти на програмний код (наприклад, обмеження на довжину рядка, відступи, використання порожніх рядків), стандарти на документування коду (наприклад, ім'я автора, дата написання, вхідні й вихідні дані, глобальні змінні, що використовуються);

– угоди щодо імен для компонентів, підпрограм, змінних, констант;

– умови й обмеження, що накладаються на дозволені угоди за кодом (наприклад, ступінь зв'язків між компонентами ПЗ і складність логічних і числових виразів);

– обмеження на використання засобів кодування.

На етапі верифікації використовуються ці стандарти як основа для порівняння виходів етапів з необхідними результатами. Керівні принципи для відбіру стандартів такі:

1. Стандарти мають давати можливість компонентам ПЗ або взаємозв'язаній множині продуктів розроблятися одноманітно.

2. Стандарти мають забезпечувати неможливість використання таких конструкцій або методів, після виконання яких отримуються результати, що не відповідають вимогам безпеки.

Складовими етапу розроблення ПЗ є такі підетапи:

– розроблення вимог до ПЗ;

– проектування ПЗ;

– кодування ПЗ;

– інтеграції ПЗ.

Під час розроблення ПЗ виробляється кілька рівнів вимог до ПЗ. Вимоги високого рівня випливають безпосередньо з аналізу вимог до системи та її архітектури, потім розвиваються на підетапи проектування ПЗ, формуючи один і більше подальших рівнів вимог, які є більш низькими. Проте якщо початковий код генерується безпосередньо з вимог високого рівня, то вони можуть розглядатися одночасно і як вимоги низького рівня. У цьому випадку до них застосовуються відповідні основоположні принципи формування.

Під час розроблення архітектури ПЗ припускається ухвалення рішення про структуру ПЗ. На підетапі проектування ПЗ визначаються як архітектура ПЗ, так і вимоги низького рівня, виходячи з яких, можна безпосередньо реалізувати програмний код.

Кожна зі складових етапу розроблення ПЗ може породжувати похідні вимоги, які прямо не зводяться до вимоги високого рівня, наприклад необхідність розроблення ПЗ підтримки переривань для вибраного цільового комп'ютера. Вимоги як високого, так і низького рівня можуть містити похідні вимоги, вплив яких на вимоги безпеки визначається на етапі оцінювання безпеки системи.

На підетапі розроблення вимог до ПЗ використовуються вихідні дані ЖЦ системи для вироблення таких вимог високого рівня, як функціональні вимоги, вимоги продуктивності, інтерфейсні вимоги й вимоги безпеки.

Вимоги високого рівня до ПЗ на підетапі проектування переробляються під час кількох ітерацій в архітектуру ПЗ, а вимоги низького рівня використовуються для реалізації програмного коду.

На підетапі кодування, виходячи з архітектури ПЗ і вимог низького рівня, виробляється програмний код ПЗ. Цільовий комп'ютер, а також програмний і об'єктний коди, які отримано на підетапі кодування, використовуються спільно з компонуванням і завантаженням даних на підетапі інтеграції для створення інтегрованої системи. Підетап інтеграції містить програмну й програмно-апаратну інтеграцію.

Усі підетапи етапу розроблення можуть починатися при задоволенні планованого проміжного критерію переходу. Підетапи вважаються завершеними, коли їхню мету, а також мету всіх пов'язаних з ними підетапів досягнуто.

Верифікація, що входить до складу етапу інтеграції, – це технічне оцінювання результатів як етапу розроблення ПЗ, так і етапу його верифікації (останній застосовується згідно з етапом планування розроблення ПЗ і планом верифікації ПЗ). Верифікація не зводиться тільки до тестування ПЗ, яке в загальному випадку не може відобразити відсутність помилок, тому використовується термін «верифікація», а не «тестування».

Метою етапу верифікації є виявлення помилок, які могли бути занесені в ПЗ на етапі його розроблення. Коригування таких помилок має

проводитися на цьому ж етапі. Основною метою етапу верифікації є перевірка такого:

– системні вимоги втілено у вимоги високого рівня;

– вимоги високого рівня втілено в архітектурі ПЗ і вимогах низького рівня;

– архітектуру і вимоги низького рівня втілено в програмному коді;

– об'єктний код відповідає вимогам до ПЗ;

– засоби, що використовуються для досягнення перелічених вище цілей, є технічно коректними й функціонально закінченими для програмного рівня.

Методи етапу верифікації базуються на комбінації оглядів, аналізуванні, розробленні тестових подій і процедур та подальшого їх виконання. Огляди й аналізування забезпечують оцінювання точності, повноти й перевірку вимог до ПЗ, архітектури ПЗ і програмного коду. Завдяки розробленню тестових подій забезпечується більш повне оцінювання внутрішньої спроможності ПЗ і повнота вимог до нього. Виконання тестових процедур забезпечує демонстрацію відповідності ПЗ призначеним вимогам.

Вхідні дані етапу верифікації: вимоги до системи, вимоги до ПЗ і архітектури ПЗ, дані про трасування, програмний та об'єктний код, план верифікації ПЗ.

Вихідні дані етапу верифікації ПЗ фіксуються в звіті ситуацій та процедур верифікації ПЗ, а також у звіті результатів верифікації ПЗ.

На етапі верифікації забезпечується перевірка відповідності реалізації вимогам до ПЗ такими методами:

– перевірка відповідності між вимогами до ПЗ і тестовими ситуаціями виконується шляхом аналізування області покриття вимог;

– відповідність між структурою коду й тестовими ситуаціями виконується шляхом аналізування області покриття структури.

До результатів етапів розроблення й верифікації ПЗ можна застосовувати різні види аналізу й оглядів. Відмінність між оглядом і аналізом полягає в тому, що під час аналізу отримуються повторювані докази коректності, а під час огляду забезпечується кількісне оцінювання коректності. Огляд може складатися з дослідження вихідних даних етапу, який виконано у вигляді діаграми. Під час аналізу може досліджуватися в деталях функціонування, продуктивність, трасування й безпека компонентів ПЗ.

Мета таких оглядів і аналізів полягає у виявленні помилок у вимогах, які могли бути занесені під час виконання підетапу розроблення вимог до ПЗ (проектування, розроблення архітектури, кодування).

Метою оглядів і аналізування результатів підетапу інтеграції є переконання в тому, що результати цього етапу є повними й коректними. Це можна виконати, детально досліджуючи скомпоновані й завантажувані

дані та карти пам'яті й перевіряючи коректність апаратних адрес, перекриття областей пам'яті, відсутність окремих компонентів ПЗ.

Метою тестування є перевірка, що ПЗ задовольняє вимогам до нього, і демонстрація, що помилки, які призводять до збою системи, було видалено з програмного забезпечення. Існують такі типи тестування:

– тестування програмно-апаратної інтеграції – для перевірки коректності виконання програмної операції в середовищі цільового комп'ютера;

– тестування програмної інтеграції – для перевірки взаємодії між вимогами до ПЗ і його компонентами, а також правильності реалізації вимог до ПЗ і його компонентів у межах архітектури ПЗ;

– низькорівневе тестування – для перевірки правильності реалізації вимог низького рівня до ПЗ.

Для досягнення мети тестування необхідно щоб тестові ситуації базувалися на вимогах до ПЗ їх було розроблено для перевірки правильності функціонування і виявлення умов, у яких можуть бути потенційні помилки.

З аналізу покриття вимог до ПЗ має випливати, які вимоги не тестувалися, а з аналізу покриття структури – які програмні структури не досліджувалися.

Етап контролю за конфігурацією, що взаємодіє з іншими етапами ЖЦ ПЗ, дає змогу досягти:

– забезпечення певної керованої конфігурації ПЗ протягом ЖЦ;

– забезпечення можливості розмножити повноцінний об'єктний код для випуску ПЗ або відтворити його у разі потреби;

– забезпечення керування вхідними й вихідними даними етапів ЖЦ ПЗ для можливості й відтворення дій етапів;

– визначення даних для оглядів і оцінювання статусу для змінення параметрів конфігурації до встановлення базового режиму;

– забезпечення керування для можливості надання уваги проблемам і запису, оцінювання і реалізації змін;

– забезпечення доступності інформації про ПЗ шляхом керування вихідними даними етапів ЖЦ ПЗ;

– достовірних оцінок відповідності ПЗ вимогам до нього;

– обов'язкового включення до параметрів конфігурації підтримки безпечної фізичної архівації, відновлення й керування.

Етап керування й контролю конфігурації містить ідентифікацію конфігурації, керування змінами, установлення базової конфігурації та архівацію ПЗ і даних ЖЦ ПЗ. Цей етап не зупиняється після сертифікації ПЗ, а продовжує функціонувати протягом всього терміну служби системи.

На етапі оцінювання якості ПЗ оцінюються вихідні дані підетапів ЖЦ ПЗ для прийняття рішення про задоволення поставлених цілей, виявлення, оцінювання й усунення помилок та узгодження ПЗ і даних ЖЦ з

вимогами сертифікації. Цей етап застосовується відповідно до етапу планування розроблення ПЗ і плану оцінювання якості ПЗ. За вихідними даними етапу оцінювання якості в Реєстрі формуються оцінки якості ПЗ або інших даних ЖЦ ПЗ.

На етапі оцінювання якості має визначатися, чи відбулося після виконання підетапів ЖЦ розроблення такого ПЗ, яке задовольняє поставленим вимогам, при цьому припускається, що ці підетапи виконувалися відповідно до прийнятих планів і стандартів.

Мета етапу оцінювання якості – переконатися в такому:

– етап розроблення ПЗ відповідає прийнятим планам і стандартам розроблення ПЗ;

– задоволено проміжний критерій для підетапів ЖЦ ПЗ;

– проведено ревізію програмного продукту.

Для досягнення цієї мети необхідно, щоб етап оцінювання мав певне значення в життєвому циклі ПЗ, при цьому виконувалися такі вимоги:

а) забезпечення створення планів і стандартів розроблення ПЗ та оцінювання їхньої коректності;

б) забезпечення проходження етапів ЖЦ ПЗ відповідно до планів і стандартів;

в) фіксація подій на етапах розроблення й інтеграції протягом ЖЦ ПЗ з метою визначення такого:

– чи знайдено, записано, оцінено, перевірено й виправлено відхилення від планів і стандартів розроблення;

– чи зафіксовано санкціоновані відхилення;

– чи використовується за планом середовище розроблення;

– чи проводиться сповіщення про проблеми, їх виправлення й перевірка згідно з планом керування конфігурацією ПЗ;

– чи успішно отримано результати етапу оцінювання безпеки системи.

На етапі керування й контролю конфігурації ПЗ має контролюватися відповідність проміжних критеріїв ЖЦ ПЗ за планом розроблення ПЗ і перевірятися дані ЖЦ ПЗ.

Перш ніж розсилати копії ПЗ для проведення сертифікації, необхідно виконати його ревізію.

На етапі керування й контролю конфігурації ПЗ має фіксуватися розвиток подій і результати ревізії ПЗ для кожного програмного продукту, що сертифікується.

Основна мета ревізії полягає в тому, щоб переконатися, що для сертифікованого програмного продукту етапи ЖЦ ПЗ завершено, їхні вихідні дані сформовано, об'єктний код контролюється і його можна виконати.

Метою забезпечення сертифікації є встановлення взаєморозуміння між виробником ПЗ і службами, що здійснюють сертифікацію, протягом

всього ЖЦ. Цей етап застосовується відповідно до етапу планування розроблення ПЗ і плану програмних аспектів сертифікації.

Виробник ПЗ має надати докази того, що ЖЦ відповідає плану розроблення ПЗ. Служба сертифікації може видавати свої звіти для виробника ПЗ або його постачальника з обговоренням тих або інших аспектів ЖЦ ПЗ. Виробник корелює зауваження і методики ЖЦ і надає необхідні дані. Виробник ПЗ зобов'язаний:

– розглядати звіти служб сертифікації;

– передавати дані про роботи, виконані над ПЗ, і реєстр конфігурації ПЗ службам сертифікації;

– передавати або робити доступними інші дані службам сертифікації.

Крім того, службам сертифікації може бути передано план програмних аспектів сертифікації.

Звичайно, якщо не заперечують служби сертифікації, заходи щодо регулюванню даних ЖЦ, які належать до розроблення типового зразка, застосовують до даних про вимоги до ПЗ, опису проектування, програмного коду, об'єктного коду, реєстру конфігурації ПЗ, звіту про роботи, що виконано над ПЗ.

Необхідно зазначити, що розглянуті керівні принципи в цілому задовольняють міжнародним і державним стандартам ISO 9000/9001, IEC, ДСТУ: стандартам в області адміністративного керування якістю й забезпечення якості та керівним положенням щодо їх застосування під час розроблення, поставки й технічного обслуговування ПЗ для комп'ютерів, що використовуються в промислових системах безпеки.

1.9. Процеси життєвого циклу програмного забезпечення загального використання

У стандарті ISO/IEC 12207 розглядаються високорівнева модель ЖЦ і фундаментальні цілі, що є істотними для розроблення високоефективного і надійного ПЗ, тобто те, що має бути досягнуто, а не те, як цього досягти.

Життєвий цикл, визначений у стандарті, можна застосувати в будь-якій організації, що бажає затвердити, а надалі й поліпшити можливості поставки, розроблення, експлуатації, розвитку й супроводження ПЗ. У моделі не припускається використання специфічних організаційних структур, філософії керування, технології або методологій розроблення ПЗ.

Архітектура моделі організовує процеси для допомоги персоналу організації в їх безперервному вдосконаленні. Процеси, визначені в стандарті, утворюють множину процесів, а організація залежно від своїх цілей може вибрати відповідну їх підмножину. Тому стандарт розроблено так, щоб його можно було пристосувати як для окремої організації, так і

для конкретного проекту. Крім того, його можна використовувати у випадках, коли ПЗ є автономним, вбудованим в апаратуру або є складовою частиною загальної системи.

Стандарт ISO/IEC 12207 не підтримує якусь конкретну модель ЖЦ ПЗ, керування ПЗ, метод розроблення або локальну промислову технологію. У ньому також не зазначається, яким чином виконувати що-небудь. Ці моменти дуже сильно залежать від умов конкретного проекту й технологічного рівня організації та залишаються в її компетенції.

Цей стандарт пов'язаний зі стандартами ISO/IEC 15504 (інформаційна технологія – оцінювання процесу розроблення ПЗ) та ISO 9001 (системи якості – модель для забезпечення якості при проектуванні, розробленні, виробництві, монтажі й обслуговуванні).

Усі дії, які можуть здійснюватися під час ЖЦ, у стандарті розбиваються на три групи процесів відповідно до дії, яку вони виконують. Кожний процес поділяється декілька дій. Кожна дія, у свою чергу, поділяється на задачі. Під задачею розуміється дія, яка перетворює входи на виходи. Процеси, дії і задачі можна виконувати послідовно, паралельно, повторно й комбіновано.

У стандарті ISO/IEC 12207 наведено такі основні процеси:

1. Процес придбання – визначаються дії замовника системи або програмного продукту.

2. Процес постачання – визначаються дії постачальника (організації), що надає програмний продукт замовнику.

3. Процес розроблення – визначаються дії розробника програми.

4. Процес експлуатації – визначаються дії оператора, організації, що надає користувачам послуги з експлуатації комп'ютерної системи в її робочому середовищі.

5. Процес супроводу – визначаються дії організації, що супроводжує ПЗ, тобто керування змінами в ПЗ для підтримки його в актуальному й працездатному стані. Цей процес містить переміщення й утилізацію ПЗ.

Супровідний процес підтримує роботу інших процесів як єдиного цілого з певною метою і сприяє успіху і якості проекту. Супровідний процес використовується зазвичай в інших процесах і складається з таких процесів:

1. Процес документування – визначаються дії для запису інформації про події, що відбуваються під час ЖЦ.

2. Процес керування конфігурацією – визначаються дії щодо керування конфігурацією.

3. Процес гарантії якості – визначаються дії, що гарантують відповідність програмних продуктів певним вимогам і дотримання встановлених планів.

4. Процес верифікації – визначаються дії (для замовника, постачальників або незалежної сторони) щодо верифікації програмних продуктів залежно від конкретного проекту ПЗ.

5. Процес атестації – визначаються дії (для замовника, постачальників або незалежної сторони) для атестації програмних продуктів проекту.

6. Процес загального огляду – визначаються дії щодо оцінювання статусу і якості продуктів. Цей процес може використати будь-яка з двох сторін.

7. Процес аудиту – визначаються дії для встановлення відповідності вимогам, планам і контрактам. Цей процес можуть використовувати дві сторони, де одна сторона (аудитор) проводить аудит програмних продуктів або дій іншої (піддається аудиту).

8. Процес розв'язання проблем – визначаються дії для аналізування й усунення проблем протягом розроблення, експлуатації, супроводу або інших процесів.

Організаційні процеси використовуються організацією на верхньому рівні для встановлення, виконання й удосконаленні структури, яку побудовано на зв'язку процесів ЖЦ і персоналу. Зазвичай їх використовують поза сферою конкретних проектів і контрактів, проте результати цих проектів і контрактів сприяють удосконаленню організації.

Організаційними є такі процеси:

1. Процес керування – визначаються основні дії щодо керування (у тому числі керування проектом) під час процесів ЖЦ.

2. Процес інфраструктури – визначаються основні дії зі встановлення структури процесу.

3. Процес удосконалення – визначаються основні дії, що виконує організація (замовник, постачальник, розробник, організація, що експлуатує, супроводжує або менеджер іншого процесу) для встановлення, оцінювання, перевірки і вдосконалень процесів ЖЦ.

4. Процес навчання – визначаються дії щодо підготовки персоналу.

1.9.1. Процес придбання програмного забезпечення

Процес придбання ПЗ стосується дій і задач замовника і починається з визначення потреб у придбанні системи або програмних продуктів. Потім готують і випускають заявки на пропозицію, вибирають постачальника і керують придбанням.

Замовник керує процесом придбання на рівні проекту, закладає інфраструктуру; корегує процес для проекту й керує процесом на організаційному рівні згідно з процесом удосконалення.

Цей процес містить такі дії:

– ініціалізація;

– підготовка замовлення для пропозиції;
– підготовка й коригування контракту;
– пошук постачальників;
– прийняття рішення й висновок.

Ініціалізація складається з таких основних завдань.

Замовник починає процес придбання, виявляючи ідею або необхідність придбання, розроблення або покращення програмної системи або програмного продукту.

Замовник визначає й аналізує системні вимоги, які мають містити безпеку, надійність та інші критичні вимоги, що стосуються тестування, і бути узгодженими зі стандартами й процедурами. Якщо замовник запрошує постачальника для виконання аналізу вимог системи, то він має схвалити проаналізовані вимоги. Замовник може встановити вимоги до ПЗ самостійно або запросити постачальника для виконання цієї задачі. Процес розроблення можна використати для виконання наведених задач.

Замовник обговорює варіанти придбання ПЗ, аналізуючи ризик по кожному можливому варіанту:

– придбання готового до використання програмного продукту, що задовольняє вимогам;
– розроблення програмного продукту самостійно;
– розроблення програмного продукту через контракт;
– комбінація перелічених вище варіантів;
– удосконалення існуючого програмного продукту.

Замовник готує план придбання, де визначає відповідні вимоги до планованого використання системи, тип контракту, який буде використано, обов'язки залучених організацій, концепцію підтримки, яку буде використано, і ризики, які беруть до уваги, а також методи керування ризиками. План треба зареєструвати і виконати.

Аквізитор має визначити і зареєструвати стратегію і умови (критерії) прийняття програмного продукту.

Підготовка заявки для пропозиції (тендеру). Замовник має зареєструвати вимоги придбання (наприклад, заявку на участь), зміст яких залежить від варіанта, вибраного при виконанні попередньої дії. Документація має містити вимоги системи, формулювання роботи, інструкції для покупців, перелік програмних продуктів, статті договору й піддоговору, технічні обмеження (наприклад, середовище використання).

Замовник має визначити, які процеси, дії і задачі стандарту належать до проекту і мають бути відповідно узгоджені. Замовник має особливо визначити прийнятні процеси супроводу та організації, що будуть їх виконувати, так що постачальники можуть визначити підхід до кожного з певних супровідних процесів.

У документації також мають визначатися точки контракту, відносно яких будуть розглядатися і перевірятися дії постачальника як частина

моніторингу. Вимоги щодо придбання мають бути поставлені організації, яку вибрано для виконання цих дій.

Решта дій цього процесу стосується вимог до підготовки контракту, його коригування (виконання й завершення тут не розглядаються).

У процесі поставки визначаються дії та задачі постачальника. Процес може бути ініційований як прийняттям рішення щодо підготовки пропозиції у відповідь на оголошений замовником тендер, так і підписанням контракту із замовником. Процес продовжується ідентифікацією процедур і ресурсів, необхідних для керування і забезпечення проекту, у тому числі розроблення планів проекту і виконання планів поставки програмного продукту аквізитору.

Цей процес складається з таких дій: уведення, підготовка відповіді, контракт, планування, виконання і контроль, огляд і оцінювання, поставка і висновок.

1.9.2. Процес розроблення програмного забезпечення

Процес розроблення визначає дії та задачі розробника для аналізування вимог, проектування, програмування, інтеграції, тестування, інсталяції і приймання, що належать до ПЗ, і може мати системні дії, якщо це передбачено в контракті.

Розробник керує процесом розроблення на рівні проекту згідно з процесом керування; подає інфраструктуру процесу згідно з процесом інфраструктури; доводить процес для проекту відповідно до процесу доведення та керує процесом на організаційному рівні відповідно до процесу вдосконалення.

Розроблення ПЗ складається з таких видів діяльності.

1. Інструментарний процес. Розробник має визначити або вибрати модель ЖЦ відповідно до призначення, значущості й складності проекту, якщо це не обумовлено в контракті. Дії і задачі цього процесу треба вибрати й відобразити в моделі ЖЦ. Ці дії і задачі можуть перекриватися або взаємодіяти і їх можна виконати ітеративно або рекурсивно.

Розробник повинен:

– документувати результати відповідно до процесу документування;

– розміщувити результати (виходи) під час конфігурації та виконувати контроль змін відповідно до цього;

– документувати і вирішувати проблеми й невідповідності, які знайдено в програмному продукті і задачах;

– інші процеси супроводу, які визначено в контракті.

Розробник має вибрати, довести й використати внутрішні стандарти, методології, процедури й мови програмування (якщо це не обумовлено в контракті), які організація має зареєструвати для виконання дій процесів розроблення й підтримки.

Розробник має розробити плани керування діями під час розроблення, які мають містити особливі стандарти, методи, засоби, дії і обов'язки, що пов'язані з розробленням і кваліфікацією всіх вимог, у тому числі з надійністю і безпекою. Ці плани треба зареєструвати й виконати.

Елементи, що офіційно не поставляються, можна використати під час розроблення або супроводу ПЗ. Однак має бути гарантія того, що експлуатація і підтримка ПЗ після поставки замовнику не залежатиме від таких елементів. Іншими словами, ці елементи стають такими, що офіційно поставляються.

2. Аналіз системних вимог. Ця дія складається із завдань, які розробник має виконати або підтримувати за умовами контракту.

Для особливо важливих систем треба визначити системні вимоги. У специфікації системних вимог має бути описано: функції і можливості системи; надійність, захист, інтерфейс, вимоги щодо експлуатації і підтримки; обмеження проектування і кваліфікаційні вимоги. Специфікації системних вимог слід зареєструвати.

Системні вимоги треба оцінити з позицій таких критеріїв: простежуваність потреб замовника, узгодженість з потребами замовника, тестованість, здійсненність системного проектування, здійсненність експлуатації і супроводу.

3. Системне проектування. Цей процес складається з таких завдань, які розробник має виконувати або супроводжувати:

– розроблення архітектури верхнього рівня, де визначаються елементи апаратного і програмного забезпечення, а також ручні операції;

– перевірка того, що всі системні вимоги повністю розподілено серед елементів, які мають бути згодом визначені як елементи апаратної конфігурації (ЕАК), елементи конфігурації ПЗ (ЕКПЗ) і ручні операції;

– реєстрація архітектури системи, системних вимоги, розподілених між елементами апаратної, конфігурації, конфігурації ПЗ і ручними операціями;

Архітектуру системи і вимоги до елементів конфігурації і ручних операцій оцінюють за такими критеріями:

– ідентифікація системних вимог;

– узгодженість із системними вимогами;

– відповідність архітектури системи і вимог стандартам і методам проектування, які використано у проекті;

– здійсненність наповнення елементів конфігурації ПЗ вимогами;

– здійсненність експлуатації і підтримки.

4. Аналіз вимог до ПЗ. Для кожного ЕКПЗ розробник має подати і зареєструвати вимоги до ПЗ, у тому числі специфікації характеристик якості:

– специфікації функцій і можливостей (у тому числі умови виконання, фізичні характеристики, умови навколишнього середовища), у яких працюватиме ПЗ;

– зовнішній інтерфейс ЕКПЗ;

– кваліфікаційні вимоги;

– специфікації безпеки, у тому числі такі, в яких описано експлуатацію і підтримку, захист від впливу навколишнього середовища і некоректних дій персоналу;

– специфікації захисту, у тому числі такі, що стосуються особливої інформації і матеріалів;

– людський чинник і специфікації людино-машинної взаємодії, у тому числі такі, що належать до ручних операцій, людино-машинних взаємодій, обмежень щодо персоналу й області застосування, які потребують концентрації людської уваги, є чутливими до помилок людини;

– вимоги щодо визначення даних і баз даних;

– вимоги щодо інсталяції і приймання ПЗ, яке поставляється в експлуатацію, і супроводу;

– документація користувача;

– вимоги щодо експлуатації, які стосуються користувача;

– вимоги щодо супроводу, які стосуються користувача.

Настанову щодо визначення характеристик якості можна знайти в ISO/IEC 9126 [3].

Розробник оцінює вимоги до ПЗ, керуючись такими критеріями:

– простежуваність системних вимог і системного проекту;

– зовнішня узгодженість з системними вимогами;

– внутрішня узгодженість;

– тестове покриття вимог;

– тестованість;

– здійсненність проекту ПЗ;

– здійсненність експлуатації і супроводу.

5. Проектування ПЗ. Для кожного ЕКПЗ проектування складається з таких задвдань:

а) розробник має перетворити вимоги для ЕКПЗ на архітектуру, що описує структуру верхнього рівня і визначає головні компоненти, при цьому слід гарантувати те, що вимоги до ЕКПЗ повністю розподілено між компонентами й далі їх уточнено для полегшення детального проектування;

б) розробник має розробити і зареєструвати:

– проект вищого рівня для зовнішньої взаємодії з ЕКПЗ і компонентами ПЗ;

– проект вищого рівня для баз даних;

– попередні версії керівництва для користувача;

– попередні тестові вимоги і план інтеграції ПЗ.

в) розробник має оцінити архітектуру ЕКПЗ і проекти інтерфейсу і баз даних, керуючись такими критеріями:

– простежуваність вимог до ЕКПЗ;

– зовнішня узгодженість з вимогами до ЕКПЗ;

– внутрішня узгодженість між компонентами;

– відповідність методів проектування і стандартів, які було використано;

– здійсненність проектування, що деталізується;

– здійсненність експлуатації і супроводу;

г) розробник має керувати загальним оглядом.

6. Детальне проектування ПЗ. Для кожного ЕКПЗ ця дія складається з таких завдань:

а) розробник має розробити детальний проект для кожного компонента ПЗ ЕКПЗ; компоненти, що містять модулі ПЗ, які можуть кодуватися, компілюватися і тестуватися, треба уточнити на нижніх рівнях; має бути гарантія того, що вимоги до ПЗ повністю локалізовано в модулях ПЗ; детальний проект слід зареєструвати;

б) розробник має розробити і задокументувати детальний проект для зовнішнього інтерфейсу ЕКПЗ між компонентами ПЗ і між модулями ПЗ; детальний проект інтерфейсу має давати змогу писати код програми без застосування додаткової інформації;

в) розробник має розробити та зареєструвати детальний проект бази даних;

г) розробник має оновлювати настанову користувача, наскільки це необхідно;

д) розробник має визначити і задокументувати тестові вимоги і розклад тестування блоків ПЗ; тестові вимоги мають містити випробування ПЗ на межі вимог;

е) розробник має оновити тестові вимоги і план інтеграції ПЗ;

ж) розробник має оцінити детальний проект ПЗ і тестові вимоги за такими критеріями:

– простежуваність вимог ЕКПЗ;

– зовнішня узгодженість з архітектурою проекту;

– внутрішня узгодженість між компонентами і модулями;

– відповідність методів проектування і використаних стандартів;

– здійсненність тестування;

– здійсненність експлуатації і супроводу.

7. Програмування і відлагодження. Для кожного ЕКПЗ ця дія складається з кількох завдань.

Розробник має розробити і задокументувати:

а) кожний модуль ПЗ і бази даних;

б) процедури тестування і дані для тестування кожного модуля і бази даних.

Розробник має тестувати кожний модуль ПЗ і бази даних, та переконатися в тому, що вони задовольняють вимогам. Результати тестування треба задокументувати.

Розробник має оновлювати настанову користувача, тестові вимоги й план інтеграції ПЗ, оцінювати код ПЗ і результати тесту за такими критеріями:

– простежуваність вимог ЕКПЗ і проекту;
– зовнішня узгодженість з вимогами ЕКПЗ та архітектурою проекту;
– внутрішня узгодженість між вимогами модулів;
– тестування модулів;
– відповідність методів кодування й використаних стандартів;
– здійсненність інтеграції ПЗ і тестування;
– здійсненність експлуатації і супроводу.

8. Інтеграція ПЗ. Для кожного ЕКПЗ розробник має розробити план інтеграції, щоб об'єднати модулі ПЗ і компоненти в ЕКПЗ. План має містити вимоги, процедури, дані, відповідальність і розклад.

Розробник має об'єднати модулі ПЗ і компоненти. Необхідна гарантія того, що кожний компонент задовольняє вимогам. Інтеграція і результати тесту мають бути задокументованими.

Розробник має оновлювати настанову користувача, якщо це потрібно.

Розробник має розробити й задокументувати для кожної кваліфікаційної вимоги ЕКПЗ повний набір тестів, ситуацій (вхід, вихід, критерії тестування) і процедури тестування для керування кваліфікаційним тестуванням ПЗ.

Розробник має оцінити план інтеграції, проект, код, тести, результати тестування та настанову користувача за такими критеріями:

– простежуваність системних вимог;
– зовнішня узгодженість із системними вимогами;
– внутрішня узгодженість;
– тестування ЕКПЗ вимог;
– відповідність використаним стандартам;
– відповідність очікуваним результатам;
– здійсненність кваліфікаційного тестування ПЗ;
– здійсненність експлуатації й супроводу.

9. Кваліфікаційне тестування ПЗ. Розробник має керувати кваліфікаційним тестуванням відповідно до кваліфікаційних вимог, особливих для ЕКПЗ. Необхідна гарантія, що програмну реалізацію кожної вимоги до ПЗ повністю протестовано. Результати кваліфікаційного тестування слід зареєструвати.

Розробник має оцінити проект, код, тести, результати тестування й настанову користувача на відповідність таким критеріям:

– тестування вимог до ЕКПЗ;

– узгодженість з очікуваними результатами;
– здійсненність системної інтеграції і тестування;
– здійсненність експлуатації і супроводу.

Розробник має підтримувати аудит, а його результати задокументувати. Якщо розробляються або інтегруються ПЗ та апаратне забезпечення, то аудит можна відкласти до кваліфікаційного тестування системи.

Після успішного завершення аудиту розробник повинен:

а) відновити і підготувати ПЗ для системної інтеграції, кваліфікаційного тестування, інсталяції або підтримки приймання ПЗ;

б) надати основну лінію проектування й кодування ЕКПЗ.

10. Системна інтеграція. ЕКПЗ має бути інтегрованим із загальною системою та іншими підсистемами в єдину систему і протестованим на відповідність вимогам, а результати інтеграції і тестування – задокументуваними.

Для кожної кваліфікаційної вимоги треба розробити й задокументувати повний набір тестів, ситуацій (вхідних, вихідних, критеріїв тестування), процедур тестування. Це дасть змогу розробнику гарантувати, що інтегрована система готова для кваліфікаційного тестування.

Інтегровану систему необхідно оцінити на відповідність таким критеріям:

– зона тестування вимог до системи;
– прийнятність використаних методів і стандартів тестування;
– узгодженість з очікуваними результатами;
– здійсненність кваліфікаційного тестування системи;
– здійсненність експлуатації і супроводу.

11. Кваліфікаційне тестування системи. Кваліфікаційне тестування системи має керуватися відповідно до кваліфікаційних вимог, визначених для системи. Повинна бути гарантія, що виконання кожної вимоги до системи протестовано повністю і система готова до поставки. Результати кваліфікаційного тестування треба задокументувати.

Систему необхідно оцінити на відповідність таким критеріям:

– зона тестування вимог до системи;
– підтвердження очікуваних результатів;
– здійсненність експлуатації і супроводу.

Розробник має підтримувати аудит, а його результати задокументувати. Цей пункт не застосовується до таких ЕКПЗ, для яких аудит було виконано раніше.

Після успішного завершення аудиту розробник має відновити й підготувати до поставки ЕКПЗ для інсталяції ПЗ і його приймання замовником, а також обгрунтувати основні напрями для проектування й кодування ЕКПЗ.

12. Інсталяція ПЗ. Розробник повинен розробити план інсталяції ПЗ у цільове середовище. Ресурси й інформація, які необхідні для установлення ПЗ, мають бути визначеними і доступними. Розробник має допомагати постачальнику під час установки ПЗ. Після того, як ПЗ встановлено в існуючу систему, розробник має підтримувати деякі дії, що виконуються паралельно. План установки слід задокументувати.

Розробник має встановити ПЗ відповідно до плану установки. Повинна бути гарантія, що ПЗ і бази даних ініціалізувалися, функціонують і припиняють роботу, як це зазначено в контракті. Процес установлення й результати треба задокументувати.

13. Підтримка приймання ПЗ. Розробник має підтримувати процеси приймання й тестування ПЗ постачальником. Приймання й тестування мають ґрунтуватися на загальному огляді, аудиті, кваліфікаційному тестуванні системи (якщо воно виконувалося). Результати приймання й тестування треба задокументувати.

1.9.3. Процес експлуатації програмного забезпечення

Процес експлуатації складається з дій і задач того, хто експлуатує розроблене ПЗ, і містить процеси експлуатації ПЗ і підтримки користувачів. Оскільки експлуатація ПЗ є складовою експлуатації системи, дії і задачі цього процесу належать і до системи.

Оператор не тільки керує експлуатацією відповідно до процесу керування, але й коригує цей процес відповідно до коригувального процесу й удосконалює цей процес на організаційному рівні згідно з процесом удосконалення.

Цей процес складається з виконання процесів тестування, експлуатації системи й підтримки користувачів.

1.9.4. Процес супроводу програмного забезпечення

Процес підтримки складається з дій і задач того, хто виконує супровід. Цей процес починається, коли необхідна модифікація через допущені помилки, невраховані проблеми, або необхідне удосконалення або адаптація коду ПЗ і відповідної документації. Його мета – модифікувати існуюче ПЗ, зберігти його цілісність. Цей процес містить процеси розповсюдження й замінення ПЗ і завершується заміненням ПЗ на нову версію.

Процес складається з процесів реалізації, аналізування проблем і модифікації, реалізації модифікації, приймання, розповсюдження й замінення ПЗ.

1.9.5. Документація життєвого циклу

Синхронізація всіх етапів розроблення відбувається з допомогою документів, які створюються на кожному етапі. Документація при цьому створюється і на прямих гілках ЖЦ – при розробленні ПЗ, і на зворотніх – при верифікації.

Результатом етапу розроблення вимог до системи є документ, де описано загальні принципи роботи системи, її взаємодію з «навколишнім середовищем» – користувачами системи, а також програмними й апаратними засобами, що забезпечують її роботу. Зазвичай паралельно з вимогами до системи створюється план верифікації і визначається її стратегія. Ці документи визначають загальний підхід до того, як буде виконуватися тестування, які методики будуть застосовуватися, які аспекти майбутньої системи необхідно ретельно перевірити. Ще одна задача, що вирішується з допомогою визначення стратегії верифікації, – виявлення місця різних процесів верифікацій і їх зв'язків з процесами розроблення.

Процес верифікації, що застосовується при роботі із системними вимогами, – це процес валідації вимог і порівняння їх з реальним очікуванням замовника. Валідація відрізняється від приймально-здавальних випробувань, які виконуються під час передання готової системи замовнику, хоча може вважатися частиною таких випробувань. Валідація є засобом довести не тільки коректність реалізації системи з точки зору замовника, але й коректність принципів, що є основою її розроблення.

Вимоги до системи є основою для процесу розроблення функціональних вимог та архітектури проекту. Під час цього процесу розробляються загальні вимоги до ПЗ системи, до функцій, які вона має виконувати. Функціональні вимоги часто містять моделі поведінки системи в штатних і нештатних ситуаціях, правила оброблення даних і визначення вимог до інтерфейсу користувача. Функціональні вимоги є основою для розроблення архітектури системи – описи її структури в термінах підсистем і структурних одиниць мови, на якій проводиться реалізація (областей, класів, модулів, функцій тощо).

На основі функціональних вимог створюються тестові вимоги – документи, що визначають ключові точки, які треба перевірити для того, щоб переконатися в коректності реализації функціональних вимог.

Однією з проблем, що виникають при написанні тестових вимог, є принципова нетестованість деяких вимог, наприклад, вимогу «Інтерфейс користувача має бути інтуїтивно зрозумілим» неможливо перевірити без чіткого визначення того, що є інтуїтивно зрозумілим інтерфейсом. Такі неконкретні функціональні вимоги зазвичай згодом видозмінюють.

Архітектурні особливості системи можуть бути джерелом для створення тестових вимог, де враховано особливості програмної реалізації системи.

На основі функціональних вимог та архітектури створюється програмний код системи, для перевірки якого на основі тестових вимог готується тестовий план – опис послідовності тестів, з допомогою яких перевіряють відповідність реалізації системи вимогам. Кожний тест має конкретний опис значень, що подаються на вхід системи, значень, що очікуються на виході, й опис сценарію виконання тесту.

Залежно від об'єкта тестування тестовий план може готуватися у вигляді або програми на якій-небудь мові програмування, або файла вхідних даних і прогнозованих значень, або інструкцій для користувача системи, де описано необхідні дії, які треба виконати для перевірки функцій системи.

Після виконання всіх тестів збирається статистика про успішність проходження тестів – відсоток тестів, для яких реальні вихідні значення відповідають очікуваними, так званих пройдених тестів. Непройдені тести є даними для аналізування причин помилок ПЗ і подальшого їх виправлення.

На етапі інтеграції здійснюється об'єднання окремих модулів системи у єдине ціле та виконання функціональних тестів.

На останньому етапі здійснюється постачання готової системи замовнику. Перед упровадженням фахівці замовника спільно з розробниками проводять приймально-здавальні випробування – перевіряють критичні для користувача функції згідно з наперед затвердженою програмою випробувань. При успішному проходженні випробувань система передається замовнику, інакше відправляється на доробку.

1.9.6. Керування конфігурацією

Найбільш важливими допоміжними видами діяльності є керування конфігурацією і контроль за документацією.

Керування конфігурацією забезпечує механізм ідентифікації, контролю й дослідження варіантів кожного елемента ПЗ. У багатьох випадках попередні варіанти, які все ще продовжують використовуватися, мають технічно обслуговуватися, контролюватися й супроводжуватися.

Система керування конфігурацією повинна:

– однозначно ідентифікувати варіанти кожного елемента ПЗ;

– ідентифікувати варіанти кожного елемента ПЗ, які разом утворюють конкретний варіант готової продукції;

– ідентифікувати стан компоновки ПЗ, що розробляється, або яке вже поставлено й змонтовано;

– керувати одночасною модернізацією конкретного елемента ПЗ, що проводиться більш ніж однією людиною;

– забезпечувати координацію робіт із модернізацією ПЗ;

– ідентифікувати й простежувати всі заходи і змінення, які викликано заявкою, що змінилася, починаючи від самого зародження до випуску продукції.

Постачальник має розробити й реалізувати план керування конфігурацією, що містить:

– назви організацій, які задіяно в керуванні конфігурацією, і ступінь відповідальності кожної з них;

– види діяльності з керування конфігурацією, що має бути здійснено;

– технічні засоби, технології і методологічні принципи, які мають бути застосовані в керуванні конфігурацією;

– стадію, на якій елементи підлягають керуванню конфігурацією.

До видів діяльності, що пов'язана з керуванням конфігурацією, належать ідентифікація і простежуваність конфігурації, контроль змінень, установлення звіту про статус конфігурації.

Постачальник має встановити і здійснити процедури з ідентифікації елементів ПЗ на всіх фазах, починаючи зі складання технічних умов, розроблення, тиражування й закінчуючи поставкою. Кожний окремий елемент ПЗ повинен мати свій власний ідентифікатор, що відрізняється від інших.

Процедури мають бути гарантією того, що для кожного варіанта елемента ПЗ можна ідентифікувати:

– функціональні й технічні вимоги;

– усі технічні засоби, які використано під час розроблення і які впливають на функціональні й технічні вимоги;

– усі інтерфейси з іншими елементами ПЗ і з апаратними засобами;

– усі документи і комп'ютерні файли, що належать до конкретного елемента ПЗ.

Для програмної продукції необхідно встановити процедури, що полегшують простежуваність її елементів.

Постачальник має встановити й виконувати процедури з ідентифікації, документального оформлення, аналізування й санкціонування будь-яких змінень в елементах ПЗ у межах керування конфігурацією. Усі змінення елементів ПЗ мають проводитися відповідно до цих процедур.

Постачальник має встановити й забезпечити протоколювання, керування й надання звітів про статус ПЗ, замовлень на змінення й реалізацію затверджених змінень.

Постачальник має встановити й забезпечити процедури з контролю всіх документів.

До цих процедур належать:

– визначення тих документів, які мають бути об'єктом контролю;
– затвердження й публікація;
– змінення, у тому числі відмінність і, якщо необхідно, випуск.

Процедури контролю за документацією мають застосовуватися до відповідних документів і містити:

а) процедурні документи, що описують систему якості, яку необхідно застосовувати протягом усього ЖЦ ПЗ;

б) документи, що описують планування й розвиток усіх видів діяльності постачальника, а також взаємодію з користувачем;

в) документи на продукцію, що описують конкретну програмну продукцію та містять:

– інформацію на вході фази розроблення;
– очікувані результати наприкінці фази розроблення;
– плани і результати перевірок і оцінювань;
– документацію для покупця і користувача;
– експлуатаційну документацію.

Усі документи мають бути вивченими і затвердженими уповноваженими посадовцями до їх публікації. Діючі процедури мають гарантувати таке:

– необхідні документи є в наявності там, де виконуються операції, важливі для ефективного функціонування системи якості;
– застарілі документи швидко вилучаються з використання.

Особливу увагу слід звернути на відповідні процедури затвердження, доступу, розподілу й архівного зберігання електронних файлів.

1.9.7. Оцінювання процесів розроблення програмних систем

У стандарті ISO/IEC 15504 реалізуються положення стандартів серії ISO 9000 щодо надання постачальникам упевненості щодо керування якістю, забезпечуючи користувачів настановою для незалежного оцінювання можливості потенційних постачальників. Користувачі отримують змогу оцінити можливість процесу за безперервною шкалою простим і порівнянним способом, не використовуючи характеристики «виконано/не виконано» аудиту якості, що базуються на ISO 9001. Крім того, стандарт ISO/IEC 15504 дає змогу регулювати сферу оцінювання для покриття конкретних найважливіших процесів, а не всіх процесів, що використовуються в організації.

У стандарті ISO/IEC 15504 запропоновано оцінювання процесу розроблення ПЗ, що може використовуватися організаціями для планування, менеджменту, поточного контролю, керування й вдосконалення придбання, поставки, розроблення, функціонування, розвитку й підтримки ПЗ.

У стандарті ISO/IEC 15504 забезпечується структурований підхід до оцінювання організацією або від її імені процесу розроблення ПЗ для таких цілей:

– розуміння стану власних процесів для вдосконалення (поліпшення) процесу розроблення ПЗ;

– визначення придатності власних процесів для задоволення специфічної вимоги або класу вимог;

– визначення придатності процесів іншої організації для специфічного контракту або класу контрактів.

У стандарті, який відповідає всім наочним областям і розмірам організацій, пропонується самостійне оцінювання, визначається керування для оцінюваних процесів, береться до уваги контекст у якому оцінювані процеси функціонують, пропонується набір рейтингів процесу (профілю процесу).

Організація, використовуючи оцінювання процесу розроблення ПЗ, затверджує:

– культуру постійного поліпшення й установлення відповідних механізмів для підтримки й супроводу цієї культури;

– інжиніринг процесів з метою задоволення ділових вимог;

– оптимальне використання ресурсів.

Користувачі отримують вигоду з використання процесу оцінювання, визначеного в цьому стандарті, ще дозволяє:

– зменшити невизначеність і ризик під час вибору постачальника програмних систем при укладенні контракту;

– розістити відповідні засоби керування в найбільш ризиковані місця життєвого циклу проекту;

– забезпечити певну кількісну основу для вибору фінансових потреб, вимог і оцінити вартість проекту щодо можливостей конкуруючих постачальників.

Основні переваги стандартизованого підходу до оцінювання процесу розроблення полягають у такому:

– запропоновано загальнодоступну модель;

– досягнуто загальне розуміння у використанні оцінювання для поліпшення процесу й визначення його можливості;

– полегшено процедуру визначення можливості поставки устаткування;

– процес розроблення ПЗ керується і регулярно є видимим з огляду на досвід використання;

– підхід можна змінити тільки за міжнародною згодою;

– гармонізуються існуючі моделі й схеми оцінювання.

Оцінюються вибрані процеси на основі моделі оцінювання, що має п'ять основних специфічних дій: планування, збір даних, верифікація даних, ранжирування процесів і документування. Процес оцінювання має

бути документованим. Крім того, експерти-консультанти мають записати об'єктивні показники виконання або використаної можливості, щоб довести досягнення рейтингів. Оцінювання процесу виконується групою принаймні з одним компетентним експертом-консультантом, компетенцію якого описано в ISO/IEC 15504-6. Цей процес можна виконувати на безперервній основі з використанням придатних інструментальних засобів для збору даних, підтверджених компетентним експертом-консультантом.

З допомогою еталонної моделі процесів і можливостей процесів формується основа для будь-якої моделі, що використовується для оцінювання процесу. Еталонна модель містить двовимірний підхід для оцінювання можливості процесу – за одним вимірюванням визначаються оцінювані процеси, за другим – описується шкала для вимірювання можливості. Будь-яка модель, що є сумісною з еталонною моделлю, може використовуватися для оцінювання, і результати будь-якого сумісного оцінювання можуть переноситися до загальної бази.

Процес розроблення ПЗ успішно поліпшується в діловому контексті, де враховуються специфічні потреби і бізнес, – цілі організації, ключові обмеження, такі, як ресурси, культура тощо.

1.9.8. Вимірювання процесу

Процеси, прийняті в організаціях, що розробляють, експлуатують, придбавають, поставляють і підтримують функціонування ПЗ, поділяють на п'ять категорій, що містять усі процеси. Категорії і процеси зіставляються з процесами, визначеними у стандарті ISO/IEC 12207 Інформаційна технологія – життєвий цикл процесу ПЗ.

Як було зазначено вище, в еталонній моделі процеси об'єднують у три групи і п'ять категорій:

– початкові процеси ЖЦ, що містять категорії процесу інжинірингу і процесу постачальник – замовник;

– підтримувальні процеси ЖЦ, що містять категорії процесу підтримки;

– організаційні процеси ЖЦ, що містять категорії процесу керування й організації.

Опис кожної категорії процесів містить характеристику процесів, що супроводжується списком імен процесів.

Індивідуальні процеси описано в термінах шести компонентів.

Ідентифікатор процесу. Ідентифікує категорію і послідовний номер усередині цієї категорії. Схема нумерації різниться між процесами різних рівней. Ідентифікатор складається з двох частин: скорочення категорії (наприклад, ENG для категорії процесу інжинірингу) і номер (наприклад, CUS 1 означає «Процес придбання» і CUS 1.2 означає «Процес другого рівня», «Процес Вибору Постачальника», який є складовою процесу «Придбання»).

Назва процесу. Описова фраза, яка виділяє принципову властивість процесу (наприклад, «Вибір постачальника»).

Тип процесу. Існує три типи процесів верхнього рівня (базисний, розширений, новий) і два процеси другого рівня (компонентний, розширений), що мають таке відношення до процесів ISO/IEC 12207:

– нові процеси є додатковими до тих, які визначено в ISO/IEC 12207;

– базисні процеси є ідентичними за призначенням процесам ISO/IEC 12207;

– розширені процеси доповнюються на існуючому процесі ISO/IEC 12207;

– компонентні процеси групують одну або більшу кількість дій ISO/IEC 12207 з того ж самого процесу;

– розширені компонентні процеси групують одну або більшу кількість дій ISO/IEC 12207 з того ж самого процесу і містять додатковий матеріал;

Мета процесу, відповідно до якої встановлюються загальні цілі виконання процесу на верхньому рівні. Необов'язковий додатковий матеріал можна включити, щоб далі визначати затвердження мети.

Результати процесу – список описів результатів процесу.

Примітки процесу – необов'язковий список інформативних приміток щодо процесу і його відношення до інших процесів.

1.9.9. Оцінювання процесів

Необхідно визначити набір мінімальних вимог для оцінювання процесів, які збільшать імовірність того, що результати оцінювання будуть цільовими, неупередженими, несуперечливими, повторюваними й достовірними і установлять обставини, за якими вони будуть порівнянними.

Під час оцінювання необхідно ураховувати контекст, у якому функціонують оцінювані процеси; проводити набори рейтингів процесу (профілі процесу); через атрибути процесу, що застосовують до всіх процесів, давати змогу їх вдосконалювити, щоб було досягнуто їхніх цілей; відповідати всім областям і розмірам організації.

Щоб збільшити несуперечність оцінювання процесів, необхідно забезпечити вимоги для його проведення. Вихід оцінювання має бути несуперечливим.

Вхід треба визначити до оцінювання і його має схвалити замовник. Вхід оцінювання має визначати таке:

1. Особа замовника оцінювання і його належність до оцінюваної організації.

2. Мета оцінювання.

3. Сфера оцінювання, у тому числі:

а) процеси, які будуть досліджуватися всередині організації;

б) найвищій рівень можливості, який має досліджуватися;

в) частина організації, яка розгортає ці процеси;

г) контекст процесу, який містить щонайменше розмір організації; демографію організації; предметну область продукту або послуг організації; розмір, уразливість і складність виробів або послуг; якісні характеристики виробів (див., наприклад, ISO/IEC 9126 Характеристики якості ПЗ).

4. Обмеження цілісності оцінювання, до яких можуть належати:

а) доступність ключових ресурсів;

б) максимальна кількість часу, що має використовуватися для оцінювання;

в) специфічні процеси, які мають бути виключеними з оцінювання;

г) мінімальний, максимальний або специфічний типовий розмір або покриття, що є бажаним для оцінювання;

д) власність на виконання оцінювання і будь-які обмеження на їх використання;

е) засоби керування інформацією, що випливає з умови конфіденційності.

5. Особи експертів-консультантів, які відповідають за оцінювання.

7. Особи експертів-консультантів і персоналу підтримки із специфічними обов'язками для оцінювання.

8. Будь-яка додаткова інформація, яку буде зібрано протягом оцінювання, щоб підтримувати поліпшення процесу або визначати можливості процесу.

Будь-які змінення у входах оцінювання мають бути узгодженими із замовником та реєструватися. Замовник оцінювання повинен перевіряти, що експерт-консультант, який бере відповідальність за оцінювання і спостерігає за ним (компетентний експерт-консультант), має необхідну компетентність і уміння. Експерт-консультант має підтвердити вимоги замовника, щоб розпочати оцінювання.

Компетентний експерт-консультант має гарантувати, що оцінювання проводиться відповідно до певних вимог і на підставі релевантних настанов в інших частинах ISO/IEC 15504.

Експерт-консультант має гарантувати, що всіх інших учасників оцінювання проінформовано щодо мети, сфери й методу оцінювання. Завершивши оцінювання, компетентний експерт-консультант має підтвердити, що вимоги було виконано.

Оцінювання має проводитися відповідно до зареєстрованого процесу і містити такі дії.

1. *Планування.* План оцінювання треба розробити і зареєструвати, у ньому має бути визначено: необхідні входи; дії, які потрібно виконати для оцінювання; ресурси і план, який призначено для цих дій; вибір і певні

обов'язки експертів-консультантів і учасників оцінювання; критерії для перевірки виконання вимог.

2. *Збір даних.* Дані, потрібні для оцінювання процесів в межах області оцінювання, треба збирати систематичним і впорядкованим способом, застосовуючи такі принципи:

– мають бути явно ідентифікована стратегія і методи збору й аналізування даних;

– має бути встановлено відповідність між процесами організації, які визначено у сфері оцінювання через сумісну модель, що використовується для оцінювання, і процесами, які визначено в еталонній моделі;

– кожний процес, ідентифікований в області оцінювання, має бути оцінено на основі цільового доказу;

– цільовий доказ, зібраний для кожного атрибуту оціненого процесу, має бути достатнім для задоволення мети оцінювання і її сфери;

– мета – затверджений доказ, що базується на показниках оцінювання атрибутів процесу, які підтримують рішення експерта-консультанта, – має бути зареєстрованим, щоб забезпечити основу для перевірки оцінювання.

Ці дані мають містити:

а) методи і досліджені робочі продукти;

б) імена осіб, що забезпечують інформацію;

в) обговорення оцінювання.

3. *Перевірка правильності даних*. Зібрані дані має бути затверджено, що гарантує достатнє покриття області оцінювання.

4. *Ранжування процесу*. Оцінювання має бути призначено і затверджено для кожного атрибута процесу до найвищого рівня можливості, який визначено в області оцінювання для певної частини організації.

Набір рейтингів атрибута процесу необхідно зареєструвати як профіль процесу для певної частини організації.

Для забезпечення повторюваності оцінювання певний набір показників у сумісній моделі має використовуватися протягом процесу оцінювання.

5. *Результати оцінювання*, у тому числі виходи, треба зареєструвати і повідомити замовнику оцінювання.

Критерії для кваліфікації компетентного експерта-консультанта слід зареєструвати.

Експерти-консультанти, що беруть участь в оцінюванні, повинні мати доступ до відповідних матеріалів, де описано виконання оцінювання, і необхідну компетентність, використовувати будь-які прилади або інструментальні засоби, які вибрано для підтримки оцінювання.

Запис оцінювання має містити:

– дату оцінювання;

– вхід оцінювання (початкові дані);

– метод оцінювання, що використовується, та інструментальні засоби;

– імена експертів-консультантів, які провели оцінювання, у тому числі компетентного експерта-консультанта, що відповідає за оцінювання;

– набір профілів процесу (один профіль для кожного оцінюваного процесу);

– розташування записів об'єктивного доказу, що забезпечує оцінювання атрибута процесу в профілях процесу, і експерта-консультанта, що відповідає за оцінювання;

– будь-яка додаткова інформація, яку було зібрано протягом процесу оцінювання та ідентифіковано на його початку для поліпшення процесу або визначення можливості виконання процесу;

– кількість часу, витраченого експертом-консультантом, і рівень можливості виконання процесу, як визначено в еталонній моделі.

1.10. Показники якості програмного забезпечення

Дефекти програмного забезпечення, які внесено під час розроблення або модифікації та не виявлено тестуванням і верифікацією, можуть призвести до відмов і вплинути на безпеку реакторної установки. Тому аварії і катастрофи під час експлуатації ІКС можливі через дефіцит функціональної безпеки (ФБ), який обумовлено дефіцитом ресурсів (рис. 6).

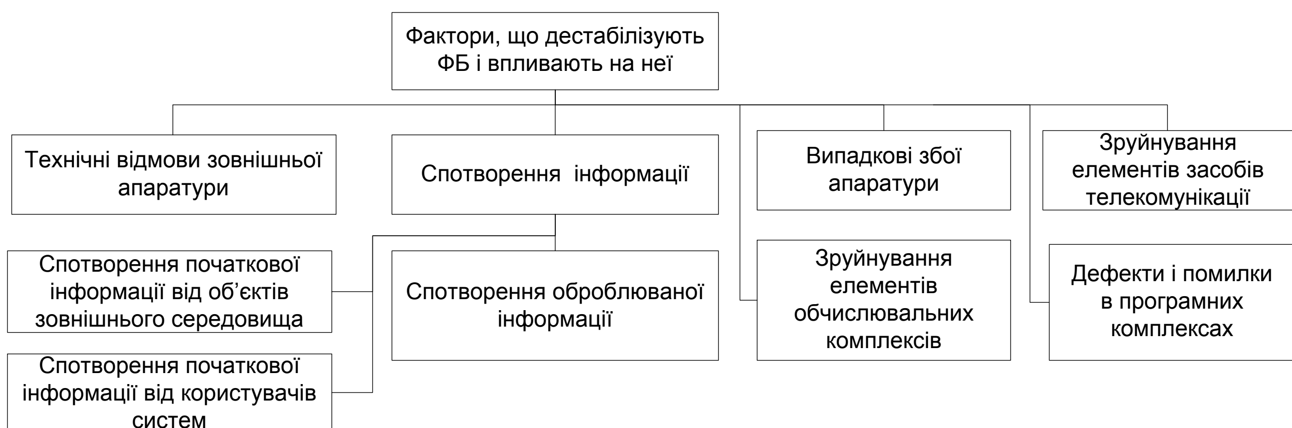

Рис. 6. Фактори дефіциту функціональної безпеки

Поняття ФБ і надійності є дуже близькими [4]. Під надійністю розуміються всі реалізації небезпечних відмов, а під ФБ – тільки ті відмови, що можуть призвести до великої або катастрофічної шкоди і впливають на безпеку системи й інформацію для споживачів. Імовірність таких подій дуже мала. Методи, фактори впливу і реальні значення показників

надійності ПЗ ІКС можуть використовуватися при оцінюванні ФБ. Тому способи оцінювання і випробувань ФБ базуються на методах визначення надійності функціонування ІКС.

Перші роботи, у яких було сформовано концепції і основні положення теорії надійності ПЗ, виникли понад двадцять років тому [5, 6]. У ГОСТ 13777–75 термін «надійність» визначено як властивість об'єкта виконувати задані функції, зберігаючи в часі значення встановлених експлуатаційних показників у межах, що відповідають заданим режимам і умовам експлуатації, технічного обслуговування, ремонту, зберігання і транспортування. Ступінь надійності характеризується ймовірністю безвідмовної роботи за певний час. За міжнародними стандартами сучасні ІКС повинні мати загальну інтенсивність відмов $\Lambda = 10^{-4}...10^{-3}$ год$^{-1}$, а ймовірність безвідмовної роботи для невідновлюваних засобів – 0,98 протягом семи років. Допустима інтенсивність аварій, наслідком яких може стати загибель великої кількості людей, $\Lambda = 10^{-9}...10^{-8}$ год$^{-1}$ [7]. Потрібний рівень ФБ досягається шляхом використання сучасних регламентованих технологічних процесів та інструментальних засобів підтримки ЖЦ ПЗ [8], за якими визначаються склад і процеси контролю виконання вимог до заданої ФБ і застосовуються методи системного аналізу, технологія проектування, розроблення і супроводу систем. Головним внеском у надійність є ДПЗ і якість процесів проектування і розроблення. Тому ПЗ має статус важливого об'єкта нормативного регулювання, що значною мірою визначає якість і безпеку ІКС і потребує вимірювання деяких метрик під час проектування і розроблення (рис. 7).

Складність прямого вимірювання досягнутого рівня ФБ обумовлює зосередження уваги на технологіях надання необхідних характеристик безпеки, які регламентуються стандартами програмної інженерії (державними – ДСТУ, Міжнародного електротехнічного комітету (МЕК) – IEC, Міжнародної організації зі стандартизації – ISO, Європейського космічного агентства – ESA) [9], які уточнюють поняття якості ПЗ шляхом введення сукупности характеристик, що впливають на його здатність задовольняти вимоги користувачів [10, 11]. Так, у стандарті ISO/IEC 9126 якість ПЗ (див. рис. 8) поділено на зовнішню і внутрішню та визначено шість її основних характеристик: функціональність, надійність, зручність у використанні, раціональність, супроводжуваність і переносність.

Функціональність – забезпує виконання функцій щодо заявлених потреб і має такі підхарактеристики: *функціональна повнота* – забезпечення відповідного набору функцій для заданих задач і цілей користувача; *правильність* – забезпечення правильних або допустимих результатів або дій з необхідним ступенем точності; *здатність до взаємодії* з іншими системами; *захищеність* – вільний доступ до інформації і даних тільки для авторизованих користувачів і систем та унеможливлення читання або модифікації для інших; *узгодженість*

функціональності – дотримання відповідних стандартів, угод, положень законів або схожих рекомендацій, що стосуються функціональності.

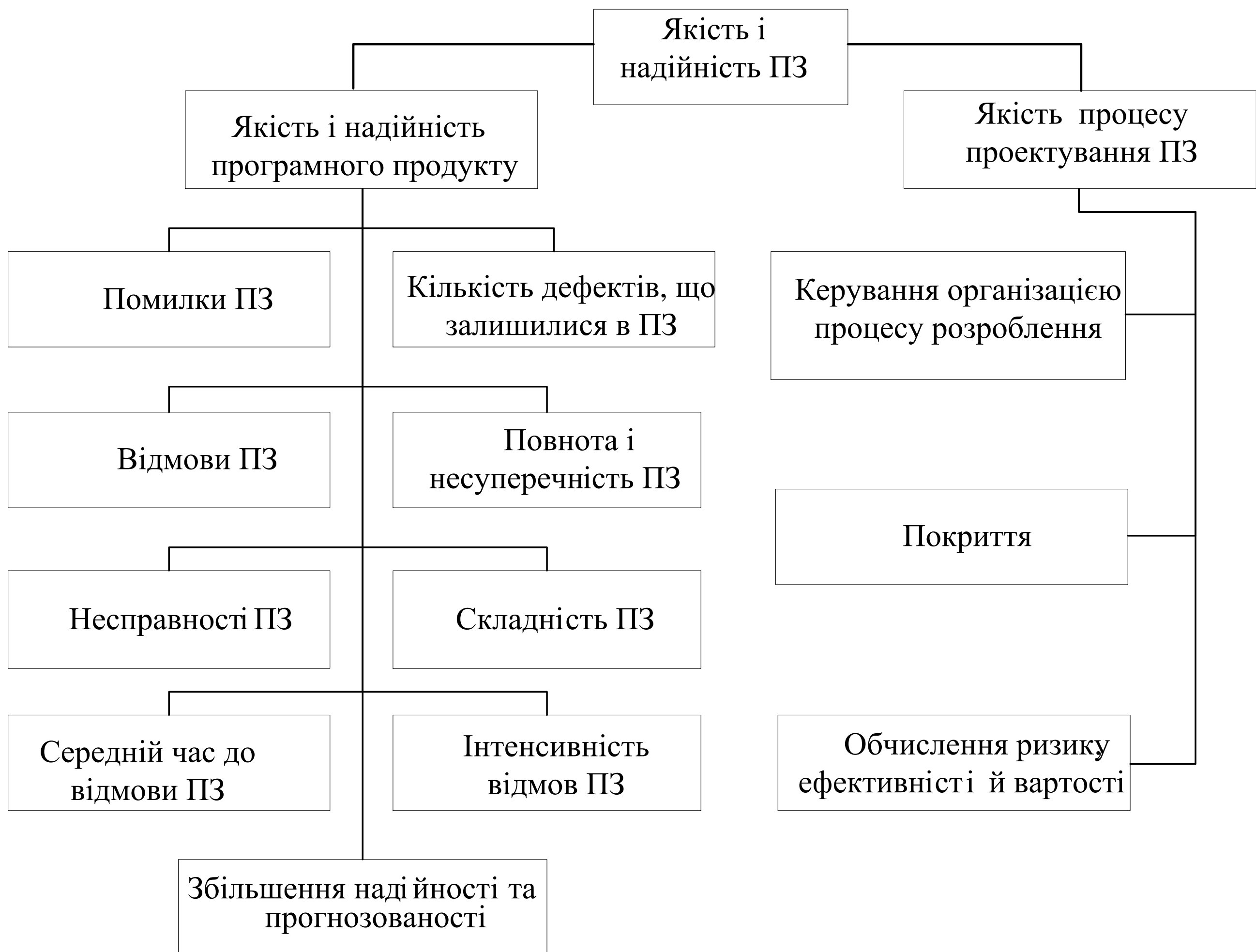

Рис. 7. Метрики, що потрібні для створення надійного ПЗ

Надійність – група властивостей, завдяки яким ПЗ може підтримувати заданий рівень працездатності у певних умовах. Надійність має такі підхарактеристики: *безвідмовність* – уникнення відмов (функціонування без відмов), які є результатом наявності дефектів; *стійкість до відхилень* – підтримка необхідного рівня працездатності у випадках активізації ДПЗ або порушення інтерфейсу; *відновлюваність* – відновлення заданого рівня працездатності, а також даних, які було зіпсовано під час виникнення відмови або після перезапуску; *узгодженість надійності* – дотримання відповідних стандартів, угод, положень законів або схожих рекомендацій, що стосуються надійності.

Зручність у використанні – зрозумілість, придатність до вивчення і привабливість для користувача має такі підхарактеристики: *зрозумілість* – забезпечення зрозумілості використання ПЗ для конкретних завдань і умов; *придатність до вивчення* – надання можливості вивчення ПЗ;

зручність інтерфейсу для користування – надання можливості керування і контролю за роботою; *привабливість* – привабливість для користувача (належить до графічного інтерфейсу); *узгодженість використання* – дотримання відповідних стандартів, угод, положень законів тощо.

Раціональність – забезпечення мінімально необхідної продуктивності з урахуванням залучених ресурсів має такі підхарактеристики: *часова раціональність* – забезпечення відповідного (припустимого) часу відгуку, оброблення та пропускної здатності при виконанні функцій у заданих умовах; *використовуваність ресурсів* – використання відповідної кількості й типу ресурсів при виконанні у заданих умовах; *узгодженість раціональності* – дотримання відповідних стандартів і рекомендацій, що стосуються ефективності.

Супроводжуваність – здатність ПЗ до модифікації (виправлення, поліпшення або адаптація ПЗ до змін середовища, вимог або функціональних специфікацій) – має такі підхарактеристики: *аналізовність* – здатність до діагностування дефектів і причин відмов ПЗ або до ідентифікації складових, які треба коригувати; *змінність* – можливість виконання заданої модифікації коду, структури, алгоритмів або програмної документації; *стабільність* – запобігання неочікуваним ефектам від модифікацій; *тестопридатність* – здатність ПЗ до валідації його змін; *узгодженість супроводжуваності* – дотримання відповідних стандартів або схожих рекомендацій, що стосуються супроводжуваності.

Мобільність – здатність ПЗ до змін організаційного, апаратного або програмного середовища – має такі підхарактеристики: *адаптовність* – здатність ПЗ до адаптації у різних середовищах без застосування дій або засобів, що різняться від передбачених; *налагоджуваність* – здатність ПЗ до інсталяції в заданому середовищі; *сумісність (безконфліктність)* – здатність до безконфліктної спільної роботи у загальному середовищі з іншим ПЗ, що використовує загальні ресурси; *взаємозамінність* – здатність ПЗ до використання іншого ПЗ замість заданого з тією самою метою й у тому самому середовищі; *узгодженість переносності* – здатність ПЗ до дотримуватися відповідних стандартів або схожих рекомендацій, що стосуються переносності.

Якість ПЗ у використанні визначається такими характеристиками (рис. 9): *ефективність* – здатність досягати накреслених цілей з точністю і повнотою в заданому контексті використання; *продуктивність* – здатність ПЗ давати змогу користувачеві витрачати відповідну кількість ресурсів; *безпека* – здатність ПЗ досягати прийнятних рівнів ризику в нанесенні шкоди людям, бізнесу, майну або навколишньому середовищу під час використання; *задоволеність* – здатність ПЗ задовольняти користувачів у заданому контексті використання.

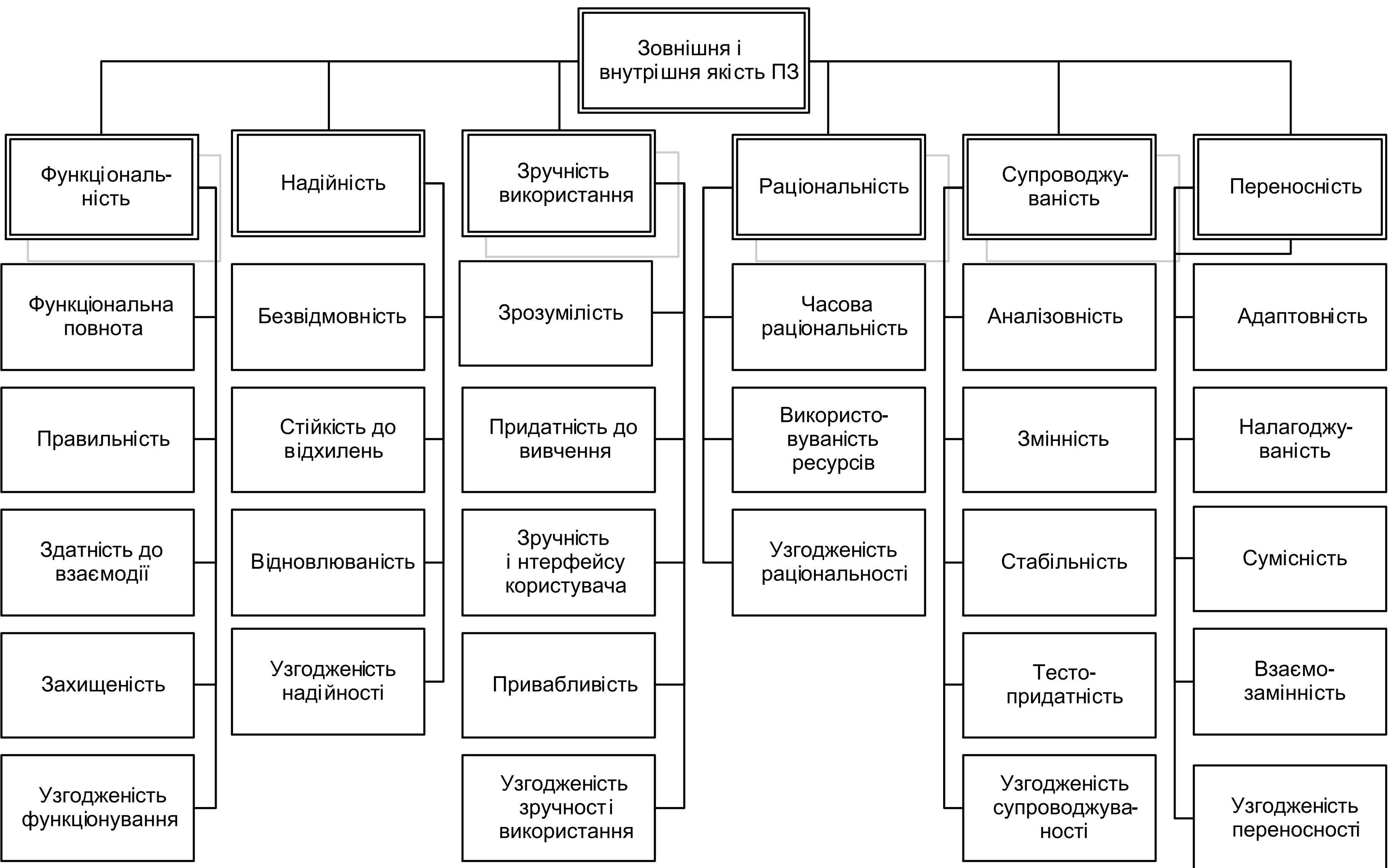

Рис. 8. Модель якості програмного забезпечення

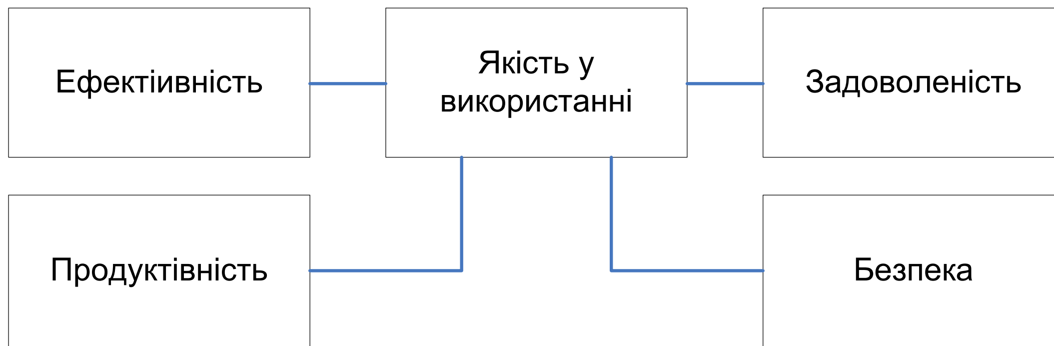

Рис. 9. Модель якості ПЗ у використанні

Необхідні умови досягнення високого рівня державних експертиз із оцінювання безпеки ПЗ ІКС, важливих для безпеки АЕС, – наявність адекватного нормативно-методичного забезпечення та широкомасштабне застосування ІЗ підтримки процесів експертизи, що відповідають сучасному динамічному розвитку стандартизації у сфері інформаційних технологій і програмної інженерії. Оцінювання характеристик якості ПЗ ІКС з урахуванням ризиків залишкових ДПЗ є актуальним завданням ризико-орієнтовних підходів до регулювання безпеки і сертифікації.

1.11. Моделі та метрики оцінювання якості ПЗ

Сучасна програмна індустрія за півстоліття накопичила багато моделей і метрик, що оцінюють окремі виробничі й експлуатаційні властивості ПЗ. Проте гонитва за їх універсальністю без урахування особливостей застосування, ігнорування етапів ЖЦ ПЗ і, нарешті, необґрунтоване їх використання в різнопланових процедурах ухвалення виробничих рішень істотно підірвало довіру до них розробників і користувачів ПЗ.

Проте аналіз технологічного досвіду лідерів виробництва ПЗ виявляє, наскільки дорого обходиться недосконалість ненаукового прогнозу трудовитрат і складності програм. Негнучкість контролю й керування розробленням свідчать про відсутність методичної підтримки. Це призводить урешті-решт до невідповідності вимогам як користувача, так і стандартів, що обумовлює необхідність повного перероблення програмного проекту. Ці обставини потребують ретельного відбору методик, моделей, методів оцінювання якості ПЗ, обліку обмежень їх придатності для різних ЖЦ і в його межах, установлення порядку їх сумісного використання, а також накопиченя й інтеграції різнорідної метричної інформації для ухвалення як своєчасних виробничих рішень, так і заключної сертифікації продукції.

Через корельовність рівня складності ПЗ з витратами на його розроблення й супровід найбільшу увагу надається метрикам, які дають змогу оцінити рівень складності та опосередковано визначити його надійність і якість.

Слід зазначити, що метрики складності програм, які буде розглянуто далі, базуються на аналізуванні програмних кодів і графів, що забезпечує єдиний підхід до автоматизації розрахунків.

Перш за все слід розглянути кількісні характеристики програмного коду. Найелементарнішою метрикою є кількість рядків коду (SLOC). Цю метрику було спочатку розроблено для оцінювання трудовитрат за проектом. Однак через те, що одну й ту саму функціональність можна розбити на декілька рядків або записати в один рядок, метрика стала майже непридатною з виникненням мов, у яких в один рядок можна записати понад одну команду. Тому розрізняють логічні й фізичні рядки коду. Логічні рядки коду – це кількість команд програми. Цей варіант так само має свої недоліки, оскільки дуже залежить як від мови, так і від стилю програмування.

Окрім SLOC до кількісних характеристик належать такі:

– кількість порожніх рядків;

– кількість коментарів;

– відсоток коментарів (відношення кількості рядків, що містять коментарі, до загальної кількості рядків);

– середня кількість рядків для функцій (класів, файлів);

– середня кількість рядків програмного коду функцій (класів, файлів);

– середня кількість рядків для модулів.

1.11.1. Метрики складності потоку керування

Ці метрики оперують або щільністю керувальних переходів усередині програм, або взаємозв'язками цих переходів. І в тому, і в іншому випадку стало традиційним подання моделей програм як керувального орієнтованого графа $G=(V,E)$, де V – вершини, що відповідають операторам, E – дуги, що відповідають переходам.

Метрика Маккейба

Уперше графічне подання моделей програм було запропоновано Маккейбом, що дало змогу кількісно оцінити цикломатичну складність графа програми (або, як її ще називають, цикломатине число Маккейба), що характеризує трудомісткість тестування програми.

Для визначення цикломатичного числа Маккейба застосовується формула

$$Z(G)=e-v+2p,$$

де e – кількість дуг орієнтованого графа G;

v – кількість вершин;

p – кількість компонентів зв'язності графа, які можна розглядати як кількість дуг і необхідно додати для перетворення графа в сильнозв'язний.

Сильнозв'язним називають граф, будь-які дві вершини якого є взаємно досяжними. Граф будь-якої коректної програми, тобто граф, що не має від точки входу недосяжних ділянок і «завислих» точок входу й виходу, перетворюється на сильнозв'язний граф шляхом замикання дугою вершини, що відповідає кінцю програми, на вершину, що відповідає точці входу в цю програму.

По суті $Z(G)$ визначає кількість лінійно незалежних контурів у сильнозв'язному графі. Інакше кажучи, цикломатичне число Маккейба – це необхідна кількість проходів для покриття всіх контурів сильнозв'язного графа або кількість тестових прогонів програми, які необхідні для вичерпного тестування за критерієм покриття кожної програмної гілки.

Цикломатичне число залежить тільки від кількості предикатів, складність яких при цьому не враховується. Наприклад, є два умовних оператори:

```
If (x>0) x=y; else;            if (x>0 && f==1|| x==0 && f==0) x=y; else;
```

Керувальні графи кожного з операторів є ізоморфними, а їхні цикломатичні числа – еквівалентними, що не відображає складності предикатів, але є дуже важливим при оцінюванні програм.

Метрика Майєрса

Виходячи з того, що цикломатичне число Маккейба не відображає складності предикатів, Г. Майєрс запропонував подання метрики складності програм у вигляді інтервалу $\left[Z(G), Z(G)+h\right]$. Для простого предиката $h=0$, а для n-місцевих предикатів $h=n-1$. Таким чином, оператору *If (x>0) x=y; else;* відповідатиме інтервал метрики складності $[2,\ \ 2]$, а оператору *if (x>0 && f==1|| x==0 && f==0) x=y; else;* – інтервал $[2,\ \ 6]$, і запропонована метрика дасть змогу розрізняти програми, які подано однаковими графами.

Метрика підрахунку точок перетину

Розглянемо метрику складності програм, що отримала назву «Підрахунок точок перетину», авторами якої є М. Вудвард, М. Хенел і Д. Хидлей. Метрику орієнтовано на аналіз програм, при створенні яких

використовувалося неструктурне кодування на таких мовах, як асемблер, Фортран тощо.

У графі програми, де кожному оператору відповідає вершина, тобто не виключено лінійні ділянки, при переданні керування від вершини A до вершини B номер оператора A дорівнює $min(A,B)$, а номер оператора B – $max(A,B)$. Точка перетину дуг виникає у таких випадках:

$$min(A,B) < min(P,Q) < max(A,B) \& max(P,Q) > max(A,B) /$$
$$min(A,B) < max(P,Q) < max(A,B) \& min(P,Q) < min(A,B) .$$

Іншими словами, точка перетину дуг виникає у разі виходу керування за межі пари вершин (A,B).

Кількість точок перетину дуг графа програми дає характеристику неструктурованості програми.

Метрика Джилба

Однією з найбільш простих є метрика Т. Джілба, у якій логічна складність програми визначається як насиченість програми виразами типу *IF–THEN–ELSE*. Це, як показує практика, досить ефективне оцінювання складності програм. При цьому вводяться дві характеристики: CL – абсолютна складність програми, що характеризується кількістю операторів умови; cl – відносна складність програми, що характеризується насиченістю програми операторами умови, тобто cl визначається як відношення CL до загальної кількості операторів.

Важливе значення має характеристика максимального рівня вкладеності умовних операторів і операторів циклів CLI, що дає змогу не тільки уточнити аналіз по операторах типу *IF–THEN–ELSE*, але й успішно застосувати метрику Джилба до аналізу складності циклічних конструкцій.

Метрика граничних значень

Ця метрика дає змогу оцінити складність програми методом граничних значень.

Уведемо декілька додаткових понять, пов'язаних з графом програми.

Нехай $G=(V,E)$ – орієнтований граф програми з єдиною початковою і єдиною кінцевою вершинами. У цьому графі кількість дуг, що входять у вершину N_-, називають від’ємним ступенем вершини, а кількість дуг, що виходять з вершини N_+– додатним ступенем вершини. Тоді вершини графа за їх загальним ступенем можна розбити на дві групи: вершини, у яких $N_+ - N_- \le 1$; вершини, у яких $N_+ - N_- \ge 2$.

Вершини першої групи назвемо приймальними вершинами, другої групи – вершинами відбору.

Для оцінювання за методом граничних значень необхідно розбити граф G на максимальну кількість підграфів G', що задовольняють таким умовам: вхід в підграф здійснюється тільки через вершину відбору; кожний підграф містить вершину (що надалі будемо звати межею підграфа), у яку можна потрапити з будь-якої іншої вершини підграфа. Наприклад, вершина відбору, що сполучається сама з собою дугою-петлею, утворює підграф.

Кількість вершин, які утворюють такий підграф, дорівнює скоригованій складності вершини відбору. Кожна приймальна вершина має скориговану складність, що дорівнює одиниці, окрім кінцевої вершини, скоригована складність якої дорівнює нулю. Скориговані складнощі всіх вершин графа G підсумовуються, утворюючи абсолютну граничну складність програми. Після цього визначається відносна гранична складність програми:

$$S_0 = 1 - \frac{(v-1)}{S_a},$$

де S_a – абсолютна гранична складність програми; v – загальна кількість вершин графа програми.

1.11.2. Метрики складності потоку даних

Ця група метрик формує кількісні характеристики про використання, конфігурацію і розміщення програмних даних.

Метрика використання глобальних змінних

Розглянемо метрику, що визначає складність програм за кількістю використання глобальних змінних.

Пара модуль–глобальна змінна позначається як (p,r), де p – модуль, що має доступ до глобальної змінної r. Залежно від наявності в програмі реального звернення до змінної r формуються фактичні й можливі пари модуль-глобальна змінна. Можливе звернення до r з допомогою p показує, що область існування r містить p.

Характеристика Aup визначає, скільки разів модулі Up дійсно використовували глобальні змінні, а число Pup – скільки разів вони могли б отримати доступ.

Відношення кількості фактичних звернень до можливих визначається таким чином:

$$Rup = \frac{Aup}{Pup}$$

За цією формулою можна знайти наближення ймовірності використання довільного модуля до довільної глобальної змінної. Очевидно, чим більше ця ймовірність, тим більше ймовірність «несанкціонованого» змінення якої-небудь змінної, що може істотно ускладнити роботи, пов'язані з модифікацією програми.

Метрика Спена

Ця метрика грунтується на локалізації звернень до даних усередині кожної програмної секції. Спен – це кількість програмних конструкцій що мають певний ідентифікатор, між першим і останнім використанням в тексті програми. Отже, ідентифікатор, який було використано *n* разів, має значення метрики Спена, що дорівнює $n-1$. Якщо програма має велике значення метрики Спена, то її тестування й налагодження буде занадто складним.

Метрика Чепіна

Суть методу полягає в оцінюванні інформаційної міцності окремо взятого програмного модуля шляхом аналізування характеру використання вхідних/вихідних змінних.

Усі змінні, що належать до списку вхідних/вихідних, поділяються на такі функціональні групи:

1) P – змінні, що вводяться, для розрахунків і забезпечення вихідних результатів; прикладом може бути програмна змінна, що містить рядок програмного коду і тільки аналізується і не модифікується лексичним аналізатором;

2) M – змінні, що модифікуються або створюються програмою;

3) C – змінні, що беруть участь в керуванні роботою програмного модуля (керувальні змінні);

4) T – змінні, що не використовуються в програмі.

Оскільки кожна змінна може виконувати одночасно декілька функцій, необхідно ураховувати її в кожній відповідній функціональній групі.

Далі вводиться значення метрики Чепіна :

$$Q = a_1P + a_2M + a_3C + a_4T\,,$$

де a_1, a_2, a_3, a_4 – вагові коефіцієнти.

Вагові коефіцієнти використано для відображення різного впливу на складність програми кожної функціональної групи. Найбільша вага дорівнює трьом, її має функціональна група C, оскільки вона впливає на

потік керування програми. Вагові коефіцієнти решти груп розподіляються таким чином: $a_1 = 1, \quad a_2 = 2, \quad a_4 = 0{,}5$.

З урахуванням вагових коефіцієнтів

$$Q = P + 2M + 3C + 0{,}5T.$$

Ваговий коефіцієнт групи T не дорівнює нулю, оскільки «паразитні» змінні не збільшують складності потоку даних програми, але іноді утруднюють її розуміння.

1.11.3. Метрики стилістики і зрозумілості програми

Найбільш простою метрикою стилістики і зрозумілості програм є оцінювання рівня коментованості програми:

$$F = \frac{N_{com}}{N_{total}},$$

де N_{com} – кількість коментарів, а N_{total} – загальна кількість рядків або операторів програми.

Таким чином, метрика F характеризує насиченість програми коментарями.

Виходячи з практичного досвіду, прийнято вважати необхідним, щоб $F \geq 0{,}1$, тобто на кожні десять рядків програми має бути щонайменше один коментар. Як випливає із досліджень, коментарі розподіляються по тексту програми нерівномірно: на початку програми їх забагато, а в середині або в кінці – замало. Це пояснюється тим, що на початку програми зазвичай розташовано оператори опису ідентифікаторів, що потребують більш «щільного» коментування. Крім того, на початку програми також розташовано загальні відомості про виконавця, характер, функціональне призначення програми тощо. Така насиченість компенсує недолік коментарів в тілі програми, тому метрика F недостатньо точно відображає коментованість функціональної частини тексту програми.

Більш вдалим є варіант, коли вся програма розбивається на n рівних сегментів і для кожного з них метрика визначається за формулою

$$F_i = sign\left(\frac{N_{com}}{N_l} - 0{,}1\right),$$

при цьому

$$F = \sum_{i=1}^{n} F_i.$$

Рівень коментованості програми вважається нормальним за умови $F = n$, інакше який-небудь фрагмент програми доповнюється коментарями до номінального рівня.

1.11.4. Метрика Холстеда

Основу метрики Холстеда [12] становлять чотири характеристики програми, які можна отримати безпосередньо шляхом аналізування програмних кодів:

n_1 – кількість унікальних операторів програми, що складається із символів-роздільників, імен процедур, функцій і знаків операцій (словника операторів);

n_2 – кількість унікальних операндів програми (словник операндів);

N_1 – загальна кількість операторів у програмі;

N_2 – загальна кількість операторів у програмі.

Завдяки отриманим кількісним характеристика підраховуються:

– словник програми

$$n = n_1 + n_2\,;$$

– довжина програми

$$N = N_1 + N_2\,;$$

– обсяг програми в бітах

$$V = N\,log_2(n).$$

Під бітом розуміється логічна одиниця інформації – символ, оператор, операнд.

Холстед уводить n^* – теоретичний словник програми, тобто словарний запас, необхідний для написання програми, з урахуванням того, що необхідну функцію вже реалізовано і, отже, програма зводиться до виклику цієї функції. Наприклад, виклик процедури виділення простого числа міг би мати такий вигляд:

CALL SIMPLE (X,Y),

де Y – масив чисел, що містить необхідне просте число X.

Теоретичний словник буде складатися з такого:

n_1^* : {CALL, SIMPLE (...)}

$n_1^* = 2$; n_2^* : {X, У}

$n_2^* = 2$

Довжина словника визначається за формулою

$$n^* = n_1^* + n_2^*$$

і буде дорівнювати чотирьом.

Використовуючи n^*, Холстед уводить формулу

$$V^* = n^* log_2(n^*)$$

для оцінювання потенційного обсягу програми, що відповідає максимально компактному її тексту для реалізації алгоритму.

Наступні характеристики є продовженням метрики Холстеда.

1. Оцінювання теоретичної довжини програми $\hat{N}$ визначається як

$$\hat{N} = n_1 log_2(n_1) + n_2 log_2(n_2),$$

де n_1 – кількість елементів словника операторів; n_2 – кількість елементів словника операндів програми.

Уводячи це оцінювання, Холстед виходить із основних концепцій теорії інформації, за аналогією з якими частота використання операторів і операндів в програмі є пропорційною двійковому логарифму кількості їх типів. Таким чином, $\hat{N}$ є правильним для потенційно коректних програм, вільних від надмірності або недосконалості (стилістичних помилок). Потенційно помилковими можна вважати такі ситуації:

а) подальша операція знищує результати попередньої без їх використання;

б) у наявності є тотожні вирази, що розв’язують однакові задачі;

в) одній і тій самій змінній призначаються різні імена тощо.

Такі ситуації приводять до змінення N без змінення n.

Для стилістично коректних програм відхилення в оцінюванні теоретичної довжини $\hat{N}$ від реальної N не перевищує 10 %, тому $\hat{N}$ слід використовувати як еталонне значення довжини програми із словником n. Довжина коректно складеної програми N, тобто програми, яка є вільною від надмірності і має словник n, не повинна відхилятися від теоретичної довжини програми $\hat{N}$ більш ніж на 10 %. Таким чином, вимірюючи n_1, n_2, N_1, N_2 і порівнюючі значення N і $\hat{N}$ для деякої програми, при більш ніж 10-відсотковому відхиленні можна говорити про наявність у програмі стилістичних помилок, тобто недосконалостей.

2. Характеристикою, що належить до метрик коректності програм, є рівень якості програмування

$$L = \frac{V^*}{V},$$

де V^* і V визначаються за наведеними вище формулами.

Ця характеристика ґрунтується на припущенні, що при зниженні стилістичної якості зменшується змістовне навантаження на кожний компонент програми і, як наслідок, розширюється обсяг реалізації алгоритму. Ураховуючи це, можна оцінити якість програмування на основі ступеня розширення тексту щодо потенційного обсягу V^*. Очевидно, що для ідеальної програми $L = 1$, а для реальної – завжди $L < 1$.

3. Іноді доцільно визначити рівень програми, не вдаючись до оцінювання її теоретичного обсягу, оскільки список параметрів програми

часто залежить від реалізації і його можна штучно поділити. Холстед пропонує апроксимувати це оцінювання виразом, що містить тільки фактичні параметри, тобто параметри реальної програми:

$$\hat{L} = \frac{2n_2}{n_1 N_2}.$$

4. Маючи в своєму розпорядженні характеристику $\hat{L}$, Холстед увів характеристику I, яку розглядав як інтелектуальний зміст конкретного алгоритму, що є інваріантним відносно мов реалізації:

$$I = \hat{L}V,$$

але після перетворень маємо $I = V^*$.

Еквівалентність I та V^* свідчить про те, що використовується характеристика інформативності програми.

Уведення характеристики I дає змогу визначити інтелектуальні витрати на створення програми. Процес створення програми умовно можна подати як ряд операцій:

1) осмислення пропозиції відомого алгоритму;

2) запис алгоритму в термінах мови програмування, тобто пошук в словнику мови відповідної інструкції, її смислове наповнення й запис.

Використовуючи цю формалізацію, можна сказати, що написання програми за наперед відомим алгоритмом є $\hat{N}$-кратна вибірка операторів і операндів із словника програми n, причому кількість порівнянь (за аналогією з алгоритмами сортування) становить $log_2(n)$.

Якщо врахувати, що кожна вибірка-порівняння містить, у свою чергу, деякі елементарні рішення, то можна поставити у відповідність змістовному навантаженню кожної конструкції програми складність і кількість цих елементарних рішень. Кількісно це можна характеризувати з допомогою характеристики L, оскільки L^{-1} має сенс розглядати як середній коефіцієнт складності, що впливає на швидкість вибірки цієї програми. Тоді обсяг необхідних інтелектуальних зусиль з написання програми можна підрахувати так:

$$E = \hat{N} log_2\left(\frac{n}{L}\right).$$

Таким чином, E є кількістю необхідних елементарних рішень при написанні програми.

Проте слід зазначити, що E адекватно характеризує лише початкові зусилля з написання програм, оскільки при підрахунку інтелектуальних зусиль не враховуються налагоджувальні роботи, які потребують інтелектуальних витрат іншого характеру.

Суть інтерпретації цієї характеристики полягає в оцінюванні витрат не на розроблення програми, а на сприйняття готової програми. При цьому замість теоретичної довжини програми $\hat{N}$ використовується її реальна довжина:

$$E' = N\,log_2\left(\frac{n}{L}\right).$$

Цю характеристику уведено виходячи з припущення, що інтелектуальні зусилля з написання й сприйняття програми є дуже близькими за своєю природою. Проте якщо при написанні програми стилістичні похибки в тексті майже не відображаються на витратах інтелектуальних ресурсів, то при спробі зрозуміти таку програму їх наявність може призвести до серйозних ускладнень. Це досить добре узгоджується з висновками щодо взаємозв'язку N і $\hat{N}$.

Перетворюючи формулу з урахуванням попередніх позначень, маємо:

$$E = \frac{V^2}{V^*},$$

що наочно ілюструє доцільність розбиття програм на окремі модулі, оскільки інтелектуальні витрати виявляються пропорційними квадрату обсягу програми, який завжди більше за суму квадратів обсягів окремих її модулів.

1.11.5. Об'єктно-орієнтовані метрики

Існує великий клас об'єктно-орієнтованих метрик, до якого належать метрики Мартіна, Чидамбера й Кемерера.

Перш ніж розглядати метрики Мартіна, необхідно ввести поняття категорії класів. Клас можна досить рідко повторно використати ізольовано від інших класів. Майже для кожного класу існує група класів, з якими він працює в кооперації і від яких не може бути легко відокремленим. Для повторного їх використання необхідно залучати всю групу класів (категорію класів). Для її існування є такі умови:

– класи в межах своєї категорії закриті від будь-яких спроб змінити їх всі разом; це означає, що якщо потрібно змінити один клас, то усі класи в цій категорії з великою ймовірністю також необхідно змінити; якщо будь-який з класів є відкритим для деякого змінення, то їх усі відкрито для такого змінення;

– класи в категорії повторно використовуються тільки разом; таким чином, якщо робиться будь-яка спроба повторного використання одного класу в категорії, то усі інші мають повторно використовуватися разом з ним;

– класи в категорії реалізують загальну функцію або досягають деякої загальної мети.

Відповідальність, незалежність і стабільність категорії можна визначити шляхом підрахунку залежності, яка визначається з допомогою трьох метрик:

– C_A – кількість класів поза цією категорією, які залежать від класів усередині цієї категорії;

– C_E – кількість класів усередині цієї категорії, які залежать від класів поза цією категорією;

– I – нестабільність, $I = \frac{C_E}{\left(C_E + C_A\right)}$. Ця метрика має діапазон значень [0, 1]; $I = 0$ – відповідає максимально стабільній категорії, $I = 1$ – максимально нестабільній.

Можна визначати метрику абстрактності категорії (якщо категорія є абстрактною, то вона досить гнучка і її можна легко розширити) таким чином:

$$A = \frac{N_A}{N_{All}},$$

де N_A – кількість абстрактних класів у категорії;

N_{All} – загальна кількість класів у категорії.

Значення цієї метрики належать інтервалу [0, 1]. Якщо абстрактність дорівнює нулю, то категорія є конкретною. Якщо абстрактність дорівнює одиниці, то категорія є повністю абстрактною.

На основі метрик Мартіна можна побудувати графік, на якому відобразити залежність між абстрактністю і нестабільністю. Якщо на ньому побудувати пряму, яку задано формулою $I + A = 1$, то на цій прямій будуть лежати категорії, що мають найкращу збалансованість між абстрактністю і нестабільністю. Ця пряма має назву головної послідовності.

Далі можна ввести ще дві метрики:

– відстань до головної послідовності $D = \left|\frac{\left(A + I - 1\right)}{\sqrt{2}}\right|$;

– нормалізірована відстань до головної послідовності $D_n = \left|A + I - 2\right|$.

Майже для всіх категорій правильним є те, що чим ближче до головної послідовності вони знаходяться, тим краще.

Метрики Чидамбера і Кемерера ґрунтуються на аналізуванні методів класу, дерева спадкування тощо.

WMC (Weighted methods per class) – сумарна складність всіх методів класу: $W_{mc} = \sum_{i=1}^{n} C_i$, де C_i – складність *i*-го методу, яку обчислено за однією із метрик (наприклад, Холстеда); якщо у всіх методів складність однакова, то $W_{mc} = n$.

DIT (Depth Inheritance tree) – глибина дерева спадкування (найбільший шлях за ієрархією класів до певного класу від класу-предка): чим більше глибина, тим краще, оскільки зі збільшенням глибини збільшується абстракція даних, зменшується насиченість класу методами, проте при досить великій глибині сильно збільшується складність розуміння і написання програми.

NOC (Number children) – кількість нащадків (безпосередніх); чим більше нащадків, тим більша абстракція даних.

CBO (Coupling between object classes) – зчеплення між класами, що виявляє кількість класів, з якими зв'язаний початковий клас. Для цієї метрики правильними є всі твердження, уведені раніше для зв'язності модулів, тобто при високому CBO зменшується абстракція даних і утруднює повторне використання класу.

RFC (Response for class) – $RFC = |RS|$, де $|RS|$ – множина методів, які потенційно можна викликати методом класу у відповідь на дані, які отримано об’єктом класу, тобто $RS = (\{M\}\{R_i\}), i = 1...n$, де M – усі можливі методи класу; R_i – усі можливі методи, які можуть бути викликані *i*-м класом. RFC – потужність цієї множини. Чим більша RFC, тим складніші тестування і відлагодження.

LCOM (Lack cohesion in Methods) – недолік зчеплення методів. Для визначення цього параметра розглянемо клас з n методами $M_1,\ M_2,\ \ldots,\ M_n$, тоді $\{I_1\},\ \{I_2\},\ \ldots,\ \{I_n\}$ – множина змінних, що використовуються у цих методах. Тепер візьмемо, що P – множина пар методів, які не мають загальних змінних, Q – множина пар методів, які мають загальні змінні, тоді $LCOM = |P| + |Q|$. Недолік зчеплення свідчить, що клас можна розбити на кілька інших класів або підкласів, так що для підвищення інкапсуляції даних і зменшення складності класів і методів краще підвищувати зчеплення.

1.11.6. Метрики надійності

Метрики надійності – це метрики, які є близькими до кількісних і базуються на кількості помилок і дефектів в програмі. Немає сенсу розглядати особливості кожної з них, досить буде їх просто перелічити:

кількість структурних змін, проведених з моменту попередньої перевірки, кількість помилок, виявлених під час перегляду коду, кількість помилок, виявлених під час тестування програми, і кількість необхідних структурних змін, необхідних для коректної роботи програми. Для великих проектів зазвичай розглядають ці показники відносно тисячі рядків коду, тобто середню кількість дефектів на тисячу рядків коду.

Хотілося б зазначити, що жодної універсальної метрики не існує. Будь-які метричні характеристики програми мають контролюватися залежно одна від одної або від конкретної задачі. Крім того, можна застосовувати гібридні метрики, проте вони також залежать від більш простих метрик і не можуть бути універсальними. Іншими словами, будь-яка метрика – це лише показник, який дуже залежить від мови і стилю програмування, тому жодну міру не можна підносити до абсолюту й ухвалювати які-небудь рішення, ґрунтуючись тільки на ній.

1.12. Методи гарантування якості програмного забезпечення

Принципи і методи гарантування надійності ПЗ спрямовано на недопущення, виявлення, виправлення й забезпечення стійкості до дефектів.

Головний недолік тестування – неможливість гарантувати відсутність ДПЗ. Це забезпечує верифікація [13], що виконується після тестування як доказ коректності всіх допустимих маршрутів з допомогою формальних методів.

Згідно з конструктивним підходом уважається, що бездефектне ПЗ будується з допомогою певних організаційних методів, а його коректність доводиться математично у статичному режимі (без реального виконання) з допомогою математичних теорем, що визначають функціональність програми. Головна складність полягає у розробленні стверджень, які треба доводити, та значній складності додаткових обчислень, що часто перевищує обсяг самої програми. Перевагою підходу є можливість виявленя ДПЗ, які не було виявлено під час тестування.

На сьогодні найбільш розвинутим є метод доказового програмування, необхідність розроблення якого пов’язана з тим, що точне знання, відтворене в умовах задачі й опису методу її розв’язання, є тільки довідковою інформацією. Це є головною причиною неефективністі програмування, що водночас стимулює створення його наукового обґрунтування. Пошуки систематичних методів доказового програмування розпочалися разом із винаходом ЕОМ. Зазн.ачимо внесок таких вчених, як Ф.Л. Бауер, Р. Берстал, Е. Дейкстра, Дж. Маккарті, М. Ніва, В. Турський, Р. Флойд, А. Хоар, В.М. Глушков, С.С. Лавров, А.А. Летичевський, А.А. Ляпунов, Е.Х. Тиугу. Існує три види доказового програмування, в основі яких – формальні математичні специфікації:

– синтезувальне програмування, у якому використовується математична специфікація – сукупність логічних формул, що має два різновиди: *логічну*, у якій програма є конструктивним доведенням теореми, яку побудовано на основі специфікації, і *трансформаційну*, у якій специфікація розглядається як рівняння і символічно перетворюється на програму. З формального погляду умови задачі записуються як сукупність логічних формул, до яких належать символи відомих і невідомих величин, операцій і функцій. Ця сукупність отримала назву *специфікації задачі*. У подальшому програми за специфікаціями синтезують за *логічним* і *аналітичним* підходами. У логічному підході специфікація трактується як теорема, а синтез програми – як пошук доведеня в певному конструктивному логічному обчисленні. В аналітичному підході специфікація розуміється як рівняння стосовно програми, що символьно перетворюється. При цьому символ невідомої програми перетворюється на систему невідомих, які, у свою чергу, замінюються або іншими невідомими, або конкретними програмними конструкціями. Коректність виконання символічних перетворень формально перевіряється на кожному кроці;

– формальне складальне програмування, у якому використовується специфікація як композиція вже відомих фрагментів;

– формальне конкретизувальне програмування, у якому використовуються змішані обчислення й конкретизації за анотаціями.

Існують інші підходи до верифікації, наприклад: *дедуктивний*, у якому використовуються автоматичне доведення теорем, мультимножини й графи, різноманітні спеціалізовані алгебри, завдяки чому ПЗ описується формально та математично доводиться наявність тих або інших його властивостей [14]; *модельний,* за яким створюється автоматна модель системи, а системні вимоги перевіряються для кожного зі станів автомата.

Використання математичних методів для верифікації відповідності ПЗ специфікації поки не набуло широкого практичного застосування. Однак формальне доведення коректності окремих підсистем вже застосовується. Наприклад, формальна верифікація за методологією «B-Method» [15] ПЗ ІКС паризького метро обсягом 100 тисяч рядків потребувала доведення близько 28 тисяч лем.

Один із методів, що забезпечує легке виявлення й ідентифікацію ДПЗ, – захисне програмування [16]. Його принципи: загальна недовіра (для кожного модуля вхідні дані мають аналізуватися в припущенні, що вони можуть бути помилковими); негайне виявлення (кожну помилку необхідно виявляти якомога раніше, що спрощує з’ясування її причин); ізолювання помилки (помилки в одному модулі мають бути ізольовані так, щоб не допустити їхнього впливу на інші модулі).

Інші методи направлено на боротьбу зі складністю ПЗ, у більшості з них усунено недоліки структурного проектування. Ці методи можна поділити на три основні групи:

– метод структурного проектування зверху-вниз, що базується на топології традиційних мов високого рівня, розглянуто у; хоча більшість існуючих продуктів написано відповідно до нього, структурний підхід обмежено приблизно 100 000 рядками;

– метод потоків даних, за яким ПЗ – перетворювач вхідних потоків на вихідні – застосовувався при розробленні систем з прямими зв'язками між вхідними й вихідними потоками, що не потребують швидкодії;

– метод об’єктно-орієнтованого проектування, за яким ПЗ – сукупність взаємодійних об'єктів, що належать до певних класів. Основою парадигми є введення абстрактних типів даних – результатів аналізу предметної області, що теоретично дає змогу зменшити складність програмного проекту і, уникнувши семантичного розриву з реальним світом, відобразити всю складність об'єкта керування й частини реального світу. Якість об’єктно-орієнтованого ПЗ вимірюється спеціальними метриками: кількістю класів, методів у класах тощо.

Один із шляхів забезпечення якості ПЗ ґрунтується на використанні принципу різноманітності, або багатоверсійності [17]. Багатоверсійні технології розроблення пов'язані з внесенням різних видів версійної надмірності на різних етапах ЖЦ ПЗ: залученням різних суб'єктів для розроблення, тестування й верифікації; створенням різних версій ПП на різних етапах розроблення. Конкретні типи ДПЗ, що вносяться до продукту на різних етапах, визначають необхідні для їх усунення типи версійної надмірності. Диверсифікація на сьогодні – методологія забезпечення найвищої якості ПЗ, що потребує багаторазового підвищення витрат через необхідність координації паралельного розроблення кількох версій.

Розроблення і впровадження в експертизу і сертифікацію автоматизованих засобів, що реалізують диверсні, відмінні від використовуваних розробниками, методи для об’єктивного оцінювання ПЗ, дадуть змогу збільшити повноту й достовірність експертних оцінок надійності. Тому актуальним є пошук нових, диверсних принципів забезпечення якості програмного забезпечення.

2. ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Місце верифікації у життєвому циклі

Верифікація – процес доведення відповідності програмних засобів і їх компонентів вимогам, визначеним протягом ЖЦ.

Визначення вимог – перший і найважливіший етап розроблення ПЗ. За статистикою, близько 65 % помилок програмістів обумовлено неправильним розумінням вимог замовника. Тому між алгоритмічними мовами, з допомогою яких створюється ПЗ, і природною мовою, на якій формуються вимоги до ПЗ, існує проміжний рівень, на якому з допомогою мови специфікацій формально визначаються вимоги як відображення аргументів на значення очікуваної функції.

Праві гілки «розщепленої» V-подібної моделі, зображеної на рис. 10, відповідають етапам ЖЦ у цілому та відображають еволюцію і види випробувань. Кожному варіанту правої гілки відповідає обсяг випробувань, визначений профілем регулювальних вимог до якості й безпеки ПЗ, що гарантує допустимий рівень ризику.

Головна ідея верифікації ПЗ – формальне доведення відповідності між програмним кодом (мовою програмування) і специфікацією (мовою специфікацій). Мови специфікацій конкретизують вимоги та полегшують програмування і супровід. Існує два види специфікацій: зовнішні (мови специфікацій задач) і внутрішні (мови специфікацій властивостей). Мови специфікацій поділяються на мови визначення вимог і мови функціональних специфікацій. Мови визначення вимог – це напівформальні мови, наприклад блок-схеми, які використовуються для визначення даних, їхніх потоків і алгоритмів. Мови функціональних специфікацій – формальні мови повного визначення програмних функцій, які побудовано на математичній основі. Функціональні специфікації є не тільки точним формалізованим завданням на програмування, але й зручним засобом для тестування і верифікації задачі до її програмування.

Основна мета верифікації полягає в підтвердженні віповідності ПЗ вимогам. Додатковою метою є виявлення і реєстрація програмних дефектів і помилок, внесених під час розроблення або модифікації ПЗ.

Верифікація є невід'ємною частиною робіт при колективному розробленні ПЗ. При цьому до задач верифікації додається контроль результатів одних розробників при передачі їх іншим.

Для підвищення ефективності використання людських ресурсів при розробленні, верифікацію близько інтегровано з процесами ЖЦ, що забезпечують проектування, розроблення й супровід ПЗ.

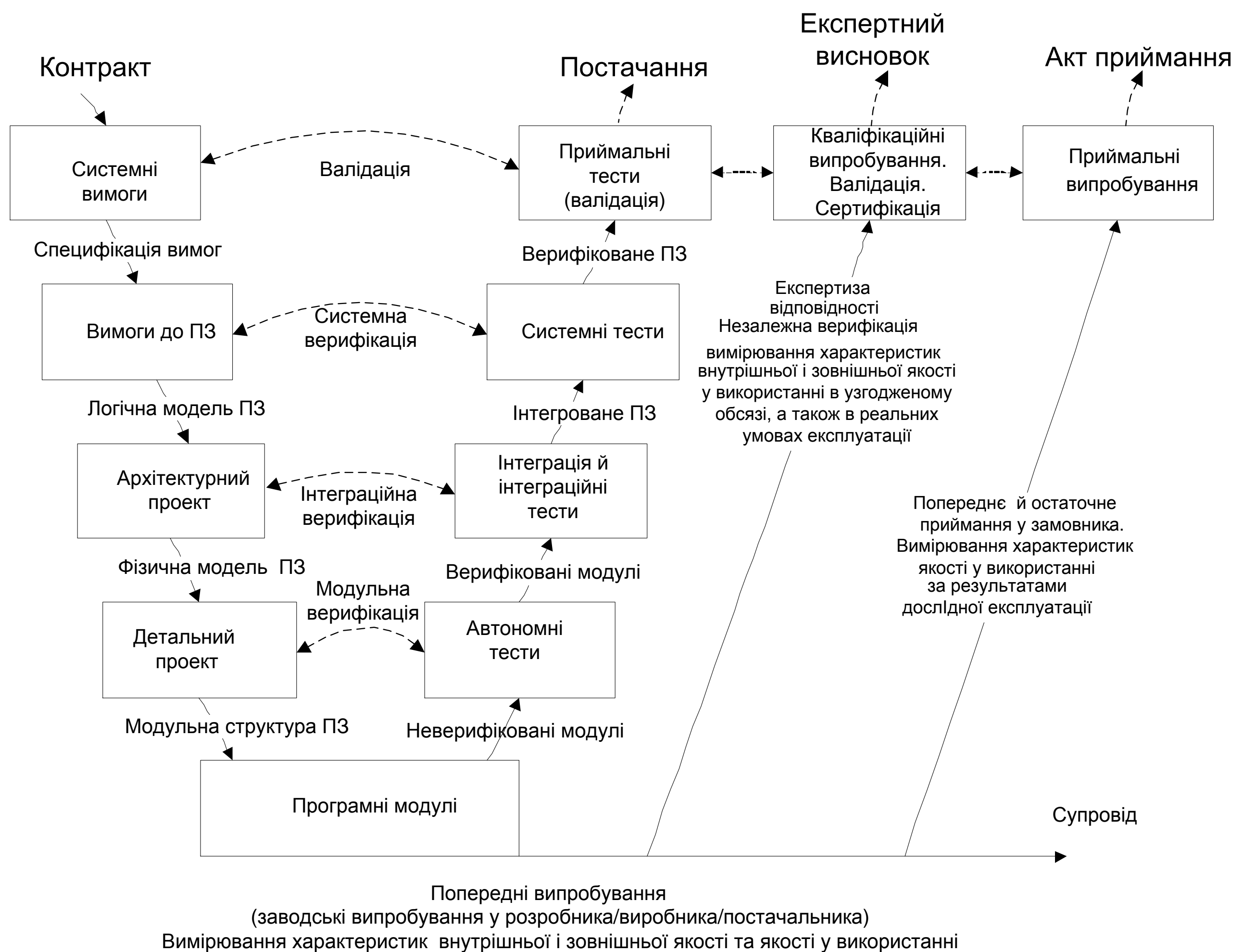

Рис. 10. Місце верифікації у ЖЦ

Слід розрізняти поняття верифікації і налагодження ПЗ. Обидва процеси спрямовано на зменшення помилок у кінцевому програмному продукті, проте налагодження – це процес, який спрямовано на локалізацію й усунення помилок в системі, а верифікація – на демонстрацію наявності помилок і умов їх виникнення. Верифікація, на відміну від налагодження, – контрольований і керований процес, що складається з аналізування причин виникнення помилок і наслідків, обумовлених їх виправленням, планування процесів пошуку помилок і їх виправлення, оцінювання отриманих результатів. Усе це дає змогу говорити про верифікацію як про процес забезпечення наперед заданого рівня якості програмного продукту, що створюється.

2.1.1. Мета і завдання верифікації

Основна мета процесу – доведення відповідності результату розроблення поставленим вимогам. Зазвичай процес верифікації проводиться зверху вниз, починаючи від загальних вимог, заданих у

вимогах користувача та/або специфікації на всю інформаційну систему, до детальних вимог на програмні модулі та їх взаємодію. Під час верифікації послідовно перевіряється таке:

– загальні вимоги до інформаційної системи коректно перероблено на специфікацію вимог високого рівня до комплексу програм, що відповідають початковим системним вимогам;

– вимоги високого рівня правильно перероблено на архітектуру ПЗ і на специфікації вимог до функціональних компонентів низького рівня, які задовольняють вимогам високого рівня;

– специфікації вимог до функціональних компонентів ПЗ, розташованих між компонентами високого й низького рівнів, задовольняють вимогам більш високого рівня;

– архітектуру ПЗ і вимоги до компонентів низького рівня коректно перероблено на програмні коди і масиви даних, що їм відповідають;

– програмні й об'єктні коди не мають помилок.

Крім того, верифікації на відповідність специфікації вимог на конкретний проект програмного засобу підлягають вимоги до технологічного забезпечення ЖЦ, а також вимоги до експлуатаційної і технологічної документації.

Мета верифікації досягається шляхом послідовного виконання комбінації з інспектування проектної документації та аналізування їхніх результатів, розроблення планів тестування і тестових вимог, тестових сценаріїв і процедур з подальшим виконанням цих процедур. Тестові сценарії призначено для перевірки внутрішньої несуперечності й повноти реалізації вимог. Виконання тестових процедур має демонструвати відповідність між програмами і початковими вимогами.

На вибір ефективних методів верифікації і послідовність їх застосування найбільше мірою впливають основні характеристики ПЗ:

– клас комплексу програм, що визначається глибиною зв'язку його функціонування з реальним часом і випадковими діями із зовнішнього середовища, а також вимогами до якості оброблення інформації і надійності функціонування;

– складність або масштаб комплексу програм і його функціональних компонентів, що є кінцевими результатами розроблення.

Наведемо деякі поняття й означення, пов'язані з процесом тестування як складовою частиною верифікації. Майєрс дає такії означення основних термінів:

– *тестування* – процесс виконання програми для виявлення помилок;

– *тестові дані* – це такі дані, що використовуються для перевірки програми;

– *тестова ситуація (test case)* містить вхідні дані для перевірки програми й передбачувані виходи залежно від входів, якщо програма працює відповідно до специфікації вимог;

– *хороша тестова ситуація* – така ситуація, що має велику ймовірність виявлення помилок;

– *вдалий тест* – це тест, що знаходить помилку;

– *помилка* – дія програміста на етапі розроблення, яка призводить до того, що в ПЗ виникає внутрішній дефект, який під час роботи програми може призвести до неправільного результату;

– *відмова* – непередбачувана реакція системи, що призводить до неочікуваного результату, який було спричинено програмними дефектами.

Таким чином, під час тестування ПЗ перевіряють відповідність ПЗ вимогам на нього, а також адекватність функціонування ПЗ в ситуаціях, не відображених у вимогах, тобто відсутність системних відмов. Крім того здійснюється перевірка функціонування ПЗ у випадку типових помилок програмістів.

2.1.2. Особливості тестування, верифікації і валідації ПЗ

Незважаючи на уявну схожість, терміни «тестування», «верифікація» і «валідація» означають різні рівні перевірки коректності роботи програмної системи. Щоб уникнути подальшої плутанини, чітко визначимо ці поняття.

Тестування програмної системи – вид діяльності під час розроблення, пов'язаний з виконанням процедур, спрямованих на виявлення (доведення наявності) помилок (невідповідностей, неповноти, двозначностей і т.д.) у поточному стані програмного проекту. Процес тестування належить насамперед до процесу перевірки коректності програмної реалізації системи, відповідності реалізації вимогам, тобто тестування – це кероване виконання програми з метою виявлення невідповідностей її функціонування і вимог.

Верифікація ПЗ – більш загальне поняття, ніж тестування. Метою верифікації є досягнення гарантії того, що об'єкт, який верифікується (вимоги або програмний код), відповідає вимогам, його реалізовано без непередбачених функцій і він задовольняє проектним специфікаціям і стандартам. Процес верифікації складається з інспекції, тестування коду, аналізування результатів тестування, формування й аналізування звітів про проблеми. Таким чином, процес тестування є складовою частиною верифікації.

Валідація програмної системи – процес, метою якого є доведення факту досягнення запланованих цілей унаслідок розроблення системи. Іншими словами, валідація – це перевірка відповідності системи очікуванням замовника.

Тестування відповідає на запитання «Як це зроблено?» або «Чи відповідає поведінка розробленої програми вимогам?», верифікація – на запитання «Що зроблено?» або «Чи відповідає розроблена система вимогам?», а валідація – на запитання «Чи зроблено те, що потрібно?» або «Чи відповідає розроблена система очікуванням замовника?».

2.2. Типи процесів верифікації

Існують кілька основних типів процесів верифікації.

Модульне тестування, яке застосовується для перевірки коректності невеликих програмних модулів (процедур, класів тощо). При тестуванні невеликого модуля розміром 100…1000 рядків є можливість перевірити якщо не всі, то принаймні більшість з логічних гілок реалізації, більшість шляхів у графі залежності даних, а також граничні значення параметрів. Відповідно до цього будуються критерії тестового покриття (покриття всіх операторів, всіх логічних гілок програми, всіх граничних даних тощо). Модульне тестування зазвичай виконується для кожного незалежного програмного модуля і є найбільш поширеним видом тестування, особливо для систем малих і середніх розмірів.

Інтеграційне тестування. Перевірка коректності всіх модулів, на жаль, не гарантує коректності функціонування системи модулів. Відповідно до плану інтеграційного тестування система будується поетапно, групи модулів додаються поступово. Так само поступово, за планом інтеграції і тестування перевіряється часткова функціональність створюваної системи.

Системне тестування застосовується для перевірки повністю реалізованого програмного продукту. На цьому етапі тестувальник перевіряє не коректність реалізації окремих процедур і методів, а всю програму в цілому, як її має бачити кінцевий користувач. Основою для тестів є загальні вимоги до програми, у тому числі не тільки коректність реалізації функцій, але й продуктивність, час відгуку, стійкість до збоїв, атак, помилок користувача тощо. Для системного й модульного тестування використовуються специфічні види критеріїв тестового покриття (наприклад, чи покрито усі типові сценарії роботи, усі сценарії з нештатними ситуаціями, парні композиції сценаріїв тощо).

Тестування навантаження дає змогу не тільки одержувати прогнозовані дані про продуктивність системи під навантаженням, яке орієнтовано на ухвалення архітектурних рішень, але й надає робочу інформацію службам технічної підтримки, а також менеджерам проектів і конфігураційним менеджерам, які відповідають за створення найбільш продуктивних конфігурацій устаткування і ПЗ. Тестування навантаження дає змогу команді з розроблення приймати більш обгрунтовані рішення, спрямовані на вироблення оптимальних архітектурних композицій.

Замовник, зі свого боку, дістає можливість проводити приймально-здавальні випробування в умовах, наближених до реальних.

Формальна інспекція є одним із способів верифікації документів і програмного коду, під час якого група фахівців здійснює незалежну перевірку їх відповідності. Незалежність перевірки забезпечується тим, що її здійснюють інспектори, які не брали участі в розробленні документів, що інспектуються.

Сертифікація ПЗ здійснюється у випадках, коли ПЗ буде використовуватися у критично важливих галузях або його відмова може призвести до значних людських жертв або втрат. Під сертифікацією розуміється процес установлення і офіційного визнання того, що розроблення ПЗ проводилося відповідно до певних вимог. Під час сертифікації відбувається взаємодія заявника, сертифікувального й наглядового органів.

Заявник – організація, що подає заявку до відповідного сертифікувального органу на отримання сертифіката (відповідності, якості, придатності тощо) виробу.

Сертифікувальний орган – організація, що розглядає заявку про проведення сертифікації ПЗ і або самостійно, або шляхом формування спеціальної комісії проводить процедури, які направлено на проведення процесу сертифікації ПЗ заявника.

Наглядовий орган – комісія фахівців, що спостерігають за процесами розроблення заявником інформаційної системи, яка сертифікується, і дають висновок про відповідність цього процесу певним вимогам.

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

У першому випадку результатом сертифікації є визнання відповідності процесів розроблення певним критеріям, а функціональності системи – певним вимогам. У другому випадку результатом є визнання відповідності процесів розроблення певним критеріям, що гарантує відповідний рівень якості продукції і її придатності для експлуатації в певних умовах. Прикладом таких стандартів може бути серія міжнародних стандартів якості ISO 9000:2000, стандартів Міжнародного електротехнічного комітету (МЕК) ISO/EC, ДСТУ або авіаційних стандартів Міністерства оборони США DO178B, AS9100, AS9006.

Тестування ПЗ, що сертифікується, має дві мети:

– продемонструвати, що ПЗ задовольняє вимогам до нього;

– продемонструвати з високим рівнем достовірності, що помилки, які можуть призвести до неприйнятних відмовних ситуацій, виявлено під час тестування.

Метою тестів для нормальних ситуацій є демонстрація здатності ПЗ давати відгук на нормальні вхідні дані й умови відповідно до вимог.

Метою тестів для ненормальних ситуацій є демонстрація здатності ПЗ адекватно реагувати на ненормальні вхідні дані й умови, інакше кажучи, це не повинно спричиняти відмову системи.

Категорії відмовних ситуацій для системи встановлюються шляхом визначення небезпеки відмовної ситуації для літака і тих, хто в ньому знаходиться. Будь-яка помилка в ПЗ може спричиняти відмову. Таким чином, рівень цілісності ПЗ, необхідний для безпечної експлуатації, залежить від відмовних ситуацій для системи.

Існує п’ять рівнів відмовних ситуацій: від неістотної до критично небезпечної. Згідно з цими рівнями вводиться поняття рівня критичності ПЗ. Від рівня критичності залежить склад документації, що надається до сертифікувального органу, а отже, і глибина процесів розроблення і верифікації системи. Наприклад, кількість типів документів і обсяг робіт з розроблення системи, необхідних для сертифікації за найнижчим та найвищим рівнем критичності DO-178B можуть відрізнятися на один-два порядки. Конкретні вимоги визначено в стандарті, за яким планується вести сертифікацію.

2.3. Інструментальні засоби автоматизованої верифікації

Розроблення ПЗ реальних ІКС – складний, багатоконтурний технологічний процес. Тестування можна умовно поділити на планування тестування, проектування тестування, виконання тестів і оцінювання результатів. Контроль виконання тесту й оцінювання його результатів є базисом, на основі якого виконуються задачі тестів і задовольняються потрібні вимоги. Ручний контроль – потенційне джерело помилок, що є одним із аргументів на користь розроблення автоматизованих інструментальних засобів (ІЗ) тестування, важливість яких стає безперечною для ІКС критичного застосування. Тому використання автоматизованих засобів тестування забезпечує поліпшену організацію тестування при автоматизації, вимірювання обсягів тестування, підвищену надійність [18]. В основу автоматизованої верифікації покладено формальні методи, з допомогою яких на перших етапах ЖЦ можна виявляти ДПЗ.

Інструментальні засоби виконують п’ять головних функцій: аналіз коду і створення БД, що відтворює архітектуру ПП; генерацію протоколів статичного аналізування (СА) програмного коду; інструментування коду шляхом його модифікації; аналізування результатів тестів і генерацію звітів. Класифікацію і перелік основних функцій ІЗ автоматизації фази аналізу вимог, проектування й розроблення ПЗ наведено на рис. 11.

Автоматизоване тестування дає змогу зменшити на 30 % ресурсні витрати та підвищити достовірність перевірки, збільшити об’єктивність. Однак застосування автоматизованих ІЗ є доцільним при роботі з ПЗ

великого обсягу, до яких належить ПЗ ІКС АЕС та ПЗ бортових комплексів, бо вони мають такі обмеження: трудомісткість підготовки до верифікації, неможливість повного покриття всіх обчислювальних маршрутів, значну обчислювальну складність, обумовлену великою кількістю тестів [19].

Класифікацію і перелік основних функцій ІЗ фази тестування відображено на рис. 12.

Існуючі ІЗ СА можна поділити на дві групи: комерційні засоби широкого використання, які можуть придбати організації науково-технічної підтримки державних регулювальних органів, та засоби, які розроблено ними спеціально для галузі атомної енергетики. У першому випадку зменшуються час та інші ресурси, необхідні для розроблення власного інструментального засобу автоматизованої верифікації, у другому – підвивищується спеціалізація ІЗ, що збільшує повноту і достовірність експертних оцінок.

Методи верифікації сьогодні перебувають у стадії досліджень. Водночас отримані результати досліджень з верифікації заслуговують на розгляд цієї проблеми як складової частини перспективної технології програмування і методології експертизи.

Тестування і верифікація – найважливіші складові забезпечення якості ПЗ, тому цю діяльність, враховуючи її ресурсомісткість, необхідно спеціально планувати, керуючись програмою забезпечення якості ПЗ із використанням атестованих ІЗ [20]. Верифікація охоплює праву гілку моделі ЖЦ ПЗ. В основу класичної верифікації покладено порівняння вимог, отриманих на лівій гілці ЖЦ ПЗ, і фактичних результатів тестування. Власне верифікація містить модульне, інтеграційне, системне й приймальне тестування [21]. На кожному з етапів використовуються спеціалізовані інструментальні засоби.

Модульне тестування можна поєднувати із СА коду. Наприклад, інструментальний засіб PC Lint дає змогу перевіряти синтаксис коду та виявляти всі непевні місця. Прикладами інших ІЗ статичного й динамічного аналізу є, які розроблено компаніями LDRA і Discover:

– LDRA Testbed – інструмент для керування якістю тестування ПЗ, з підвищеними вимогами до надійності. Для досягнення потрібного рівня якості використовують методи статичного й динамічного аналізування длядослідження коду й перевірки його відповідності стандарту алгоритмічної мови, а також формування звітів про якість і структуру коду. Це дає змогу знаходити фрагменти, які потребують підвищеної уваги. Динамічний аналіз шляхом запуску інструментованої версії з тестовими даними виявляє ДПЗ під час виконання програми;

– Discover – ІЗ підтримки розроблення, який складається з інтегрованих утиліт-аналізаторів програмного коду і створює інформаційну базу даних, що фіксує взаємозв’язки сутностей коду й відновлює архітектуру продукту.

Рис. 11. Інструментальні засоби верифікації лівої гілки ЖЦПЗ

Фаза тестування

Рис. 12. Інструментальні засоби верифікації правої гілки ЖЦПЗ

2.4. Тестування програмного коду

2.4.1. Задачі і мета тестування програмного коду

Тестування програмного коду – це процес виконання програмного коду, спрямований на виявлення ДПЗ. Під дефектом тут розуміється ділянка програмного коду, виконання якого за певних умов призводить до несподіваної реакції системи (тобто функціональності, що не відповідає вимогам), яка може спричинити збої в її роботі й відмови, У цьому випадку говорять про істотні дефекти програмного коду. Деякі ДПЗ спричиняють незначні проблеми і не порушують процес функціонування системи, але трохи ускладнюють роботу з нею, у цьому випадку говорять про середні або малозначні дефекти.

Задача тестування при такому підході – визначення умов, за яких виявляються дефекти системи, та протоколювання цих умов. До задачі тестування зазвичай не належить виявлення конкретних дефектних ділянок програмного коду і ніколи не належить виправлення дефектів – це задача налагодження, що виконується за результатами тестування системи.

Мета тестування програмного коду – мінімізація кількості дефектів, особливо істотних, у кінцевому продукті. Тестування саме по собі не може гарантувати повної відсутності дефектів у програмному коді системи. Проте в поєднанні з процесами верифікації і валідації, які направлено на усунення суперечності й неповноти проектної документації (зокрема, вимог на систему), правильно організоване тестування дає гарантію того, що система задовольняє вимогам і поводиться відповідно до них у всіх передбачених ситуаціях.

При розробленні систем підвищеної надійності, наприклад авіаційних, гарантії надійності досягаються з допомогою чіткої організації процесу тестування, визначення його зв'язку з рештою процесів ЖЦ, уведення кількісних характеристик, що дають змогу оцінювати успішність тестування. При цьому чим вище вимоги до надійності системи (її рівень критичності), тим більш жорсткі вимоги ставляться до тестування.

Оскільки сучасні програмні системи мають значні розміри, при тестуванні їхнього програмного коду використовується метод функціональної декомпозиції. Система розбивається на окремі модулі (класи, простори імен тощо), що повинні відповідати певним вимогам функціональності та мають відповідні інтерфейси. Після цього окремо тестується кожний модуль – виконується модульне тестування. Потім виконується інтеграція окремих модулів у більш великі конфігурації – виконується інтеграційне тестування і, нарешті, тестується система в цілому – виконується системне тестування.

Модульне, інтеграційне й системне тестування мають багато спільного, тому головну увагу буде приділено модульному тестуванню, особливості інтеграційного і системного тестування буде розглянуто пізніше.

Під час модульного тестування кожний модуль тестується як на відповідність вимогам, так і на відсутність проблемних ділянок програмного коду, що можуть спричинити відмови і збої в роботі системи. Зазвичай модулі не працюють поза системою – вони приймають дані від інших модулів, переробляють їх і передають далі. Для того щоб ізолювати модуль від системи і виключити вплив потенційних помилок системи, а також забезпечити модуль всіма необхідними даними, використовується тестове оточення.

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

Типова процедура тестування полягає в підготовці й виконанні тестових завдань (або просто тестів). Кожний тест перевіряє одну «ситуацію» в спефиціації модуля і складається зі списку значень, що надходять на вхід модуля, опису запуску й виконання перероблення даних – тестового сценарію і списку значень, що очікуються на виході модуля у разі його коректної реакції. Тестові сценарії складають так, щоб виключити звернення до внутрішніх даних модуля, уся взаємодія має відбуватися тільки через його зовнішні інтерфейси.

Виконання тесту підтримується тестовим оточенням, що містить програмну реалізацію тестового сценарію. Виконання починається з передання модулю вхідних даних і запуску сценарію. Реальні вихідні дані, які отримано від модуля внаслідок виконання сценарію, зберігаються і порівнюються з очікуваними. У разі їх збігу тест вважається пройденим, інакше – не пройденим. Кожний непройдений тест свідчить про дефект або модуля, або тестового оточення, або ж власного опису.

Сукупністю описів тестів є тестовий план – основний документ, за яким визначається процедура тестування програмного модуля. У тестовому плані задано не тільки самі тести, але й порядок їх проходження, який також може бути важливим.

При тестуванні часто буває необхідно враховувати не тільки вимоги до системи, але й структуру програмного коду модуля, що тестується. У цьому випадку тести складаються так, щоб знаходити типові помилки програмістів, спричинені неправильною інтерпретацією вимог. Застосовуються перевірки граничних умов, класів еквівалентності. Відсутність в системі можливостей, не заданих вимогами, гарантують різне оцінювання покриття програмного коду тестами, тобто оцінювання того, який відсоток тих або інших мовних конструкцій виконано унаслідок тестування.

2.4.2. Методи тестування програмного забезпечення

Метод чорної скрині

Основна ідея тестування системи за методом чорної скрині полягає в тому, що тестувальник має доступ тільки до системи і вимог, які описують функціональність системи. Працювати з програмною системою можна тільки подаючи на її входи деякі зовнішні дані і спостерігаючи за результатом. Усі внутрішні особливості реалізації системи приховано від тестувальника. Таким чином, система і є «чорною скринею», правильність функціональності якої відносно вимог і треба перевірити.

З огляду на програмной код чорна скриня – множина класів (або модулів) з відомими зовнішніми інтерфейсами, але недосяжним програмним кодом, що їх реалізує.

Основна задача тестувальника при цьому полягає в послідовній перевірці відповідності функціональності системи вимогам. Крім того, тестувальник повинен перевірити роботу системи в критичних ситуаціях: що відбувається у разі подання неправильних значень вхідних даних. В ідеальній ситуації всі варіанти критичних ситуацій має бути описано у вимогах на систему, і тестувальнику залишається тільки визначити конкретні перевірки цих вимог. Проте в реальності під час тестування зазвичай виявляються два типи проблем системи:

– функціональність системи не відповідає вимогам;

– неадекватна функціональність програмної системи в ситуаціях, не передбачених у вимогах.

Звіти про проблеми документуються і передаються розробникам. При цьому проблеми першого типу зазвичай спричиняють змінення програмного коду, набагато рідше – змінення вимог, що у цьому випадку може бути потрібним, зважаючи на їх суперечність (декілька різних вимог описують різні реакції системи в одній і тій самій ситуації) або некоректність (вимоги не відповідають дійсності).

Проблеми другого типу однозначно потребують змінення вимог через їх неповноту – у вимогах явно пропущено ситуацію, що призводить до неадекватної реакції системи. При цьому під неадекватною реакцією може розумітися як повний крах системи, так і взагалі будь-яка реакція, що не описано у вимогах.

Тестування методом чорної скрині називають також тестуванням за вимогами, оскільки це єдине джерело інформації для розроблення тестового плану.

Метод скляної (білої) скрині

При тестуванні системи як скляної скрині тестувальник має доступ не тільки до вимог до системи, системного інтерфейсу, але й до її внутрішньої структури – він бачить її програмний код.

Доступність програмного коду збільшує можливості тестувальника оскільки він може бачити відповідність вимог ділянкам коду і оцінити рівень покриття коду вимогами. Програмний код, для якого немає вимог, називають кодом, який не покрито вимогами. Такий код є потенційним джерелом неадекватної реакції системи. Крім того, прозорість системи дає змогу поглибити аналіз її ділянок, що спричиняють проблеми: часто одна проблема нейтралізує іншу і вони ніколи не виникають одночасно.

Тестування моделей програм

Тестування моделей дещо відрізняється від класичних методів верифікації ПЗ [5]. Перш за все це пов'язано з тим, що об'єкт тестування – не сама система, а її модель, яку розроблено формальними засобами. Якщо не брати до уваги питання перевірки коректності й застосовності самої моделі (вважається, що її коректність і відповідність початковій системі можна довести формальними засобами), то тестувальник отримує у своє розпорядження доситть потужний інструмент аналізу загальної цілісності системи. Працюючи з моделлю, можна створити ситуації, які в тестовій лабораторії для реальної системи створити неможливо. Працюючи з моделлю програмного коду системи, можна аналізувати його властивості й такі параметри системи, як оптимальність алгоритмів або її стійкість.

Проте тестування моделей не отримало сьогодні широкого розповсюдження через труднощі з розроблення формального опису функціонування системи.

Аналіз програмного коду (інспекції)

У багатьох ситуаціях тестування функціональності системи в цілому є неможливим – окремі ділянки програмного коду можуть ніколи не виконуватися, при цьому їх буде покрито вимогами. Прикладом таких ділянок коду є обробники виняткових ситуацій. Якщо, наприклад, два модулі передають один одному числові значення і функції перевірки коректності значень працюють в обох модулях, то функцію перевірки модуля-приймача ніколи не буде активізовано, оскільки всі помилкові значення буде відсічено ще в передавачі.

У цьому випадку виконується ручне аналізування програмного коду на коректність, що також має назву перегляду, або інспекції коду. Якщо після інспекції виявляються проблемні ділянки, то інформація про це

передається розробникам для виправлення нарівні з результатами звичайних тестів.

2.4.3. Тестувальне оточення

Основний обсяг тестування будь-якої складної системи зазвичай виконується в автоматичному режимі. Крім того, система, що тестується, розбивається на кілька окремих модулів, кожний з яких тестується спочатку окремо від інших, а потім в комплексі.

Це означає, що для виконання тестування необхідно створити деяке середовище, що забезпечить запуск і виконання модулів для тестування, а також надасть їм вхідні дані й результат роботи системи на заданих вхідних даних. Після цього середовище має порівняти реальні вихідні дані з очікуваними і на основі цього порівняння зробити висновок про відповідність реакції модуля тій реакції, яку задано у специфікації.

Тестувальне оточення також може використовуватися для вилучення окремих модулів із системи. Відокремлення модулів системи на ранніх етапах тестування дає змогу більш точно локалізувати проблеми, що виникають у програмному коді. Для підтримки автономної роботи модуля тестове оточення повинно моделювати поведінку всіх модулів, до функцій або даних яких звертається модуль, що тестується.

Оскільки тестувальне оточення є програмою (причому, часто не на тій мові програмування, на якій написано систему), його треба протестувати. Метою тестування тестувального оточення є доведення того, що тестове оточення ніяким чином не спотворює виконання модуля, що тестується, та адекватно моделює системи.

Тестувальне оточення для програмного коду на структурних мовах програмування, схема якого зображена на рис.13, складається з двох компонентів – драйвера, який забезпечує запуск і виконання модуля, що тестується, і заглушок, які моделюють функції, що викликаються з цього модуля.

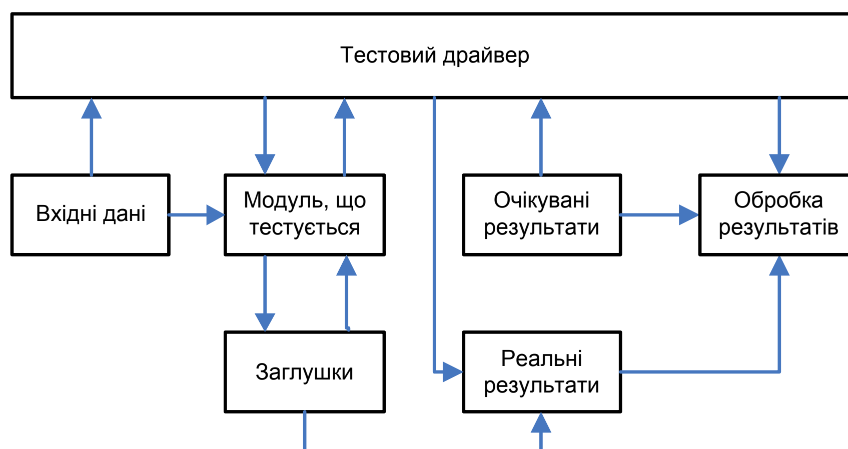

Рис. 13. Узагальнена схема тестового оточення

Розроблення тестового драйвера є окремою задачею тестування, сам драйвер необхідно протестувати, 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.

Тестовий драйвер може виконувати такі функції:

– виклик модуля, що тестується;

– передача в модуль вхідних значень та приймання результатів;

– передача вихідних результатів для подальшого оброблення;

– протоколювання процесу тестування і ключових точок програми.

Заглушки можуть або не виконувати ніяких дій, або виконувати такі функції:

– забезпечувати коректність редагування зв’язків модуля;

– виводити повідомлення про те, що заглушку було викликано;

– виводити повідомлення зі значеннями параметрів, переданих у функцію;

– повертати значення, яке наперед задано у вхідних тестових даних;

– виводити значення, яке наперед задано у вхідних тестових даних;

– приймати від ПЗ значення, що тестуються, і передавати їх у драйвер.

Для тестування програмного коду, написаного на процедурній мові програмування, використовуються драйвери, що є програмою з точкою входу (наприклад, функцією *main()*), функціями запуску модуля, що тестується, і функціями накопичення результатів. Зазвичай драйвер має принаймі одну функцію – точку входу, якій передається керування при його виклику.

Функції-заглушки можуть знаходитися у тому ж файлі програмного коду, що й основний текст драйвера. Імена і параметри заглушок мають збігатися з іменами й параметрами функцій реальної системи, що «заглушуються». Це вимога є важливою не стільки для коректної інтеграції системи (при використанні тестового драйвера і ПЗ, що тестується, може використовуватися зведення типів), скільки для того, щоб максимально точно моделювати функціональність реальної системи при передані даних. Наприклад, якщо в реальній програмній системі є функція *double function(double value),* то у випадку, коли замість параметра типу *double* було використано *float*, зменшення точності може спричинити непередбачувані результати в модулі, що тестується.

При модульному тестуванні необхідно використовувати заглушки замість реальних функций. Це дасть змогу імітувати роботу ще нествореного ПЗ і полегшить сам процес тестування. Заглушки можуть виводити трасувальне повідомлення, здійснювати будь-які маніпуляції з даними і таким чином моделювати роботу складних програмних модулів, які ще не завершили тестування.

2.4.4. Тестові класи

Тестове оточення для об'єктно-орієнтованого ПЗ виконує ті самі функції, що й для структурних програм (на процедурних мовах). Проте, у нього є деякі особливості, що пов'язані із застосуванням парадігм ООП, пов’язаних з успадковуванням та інкапсуляцією.

Якщо під час тестування структурних програм мінімальним об'єктом, що тестується, 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 є недосяжними ззовні. У цьому випадку тестувальник позбавлений можливості звертатися в своїх тестах до даних класу і довільно викликати методи. Усе, що він може, – викликати методи зовнішнього інтерфейсу класу.

Існує декілька підходів до тестування класів, кожний з яких накладає свої обмеження на структуру драйвера і заглушок.

1. Драйвер створює один або більше об'єктів класу, що тестується. Усі звернення до об'єктів відбуваються тільки з використанням їхнього зовнішнього інтерфейсу. Текстом драйвера в цьому випадку є так званий тестувальний клас, що містить по одному методу для кожного тесту. Процес тестування полягає в послідовному виклику цих методів. Замість заглушок у складі тестового оточення є програмний код реальної системи, відповідно немає ізоляції класу, що тестується. Проте саме такий підхід до тестування прийнято у більшості методологій і середовищ розроблення.

2. Заглушки створюються аналогічно попередньому підходу, але для всіх класів, які використовує певний клас.

3. Програмний код класу модифікується таким чином, щоб відкрити доступ до всіх його властивостей і методів. Тестове оточення в цьому випадку є повністю аналогічним оточенню для тестування структурних програм.

4. Використовуються спеціальні засоби доступу до закритих даних і методів класу на рівні об'єктного коду, які є властивими середовищам розроблення програм.

Основною перевагою перших двох методів є те, що при їх використанні клас працює таким же чином, як і в реальній системі. Проте в цьому випадку не можна гарантувати того, що під час тестування буде виконано весь програмний код класу і не залишиться непротестованих методів.

Основним недоліком третього методу є те, що після змінення програмних кодів модуля, що тестується, не можна дати гарантію, що методи класу будуть виконуватися так само, як і на початку тестування. Зокрема, це пов'язано з тим, що змінення захисту даних класу впливає на успадковування даних і методів іншими класами.

Тестування успадковування – окрема складна задача в об'єктно-орієнтованих системах. Після того, як базовий клас протестовано, необхідно тестувати класи-нащадки. Проте для базового класу не можна створювати заглушки, оскільки в цьому випадку можна пропустити можливі проблеми поліморфізму. Якщо клас-нащадок використовує методи базового класу для оброблення власних даних, то необхідно переконатися, що ці методи працюють.

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

2.4.5. Генератори тестових сигналів

У значній частині складних програм використовується міжпроцесна взаємодія. Це обумовлено еволюцією підходів до проектування програмних систем.

1. Монолітні програми, що мають в коді всі необхідні для роботи інструкції. Обмін даними усередині таких програм здійснюється шляхом передання параметрів функцій і використання глобальних змінних. Під час запуску таких програм утворюється один процес, який і виконує необхідну функціональність програмної сиситеми.

2. Модульні програми, які складаються з окремих програмних модулів з певними інтерфейсами викликів. Об'єднання модулів у програму може відбуватися на етапі або складання виконуваного файла (статична збірка, або static linking), або виконання програми (динамічна збірка, або dynamic linking). Перевага модульних програм полягає у досягненні деякого рівня універсальності: один модуль можна замінити іншим. Проте модульна програма все одно є єдиним процесом, а дані, які необхідні для вирішення завдання, передаються всередині процесу як параметри функцій.

3. Програми, що використовують міжпроцесну взаємодію, утворюють програмний комплекс, який призначено для вирішення загального завдання.

Кожна програма утворює один або більше процесів. Кожний з процесів використовує для вирішення завдання або свої власні дані й обмінюється з іншими процесами тільки результатом своєї роботи, або працює із загальною областю даних, що поділяється між різними процесами. Для вирішення особливо складних завдань процеси можуть функціонувати на різних фізичних комп'ютерах і взаємодіяти через мережу. Перевага використання міжпроцесної взаємодії полягає у ще більшій універсальності – процеси, що взаємодіють, можна замінити незалежно один від одного при збереженні інтерфейсу взаємодії. Ще одна перевага полягає у тому, що обчислювальне навантаження розподіляється між процесами. Це дає змогу операційній системі керувати

пріоритетами виконання окремих частин програмного комплексу й виділяти певну кількість ресурсів.

При виконанні багатьох процесів, що вирішують загальне завдання, використовується декілька типових механізмів взаємодії, які призначено для такого:

– передання даних від одного процесу до іншого;

– одночасне використання одних і тих самих даних декількома процесами;

– сповіщення про змінення стану процесів.

У всіх цих випадках типова структура кожного процесу є кінцевим автоматом з набором станів і переходів між ними. Перебуваючи у певному стані, процес виконує оброблення даних, а при переході між станами – надсилає дані іншим процесам або приймає дані від них.

2.4.6. Тести

Тестове оточення забезпечує процес тестування необхідною інфраструктурою і підтримує її. Безпосередньо для тестування окрім тестового оточення необхідно визначити перевірні завдання, які буде вирішувати система або її частина. Такі перевіні завдання називають тестовими завданнями або просто тестами.

Кожний тест складається із вхідних значень для системи, опису сценарію роботи тесту й очікуваних результатів. Метою виконання будь-якого тесту є або демонстрація наявності в системі дефекту, або доведення його відсутності.

Створення тестів

Основним джерелом інформації для створення тестів є різноманітна документація на систему, наприклад функціональні вимоги й вимоги до інтерфейсу.

Функціональні вимоги описують реакцію системи як «чорної скрині», тобто виключно з огляду на те, що має робити система в різних ситуаціях. Іншими словами, функціональні вимоги визначають реакцію системи на різні вхідні дії.

Наприклад, функціональні вимоги на програмний модуль підрахування й перевірки контрольної суми можуть мати такий вигляд:

Функциональні вимоги на модуль контрольної суми

Модуль приймає на вхід записи-структури:

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

Якщо всі байти мають нульове значення, то змінна Empty отримує значення TRUE;

Модуль повинен виконувати кілька функцій:

1. Функція підрахування контрольної суми **Set_CRC** має інтерфейс:

void Set_CRC(record_type record);

Вхід: запис **record** з невизначеною контрольною сумою **CRC**.

Вихід: запис **record з** підрахованим за певними правилами значенням **CRC.** Змінна **Empty.**

2. Функція перевірки контрольної суми **Check_CRC** має інтерфейс:

bool Check_CRC(record_type record);

Вхід: запис r**ec_ms** з визначеною контрольною сумою **CRC**.

Вихід: повертається значення **true** или **false**. Змінна **Empty.**

Функціональні вимоги

1. Ініціалізація модуля

Змінній **Empty** має бути присвоєно **true**.

2. Підрахування контрольної суми запису

а) розрахунок контрольної суми

Процедура **Set_CRC** має обчислити контрольну суму запису r**ec_ms** за алгоритмом CRC32. Під час розрахунків контрольної суми значення **CRC** не повинно використовуватися. На основі проведених розрахунків треба обчислити та визначити значення **CRC** таким чином, щоб разом із встановленим значенням цієї змінної контрольная сума дорівнювала нулю.

б) установлення значення змінної Empty

Якщо всі байти полей запису (окрім CRC) мають нульове значення, то значення **Empty** має бути встановленим у **true,** інакше – у **false**.

3. Перевірка контрольної суми запису

а) перевірка контрольної суми

Процедура має підраховувати відповідно до алгоритму CRC32 контрольну суму **rec_ms**. Результат має бути **true**, якщо підраховане значення дорівнює нулю, інакше – **false**.

б) установлення значення змінної Empty

Якщо всі байти запису, у тому числі CRC, мають нульове значення, то змінна **Empty** повинна мати значення **true,** інакше – **false**.

Початковий етап роботи тестувальника полягає у формуванні тестових вимог, що відповідають функціональним вимогам. Основна мета тестових вимог – визначити, яку функціональність системи має бути протестовано. У найпростішому випадку одній функціональній вимозі відповідає одна з тестових вимог, але частіше тестові вимоги деталізують функціональні вимоги.

Тестові вимоги визначають, що слід протестувати, але не визначають, як це зробити.

Наприклад, для перелічених функціональних вимог можна сформулювати такі тестові вимоги.

1. Перевірка ініціалізації модуля: перевірити, що початковим значенням програмної змінної Empty є **true**.

2. Перевірка правильного обчислення контрольної суми:

а) перевірити, що в процедурі Set_CRC обчислення контрольної суми проводиться за правилами алгоритму CRC32, як визначено в секції 2а функціональних вимог;

б) перевірити, що обчислене значення контрольної суми не залежить від початкового значення поля CRC;

в) перевірити, що обчислене значення контрольної суми не залежить від значень байтів вирівнювання полів запису;

г) перевірити, що значення програмної змінної Empty встановлюється при кожному виклику функції Set_CRC залежно від значень полів запису, як визначено в секції 2б функціональних вимог.

3. Перевірка процедури Check_CRC:

а) перевірити, що при зверненні до процедури Check_CRC обчислення контрольної суми проводиться за правилами алгоритму CRC32, як визначено в секції 3а функціональних вимог;

б) перевірити, що результатом є **true**, якщо контрольна сума запису правильна, та **false** – в інших випадках;

в) упевнитися, що перевірка правильності значення контрольної суми не залежить від значень байтів вирівнювання полів запису.

г) перевірити, що значення програмної змінної Empty встановлюється при кожному виклику функції Check_CRC залежно від значень запису, як визначено в секції 3б функціональних вимог.

Особливості реалізації тестового оточення і конкретні значення, що подаються на вхід системи й очікуються на її виході, визначаються тестами. Одній тестовій вимозі відповідає щонайменше один тест.

Типи тестів

Розглянемо різні класи тестів, спрямованих на виявлення різних дефектів у роботі програмної системи.

Допустимі дані

Частіше за все дефекти виявляються під час оброблення нестандартних даних, які не передбачено вимогами, – при введенні неправильних символів, порожніх рядків або під час дуже великої швидкості введення інформації. Проте перед пошуком таких дефектів необхідно упевнитися, що програма коректно обробляє правильні дані, які передбачено специфікацією, тобто перевірити роботу основних алгоритмів. Так, для функції обчислення контрольної суми допустимими вхідними даними буде довільний запис, що містить дані в усіх полях, окрім поля контрольної суми CRC:

```
record_type tv1; int i              //декларація тестового запису
Tv1.A = false;                      //занесення данних до змінної А запису
for (i=0;i<20;i++)  tv1.B[i] = i;   // занесення даних до масиву В запису
for (i=0;i<5;i++)
{ tv1.C[i] = i+5;                   // занесення даних до елемента С
  tv1.D[0] = i+8;                   // занесення даних до елемента D
  tv1.CRC = 0;                      // онулення контрольної суми
  Set_CRC(tv1);                     // обчислення контрольної суми
printf("%d\n", tv1.CRC);}           // циклічне друкування суми
```

Сценарієм буде виклик функції Set_CRC, а очікуваним вихідним значенням – коректне значення поля CRC, яке розраховано за алгоритмом CRC32.

Зазвичай для перевірки допустимих даних достатньо одного тесту, але функціональні вимоги можуть визначати різні групи допустимих даних, що об'єднуються у класи еквівалентністі. У цьому випадку необхідно визначати щонайменше один тест для одного класу еквівалентності.

Граничні дані

Окремий вид допустимих даних, передача яких в систему може розкрити дефект, це – граничні дані, тобто, наприклад, числа, значення яких є граничними для їхнього типу, або рядка граничної чи нульової довжини. Зазвичай з допомогою тестування граничних умов виявляються проблеми з арифметичним порівнянням чисел або з ітераторами циклів.

Для тестування функції Set_CRC на граничних умовах визначають два тести з мінімальними і максимальними значеннями полів запису:

Тестування з мінімальними значеннями полів запису:

```
record_type test_value2;   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);
```

Тестування з максимальними значеннями полів запису:

```
record_type test_value3; int i;  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 виявитися й у випадку, якщо системі не передаються дані або передаються дані нульового розміру. Для тестування функції Set_CRC за відсутності даних можна викликати її, передавши як параметр неініціалізовану структуру. Проте такий тест не є точним прикладом відсутності даних, швидше за все, це приклад випадкових (можливо, неправильних) даних:

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

Повторне введення даних

У разі повторного передання на вхід системи даних можуть виникати непередбачені результати. Зазвичай дефекти такого типу виявляються внаслідок того, що система не встановлює внутрішні змінні в початковий стан, або через похибки 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);
```

Неправильні дані

Перевіряючи функціональність системи, необхідно не забувати перевіряти її реакцію при переданні їй даних, які не передбачено вимогами, – занадто довгих або занадто коротких рядків, неправильних символів, чисел за межами обчислюваного діапазону тощо. Неправильні дані, як і допустимі, також можна поділяти на різні класи еквівалентності. Прикладом неправильних даних для функції Set_CRC може бути запис з іншою структурою, який передано у функцію шляхом зведення типів. Якщо при обчисленні контрольної суми використовуються імена полів запису, то контрольна сума може виявитися обчисленою неправильно або відбутися затирання областей пам'яті, не призначених для зберігання даних.

Якщо у тестувальному оточенні буде визначено іншу, відмінну від попередньої структуру запису

```
struct record_type2 {  int F;  int G[45];  int H[8]; unsigned int CRC;  int K[2]; }
```

і декларовано відповідну програмну змінну

record_type2 test_value6;

то після зведення програмної змінної до попереднього типу у програмі можливі непередбачувані ситуації:

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

Реініціалізація системи

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

Прикладом реініціалізації модуля обчислення CRC може бути примусове обнулення програмної 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_value7.CRC);
empty=true; Set_CRC(test_value7);
printf("%d\n", test_value7.CRC);
```

Стійкість системи

Під стійкістю системи розуміють її здатність витримувати нештатне навантаження, не передбачене вимогами, наприклад збереження системою працездатності після 10 тисяч викликів. Відповідний тест функції Set_CRC можна реалізувати таким чином:

```
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_value8.CRC);
```

Аналогічну перевірку можна зробити шляхом перегляду програмного коду (якщо він є доступним при тестуванні) на основі відсутності «історії» в реалізації програми, тобто відсутності даних, значення яких може змінюватися залежно від кількості запусків програми. Таким чином, у деяких випадках тестування можна замінити аналізуванням коду.

Нештатні стани середовища виконання

Нештатні стани середовища виконання (наприклад, вичерпання пам'яті, дискового простору або тривалий брак процесорного часу) можуть перешкоджати роботі системи або зробити її неможливою. Основна задача системи в такій ситуації – коректно завершити або припинити свою роботу.

Прикладом тесту, що створює нештатний стан середовища для функції Set_CRC, є виділення всієї вільної пам'яті перед викликом функції. Якщо Set_CRC використовує динамічну пам'ять, то в ній мають бути перевірки на можливість виділення пам'яті, інакше виконання функції може спричинити 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);
```

Граничні умови

У тестах, що відповідають тестовим вимогам, зазвичай використовують вхідні дані, що знаходяться всередині допустимого інтервалу. Одним із способів перевірки стійкості системи на значеннях, близьких до граничних, є створення для кожного входу щонайменше трьох тестів:

– значення всередині діапазону;
– мінімальне допустиме значення;
– максимальне допустиме значення.

Для гарантування впевненості у функціональності системи використовують п'ять тестів:

- мінімальне допустиме значення;
- значення = мінімальне допустиме значення + δ;
- значення всередині інтервалу;
- значення = максимальне допустиме значення – δ;
- максимальне допустиме значення,

де δ – для цілочисельних даних дорівнює одиниці, а для чисел з плаваючою крапкою визначається ціною найменшого розряду у машинному поданні числа.

Такий спосіб має назву перевірки на граничних значеннях і дає змогу виявляти проблеми, що пов'язані з виходом за межі інтервалу. Наприклад,

якщо до функції

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

будуть передані значення 255 і 255, то за відсутності спеціального оброблення ситуації переповнення сума буде обчислюватися неправильно.

Цю перевірку доречно використовувати для контролю операцій з індексами масивів. Наприклад, фрагмент коду

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

має дефект, що виявляється під час переданя функції масиву одиничної довжини.

Перевірка робастністі системи (вихід даних за межі інтервалу)

Робастність системи – це ступінь її чутливості до чинників, які не враховано на етапах проектування, наприклад до неточності основного алгоритму, що призводить до помилок округлення при обчисленнях, збоїв у зовнішньому середовищі або до даних, значення яких знаходяться зовні допустимого діапазону. Частіше за все під робастністю програмних систем розуміють саме стійкість до некоректних даних. Система має бути здатна коректно обробляти такі дані шляхом видання відповідних повідомлень про помилки. Збої і відмови системи на таких даних є недопустимими.

Для тестування робастності до граничних даних додаються ще два тести:

– мінімальне значення –δ;

– максимальне значення +δ,

де δ для цілочисельних даних дорівнює одиниці, а в інших випадках визначається типом даних.

Ці тести перевіряють коректність функціональності системи за межами допустимого діапазону. Крім того, у випадках тестування операцій порівняння додатково надаються гарантії того, що в них не допущено друкарської помилки.

Часто стверджують, що значення всередені інтервалу є надлишковим і його тестування не потрібне. Однак перевірка реакції системи на внутрішньому значенні є корисною у випадку обмеження інтервалу складними логічними умовами. Рекомендується окремо перевіряти реакцію системи на нульовому значенні (навіть якщо воно всередині інтервалу), тому що часто це значення обробляється некоректно (наприклад, у випадку ділення на нуль).

Класи еквівалентності даних

Під час розроблення тестів може виникнути ситуація, коли різні вхідні дані призводять до одних і тих самих реакцій системи. Якщо при цьому такі вхідні значення мають щось загальне, то їх можна об'єднати в класи еквівалентності, тобто виконати еквівалентне розбиття множини допустимих вхідних значень.

Розбиття на класи еквівалентності зменшує необхідну кількість тестів. Якщо в тестових вимогах не обумовлено інше, при тестуванні достатньо виконати тільки один тест для кожного класу еквівалентності. Розбиття на класи еквівалентності є особливо корисним, коли на вхід системи можна подати велику кількість різних значень; тестування кожного можливого значення призвело б до дуже великого обсягу тестування.

Розглянуті вище граничні умови можуть бути прикладом класів еквівалентності:

– значення з середини інтервалу;

– граничні значення;

– неприпустимі значення за межами інтервалу.

Таким чином, тестування граничних умов і тестування робастності є окремими випадками тестування з використанням класів еквівалентності: замість того щоб тестувати всі неприпустимі значення, вибираються тільки суміжні з граничними значеннями.

При визначенні класів еквівалентності слід керуватися такими правилами:

– завжди буде щонайменше два класи: коректний і некоректний;

– якщо за вхідною умовою визначається діапазон значень, то зазвичай є три класи: менше нижньої межі діапазону, усередині діапазону й більше верхньої межі діапазону (значення на кінцях діапазону можна вважати граничними значеннями);

– якщо елементи діапазону обробляються по-різному, то кожному варіанту оброблення будуть відповідати різні вимоги;

Іншим прикладом розбиття на класи еквівалентності є тестування відкриття файла за його іменем. Унаслідок тестування має бути визначено, чи всі варіанти імен оброблено системою згідно з такими тестовими вимогами:

1) система має виводити повідомлення про помилку в разі наявності в імені файла символів, що не є латинськими літерами або цифрами;

2) система має виводити повідомлення про помилку, коли довжина імені файла перевищує 11 символів;

3) система має бути незалежною від регістрів символів імені файла;

4) система має відкривати файл з іменами, які не суперечать вимогам 1 – 3.

Вхідними значеннями тестів є різні імена файлів, вихідними – реакція системи (помилка або успішне відкриття).

Можна вирізнити такі класи еквівалентності:

а) за довжиною імені:

– довжина імені менше 11 символів;

– довжина імені дорівнює 11 символам;

– довжина імені більше 11 символів;

б) за символами:

– ім'я, що складається з цифр і літер будь-якого регістра;

– ім'я, що складається з цифр і літер нижнього регістра;

– ім'я, що складається з цифр і літер верхнього регістра;

– ім'я, що складається тільки з цифр;

– ім'я, що складається тільки з літер;

– ім'я, що містить розділові знаки (не літеро-цифрові символи);

– ім'я, що містить керувальні символи (не літеро-цифрові символи);

Ці класи еквівалентності ілюструють, що перевірки на межах інтервалів можна застосовувати не тільки для тестування арифметичних операцій і операцій порівняння. Майже для будь-яких даних можна визначити мінімальні й максимальні допустимі значення.

Тестування операцій порівняння

Розбиття на класи еквівалентності широко використовується при тестуванні коректності реалізації арифметичних операцій і операцій порівняння. Для тестування виконується розбиття діапазону вхідних змінних на класи еквівалентності за методом аналізування граничних значень цих змінних. Тестові набори наведено в табл. 2, 3.

Таблиця 2

Тестові набори для порівняння константи і змінної

Операція порівняння	Вхід a					
	c - δ	c + δ	c	min	max	≠ c
a > c	F	T	*F*	F	T	-
a ≥ c	F	T	*T*	F	T	-
a < c	T	F	*F*	T	F	-
a ≤ c	T	F	*T*	T	F	-
a = c	-	-	*T*	F	F	F
a ≠ c	-	-	*F*	T	T	T

Тут *a* – програмна змінна; *c* – константа, що порівнюється зі змінною; *δ* –дискретність зміненя *a* і *c*, що для цілих чисел дорівнює одиниці, а в інших випадках визначається типом даних*; min, max* – мінімальне й максимальне значення змінної; *T, F* – очікувані результати (true, false);

курсив означає, що ці тестові набори виконуються для цілочислових даних; знак мінус – що тест для цієї операції не виконується.

Таблиця 3

Тестові набори функціональних блоків порівняння двох змінних

Операція		a > b			a ≥ b			a < b			a ≤ b			a = b			a ≠ b		
Вхід a		min	val	max	min	val	max	min	val	max	min	val	max	min	val	max	min	val	max
Вхід b	max	F	-	-	F	-	-	T	-	-	T	-	-	F	-	-	T	-	-
	val + δ	-	F	-	-	F	-	-	T	-	T	-	-	-	-	-	-	-	-
	val	-	*F*	-	-	*T*	-	-	*F*	-	-	*T*	-	-	*T*	-	-	*F*	-
	val - δ	-	T	-	-	T	-	-	F	-	-	F	-	-	-	-	-	-	-
	min	-	-	T	-	-	T	-	-	F	-	-	F	-	-	F	-	-	T
	val2	-	-	-	-	-	-	-	-	-	-	-	-	-	F	-	-	T	-

Тут a, *b* – програмні змінні; *val, val2* – значення із середини діапазону, отриманого перетином діапазонів значень змінних *a і b.*

2.5. Тестові плани

Тести не існують самі по собі – кожний тест перевіряє одну ситуацію в роботі системи, але вся сукупність тестів повинна повністю перевіряти всю функціональність системи. У зв'язку з цим описи тестів об'єднують у документи, що мають назву тестових планів.

Тестовий план є документом, у якому перелічено всі тести, що є необхідними для тестування системи, або частина тестів, які з'єднано за певною ознакою.

Тестовий план може бути написаний на звичайній або формальній мові, в останньому випадку можливе передання тестового плану на вхід тестувального оточення для автоматичного виконання наведених у тестовому плані тестів.

Існує кілька причин для об'єднання описів тестів у єдиний документ або декілька документів.

Єдина схема ідентифікації і трасування тестів. Оскільки тести пишуться на основі функціональних або тестових вимог, при тестуванні необхідно упевнитися, що для кожної вимоги існує хоча б один тест. Це досягається введенням єдиної схеми ідентифікації тестів (наприклад, наскрізної нумерації) і посилань на вимоги, на основі яких написано тест.

Об'єднання тестів у смислові групи. Тести, призначені для перевірки одних і тих самих модулів системи, раціонально об'єднувати в смислові групи. Це пов'язано з тим, що в таких тестах є дуже схожими вхідні дані й сценарії, а групування дає змогу виявити друкарські помилки й помилки в тестах.

Внесення змінень в тести. При зміненні системи, що тестується, під час її ЖЦ неминуче доводиться змінювати тести. Загальні огляди тестових вимог і тестових планів дають змогу виявити, які тести слід змінити або видалити, а в яких смислових групах створити нові тести, що будуть перевіряти нову функціональність.

Визначення послідовності тестування. Одна з важливих властивостей тесту – його незалежність. Результат виконання тесту не повинен змінюватися залежно від того, які тести виконувалися до нього. Зазвичай незалежність тестів досягається шляхом повної реініціалізації тестового оточення перед виконанням кожного нового тесту. Однак часто виникають ситуації, у яких для економії часу виконання тестів їх об'єднують у послідовності, у яких кожний наступний тест використовує стан тестового оточення або системи, що тестується, якого досягнуто під час попереднього тесту. Такі зв'язані тести слід окремо помітити для збереження коректного порядку їх проходження.

Розглянемо типову структуру тестового плану, написаного на звичайній мові, який містить тести для перевірки роботи модуля розрахунку контрольних сум. Кожний тест має унікальний номер і посилання на тестову вимогу, на основі якого його написано.

Загальний опис тесту допомагає при супроводі тестових планів – внесенні змінень при зміненні системи, інспекціях тестових планів, що виявляє неузгодженість, тощо.

У кожному тесті також обов'язково перелічено всі вхідні значення й очікувані вихідні значення, а також сценарій, у якому описано послідовність дій тестового оточення для виконання тестового завдання.

Тестовий план

Тест 1

Номер тестової вимоги: 2а, 2б.

Опис тесту: у цьому тесті перевіряється правильність обчислення значення контрольної суми (CRC) при ненульовому значенні поля CRC і нульових значеннях елементів запису.

Вхідні дані: CRC = 12345, A = 0, B = 0, C = 0, D = 0.
Очікувані дані: CRC = 0, A = 0, B = 0, C = 0, D = 0, Empty = TRUE.
Сценарій тесту:
1. Установка значення поля CRC в 12345.
2. Установка значень полів A – D в нуль.
3. Виклик функції Set_CRC.
4. Перевірка значень CRC на нуль і Empty на TRUE.
Тест 2
Номер тестової вимоги: 2a.
Опис тесту: у цьому тесті перевіряється відповідність алгоритму обчислення поля CRC, заданому в специфікації вимог.
Вхідні дані: CRC = 0, поля A – D заповнено байтами 01010101_2.
Очікувані вихідні дані: CRC = 0111100_2, Empty = FALSE.
Сценарій тесту:
1. Установка значення поля CRC в нуль.
2. Заповнення байтів A – D значеннями 01010101_2.
3. Виклик функції Set_CRC.
4. Перевірка значень CRC на 0111100_2 і Empty на FALSE.
Тест 3
Номер тестової вимоги: 2a.
Опис тесту: у цьому тесті перевіряється незмінність полів A – D запису при обчисленні поля CRC (підрахунку контрольної суми).
Вхідні дані: CRC = 0, A– D заповнено байтами 01010101_2.
Очікувані вихідні дані: A-D заповнено байтами 01010101_2.
Сценарій тесту:
1. Установка значення поля CRC в нуль.
2. Заповнення байтів A – D значеннями 01010101_2.
3. Виклик функції Set_CRC.
4. Перевірка значень байт полів A – D на 01010101_2.

Така структура тестового плану дає змогу описувати тести з абсолютно різними наборами вхідних і вихідних даних і сценаріями, проте при великій кількості тестів ця схема є дуже громіздкою.

2.6. Оцінювання якості коду, що тестується

Під час виконання кожного тестового завдання тестове оточення порівнює очікувані й реальні вихідні значення. Якщо ці значення збігаються, тест уважається пройденим, оскільки система видала саме ті вихідні значення, які очікувалися, інакше тест буде непройденим.

Кожний непройдений тест може допомогти виявити потенційний дефект у системі, що тестується, а загальна їх кількість дає змогу

оцінювати якість програмного коду і обсяг змін, які необхідно в нього внести для усунення дефектів.

Для проведення такого інтегрального оцінювання після виконання всіх тестових прикладів тестовим оточенням збирається статистика виконання, яка записується у файл звіту про виконання тестів. Існує декілька ступенів докладної статистики виконання тестів, з яких кожна наступна є розширенням попередньої:

– відображня кількості пройдених і непройдених тестів, а також їх загальної кількості;

– додатково до попередньої стратегії відображення іденти-фікаторів невиконаних тестових завдань дає змогу локалізувати тести, що потенційно виявили дефект;

– додатково до попередньої стратегії відображення очікуваних і реальних вихідних даних, що не збіглися, дає змогу проводити більш глибоке аналізування причин неуспішного проходження тестів;

– повне відображення очікуваних і реальних вихідних даних з відмітками про збіг і незбіг та відмітками про успішне (неуспішне) завершення для кожного з тестів.

2.6.1. Покриття програмного коду

Поняття покриття

Одним із параметрів, за якими оцінюється якість системи тестів – це її повнота, тобто відносна величина частки функціональності системи, що перевіряється тестами. Зазвичай за міру повноти беруть відношення обсягу перевіреної частини системи до її обсягу в цілому. Повна система тестів дає змогу стверджувати, що система реалізує всю функціональність, наведену у вимогах, і, що є ще більш важливим, – не реалізує ніякої іншої функціональності.

За одним із методів повнота системи тестів визначається як відношення кількості тестових вимог, для яких існують тести, до їх загальної кількості, тобто у цьому випадку йдеться про покриття тестами тестових вимог. Одиницею вимірювання ступеня покриття є відсоток тестових вимог, для яких існують тести, що має назву відсотка покриття тестовими вимогами.

Покриття вимог дає змогу оцінити ступінь повноти системи тестів відносно функціональності системи, але не її програмної реалізації. Одну й ту саму функцію можна реалізувати з допомогою абсолютно різних алгоритмів, що потребують різного підходу до організації тестування.

Для більш детального оцінювання повноти системи тестів при тестуванні методом скляної скрині аналізується покриття програмного коду, що має назву структурного покриття.

Під час роботи кожного тесту виконується деяка ділянка програмного коду системи, при проведенні всієї системи тестів – усі ділянки програмного коду, які задіює ця система тестів. Якщо є ділянки коду, які не виконано при проведенні системи тестів, то ця система є потенційно неповною (тобто не перевіряє всю функціональність системи) або містить ділянки захисного коду, що не використовується (наприклад, «закладки» або фрагменти коду, які призначено для подальшої модифікації системи). Таким чином, відсутність покриття яких-небудь ділянок коду є сигналом до переоблення тестів або коду (а іноді й вимог).

Аналізування покриття програмного коду можна розпочинати тільки після досягнення повного покриття вимог. Повне покриття програмного коду не є гарантією того, що тести перевіряють усі вимоги до системи. Одна з типових помилок малодосвідченого тестувальника – починати з покриття коду, забуваючи про покриття вимог.

Покриття операторів програмного коду

Для забезпечення повного покриття програмного коду на рівні операторів необхідно, щоб унаслідок виконання системи тестів кожний оператор було виконано хоча б один раз.

Особливість цього рівня покриття полягає в тому, що на ньому утруднено аналізування покриття деяких керувальних структур.

Наприклад, для повного покриття коду

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

достатньо одного тесту:

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

Навіть якщо у складі тестів не буде тесту, що перевіряє роботу фрагмента при значенні *condition = false*, то код буде покритим. Проте у разі *condition = false* виконання фрагменту спричинить помилку.

Аналогічні проблеми виникають під час перевірки циклів *do*, while – при такому рівні покриття достатньо виконати цикл тільки один раз, при цьому метод є абсолютно нечутливим до логічних операцій AND/OR.

Іншою особливістю цього методу є залежність рівня покриття від структури програмного коду. На практиці 100-відсоткове покриття коду часто не потребується, достатньо встановити допустимий рівень покриття, наприклад 75-відсотковий. Проблеми можуть виникнути при покритті такого фрагмента:

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

Якщо functionA() містить 99 операторів, a functionB() – один оператор, то єдиного тестового завдання, що встановлює *condition* в *true*, буде достатньо для досягнення необхідного рівня покриття. При цьому

аналогічний тест, що встановлює значення *condition* в *false,* дасть дуже низький рівень покриття.

Покриття гілок умовних операторів

Для забезпечення повного покриття гілок умовних операторів кожна точка входу й виходу в програмі й у всіх її функціях має бути виконана принаймні один раз і всі логічні вирази в програмі набути кожного з можливих значень хоча б один раз. Таким чином, для цього покриття потрібно щонайменше два тестових завдання.

На відміну від попереднього рівня покриття у цьому методі враховано покриття умовних операторів з порожніми гілками. Так, для покриття ділянки програмного коду

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

потрібні такі тести:

Тест 1. вхід: condition = true; очікуваний вихід: a = 1;

Тест 2. вхід: condition = false; очікуваний вихід: a = 0;

Особливістю покриття є те, що в ньому не враховуються логічні вирази і значення компонент, які отримують завдяки викликам функцій. Наприклад, для фрагмента

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

повного покриття гілок можна досягти такими тестами:

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

В обох тестах не працює *function1()*, але покриття коду буде повним. Для перевірки виклику *function1()* необхідно додати ще один тест:

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

Для більш повного аналізу компонентів умов в логічних операторах існує декілька методів, у яких ураховано структуру компонентів умов і значення, яких вони набувають при виконанні тестів.

Покриття умов

Для забезпечення повного покриття кожний компонент логічної умови внаслідок виконання тестів повинен набувати всіх можливих значень, але при цьому не потрібно, щоб сама логічна умова набувала всіх можливих значень, наприклад:

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

Для покриття умов потрібні такі тести:

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

При цьому логічна умова буде набувати тільки true. Таким чином, при повному покритті умов не буде досягатися покриття гілок.

Покриття гілок/умов

Для забезпечення повного покриття за цим методом необхідно, щоб логічна умова і кожний з її компонентів набували всіх можливих значень.

Для покриття попереднього фрагмента з умовою *condition1 || condition2* потрібні такі тести:

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

Однак ці тести не дають змоги протестувати правильність логічної функції – замість OR у коді могло бути помилково записано AND.

Покриття за всіма умовами

Для виявлення неправильно заданих логічних функцій було запропоновано метод покриття за всіма умовами. За цим методом покриття має бути перевірено всі можливі набори значень компонентів логічних умов. Іншими словами, для n компонентів потрібно буде 2n тестів, кожний з яких перевіряє один набір значень. Тести, що є необхідними для повного покриття за цим методом, дають повну таблицю істинності для логічного виразу.

Незважаючи на очевидну повноту системи тестів, що забезпечує такий рівень покриття, цей метод нечасто застосовують на практиці через його складність.

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

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

потрібна різна кількість тестів: для першої умови – шість, а для другої – 11.

Метод MC/DC

Для зменшення кількості тестів логічних умов фірмою Boeing було розроблено модифікований метод покриття гілок (умов) (Modified Condition/Decision Coverage або MC/DC) [22]. Цей метод широко використовують при верифікації бортового авіаційного ПЗ в США.

Для забезпечення повного покриття необхідне виконання таких умов:

– кожна логічна умова повинна приймати всі можливі значення;

– кожний із компонентів логічної умови повинен хоча б один раз набувати всі можливі значення;

– має бути показано незалежний вплив кожного з компонентів на значення логічної умови, тобто вплив при фіксованих значеннях інших

компонентів.

Покриття потребує досить великої кількості тестів, для того щоб перевірити кожну умову, яка може вплинути на результат виразу, проте ця кількість значно менша, ніж потребується для методу покриття за всіма умовами. У табл. 4, 5 наведено приклади тестових наборів, необхідних для тестування логічних блоків за MC/DC. Так, наприклад, для блоку OR, який має n входів, достатньо n + 1 тестів. Перший тест свідчить, що при нульових значеннях входів значення виходу також буде нульовим. У кожному з наступних n тестів значення кожного входу встановлюється на одиницю, що свідчить про незалежний вплив входів на значення виходу.

Таблиця 4

Тести для багатовходових AND і NOT-AND блоків

Вхід/ Вихід	Номер набору тестових даних											
	Багатовходовий AND блок						Багатовходовий NOT-AND блок					
	1	2	3	4	…	n+1	1	2	3	4	…	n+1
Вхід 1	T	F	T	T	···	T	T	F	T	T	···	T
Вхід 2	T	T	F	T	···	T	T	T	F	T	···	T
Вхід 3	T	T	T	F	···	T	T	T	T	F	···	T
…	…	…	…	…	…	…	…	…	…	…	…	…
Вхід n	T	T	T	T	···	F	T	T	T	T	···	F
Вихід	T	F	F	F	···	F	F	T	T	T	···	T

Таблиця 5

Тести для багатовходових OR і NOT-OR блоків

Вхід/ Вихід	Номер набору тестових даних											
	Багатовходовий OR блок						Багатовходовий NOT-OR блок					
	1	2	3	4	…	n+1	1	2	3	4	…	n+1
Вхід 1	F	T	F	F	···	F	F	T	F	F	···	T
Вхід 2	F	F	T	F	···	F	F	F	T	F	···	T
Вхід 3	F	F	F	T	···	F	F	F	F	T	···	T
…	…	…	…	…	…	…	…	…	…	…	…	T
Вхід n	F	F	F	F	···	T	F	F	F	F	···	T
Вихід	F	T	T	T	···	T	T	F	F	F	···	F

2.6.2. Аналіз покриття

Метою аналізу повноти покриття коду є виявлення його ділянок, які не виконуються при виконанні тестів. Тести, що базуються на вимогах, можуть не забезпечувати повного виконання всієї структури коду. Тому для поліпшення покриття аналізується повнота покриття коду тестами і, якщо це необхідно, проводяться додаткові перевірки, спрямовані на

визначення причин недостатнього покриття і шляхів їх усунення. Зазвичай аналізування покриття виконують з урахуванням таких угод:

– аналізування має підтвердити, що повнота покриття тестами структури коду відповідає необхідному виду покриття і заданому мінімально допустимому відсотку покриття;

– аналізування повноти покриття тестами структури коду можна виконати з використанням програмного коду, якщо ПЗ не належить до рівня А (див. розд. 1). Для рівня А необхідно перевірити об'єктний код, чи трасується він у програмний код. Якщо об'єктний код не трасується, то необхідно провести перевірки об'єктного коду щодо правильності генерації послідовності команд. Прикладом об'єктного коду, який прямо не трасується в початковий текст, але генерується компілятором, може бути перевірка виходу за задані межі масиву;

– під час аналізування має підтвердитися правильність передачі даних і керування компонентами коду.

Під час аналізування повноти покриття тестами можуть виявитися фрагменти коду, що не тестувалися. У цьому випадку потрібні додаткові дії для перевірки ПЗ. Ці фрагменти коду можуть бути результатом такого:

– недоліки у формуванні тестів або тестових процедур, що базуються на вимогах; у цьому випадку слід доповнити набір тестів або змінити тестові процедури для забезпечення покриття всього коду. При цьому можливим є перегляд методу (методів), що використовується для аналізування повноти тестів на основі вимог;

– неадекватність у вимогах на ПЗ; у цьому випадку повинні бути модифіковані вимоги на ПЗ, створені та виконані додаткові тести і тестові процедури;

– «мертвий» код – цей код слід видалити і проаналізувати програмний код для оцінювання ефекту видалення «мертвого» фрагменту, крім того весь програмний код необхідно повторно перевірити;

– код, що дезактивувався, – для коду, що дезактивувався, і який не передбачено для виконання в кожній конфігурації, поєднання аналізу й тестів має продемонструвати можливості засобів, що запобігають ненавмисному виконанню такого коду, ізолюють або усувають його. Для коду, що дезактивувався, який виконується тільки при певних конфігураціях, слід встановити нормальну експлуатаційну конфігурацію для виконання цього коду і для неї розробити додаткові тести й тестові процедури, що задовольняють повноті покриття тестами структури коду;

– надмірність умови – логіку роботи такої умови треба переглянути. Наприклад, в умові if(A && B || !B) принципово неможливо перевірити, що A && B буде False у разі, коли A=True і B=False, оскільки друга частина умови (!B) має значення True і загальний результат буде True;

– захисний код – ця частина коду використовується для запобігання винятковим ситуаціям, які можуть виникнути під час роботи програми. Це

може бути, наприклад, гілка default в операторі вибору switch, причому вхідна умова оператора switch може набувати певних значень, які він описує, і, як наслідок, гілку default ніколи не буде виконано.

2.7. Повторюваність тестування

2.7.1. Мета і задачі забезпечення повторюваності тестування

Тестування програмної системи – не разовий захід, а постійний процес, який є активним протягом усього ЖЦ розроблення системи. Протягом цього процесу система неминуче змінюється або внаслідок виправлення помилок, або розширення її функціональності. Задача тестувальника в такій ситуації – підтвердити, що нова або виправлена функціональність не спричинила нових помилок, а якщо помилки все ж таки виникли – визначити причини їх виникнення.

Найпростіший, але водночас дієвий спосіб такого підтвердження – повне виконання всіх тестів після кожного істотного змінення системи й порівняння результатів виконання тестів до і після змінення.

Якщо результати виконання тестів до внесення змінення були позитивними (усі тести проходили успішно), то неуспішно пройдені тести можуть означати, що в системі виникли нові дефекти, спричинені виправленням попередніх.

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

1. Усі тести пройдено успішно. У цьому випадку змінення не стосуються вже протестованих функцій, але може потребуватися розроблення нових тестів для нових функцій системи.

2. Частина тестів, що раніше виконувалися успішно, завершується з негативним результатом. Причин цьому може бути декілька:

– коректне змінення функціональності системи, що тестується, унаслідок якого тестове завдання перестало відповідати вимогам;

– некоректне змінення функціональності системи, унаслідок якого тестовий приклад виявив розбіжність з вимогами;

– вплив залишкових даних від попередніх тестових завдань, які залишалися раніше непоміченими.

3. Виконання тестів аварійно завершується на самому початку або при виконанні певного тесту. Ця проблема частіше за все пов'язана зі зміненням зовнішнього оточення частини системи, що тестується, яке моделює тестове оточення. Унаслідок таких змінень можуть змінюватися зовнішні інтерфейси, а також склад і формат вхідних і вихідних даних. Після цього 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, які система видає в новому форматі.

Перші дві причини можна помітити тільки з допомогою аналізування змінень у функціональних і тестових вимогах, а також поточного стану тестувальних планів і тестувального оточення. За результатами цього аналізування в першому випадку тестувальник вносить змінення в тест (можливо, розробляються нові тестові завдання), у другому випадку тестувальник повідомляє розробників про наявність дефекту.

Якщо для виконання тестів потрібно об’єднати програмні модулі тестового оточення і системи в єдиний код, то при зміненні інтерфейсів системи може виникнути ситуація, коли неможливо не тільки виконати тести, а навіть об’єднати оточення й системи. У цьому випадку також необхідно проаналізувати змінення, внесені в систему, і модифікувати відповідно до них тестове оточення.

У деяких випадках повторно виконати всі тести неможливо. Це може бути пов'язане з великим часом виконання всіх тестів і обмеженим часом, який відведено на процес тестування. У цьому випадку часто застосовують вибіркове тестування окремих частин системи, які було змінено. Повне тестування при такому підході проводиться тільки після накопичення достатньо великої кількості змінень або на ключових стадіях проекту.

Процес, що містить повторне виконання тестів, називають регресійним тестуванням, яке складається з таких стадій:

– аналізування змін у системі;
– вибір тестів для перевірки системи;
– виконання тестових завдань;
– аналізування результатів виконання;
– модифікація тестового оточення, тестів або сповіщення розробників про дефект системи тестування.

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

– забезпечення можливості повного виконання всіх тестів, які перевіряють функціональність системи, або проведення аналізу, що дає змогу виявити тести, які необхідно повторно виконати для тестування функціональності, що змінилася;
– розроблення тестів і тестового оточення з використанням методик, які полегшують модифікацію при зміненнях в системі, що тестується;
– розроблення тестів, структура яких повністю виключає їхній взаємний вплив за залишковими даними.

Для забезпечення повторюваності тестування потрібне постійне забезпечення тестувальників і розробників актуальною інформацією про поточний стан системи і коректність змін, внесених під час розроблення системи.

2.7.2. Передумови для виконання тестів

Вхідні дані в кожному тесті явно задають початковий стан системи, що тестується, і режими її роботи при виконанні тестового сценарію. Проте неявний вплив на виконання тесту має і стан тестового оточення. Під станом тут розуміється набір параметрів, змінення будь-якого з них може вплинути або на результат виконання тесту, або на можливість його коректної роботи й завершення. Наприклад, для виконання тесту системі може потребуватися значна місткість дискової або оперативної пам'яті. Якщо перед виконанням тесту тестове оточення зарезервує цю пам'ять під свої потреби, виконання тесту стане неможливим. Така ж сама ситуація може виникнути й у випадку, якщо оточення не звільнить пам'ять після виконання попереднього тесту.

Цієї інформації зазвичай немає в тестових планах, проте необхідний для виконання тестових завдань стан тестового оточення слід враховувати при розробленні тестів.

Потрібно оформити перевірки на допустимість стану тестового оточення у вигляді передумов для виконання тесту. Це дає змогу діагностувати ситуації, що виникають під час вибіркового тестування й спричиняють відмови тестового оточення.

Для полегшення проведення регресійного тестування (і тестування взагалі) тести поділяють на групи. Кожна група містить набір тестів, які перевіряють окрему локальну частину функціональності системи. Тести для часткового регресійного тестування слід відбирати відразу групами.

Для полегшення проведення вибіркового регресійного тестування кожне тестове завдання має бути повністю автономним – хід його виконання і тим паче результат не повинні залежати від попередніх тестових завдань. Тим самим при вибірковому тестуванні результат тестування не буде залежити від вибраного набору тестів (тестового набору). Проте на практиці створення автономних тестів часто є неможливим з різних причин (зазвичай через довгий час їх виконання).

У разі, коли в наборі тести не є автономними, кажуть про тестову залежність. Існує два види тестової залежності – така, що передбачена структурою тестів, і паразитна.

Часто коректність виконання тестів визначається порядком їх виконання. Така тестова залежність потребує документування й супроводу, як і описи тестів. Існують два види документування тестової залежності:

– явне визначення допустимого порядку виконання тестів;

– визначення допустимого порядку виконання тестів з допомогою передумов.

Перший спосіб є більш зручним при порівняно невеликій загальній кількості тестів, а у разі розбиття групи – при невеликому розмірі груп

тестів. При другому способі коректність порядку виконання тестів визначається шляхом перевірки того, що або система, яка тестується, або тестове оточення знаходиться в необхідному стані для виконання тесту.

Паразитну тестову залежність зазвичай спричиняє некоректне складання тестових планів. Паразитна залежність (як і передбачувана) виявляється в тому, що кілька тестів коректно працюють тільки в тому випадку, якщо до них було виконано інші тести, причому така залежність не була передбачена тестувальником. Природа паразитної тестової залежності схожа з природою помилок використання неініціалізованих або залишкових даних в динамічній пам'яті при програмуванні.

2.8. Документування верифікації і тестування

2.8.1. Проектна документація і її призначення

Під час роботи над проектом складної програмної системи створюється велика кількість проектної документації. Основне її призначення – координація дій великої кількості розробників протягом більш-менш тривалих проміжків часу: під час первинного розроблення системи, виконання робіт з її модифікації, супроводу. Структурний склад проектної документації в більшості проектів є майже однаковим – це вимоги до систем різного рівня (системні, функціональні й структурні), опис архітектури, програмний код, тести і документи, що супроводжують процес впровадження (інструкції щодо установлення, налагодження, які призначено для користувача).

Оскільки верифікація програмної системи виконується протягом всього життєвого циклу розроблення 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 вибраних критеріїв оцінювання якості, всі аспекти функціонування системи протестовано. Крім того, тестова документація використовується при внесенні змін в систему для перевірки того, що як попередня, так і нова функціональність є коректними.

Перед початком верифікації менеджером тестування створюється документ, що має назву плану верифікації (або плану тестування, але це не те саме, що тестувальний план). План тестування – організаційний документ, що містить вимоги до виконання тестування в конкретному проекті. У ньому визначено загальні підходи до узгодження процесів розроблення і верифікації, методики проведення верифікації, склад тестової документації та її взаємозв'язок з документацією розробників, терміни різних етапів верифікації, різні функції і кваліфікація тестувальників, які необхідні для виконання всіх робіт з тестування, вимоги

до інструментів тестування і тестових стендів, оцінено ризики й наведено шляхи для їх подолання.

У цьому документі також визначено вимоги власне до тестової документації – до тестових вимог, тестувальних планів, звітів про виконання тестування.

На підставі тестових вимог і проектної документації розробників також створюється тестове оточення, необхідне для коректного виконання тестів на тестових стендах – драйвери, заглушки, настроювальні файли тощо.

Після виконання тестів (автоматично або вручну) складаються звіти про тестування, що містять інформацію про невідповідності вимогам, які було виявлено під час тестування, а також звіти про покриття, що містять інформацію про те, яку частку програмного коду системи було задіяно під час виконання тестування.

Після виявлення невідповідностей створюються звіти про проблеми – документи, які надходять для аналізування в групу розробників з метою визначення причини виникнення невідповідності.

Змінення в систему вносяться після всебічного вивчення цих звітів і локалізації проблем, що спричинили невідповідність вимогам. Для того щоб процес контролювався і будь-яке змінення протоколювалося (і зв'язувалося з тестами, з допомогою яких було виявлено проблему), створюється запит на змінення системи (рис. 14). Після завершення всіх робіт за запитом на змінення процес тестування повторюється доти, доки не буде досягнуто прийнятний рівень якості програмної системи.

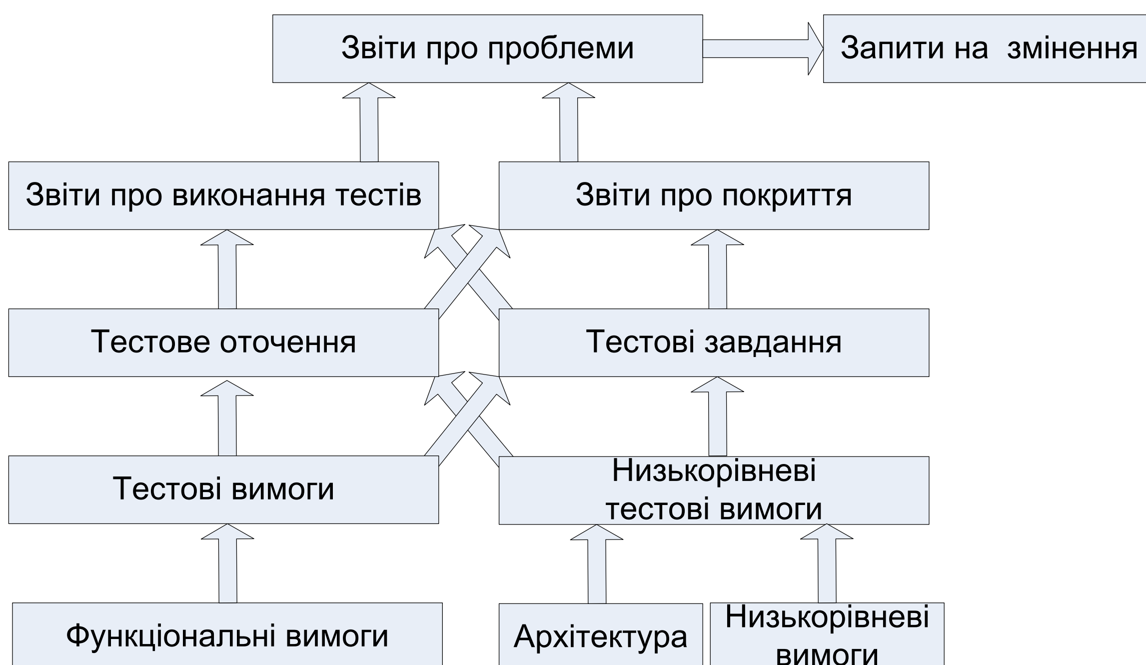

Рис. 14. Документація процесу верифікації

Слід особливо зазначити, що всі документи повинні мати унікальні ідентифікатори і зберігатися в єдиній базі документів проекту. Це дасть змогу зберегти керованість процесом тестування і підтримувати необхідну якість системи, що розробляється. Немає гіршої ситуації, коли знайдену проблему не було виправлено через те, що звіт про неї загубився і не потрапив до розробника.

2.8.2. Стратегія і плани верифікації

Першим документом, що належить до технологічної документації процесу верифікації, є стратегія тестування. Стратегія верифікації визначає загальні підходи і методики верифікації, необхідні рівні верифікації проектної документації і програмного коду, технології та інструментальні засоби.

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

Для кожного етапу визначаються:

– типи вхідних і вихідних документів;
– загальна процедура верифікації;
– функції і відповідальності посадових осіб;
– формати й угоди щодо ідентифікації вихідних документів;
– критерії оцінювання результативності етапу.

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

– план верифікації системних вимог;
– план верифікації архітектури;
– план тестування програмного коду;
– план тестування й інтеграції модулів;
– план системного тестування;
– план тестування навантаження;
– план напівнатурних випробувань;
– план приймально-здавальних випробувань.

Основна задача плану тестування як документа – визначення меж тестування, підходу до тестування, ресурсів, що потребуються для тестування, плану-графіка тестування. У плані тестування визначаються елементи, що тестуються, і функції системи, а також завдання, що вирішуються під час тестування, співробітники, які відповідають за тестування, і ризики, пов'язані з цим планом. Така форма плану тестування є досить повною і містить не тільки технічні аспекти, пов'язані власне з описом тестових прикладів, але й організаційні, пов'язані із

загальним керуванням процесом тестування. На практиці обсяги технічних і організаційних розділів планів тестування можуть досить сильно варіюватися. Проте мінімально необхідні елементи, які рекомендується містити в кожний план тестування, є такими:

– ідентифікатор плану тестування і номер його версії, який дає змогу однозначно знаходити потрібний план тестування і його останню актуальну версію в базі даних проекту;

– загальний опис тестувального плану;

– посилання на інші документи – стандарти, плани тестування, тестові вимоги, результати виконання тестів;

– визначення областей системи з посиланням на проектну документацію, програмний та об'єктний коди, що піддаються верифікації з визначенням її типу (автоматизовані тести, формальні інспекції, тестування на моделях, напівнатурні випробування тощо);

– визначення підходів до тестування – загальних методик, яких слід дотримуватися при тестуванні системи; незважаючи на те, що більшість тестів можуть досить сильно різнитися, загальні методи й підходи до їхньої побудови можуть бути єдиними;

– критерій успішності/неуспішності проходження тестів (pass/fail);

– тестові документи – зазвичай план тестування має в додатку всі тестові документи нижчого рівня – тестові вимоги, тестові плани, результати тестування, дані про покриття; у випадку, коли заносити ці документи до складу плану тестування недоцільно (наприклад, у разі їх значного обсягу), рекомендується поміщати посилання на ці документи;

– вимоги до середовища тестування, де описано вимоги до апаратних і програмних засобів, необхідних для проведення тестування. Якщо ПЗ є вбудованим, то програмна система працює на спеціальному апаратному забезпеченні, а інструментальні засоби для тестування – на звичайних PC загального призначення. Для виконання тестування в таких умовах потрібне використання або емуляторів, або програмно-апаратного комплексу для сполучення спеціального апаратного забезпечення з PC. Крім того, до складу програмних засобів тестування зазвичай належать крос-засоби розроблення. У випадку, коли тестується система загального призначення, у цьому розділі просто перелічуються вимоги до обладнання, необхідного для тестування, які зазвичай є дещо вищими, ніж вимоги до обладнання, достатнього для «штатної» роботи системи;

– людські ресурси і рівень їх підготовки, де наводяться склад групи тестування, необхідний для успішного завершення тестування в поставлені терміни, а також знання, які необхідні виконавцям;

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

– список ризиків, які можуть перешкоджати завершенню тестування в строк або з необхідним рівнем якості. Зазвичай для кожного ризику оцінюється ймовірність його виникнення, а також наводяться загальні

шляхи, з допомогою яких можна уникнути ризику або ліквідувати його наслідки.

Стратегія і плани тестування дещо відрізняються від іншої документації, що належить до процесу тестування. Перш за все це пов'язано з тим, що в цих документах досить багато уваги приділяється тому, як має бути організовано процес тестування, а не тому, як тестувати саму систему.

2.8.3. Тестові вимоги

Тестові вимоги – основний документ для тестувальника, який визначає функціональність системи з огляду на те, що має бути перевірено, щоб упевнитися в її коректному функціонуванні, а також на підставі якого зовнішнього ефекту можна переконатися, що функції реалізовано коректно.

Існує два підходи до написання тестових вимог – функціональні й структурні. Тестові вимоги, написані на основі функціонального підходу, грунтуються на системних вимогах і вимогах до ПЗ системи.

Тестові вимоги, які базуються на структурному підході, пишуться з урахуванням архітектури системи і програмних кодів (рис. 15). Через таку відмінність функціональний і структурний підходи часто називають підходами чорної і скляної скринь. Структурні тестові вимоги є важливими для систем з підвищенними вимогами до надійності, тобто для випадків, коли необхідно перевірити не тільки, наскільки коректно система в цілому відпрацьовує сценарії своєї роботи (коректні й некоректні з погляду користувача), але і як у різних нестандартних ситуаціях будуть функціонувати окремі її компоненти.

На практиці майже завжди застосовують обидва підходи до розроблення тестових вимог, унаслідок чого створюються тестові вимоги верхнього і нижнього рівнів, за якими складаються тестові плани.

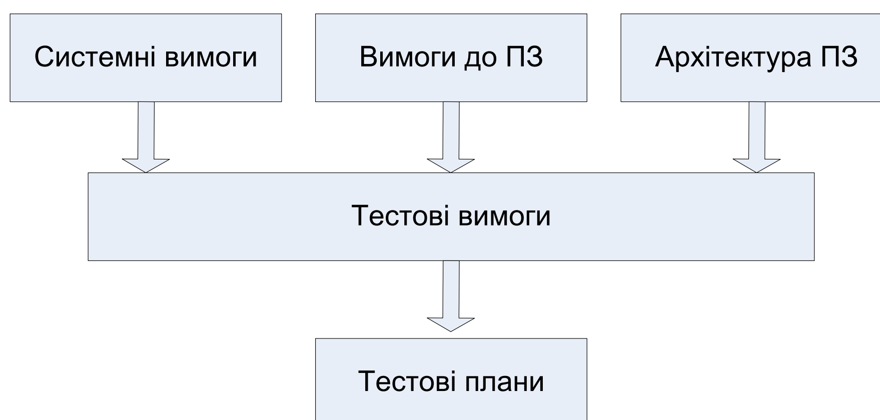

Рис. 15. Місце тестових вимог у проектній документації

Зазвичай структура розділу тестових вимог відповідає структурі розділу функціональних вимог на систему. Задача кожної вимоги полягає у визначенні того, *що* саме має бути перевіреним. Техніка виконання кожної такої перевірки – головна задача тестового плану. Звичайний формат опису окремої вимоги є таким:

а) перевірити, що у випадку певних зовнішніх впливів відбувається певна реакція програмного забезпечення;

б) тестові вимоги, написані у межах функціонального підходу, зазвичай поділяються на такі групи:

– контроль вхідних даних;
– оброблення помилок (введення, обчислень);
– отримання основного результату;
– оброблення особливих ситуацій;
– оформлення і висновок результатів.

Конкретизація програмної реалізації може потребувати уточнення або розширення реакцій на різні ситуації, що виникають під час функціонування програмної системи. У цьому випадку рекомендується оформити додаткові тестові вимоги низького рівня для структурної перевірки системи.

Сукупність тестових вимог має характеризуватися повнотою, несуперечністю і здійсненністю верифікації.

Зазвичай одній системній або функціональній вимозі відповідає щонайменше одна тестова вимога. Якщо усі перевірки, що задаються тестовими вимогами, покривають всю функціональність системи, визначену в системних вимогах і вимогах до ПЗ, то кажуть про повноту тестових вимог. При зміненні вимог до системи для підтримки повноти мають змінюватися й тестові вимоги.

Як системні вимоги до ПЗ, так і тестові вимоги мають бути записані у формі, що дає можливість здійснення верифікації, тобто для кожної вимоги має існувати можливість визначити чіткий критерій перевірки, який дасть змогу перевирити, чи виконується ця вимога в реалізованій системі.

Прикладом вимоги, що неверифікується, може бути така: с*истема повинна мати інтуїтивно зрозумілий інтерфейс, який призначено для користувача.*

Текст вимоги потрібно переписати таким чином: *перевірити, що система має інтуїтивно зрозумілий інтерфейс, який призначено для користувача*

Без чіткого визначення критеріїв інтуїтивної зрозумілості перевірити таку вимогу з допомогою написання тестів неможливо. Проте якщо супроводжувати таку вимогу кількісними або якісними характеристиками інтуїтивно зрозумілого інтерфейсу, то написання тестів за вимогами стає можливим. Так, серед критеріїв інтуїтивної зрозумілості можуть бути такі:

глибина вкладеності меню не більше трьох, наявність спливних підказок на кожному елементі керування кожної екранної форми тощо.

При великій кількості тестових вимог і частих змінах може виникнути ситуація, у якій різні вимоги перестають бути узгодженими. У цьому випадку такі вимоги мають критерії перевірки, що взаємовиключають один одного. Наприклад, у простому випадку одна тестова вимога на призначений для користувача інтерфейс може декларувати необхідність перевірки того, що введений користувачем пароль має довжину не більше 16 символів, а тестова вимога до бази даних системи – те, що допустимий розмір пароля, який 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 тестових вимог. Тестові вимоги, за якими складаються тестові плани для тестування системи, є несуперечними, оскільки суперечність зазвичай усувається на рівні верифікації проектної документації. Проте її можна виявити й пізніше, унаслідок спроби створити адекватні тести.

2.8.4. Тестові плани

Зв'язок тестових планів з іншою проектною документацією

На основі тестових вимог складаються тестові плани – програми випробувань (перевірки, тестування) програмної реалізації системи. На відміну від тестових вимог в тестових планах описуються конкретні способи перевірки функціональності системи, тобто те, як повинна перевірятися функціональність. Зазвичай тестовий план складається з окремих тестів, кожний з яких перевіряє деяку функцію або набір функцій системи. Для кожного тесту однозначно знаходиться критерій успішного проходження (pass/fail criteria), з допомогою якого можна визначити відповідність функціонування системи заданій у вимогах.

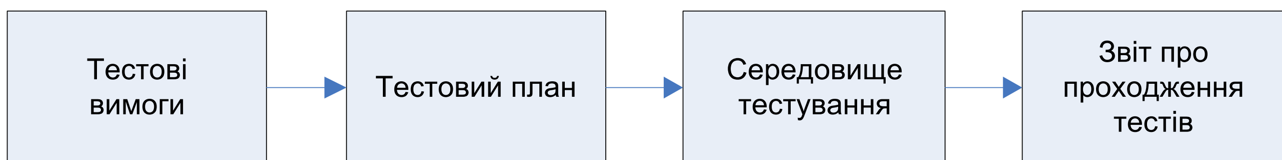

Рис. 16. Місце тестових планів у проектній документації

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

Структура тестового плану може відповідати структурі тестових вимог або логіці зовнішнього функціонування системи. У кожному пункті тестового плану описується, як проводиться перевірка правильності функціонування програмної реалізації, і містяться:

– посилання на вимоги, що перевіряються у цьому пункті;
– конкретна вхідна дія на програму (значення вхідних даних);
– очікувана реакція програми (повідомленя, значення результатів);
– послідовність дій, необхідних для виконання пунктів плану.

До складу тестового плану рекомендується додатково включати пункти, за якими можна перевірити гілки програми, що не виконувалися при перевірці задоволення функціональних вимог. Такі пункти тестового плану можуть мати позначку «Для повноти покриття».

Тестовий план може готуватися у формалізованій формі й бути вхідним документом для тестового оточування, за яким тести будуть виконуватися в автоматичному режимі з автоматичною фіксацією результатів. У випадку, коли тестовий план готується у вигляді текстового документа, можливим є тільки ручне тестування системи.

Форми подання тестових планів

Форма подання тестового плану залежить від того, яким чином його буде використано під час тестування. При ручному тестуванні зручно подавати тестові плани у вигляді текстових документів. Кожний тест у такому разі містить перелік послідовності дій, які необхідно виконати для проведення тестування: сценарій тесту, а також очікувані відгуки системи на ці дії. Така форма подання тестового плану є незручною для автоматизації тестування, оскільки природна мова майже не піддається формалізації.

Для автоматизованого тестування сценарій тесту може записуватися на якій-небудь формальній мові, у цьому випадку можна безпосередньо використати тестові плани як вхідні дані для середовища тестування.

Іншою формою подання тестових планів є таблиця, яка найбільш часто використовується при чітко і формально визначених вхідних потоках даних системи. Кожний стовпець таблиці може бути тестом, кожний рядок – описом вхідного потоку даних. Очікувані значення для цього тесту записуються в аналогічній таблиці, у якій в рядках перелічуються вихідні потоки даних.

Третьою формою подання тестів є визначення їх у вигляді кінцевого автомата. Така форма використовується при тестуванні протоколів зв'язку або програмних модулів, які взаємодіють із зовнішнім світом з допомогою обміну повідомленнями за наперед заданим інтерфейсом. Модуль при цьому можна подати як кінцевий автомат з набором станів, а тестовий план буде складатися з двох частин – опису переходів між станами і їхніми

параметрами та тестових завдань, у яких задається маршрут переходу між станами, параметри переходів і очікувані значення. Таке подання тестового плану може бути придатним як для ручного, так і для автоматизованого тестування.

2.8.5. Тестові сценарії

Сценарії є дуже зручними для ручного тестування, при цьому тестовий план має вигляд текстового документа, у якому кожне тестове завдання наведено в окремому розділі. Для кожного тестового завдання записується:

– ідентифікатор;
– опис тесту і його мета;
– посилання на частину системи, що тестується;
– посилання на використану проектну доументацю (тестові вимоги);
– перелік дій сценарію;
– системна реакція, що очікується на кожний пункт сценарію.

Ідеться про те, що дії сценарію необхідно описати так, щоб їх могла відтворити людина з будь-яким рівнем підготовки.

Характеристику очікуваної реакції системи необхідно записати так, щоб можна було однозначно судити про те, чи відповідає реакція очікуваній.

Іноді такі тестові плани поєднують зі звітами про проведення тестування, додаючи до таблиці опису сценарію третій і четвертий стовпець – «Реальний результат» і «Відповідає», до якого заносяться реальна реакція системи і вказівка на збіг/незбіг результатів. У кінці опису кожного тесту додається графа «Пройдений/не пройдений», до якої заноситься інформація про те, чи пройдено тест у цілому. У кінці всього тестового плану, поєднаного зі звітом, розташовують графу «Тестові приклади пройдено/всього», до якої заноситься загальна й пройдена кількість тестів.

Сценарії тестування для автоматичного тестування часто описують на тій або іншій мові програмування. Можливою є і більш близька до природної мови форма підготовки тестів.

Найчастіше сценарії кожного тесту складаються з послідовності викликів функцій, що надсилають дані у середовище тестування.

2.8.6. Тестові таблиці

Табличне подання тестів є зручним при чітко формалізованих вхідних і вихідних потоках даних системи. Таблиці використовують для спрощення роботи з підготовки й супроводу великої кількості однотипних тестів. Тестове оточення, у якому використовується табличний опис тестів як

вхідних даних, має інтерпретатор таблиць, що перетворює її на послідовність команд, які виконує середовище для проведення тестування, тобто своєрідний сценарій.

У разі, коли однотипними є не тільки вхідні й вихідні дані, але і їхні значення, може використовуватися альтернативна форма подання табличних даних. Тести в ній також нумеруються по горизонталі, а вхідні потоки даних – по вертикалі. Проте під кожним із потоків даних перелічуються можливі вхідні значення, а спеціальною міткою позначають, що це вхідне значення необхідно подати в певне тестове завдання.

2.8.7. Кінцеві автомати

Форма тестових планів у вигляді кінцевих автоматів є зручною при тестуванні програмних модулів або систем, функціональність яких також можна описати у вигляді кінцевого автомата. У цьому випадку процес тестування є обмін повідомленнями між двома кінцевими автоматами, що змінюють свій стан під час обміну. Критерієм повноти такого тестування буде досяжність усіх станів системи, що тестується, усіма можливими способами.

Тестовий план складається з двох частин – визначення кінцевого автомата, що тестується, і сценаріїв переходу між станами – тестів.

Розглянемо такий тестовий план на прикладі. Нехай модуль, що тестується, є простим кінцевим автоматом із трьома станами (рис. 17): «Початковий стан», «Приймання даних» і «Помилка».

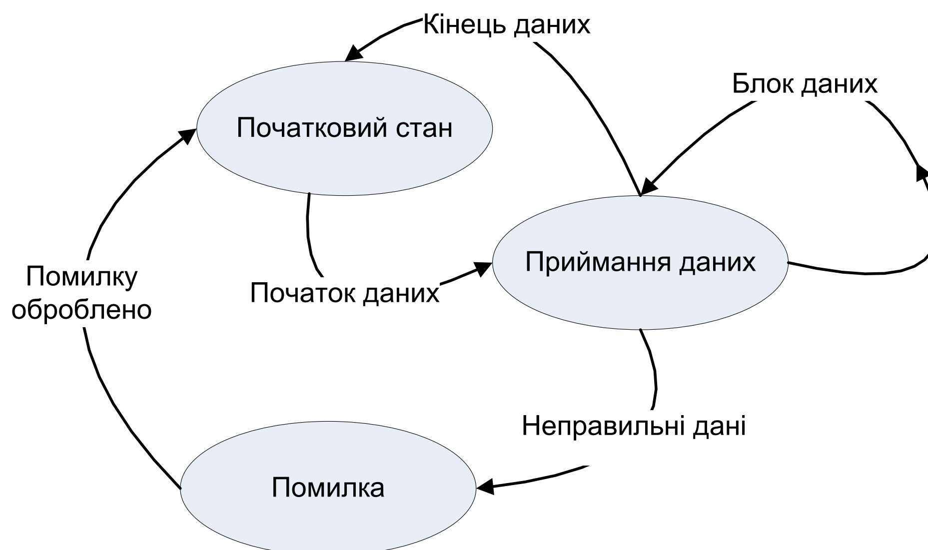

Рис. 17. Граф переходів кінцевого автомата

Автомат починає свою роботу в початковому стані, з якого його можна перевести в стан «Приймання даних» після отримання

повідомлення «Початок даних». Він залишається в цьому стані після отримання кожного наступного правильного блоку даних і може перейти в стан «Помилка» після отримання неправильного блоку даних або в «Початковий стан» після отримання повідомлення «Кінець даних». При переході в стан «Помилка» він передає повідомлення «Виникла помилка». Із стану «Помилка» він може переходити в початковий стан після отримання повідомлення «Помилку оброблено».

Метою тестування буде проведення автомата, що тестується, по всіх станах всіма можливими способами. Один із можливих варіантів полягає в будованні тестувального автомата з еквівалентними станами. Керування таким автоматом буде проводитися з допомогою описів тестів, а не зовнішніх повідомлень.

Тестувальний автомат буде мати три стани – «Початковий стан», «Передання даних» і «Оброблення помилки». Переходячи з початкового стану в стан «Передання даних», він передає повідомлення «Початок даних», у стані «Передання даних» він буде передавати блоки даних, які описано в тестовому завданні, у тому числі, можливо, помилкові. При отриманні повідомлення «Виникла помилка» автомат перейде в стан «Оброблення помилки», з якого перейде в початковий стан, передавши повідомлення «Помилку оброблено». У початковий стан тестувальний автомат може перейти й у разі завершення послідовності оброблення блоків даних, описаних в тестовому прикладі, у цьому випадку він надішле повідомлення «Кінець даних».

Розглянемо визначення тестувального автомату у тестовому плані:

```
STATES DEFINITION:
State1=Початковий стан
State2=Передання даних
State3=Оброблення помилки
PASS DEFINITION
Pass1=State1->State2 with function call
BeginData(Param1)
Pass2=State2->State2 with function call
SendData(Param1)
….
Pass5=State2->State3 external with function call
ErrorReceived(Message)
```

У секції STATES DEFINITON визначено всі стани тестувального автомата, а у PASS DEFINITION – переходи між станами. Перехід зі стану M у стан N визначається StateN->StateM. При переході викликається функція тестового драйвера. Коли буде необхідно, їй передають параметри. Якщо перехід має відбуватися при отриманні зовнішнього повідомлення, то це позначається словом external. При цьому викликається функція, що обробляє отримане повідомлення.

Тести будуть мати такий вигляд:

```
TESTCASE 1
Data:
  egBlock=\027 sndBlock[0]='H' sndBlock[1]='i' errBlock=0
Script:
 Pass1(begBlock) Pass2(sndBlock[0]) Pass2(sndBlock[1])
 Pass2(errBlock)   Pass5(message)
```

Секція Data визначає дані для повідомлень, що передаються автоматом, секції Script – послідовності переходів зі станів з данними, що відсилаються.

2.9. Генератори тестів

У деяких випадках для спрощення процедури тестування використовуються спеціальні ІЗ, що автоматично генерують тести. Ці системи різняться методами генерації тестів, що використовуються, а одержані тести – областями застосування.

Існують такі способи генерації тестів:

– за формалізованими вимогами;

– випадковим чином;

– за програмним кодом.

Перший спосіб генерації тестів можна застосувати для тестування системи як «чорної скрині», але потребує підготовки тестових вимог на формальній мові, наприклад RDL (Requirements Definition Language). Потім за вимогами будуються тести, які перевіряють функціональність системи. У цьому випадку досягається основна мета верифікації – перевірка відповідності функціональності системи вимогам.

На жаль, цей шлях є досить трудомістким і економія часу від автоматичної генерації тестів часто зводиться нанівець необхідністю виділення додаткового часу на переведення усіх вимог у формальну форму, тому рекомендується застосовувати цей метод тільки для тестування систем, вимоги на які можна досить легко формалізувати з використанням тієї або іншої мови.

Другий метод генерації тестів базується на випадкових даних. Тут не може йтися про систематизоване тестування і гарантії якості системи. Такий підхід можна застосовувати тільки тоді, коли необхідно перевірити функціональність системи у випадку передання великої кількості неправильних даних або визначити кількісні параметри її функціонування системи під великим навантаженням.

Третій метод тестування базується на аналізуванні програмних кодів системи і розробленні тестових завдань з огляду на виконання кожної логічної умови й кожного оператора. Унаслідок цього досягається дуже високий рівень покриття програмного коду. Проте в цьому випадку тести

перевіряють не те, що система повинна робити відповідно до вимог, а те, як вона здійснює запрограмовану функціональність. Перед тестувальником у цьому випадку постає задача аналізування програмного коду системи на відповідність вимогам, що часто є задачею не менш складною, ніж написання тестів для перевірки вимог вручну. Зазвичай рекомендується спочатку написати всі тести щодо вимог, а потім, якщо необхідно, скористатися генератором тестів за програмним кодом. При цьому метою використання генератора буде не досягнення максимального покриття будь-якими способами, а аналіз причин непокриття при виконанні тестових вимог і корекція вимог.

2.10. Звіти про проходження тестів

Звіти про проходження тестів – основне (а іноді єдине) джерело для висновку про відповідність протестованої системи вимогам. Після виконання всіх тестів, визначених тестовими планами, середовище тестування створює звіт про те, наскільки успішно система виконала ці тести. Такий звіт містить інформацію про кожний тест (ідентифікатор) і результат його виконання – успіх або невдачу.

За наслідками аналізування звітів про проходження тестів може бути виявлено як дефекти системи, так і некоректно визначені або суперечливі вимоги. В обох випадках на основі результатів аналізу створюються запити на змінення вимог і/або коду системи. Після коректного виправлення дефектів усі тести під час регресійного тестування мають успішно виконуватися.

Звіти про проходження тестів можуть бути основою для відстеження стану проекту: якщо з часом кількість дефектів (неуспішно виконаних тестів), що виявляються, зменшується за умови збереження якості тестування то це свідчить про підвищення якості системи, яка розробляється. З іншого боку, при внесенні значних змін у систему кількість дефектів неминуче збільшується. Таким чином, ідеальний графік залежності кількості дефектів від часу схожий на синусоїду з амплітудою, що зменшується на кожному напівперіоді.

Найчастіше звіт про проходження тестів складається із загального звіту про проходження тестів, звіту про проблеми, які виявлено за результатами виконання тестів, і загальної статистики проходження тестів. У цьому курсі звіт про проходження тестів уважається єдиним документом, розділеним на три частини:

– загальна (заголовна інформація);

– результати виконання тестів;

– підсумкова інформація про виконання тестів (загальна статистика щодо виконаних тестів).

У заголовній частині звіту про проходження тестів ідентифікується звіт і протоколюються те, які частина і версія системи тестувалися, а також яка конфігурація тестового середовища використовувалася для виконання тестів.

Загальна частина звіту про виконання тестів містить:

– назву проекту або системи;

– загальний ідентифікатор групи тестів, занесених у звіт;

– ідентифікатор модуля, що тестується, або групи модулів і номера їхніх версій;

– посилання на розділи і версії тестових або функціональних вимог, за якими написано тести;

– час початку виконання тесту і його тривалість;

– конфігурацію тестового середовища, на якому виконували тест;

– імена й прізвища авторів тестів і/або осіб, що виконували тести.

Наступна частина звіту про проходження тестів повинна містити інформацію про результат виконання кожного тесту: чи завершився він успішно, чи внаслідок його виконання було виявлено які-небудь невідповідності очікуваним результатам. У деяких проектах ця частина звіту може мати одну із двох форм – повну або стислу. Повна форма має всю інформацію про тест, стисла – тільки інформацію про знайдені під час тестування невідповідності очікуваних і реальних результатів.

Зазвичай кожний запис про результат проходження тестів у повній формі містить таку інформацію:

– ідентифікатор тесту;

– короткий опис тесту;

– перелік усіх вхідних значень тесту;

– перелік усіх очікуваних і реальних результатів тесту;

– для кожної пари «очікуваний – реальний результат» – ознаку про збіг або розбіжність цих значень;

– повідомлення про виконання тесту.

У стислій формі кожний запис зазвичай містить:

– ідентифікатор тесту;

– перелік очікуваних і реальних результатів тесту, що розбіглися;

– для кожної пари «очікуваний – реальний результат» дані про збіг або розбіжність цих значень;

– повідомлення про виконання тесту.

Завершальна частина звіту має надавати коротку підсумкову інформацію про виконання всіх тестів, для яких складався звіт, і містити:

– загальну кількість виконаних тестів;

– кількість успішно й неуспішно виконаних тестів;

– загальну кількість перевірених значень;

– кількість результатів, реальні значення яких не відповідали очікуваним.

Часто у звіті про виконання тестів окрім кількісної статистики є розділ з докладним поясненням причин неуспішних тестів. Кожний пункт такого пояснення зазвичай містить:

– ідентифікатори тестів, завдяки неуспішному виконанню яких виявлено проблему;

– посилання на розділи вимог, на основі яких написано тести;

– посилання на ті ділянки програмного коду, де виявлено проблеми;

– опис суті проблеми та можливі шляхи її вирішення з погляду тестувальника;

Цей розділ може бути основою для складання звітів про проблеми або частково заміняти їх.

2.10.1. Звіти автоматичного і ручного тестування

Деякі тести не можна виконати в автоматичному режимі й тому потребують ручної роботи тестувальника. Результати виконання ручних тестів можна заносити до того самого документа, що й результати виконання автоматичних тестів. Особливо часто це робиться у випадку, якщо автоматичні й ручні тести перевіряють одну й ту саму функціональну частину системи. У цьому випадку при генерації звіту про проходження тестів для ручних тестів генерується форма, до якої тестувальник заносить дані про результати тестування, що може полягати або у виконанні тестового сценарію, заданого в тестовому плані, або в експертному аналізі ділянок програмного коду системи, які не можна виконати під час автоматичного тестування.

Форма для ручного тестування зазвичай має:

– ідентифікатор ручного тесту;

– опис сценарію ручного тесту або задачі експертного аналізу;

– ім'я особи, що проводила ручне тестування;

– версії вимог, на основі яких проводилося ручне тестування;

– посилання на ділянки програмного коду, для яких виконувалося ручне тестування;

– інформацію про відповідність програмного коду вимогам (результат ручного тестування);

– інформацію про потенційно можливі проблеми у межах діапазону вхідних значень і за його межами;

– інформацію про можливість тестового покриття програмного коду, при досягненні умов, наведених у вимогах;

– інформацію про підсумковий результат ручного тесту (успішно/неуспішно).

2.10.2. Звіти про покриття програмного коду

Ступінь покриття програмного коду тестами – важливий кількісний показник, що дає змогу оцінити якість системи тестів, а в деяких випадках і якість програмної системи. Дані про ступінь покриття надаються у звітах про покриття, що генеруються під час виконання тестів ІЗ, які підтримують процес тестування. Формат звітів про покриття зазвичай є єдиним у межах проекту і часто залежить від особливостей інструментальних засобів тестування.

У звіті про покриття в стандартизованій формі визначають ділянки програмного коду системи (або її частини), які не було виконано під час тестування, тобто не було покрито тестами. Причини непокриття аналізують тестувальники. За висновками аналізу складаються звіти про проблеми і запити на зміну – документи, у яких описуються об'єкти розроблення, які необхідно змінити, і причини цих змін.

Недостатнє покриття може свідчити про неповноту системи тестів або тестових вимог. У цьому випадку в запиті про зміну зазначається необхідність розширення системи тестів або тестових вимог. Іншою причиною недостатнього покриття можуть бути ділянки захисного коду, які ніколи не буде виконано навіть під час нештатної роботи системи. У цьому випадку в запиті на зміни зазначається необхідність модифікації текстів, або підкреслюється, що для цієї ділянки програмної системи не потрібне покриття. Третьою причиною недостатнього покриття може бути неузгодженість вимог і програмного коду системи, унаслідок чого в коді можуть залишитися такі ділянки, що не виконуються, або навпаки, виникнути ділянки, які розраховані на майбутнє (що реалізують функціональність, не визначену у вимогах). У цьому випадку в запиті на зміну зазначається необхідність модифікації вимог і/або коду системи для зведення їх в узгоджений стан.

2.10.3. Форми звітів про покриття

Типовий звіт про покриття є списком структурних елементів програмного коду (функцій або методів), що містить для кожного структурного елемента таку інформацію:

– назва функції або методу;

– тип покриття (по рядках, гілках, MC/DC тощо);

– кількість елементів у функції або методі (рядків, гілок, логічних умов), що покриваються;

– ступінь покриття функції або методу (у відсотках або абсолютний);

– список непокритих елементів (у вигляді ділянок непокритого програмного коду з номерами рядків).

Крім того, звіт про покриття має інформацію, що дає змогу ідентифікувати звіт і загальний підсумок, тобто загальний ступінь покриття всіх функцій. Приклад звіту:

```
Coverage Report Generated for file Model.cpp
1) function main_Menu()
Coverage: Instructions
Elements: 2100 structured lines of code (SLOCs)
Covered: 87 lines (87%)
Not covered:
241 default:
222 return -1;
216 break;
Coverage: Branches
Elements: 512 branches
Covered: 496 branches (80%)
Not covered (starting and ending lines only):
default:
break;
2) function Menu_item_Help()
Coverage: Instructions
Elements: 210 structured lines of code (SLOCs)
Covered: 210 lines (100%)
Coverage: Branches
Elements: 3 branches
Covered: 3 branches (100%)
Total functions: 2
Total instructions coverage: 97.3%
Total branches coverage: 85%
```

Звіт про покриття можна створювати як для всього проекту, так і для всіх або окремих функцій програмного модуля.

У випадку, коли розмір функцій, для яких генерується вибірковий звіт, є невеликим, можна застосовувати іншу форму звіту про покриття, у якому покритий і непокритий програмний код позначають різними кольорами. Таку форму не можна застосовувати для покриття гілок і логічних умов, але можна для покриття по рядках. Зеленим кольором позначають ділянки методу, які виконано під час тестування, а червоним – які не виконано. Конкретна форма звіту про покриття визначається інструментарієм і технологічними процесами проекту.

Звіт має містити:

– кількість логічних виразів, що можуть набувати значення true-false;
– кількість виразів, покритих на обидва можливих значення;
– кількість виразів, покритих на якесь одне із значень;
– кількість виразів, які не покрито взагалі;

– кількість логічних умов усередині логічних виразів;

– кількість гілок операторів вибору select/switch (гілка вважається покритою, якщо вираз, що стоїть в умові оператора, набуває значення, яке відповідає певному варіанту вибору (case), унаслідок чого гілка виконується);

– кількість точок входу/виходу у функціях (точка входу у функцію вважається покритою, якщо ця функція була викликана хоча б один раз; точками виходу є точка закінчення функції (}), коли керування передається в місце виклику цієї функції, а також оператори повернення (return), після виконання яких керування також передається в місце виклику функції).

Наведемо звіт про загальне покриття фрагменту коду із 33 модулів:

```
Overall Summary (MCDC): 33 Files
 1405 true-false decisions
      674 48.0% covered
      494 35.2% partially covered
      237 16.9% not covered
      1928 conditions in the decisions
     1068 55.4% covered
     571 29.6% partially covered
     289 15.0% not covered
  278 case branches
      154 55.4% covered
      124 44.6% not covered
   581 coverage events
   504 86.7% covered
   77 13.3% not covered
```

Розглянемо детально кожну з характеристик звіту:

– *true-false decisions* – загальна кількість логічних виразів (1405), із якої на обидва можливі значення (True и False) було покрито тільки 674 і ще 494 умови було покрито тільки на одне значення, а 237 умов не було покрито жодного виразу;

– *conditions in the decisions* – кількість логічних умов у логічних виразах; непокриття логічних умов призводить до непокриття логічних виразів, якщо всі умови буде покрито, то всі вирази також буде покрито;

– *case branches* – кількість операторів select;

– *coverage events* – кількість точок входу/виходу у функціях.

Потім виводиться інформація для першого модуля: назва модуля, шлях до файла на диску, дата останнього змінення, контрольна сума. Після цього виводиться сумарна інформація про покриття тільки для цього модуля (для всіх його функцій, кількість яких також визначається).

Структура цієї частини звіту є аналогічною попередній:

1. File <Model.cpp> in <Path_to_model>Last Modified: Oct 15 18:34:14 2012 Checksum: ADDBBB12
File Summary (Extended MCDC): 6 Functions
24 true-false decisions
10 41.7% covered 11 45.8% partially covered
3 12.5% not covered 39 conditions in the decisions
20 51.3% covered 15 38.5% partially covered
4 10.3% not covered 0 case branches 12 coverage events12
100.0% covered 0 0.0% not covered

Далі у звіті має бути сумарна інформація для кожної із функцій окремо, а також усіх ключових ділянок функції, що впливають на покриття: точки входу/виходу, логічні вирази, логічні умови, оператори вибору. Для кожної з цих ключових ділянок виводиться інформація про ступінь покриття, де можливими значеннями можуть бути такі:

COVERED – ділянку повністю покрито в поняттях типу, до якого його приписано;

PARTIALLY COVERED – ділянку частково покрито; може застосовуватися тільки до логічних виразів і умов, указує на те, що вираз (умову) покрито на якесь одне із двох можливих значень;

NOT COVERED – ділянку не покрито, це означає, що ця ділянка не виконувалася під час тестування скомпільованого програмного коду.

Типовий звіт про покриття функцій має такий вигляд:

1.1 Function <Function_1> void Function_1(void)
Function Summary (MCDC):
3 true-false decisions
0 0.0% covered
2 66.7% partially covered
1 33.3% not covered
5 conditions in the decisions
1 20.0% covered
3 60.0% partially covered
1 20.0% not covered
0 case branches
2 coverage events
2 100.0% covered
0 0.0% not covered
coverage line 266: function entry COVERED
decision line 267: if statement if ((A == B) && (...
T F
1: *ttt *fxx ((A == ...)&& COVERED
2: *ttt tfx ((C...)&& PARTIALLY covered

```
3: *ttt ttf ((D == 0) PARTIALLY covered
decision line 271: if statement if (E <= 0)
T F
1: t *f (E <= 0) PARTIALLY covered
decision line 276: if statement if (D == 0)
T F
1: t f (D...) NOT covered
coverage line 291: function end
}
COVERED
```

Слід звернути увагу на те, яким чином відбувається розбір логічних виразів. Наприклад, для умовного оператора

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

будується таблиця, де в кожний рядок поміщається інформація для кожної з логічних умов, що входять до цього логічного виразу. Нумерація логічних умов починається з одиниці (у цьому прикладі їх три).

У першому стовпці наводиться комбінація значень логічних умов (з участю цієї умови), яку необхідно виставити, щоб логічний вираз набув значення True. У цьому випадку, оскільки всі умови сполучені логічним оператором «AND» (&&), такою комбінацією може бути тільки TTT. Якщо наведена комбінація була сформована під час тестування програми, то її позначають символом (*).

У другому стовпці наводиться комбінація значень логічних умов (з участю цієї умови), яку необхідно виставити, щоб логічний вираз набув значення False. У цьому випадку достатньо, щоб хоча б одна умова набула значення False, при цьому значення всіх подальших умов не враховуються (це відображається символом X). Третій стовпчик – це фрагмент тексту логічної умови, що дає змогу легше орієнтуватися в коді під час аналізування покриття.

2.10.4. Покриття на рівні програмних і машинних кодів

У деяких випадках ІЗ аналізують покриття програмного коду тестами на рівні не програмних кодів, а машинних інструкцій. У цьому випадку ступінь покриття залежить також від того, який машинний код генерується компілятором.

Оскільки ступінь покриття може змінюватися залежно від оптимізації, у деяких випадках навіть при повному виконанні всіх операторів високого рівня, на якому написано програмну систему, не вдається досягти повного покриття на рівні виконуваного коду.

Наприклад, для фрагмента програмного коду

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

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

деякі компілятори можуть створити таблицю можливих значень змінної key, у якій будуть всі значення від 0 до 255. Реально в програмній системі можливими є тільки значення констант CC1 – CC3. Таким чином, тільки три гілки із 255 будуть покритими і ніякі стандартні засоби не допоможуть підвищити ступінь покриття.

У цьому випадку звіт про покриття містить додаткову інформацію про причини неможливості забезпечення повного покриття.

Збір інформації про покриття на рівні виконуваного коду найчастіше застосовується у висококритичних програмних системах, у яких не допускається наявність «мертвого» виконуваного коду, що потенційно може призвести до збою або відмови під час роботи системи. До таких систем можна віднести насамперед авіаційні бортові системи, медичні системи і системи забезпечення безпеки інформації.

2.11. Звіти про проблеми

2.11.1. Зв'язок звітів з іншою проектною документацією

Кожна невідповідність вимогам, яку знайдено тестувальником, слід задокументувати у вигляді звіту про проблему. Імовірність виявлення і виправлення помилки, що спричинила цю невідповідність, залежить від того, наскільки якісно її задокументовано. Звіти про проблеми можуть надходити не тільки від тестувальників, але й від фахівців технічної підтримки або користувачів. Їхня загальна мета – указати на наявність проблеми в системі, яку необхідно усунути. Якщо звіт складено некоректно, то розробник не зможе усунути проблему, тому можна вважати цей звіт одним із найважливіших документів у ланцюжку тестової документації.

Головним, що слід включити в звіт про помилку, є:

– спосіб відтворення проблеми – для того щоб розробник зміг усунути проблему, він повинен розібратися в її причинах і самостійно відтворити її; один із найважчих випадків під час розроблення виникає при невідтворенні проблеми, тобто такої проблеми для якої спосіб її виклику є

точно невідомим; знаходження такого способу – одна з найголовніших задач в роботі тестувальника;

– аналізування проблеми з коротким її описом – найкраще наводити опис у тих самих термінах, у яких складено вимоги на частину системи, де знайдено проблему, у цьому випадку мінімізується ймовірність нерозуміння суті проблеми.

Будь-який звіт про проблему необхідно складати негайно після її виявлення. Якщо звіт буде складено значно пізніше, то збільшується ймовірність того, що до нього не потрапить важлива інформація, яка допоможе усунути причину проблеми в найкоротші терміни.

2.11.2. Структура звіту про проблему

Звіт про проблему має такі дані.

1. Об'єкт, де знайдено проблему. Для документації – це назва документа, розділ, автор, версія, для програмних кодів – назва модуля, функції/методу або номери рядків, версія.

2. Випуск і версія системи, що визначає місце, звідки було взято об'єкт із знайденою проблемою. Зазвичай потрібна окрема ідентифікація версії системи (а не тільки версії програмних кодів), оскільки може виникнути плутанина з повторно виявленими проблемами. У цьому випадку, якщо проблему вже було колись виявлено розробником і потім вона знову виникла через те, що до системи потрапила не найостанніша версія програмного модуля, то розробник може вирішити, що йому надійшов попередній звіт і проблеми насправді не існує.

3.Тип звіту:

– *помилка кодування* – код не відповідає вимогам;

– *помилка проектування* – тестувальник не згоден з проектною документацією;

– *пропозиція* – у тестувальника виникла ідея вдосконалити код;

– *розбіжність з документацією* – функціональність ПЗ не відповідає інструкції користувача чи проектній документації, або її взагалі ніде не описано, при цьому у тестувальника немає підстав оголошувати, де саме знаходиться помилка;

– *взаємодія з апаратурою* – неправильна діагностика поганого стану пристрою, помилка в інтерфейсі з пристроєм.

– *питання* – тестувальник не впевнений, що це проблема, і йому потрібна додаткова інформація.

4. Ступінь важливості. Завичай це поле кодують від одиниці (трохи) до 10 (фатально), проте способів обгрунтованого оцінювання немає, дуже складно визначити, наскільки фатальною може виявитися, наприклад, одна друкарська помилка в інструкції користувача.

5. Суть проблеми. Коротке (не більше двох рядків) визначення проблеми. Навіть якщо дві проблеми є дуже схожими, їхні описи повинні різнитися.

6. Чи можна відтворити проблемну ситуацію? Відповідь: *Так, Ні, Не завжди*. Відповідь «Не завжди» ставиться, якщо проблема має нерегулярний характер. Потрібно описувати, коли її виявлено, а коли ні (наприклад, невчасно було натиснуто не ту клавішу).

7. Докладний опис проблеми і спосіб її відтворення. При цьому потрібні подробиці в описі умов відтворення і причини оголошення реакції системи помилкою.

2.12. Таблиці трасування

2.12.1. Зв'язок таблиць трасування з іншою документацією

На кожному етапі життєвого циклу розроблення програмної системи створюється різноманітна проектна документація. Зазвичай документація кожного подальшого етапу створюється на базі документації попереднього етапу. Для спрощення навігації по різних документах і в тому числі для спрощення верифікації документації і самої системи часто використовують перехресні посилання між розділами документів.

Наприклад, часто кожну вимогу в документах позначають унікальним ідентифікатором – якорем (anchor). Якорі у вимогах можуть мати такий формат:

[ANCHOR: код].

Тут код записується у вигляді AAA RR NNNNNN, де AAA – тип документа (SYS, ORD і т.ін.), RR – номер розділу верхнього рівня, у якому міститься якір; NNNNNN – номер посилання з провідними нулями.

Вимога у такому форматі буде мати такий вигляд:

Для кожного обчислюваного атрибута необхідно визначити 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 в посиланні наводиться код якоря для відповідної вимоги. Наприклад, якщо посилання записується в тексті й має формат

[REF: код],

де код має той самий формат, що й для якоря, то посилання на вимоги буде мати такий вигляд:

Для доступу до назви ролі у формулі розрахунку значення обчислюваного атрибута слід використовувати мнемоніку: [RoleName]. [ANCHOR: SRD 02 000058] [REF:SYS_02_000084].

Система якорів і посилань використовує ті самі ідеї, що й звичайний гіпертекст, проте часто виникає необхідність не тільки зазначати сам факт

зв'язку між вимогами або розділами документів, але й додатково вказати тип зв'язку. Наприклад, можна виділити такі 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). У цьому випадку можна вказувати тип посилання поряд із кодом якоря, на який вона посилається. Проте часто використовується й інший метод організації посилань між документами – таблиці трасувань.

У загальному випадку в рядках і стовпцях таблиці трасування вказано ідентифікатори якорів, на які і з яких іде посилання, а на перетині рядків і стовпців заявляється або факт наявності посилання, або його тип.

Таблиці трасувань можна використовувати для машинного аналізування цілісності посилань проектної документації або для швидкої навігації у великих обсягах документів.

2.12.2. Можливі форми таблиць трасування

Як було зазначено в попередньому розділі, в одній із форм таблиць трасувань використовуються ідентифікатори якорів в рядках і стовпцях і типу зв'язку в комірках на їх перетині. Таблиця трасувань буде мати такий вигляд:

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

У першому стовпці записано якорі вимог до ПЗ, у першому рядку – якорі системних вимог. На перетині вказано типи посилань – *cross-ref* – звичайне інформаційне перехресне посилання, *based on* – вимога до ПЗ, що ґрунтується на цій системній вимозі, *variant* – може використовуватися або одна, або інша вимога до ПЗ залежно від варіанта системи.

2.13. Модульне тестування

Процес верифікації триває протягом майже всього ЖЦ системи і працює паралельно з процесом розроблення. Система зазвичай розробляється на різних рівнях: спочатку – концепція системи, системні вимоги, потім – архітектура системи, її розбиття на модулі, наприкінці – окремі модулі. Послідовність цих рівнів залежить від типу ЖЦ, але їхній склад майже завжди однаковий. Процес верифікації також розбивається на окремі рівні:

– системне тестування, під час якого тестується система в цілому;

– інтеграційне тестування, під час якого тестуються групи взаємодійних модулів і компонентів системи;

– модульне тестування, під час якого тестуються окремі компоненти.

Технічні аспекти методів розроблення і проведення тестування на кожному з трьох рівнів було розглянуто в попередніх темах. На кожному з рівнів розробляються тестове оточення, автоматизовані тести, проводяться формальні інспекції. Проте кожний з цих трьох рівнів має свої організаційні особливості, що буде розглянуто далі.

2.13.1. Мета і задачі модульного тестування

Кожна складна програмна система має декілька частин – модулів, що виконують ту або іншу функцію у складі системи. Для того щоб упевнитися в коректній роботі системи в цілому, необхідно спочатку тестувати кожний модуль системи. У разі виникнення проблем при тестуванні системи в цілому це дає змогу швидше виявити модулі, що спричинили проблему, й усунути відповідні дефекти в них. Окреме тестування модулів отримало назву модульного тестування (unit testing).

Для кожного модуля розробляється тестове оточення, що має драйвер і заглушки. Створюються тестові вимоги й тестові плани, що визначають конкретні тести.

Основна мета модульного тестування – упевнитися у відповідності вимогам кожного окремого модуля системи перед тим, як буде проведено його інтеграцію до складу системи.

При цьому під час модульного тестування вирішуються такі основні завдання:

1. Пошук і документування невідповідностей вимогам.
2. Підтримка розроблення і рефакторингу низькорівневої архітектури системи і міжмодульної взаємодії.
3. Підтримка рефакторингу модулів.
4. Підтримка усунення дефектів і відлагодження.

Перше завдання – класичне завдання тестування, що містить не тільки розроблення тестового оточення й тестових прикладів, але й виконання тестів, протоколювання результатів виконання, складання звітів про проблеми.

Друге завдання більш властиве «легким» методологіям типу XP, у яких застосовується принцип тестування перед розробленням (Test-driven development), де основним джерелом вимог для програмного модуля є тест, написаний до реалізації самого модуля. Проте навіть при класичній схемі тестування модульні тести можуть виявити проблеми в дизайні системи та нелогічні або заплутані механізми роботи з модулем.

Третє завдання пов'язане з підтримкою процесу змінення системи. Досить часто під час розроблення потребується проводити рефакторинг

модулів або їхніх груп – оптимізацію або повне перероблення програмного коду з метою підвищення його супроводжуваності, швидкості або надійності роботи. Модульні тести при цьому є потужним інструментом для перевірки того, що новий варіант коду виконує ті самі функції, що й попередній.

Четверте завдання пов'язане із зворотним зв'язком, який одержують розробники від тестувальників у вигляді звітів про проблеми. Докладні звіти про проблеми, складені на етапі модульного тестування, дають змогу локалізувати й усунути багато дефектів у програмній системі на ранніх стадіях її розроблення або розроблення її нової функціональності.

Унаслідок того, що модулі зазвичай є невеликими за розміром, модульне тестування вважається найбільш простим етапом тестування системи. Проте, незважаючи на зовнішню простоту, з модульним тестуванням пов'язано дві проблеми.

Перша з них полягає в тому, що не існує єдиного принципу визначення того, що точно є окремим модулем.

Друга полягає у відмінностях в трактуванні самого поняття модульного тестування – під цим розуміється відособлене тестування модуля, робота якого підтримується тільки тестовим оточенням, перевірка коректності роботи модуля у складі вже розробленої системи.

2.13.2. Поняття модуля і його границь. Тестування класів

Традиційним визначенням модуля є таке: «модуль – це компонент мінімального розміру, який можна незалежно протестувати під час верифікації програмної системи». У реальності часто виникають проблеми з тим, що вважати модулем. Існує декілька підходів до цього визначення:

а) модуль – це частина програмного коду, що виконує одну функцію, яку визначено у функціональних вимогах;

б) модуль – це програмний модуль, тобто мінімальний компільований елемент програмної системи;

в) модуль – це задача в списку задач проекту;

г) модуль – це ділянка коду, який може вміщатися на одному екрані або одному аркуші паперу;

д) модуль – це один клас або множина класів з єдиним інтерфейсом.

Зазвичай за модуль, що тестується, береться або програмний модуль (одиниця компіляції), якщо система розробляється на процедурній мові програмування, або клас, якщо система розробляється на об'єктно-орієнтованій мові.

Якщо систему написано на процедурних мовах, то для кожного модуля розробляється тестовий драйвер, який викликає функції модуля і збирає результати їхньої роботи, і набір заглушок, що імітують

функціональність, яку реалізовано в інших модулях і яка не потрапляє під тестування цього модуля.

При тестуванні об'єктно-орієнтованих систем існує кілька особливостей, перш за все викликаних інкапсуляцією даних і методів в класах. У цьому випадку використання окремих методів як модулів, що тестуються, є недоцільним через те, що тестування кожного методу потребує розроблення тестового оточення, схожого за складністю з уже написаним програмним кодом класу. Крім того, декомпозиція класу порушує принцип інкапсуляції, за яким об'єкти кожного класу повинні функціонувати як єдине ціле відносно інших об'єктів.

Процес тестування класів як модулів іноді називають компонентним тестуванням. Під час такого тестування перевіряється взаємодія методів усередині класу й правильність доступу методів до його внутрішніх даних. При такому тестуванні можна виявити не тільки стандартні дефекти, що пов'язані з виходами за межі діапазону або неправильно реалізованими вимогами, але й специфічні дефекти об'єктно-орієнтованого ПЗ, а саме:

– дефекти інкапсуляції, унаслідок чого, наприклад, приховані дані класу є недосяжними з допомогою відповідних публічних методів;

– дефекти спадкування, через які схема спадкування блокує важливі дані або методи від класів-нащадків;

– дефекти поліморфізму, через які поліморфність поширюється не на всі можливі класи;

– дефекти створення об’єктів класу, через які в них не встановлюються коректні значення з параметрів і внутрішніх даних класу, проте вибір класу як модуля, що тестується, має деякі проблеми.

2.13.3. Визначення ступеня повноти тестування класу

У тому випадку, коли модуль для тестування є вибраним класом, не зовсім ясно, як визначати ступінь повноти його тестування. З одного боку, можна використовувати класичний критерій повноти покриття програмного коду тестами: якщо повністю виконано всі структурні елементи всіх методів (як публічних, так і прихованих), то тести можна вважати повними. Проте існує альтернативний підхід до тестування класу, за яким усі публічні методи мають надавати користувачу цього класу узгоджену схему роботи і достатньо перевірити типові коректні й некоректні сценарії роботи з цим класом. Наприклад, у класі, об'єктами якого є записи в телефонному записнику, одним із типових сценаріїв роботи будуть такі запитання: «Створити запис?», «Шукати запис і знайти його?», «Видалити запис?», «Шукати запис повторно і отримати повідомлення про помилку?».

Відмінності в цих двох методах тестування схожі з відмінностями між тестуванням «чорної» і «білої» скрині, але насправді другий підхід тестування класів відрізняється від першого тим, що функціональні вимоги

на систему можуть бути складені на рівні більш високому, ніж окремі класи, і адекватність тестових сценаріїв вимогам підтверджує тестувальник.

Деякі методи класу призначено не для видачі інформації користувачу, а для змінення внутрішніх даних об'єкта класу. Значення внутрішніх даних об'єкта є його станом в кожний окремий момент часу, а викликані методи, що змінюють дані, змінюють і стан об'єкта. При тестуванні класів необхідно перевіряти, чи адекватно реагує клас на зовнішні виклики в будь-якому зі станів об'єктів. Проте часто через інкапсуляцію даних неможливо визначити внутрішній стан об'єктів класу програмними способами усередині драйвера. У цьому випадку може допомогти складання графа переходів об'єкта як кінцевого автомата з певним набором станів і переходів. Така схема може належати до низькорівневої проектної документації (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 ручне аналізування програмного коду і визначення його відповідності вимогам. Автоматизоване тестування може лише визначити, чи в усіх виявлених станах здійснювалися переходи і чи всі можливі реакції перевірялися.

Модульні тести – потужний інструмент перевірки коректності змін, занесених в програмний код під час рефакторинґу. Проте внаслідок рефакторинґу тільки одного класу зазвичай не змінюється його зовнішній інтерфейс з іншими класами, бо найчастіше під час рефакторинґу змінюються відразу декілька класів. Унаслідок звичайних еволюційних змін системи може змінюватися зовнішній інтерфейс класу, причому як за формальними (змінюються назви, склад і параметри методів), так і за функціональними (при збереженні зовнішнього інтерфейсу міняється логіка роботи методів) ознаками. Для проведення модульного тестування класу після таких змінень потрібне змінення драйвера і, можливо, заглушок. Крім того, необхідно також проводити й інтеграційне тестування цього класу разом зі всіма класами, які пов'язані з ним через дані або через керування.

Незалежно від того, на які модулі під час тестування розділяється система, рекомендується визначити принципи виділення модулів в плані й стратегії тестування, а також скласти на основі архітектури системи нову структурну схему, на якій і позначити всі модулі. Це дасть змогу спрогнозувати склад і складність драйверів і заглушок, потрібних для модульного тестування системи. Така схема може використовуватися пізніше на етапі модульного тестування для виділення груп модулів, для поступової інтеграції і тестування.

2.13.4. Особливості тестового оточення

Незалежно від того, яка одиниця програмного коду системи вибирається за мінімальний модуль під час тестування, існує ще одна відмінність у підходах до модульного тестування.

Перший підхід до модульного тестування ґрунтується на припущенні, що функціональність кожного модуля має перевірятися в автономному режимі без його інтеграції з системою. При такому підході для кожного модуля створюється тестовий драйвер і заглушки, з допомогою яких виконується набір тестів. Тільки після усунення всіх дефектів в автономному режимі проводиться інтеграція модуля в систему й наступне тестування. Перевагою цього підходу є більш проста локалізація помилок в модулі, оскільки під час автономного тестування виключається вплив решти частин системи, який може спричинити маскування дефектів (ефект парної кількості помилок). Основний недолік цього методу – підвищена трудомісткість написання драйверів і заглушок, оскільки заглушки мають адекватно моделювати функціональність системи в різних ситуаціях, а драйвер не тільки створювати тестове оточення, але й імітувати внутрішній стан системи, у складі якої функціонує модуль.

Другий підхід побудовано на припущенні, що модуль все одно працює у складі системи, і якщо модулі інтегрувати в систему по одному, то можна тестувати модуль у складі всієї системи. Цей підхід є властивим більшості сучасних «полегшених» методологій розроблення.

Унаслідок застосування такого підходу різко зменшуються трудовитрати на розроблення заглушок і драйверів – заглушкою є частина системи, що вже пройшла тестування, а драйвер виконує тільки функції передання і приймання даних, не моделюючи внутрішній стан системи.

Проте використання цього методу підвищує складність написання тестів. Для приведення заглушок в стан тестування системи зазвичай необхідно тільки встановлення значення тестових змінних, а приведення в належний стан частини реальної системи потребує виконання складного сценарію, що переведе програмну систему у цей стан. Для цього кожний тест матиме такий сценарій. Крім того, цей підхід не завжди дає змогу локалізувати помилки, які приховано усередині модуля і можуть виявитися під час інтеграції з іншими модулями.

2.13.5. Організація модульного тестування

Модульне тестування з погляду тестувальника – це комплекс робіт з виявлення дефектів в модулях, що тестуються. До цієї роботи належать аналізування вимог, розроблення тестових вимог і тестових планів, розроблення тестового оточення, виконання тестів, накопичення інформації про їх проходження.

Проте з погляду керівника групи тестування (або керівника проекту, якщо в ньому не виділено окрему групу тестування), модульне тестування є більш широким поняттям. Для того щоб процес модульного тестування міг функціонувати спільно з іншими процесами розроблення, він повинен мати декілька фаз: планування процесу, розроблення тестів, виконання тестів, накопичення статистичних даних, керування звітами про виявлені дефекти.

Процес модульного тестування складається з таких фаз:

1. Фаза планування тестування:

– планування основних підходів до тестування, ресурсне планування і календарне планування;

– визначення властивостей, що підлягають тестуванню;

– уточнення основного плану, який сформовано на етапі планування.

2. Фаза отримання набору тестів:

– розроблення набору тестів;

– реалізації уточненого плану.

3. Фаза вимірювання модуля:

– виконання тестових процедур;

– визначення достатності тестування;

– оцінювання результатів тестування і модуля.

Під час етапу планування основних підходів як вхідні дані використовується загальний план проекту (модульне тестування як частина проектних робіт має укладатися в загальний графік) і вимоги до системи (для оцінювання трудомісткості робіт і будь-якого планування необхідно проводити аналіз складності системи на основі вимог до неї).

Основні задачі планування модульного тестування:

– визначення загального підходу до тестування модулів – визначаються ризики і на їх основі – ступінь повноти й обхвату тестування системи; визначаються джерела вхідних і вихідних даних, технології перевірки результатів тестування і формати запису даних про проведене тестування, описується зовнішній інтерфейс модулів, що тестуються, і їхнє інформаційне оточення;

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

– визначення вимог до завершення тестування – визначаються умови, перевірка яких дасть змогу стверджувати, що тестування модуля завершено, а також визначаються умови, за якими подальше тестування модуля вважається неможливим до його зміни й модифікації; прикладом таких умов може бути досягнення певного рівня покриття програмного коду тестами і неможливість компіляції модуля;

– визначення вимог до технічних (комп'ютери і ПЗ) і людських

(тестувальники) ресурсів, необхідних для розроблення і виконання тестів, а також для аналізування результатів тестування – при вирішенні цього завдання необхідно визначити не тільки вимоги до програмного й апаратного забезпечення, необхідної кваліфікації людей, але й необхідну кількість ресурсів;

– визначення загального плану-графіка робіт – на основі загального плану проекту складається план робіт з модульного тестування; основний критерій початку роботи з тестування – готовність модулів, тобто загальний план робіт з тестування узгоджується за датами початку і закінчення робіт загального плану розроблення.

Після завершення етапу планування починається етап визначення властивостей системи, що підлягають тестуванню. Основними завданнями є такі:

– вивчення функціональних вимог – визначення тестопридатності вимог, у разі необхідності вимоги уточнюються;

– визначення додаткових вимог і зв'язаних процедур – визначення вимог, що не потрапили до функціональних, але їх можна протестувати на рівні модульного тестування (наприклад, це можуть бути вимоги до продуктивності системи, що належать до складу системних вимог);

– визначення станів модуля – якщо модуль можна подати у вигляді кінцевого автомата з певним набором станів, то кожний стан має бути ідентифікованим, а вимоги, що належать до нього, – виділеними;

– визначення характеристик вхідних і вихідних даних – для всіх даних, що надходять/виходять з модуля, необхідно визначити формати, частоту надходження, допустимі значення тощо;

– вибір елементів, що тестуються, – у разі, коли не можна застосовувати повне тестування, необхідно вибрати елементи модуля, що будуть тестуватися; основне джерело інформації тут – дані про ризики, проаналізовані на рівні структури програмного коду модуля; для тестування у першу чергу слід відбирати елементи з максимальним ступенем ризику; на завершення фази планування уточнюється основний план, а саме загальний підхід до тестування, формулюються спеціальні й додаткові вимоги до ресурсів, складається детальний план-графік робіт.

Після завершення планування починається фаза розроблення тестів. Тести розробляються за тими планами й вимогами, які створено на попередньому етапі. Таким чином, якщо на першому етапі основну роботу виконує керівник групи тестування, то на другому – тестувальник, що діє відповідно до вказівок керівника.

Фаза розроблення тестів починається з розроблення набору тестів для тестування модуля. Основними документами, що використовуються на цьому етапі, є функціональні вимоги до модуля, архітектура модуля, список елементів, які піддаються тестуванню, план-графік робіт, тести і результати тестування попередньої версії.

Під час цього етапу необхідно вирішити такі завдання:

– розроблення архітектури тестового набору – під тестовим набором тут розуміється не набір конкретних тестів, а загальна структура системи тестів для перевірки функціональності модуля, організація тестів у такій системі зазвичай відображає структуру функціональних вимог і часто є ієрархією, на кожному рівні якої визначається свій набір тестів;

– розроблення явних тестових процедур (тестових вимог) – у разі досить докладних функціональних вимог і чітко прописаної концепції розроблення тестів явні тестові процедури можуть і не розроблятися, однак у випадку існування необхідних ресурсів розроблення тестових вимог дасть змогу більш чітко інтерпретувати ті функціональні вимоги, що будуть тестуватися, і знизити ризик їх неоднозначної інтерпретації;

– розроблення тестів – тести мають відповідати вимогам до повноти тестування і базуватися на тестових або функціональних вимогах; цей вид діяльності є найтривалішим у часі;

– розроблення тестів, що ґрунтуються на архітектурі (якщо це необхідно), – деякі тести побудовано не на функціональних вимогах, а на особливостях архітектури модуля; для розроблення тестів, що грунтуються на архітектурі, необхідно використовувати підхід «скляної скрині»; ці тести пишуться на основі низькорівневих тестових або системних вимог, якщо їх розробляли на одному з попередніх етапів;

– складання специфікації тестів – результатом діяльності тестувальника на цьому етапі є документ, формат якого відповідає стандарту IEEE 829 [23].

На наступному етапі проводиться реалізація тестів (наприклад, у вигляді тестового оточення й формалізованих описів тестів); формуються тестові набори даних, що використовуються у тестах; створюється тестове оточення; здійснюється інтеграція тестового оточення з модулем.

Після того як всі тести реалізовано, їх виконують в ручному або автоматичному режимі. Незалежно від виду тестування під час цього етапу вирішуються два завдання: виконання тестів та накопичення й аналізування результатів тестування.

Накопиченню підлягає така інформація:

– результат виконання кожного тесту (пройшов/не пройшов);

– інформація про інформаційне оточення системи у випадку, якщо тест не пройшов;

– інформація про ресурси, що були потрібні для виконання тесту; за підсумками аналізу цієї інформації складаються запити на змінення проектної документації, програмного коду модуля або тестового оточення.

Етапи розроблення (дороблення), реалізації й виконання тестів продовжуються доти, доки не буде досягнуто критерій завершення модульного тестування. Прикладом такого критерію може бути відсутність непройдених тестів при 75-відсотковому покритті рядків програмного коду.

Після припинення тестування виконуються такі роботи з оцінювання проведеного тестування:

– описуються відмінності реального процесу тестування від запланованого;

– описуються відмінності реакції модуля від описаної у вимогах (з метою подальшої корекції вимог);

– складається загальний звіт про проходження тестів, що містить також інформацію про покриття.

Завершивши модульне тестування, необхідно перевірити, що всі створені артефакти – документи, програмний код, файли звітів і даних – занесено до репозиторію або бази даних проекту разом з даними, що використовувалися і створювалися під час розроблення програмної системи.

2.14. Інтеграційне тестування

2.14.1. Мета і задачі інтеграційного тестування

Результатом тестування і верифікації окремих модулів, що складають програмну систему, є висновок про те, що ці модулі – внутрішньо несуперечливі і відповідають вимогам. Проте окремі модулі майже не функціонують самі по собі, тому наступна задача після тестування окремих модулів – тестування коректності взаємодії кількох модулів, з'єднаних в єдине ціле. Таке тестування називають інтеграційним, його мета – упевнитися в коректності спільної роботи компонентів системи.

Інтеграційне тестування називають ще тестуванням архітектури системи. З одного боку, це пов'язане з тим, що інтеграційні тести містять перевірки всіх можливих видів взаємодій між програмними модулями й елементами, які визначають в архітектурі системи. Таким чином, з допомогою інтеграційних тестів перевіряють повноту взаємодій в реалізації системи, що тестується. З іншого боку, результати виконання інтеграційних тестів – одне з основних джерел інформації для процесу поліпшення й уточнення архітектури системи, міжмодульних і міжкомпонентних інтерфейсів. Іншими словами, інтеграційні тести перевіряють коректність взаємодії компонентів системи.

Прикладом перевірки коректності взаємодії можуть бути два модулі, один з яких накопичує повідомлення протоколу про прийняті файли, а другий виводить цей протокол на екран. У функціональних вимогах на систему записано, що повідомлення мають виводитися у зворотному хронологічному порядку. Проте модуль зберігання повідомлень зберігає їх у прямому порядку, а модуль виведення – використовує стек для виведення в зворотному порядку. Модульні тести кожний окремо не

ефективні – цілком реальною є зворотна ситуація, при якій повідомлення зберігаються в зворотному порядку, а виводяться по черзі. Знайти потенційну проблему можна, тільки перевіривши взаємодію модулів з допомогою інтеграційних тестів. Ключовим моментом тут є те, що система виводить повідомлення в зворотному хронологічному порядку, тобто перевіривши модуль виведення і знайшовши, що він виводить повідомлення в прямому порядку, не можна гарантувати, що було знайдено дефект.

Унаслідок проведення інтеграційного тестування й усунення всіх виявлених дефектів виходить узгоджена й цілісна архітектура програмної системи, тобто інтеграційне тестування – це тестування архітектури й низькорівневих функціональних вимог. Інтеграційне тестування зазвичай є ітеративним процесом, коли перевіряється, що функціональність все більше і більше збільшується зі збільшенням сукупності модулів.

2.14.2. Організація інтеграційного тестування

Структурна класифікація методів інтеграційного тестування

Зазвичай інтеграційне тестування проводять уже після закінчення модульного тестування для всіх модулів. Проте це зовсім не так. Існує кілька методів проведення інтеграційного тестування:

– висхідне тестування;
– монолітне тестування;
– низхідне тестування.

Усі ці методики ґрунтуються на знаннях про архітектуру системи, яка часто подається у вигляді структурних діаграм або діаграм викликів функцій. Кожний вузол на такій діаграмі є програмним модулем, а стрілки між ними – залежністю за викликами між модулями. Основна відмінність методик інтеграційного тестування полягає у напрямі руху за цими діаграмами і в широті обхвату за одну ітерацію.

Висхідне тестування. При використанні цього методу спочатку тестуються всі програмні модулі, що належать до складу системи, і тільки тоді їх об'єднують для інтеграційного тестування (рис. 18).

При цьому значно спрощується локалізація помилок: якщо модулі протестовано окремо, то помилка під час їхньої спільної роботи є проблемою їхнього інтерфейсу. При такому підході область пошуку проблем для тестувальника досить вузька, а тому набагато вище ймовірність ідентифікації дефектів. Проте висхідний метод тестування має істотний недолік – необхідність в розробленні драйвера і заглушок не тільки для модульного тестування перед проведенням інтеграційного тестування, але й для всіх етапів інтеграційного тестування.

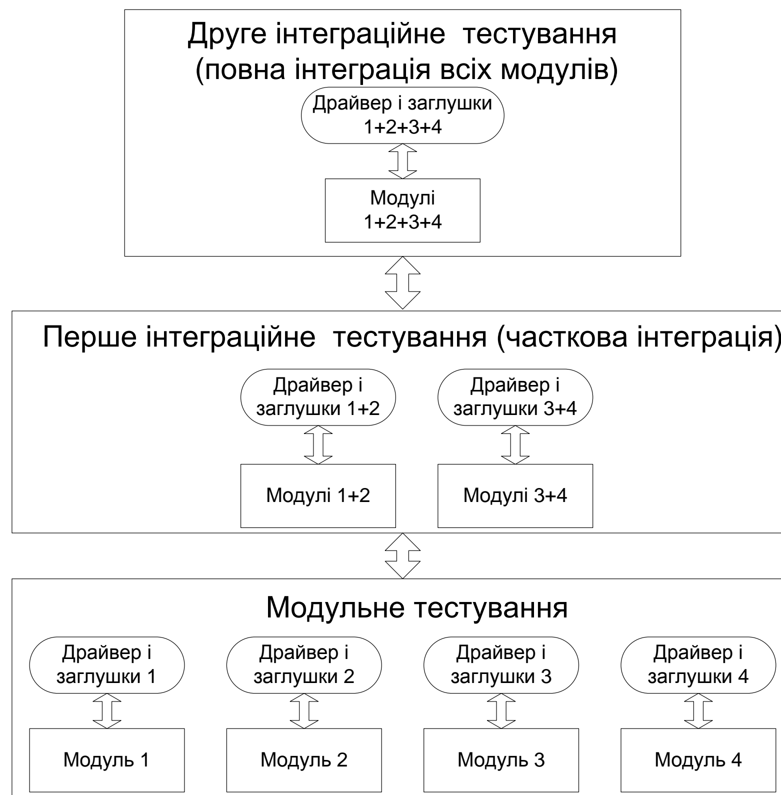

Рис. 18. Послідовність розроблення драйверів і заглушок висхідного інтеграційного тестування

З одного боку, драйвери і заглушки – потужний інструмент тестування, з іншого – їх розроблення потребує значних ресурсів, особливо через змінення складу модулів, що інтегруються. Іншими словами, потрібен один набір драйверів для окремого тестування кожного модуля, окремий драйвер і заглушки – для тестування інтеграції двох модулів з набору, окремий – для тестування інтеграції трьох модулів і т.д. Це пов'язано насамперед з тим, що при інтеграції модулів немає необхідності в деяких заглушках, а також потрібна зміна драйвера, що підтримує нові тести, які стосуються кількох модулів.

При монолітному тестуванні припускається, що окремі компоненти системи детального тестування не проходили. Основна перевага цього методу – відсутність необхідності розроблення тестового оточення, драйверів і заглушок. Після розроблення всіх модулів виконується їх інтеграція, після чого система перевіряється вся в цілому. Цей підхід не треба плутати із системним тестуванням. Не зважаючи на те, що при монолітному тестуванні перевіряється робота всієї системи в цілому, основна задача цього тестування – визначення проблеми взаємодії окремих модулів системи. Задачею ж системного тестування є оцінювання якісних і кількісних характеристик системи для кінцевого користувача.

Монолітне тестування має такі недоліки:

– дуже важко виявити джерело помилки (ідентифікувати помилковий фрагмент коду) – у більшості модулів слід припускати наявність помилки; проблема полягає у визначенні того, яка з помилок у всіх залучених модулях призвела до отриманого результату, при цьому можливе накладення ефектів кількох помилок, а помилка в одному модулі може блокувати тестування іншого;

– важко організувати виправлення помилок – під час тестування тестувальник фіксує знайдену проблему; дефект в системі, що спричинив цю проблему, буде усувати розробник; оскільки програмні модулі пишуться зазвичай різними людьми, постає проблема: хто з них відповідає за пошук усунення дефекту, при цьому через «колективну безвідповідальності» швидкість усунення дефектів може різко зменшитися;

– процес тестування погано автоматизується – перевага (немає додаткового ПЗ, що супроводжує процес тестування) перетворюється на недолік, кожна внесена зміна потребує повторення всіх тестів.

При низхідному тестуванні припускається, що процес інтеграційного тестування здійснюється після процесу розроблення. Спочатку при низхідному підході тестують тільки верхній рівень керування системи, без модулів нижчих рівнів. Потім поступово з високорівневими модулями інтегруються низькорівневі. Послідовність інтеграції показано на рис.19.

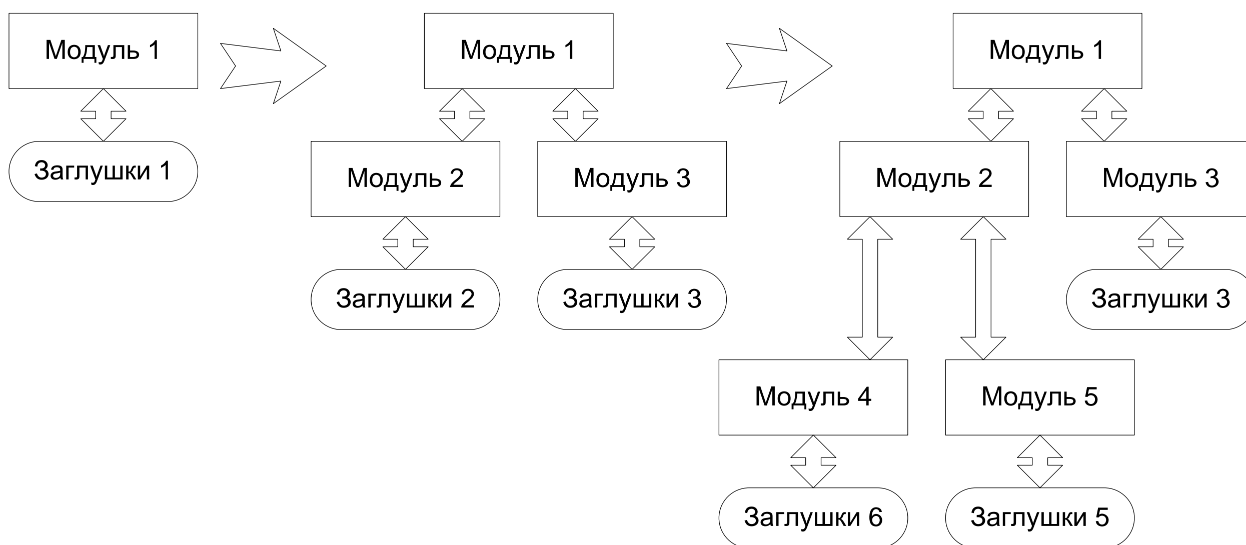

Рис. 19. Інтеграція модулів під час низхідного тестування

Унаслідок застосування такого методу немає необхідності в драйверах (драйвером є високорівневий модуль системи), проте зберігається потреба в заглушках.

У фахівців в області тестування різні думки щодо того, який із методів є більш зручним при реальному тестуванні програмних систем. Г. Майерс уважає, що кожний з підходів має свої переваги й недоліки, але

в цілому висхідний метод є кращим, а інші – що низхідне тестування є найбільш прийнятним у реальних ситуаціях. У літературі часто наводиться метод інтеграційного тестування об'єктно-орієнтованих програмних систем, який базується на виділені кластерів класів, що мають разом деяку замкнуту й закінчену функціональність. По суті такий підхід не є новим типом інтеграційного тестування, просто змінюється мінімальний елемент, який одержують під час інтеграції. При інтеграції модулів на процедурних мовах програмування можна інтегрувати будь-яку кількість модулів за умови розроблення заглушок. При інтеграції класів в кластери існує обмеження на закінченість функціональності кластера. Проте навіть у разі об'єктно-орієнтованих систем можна інтегрувати будь-яку кількість класів з допомогою класів-заглушок.

Незалежно від того, який метод інтеграційного тестування було застосовано, необхідно враховувати ступінь покриття інтеграційними тестами функціональності системи. Існує спосіб оцінювання ступеня покриття, який базується на керувальних викликах між функціями і потоками даних. При такому оцінюванні коди всіх модулів на структурній діаграмі системи має бути виконано (покрито всі вузли), усі виклики слід виконати хоча б один раз (усі зв'язки між вузлами на структурній діаграмі мають бути покритими), усі послідовності викликів також слід виконати хоча б один раз (усі шляхи на структурній діаграмі мають бути покритими).

Часова класифікація методів інтеграційного тестування

На практиці в різних частинах проекту розглянуті методи частіше за все застосовуються в сукупності. Кожний модуль тестують окремо у міру готовності, а потім додають до вже готової композиції. Для одних частин тестування виходить низхідним, для інших – висхідним. У зв'язку з цим корисно розглянути ще одну класифікацію типів інтеграційного тестування – класифікацію за часом інтеграції.

У межах цієї класифікації вирізняють:

– тестування з пізньою інтеграцією;

– тестування з постійною інтеграцією;

– тестування з регулярною або пошаровою інтеграцією.

Тестування з пізньою інтеграцією – майже повний аналог монолітного тестування. Інтеграційне тестування за такою схемою відстрочується на найпізніші проектні дати. Цей підхід виправдовує себе, якщо система є конгломератом слабо зв'язаних між собою модулів, що взаємодіють за будь-яким стандартним інтерфейсом, визначеним поза проектом (наприклад, у випадку, якщо система складається з окремих Web-сервісів). Тестування з пізньою інтеграцією можна подати як ланцюжок:

R-C-U-R-C-U-R-C-U-I-R-C-U-R-C-U-I,

де R – розроблення вимог на окремий модуль; C – розроблення програмного коду; U – (unit testing) модульне тестування; I – інтеграційне тестування всього, що було зроблено раніше.

У разі тестування з постійною інтеграцією кожний новий модуль системи після розроблення відразу ж інтегрується зі всією рештою системи. При цьому тести для модуля перевіряють як його внутрішню функціональність, так і взаємодію з іншими модулями системи. Таким чином, цей підхід об’єднує в собі модульне й інтеграційне тестування. Розробляти заглушки при такому підході зовсім не потрібно, але можуть знадобитися драйвери. На сьогодні 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 є дещо утрудненою, але все ж таки значно нижчою, ніж при монолітному тестуванні. Велика частина таких помилок виявляється досить рано саме завдяки частоті інтеграції і того, що за одну ітерацію тестування перевіряється порівняно невелика кількість міжмодульних інтерфейсів.

Тестування з постійною інтеграцією можна подати у вигляді ланцюжка R-C-I-R-C-I-R-C-I, що не має окремої фази модульного тестування, заміненого на тестування інтеграції.

При тестуванні з регулярною або пошаровою інтеграцією інтеграційному тестуванню підлягають сильно зв'язані між собою групи модулів (шари), які потім також інтегруються між собою. Такий вид інтеграційного тестування називають також ієрархічним інтеграційним тестуванням, оскільки збільшення інтегрованих частин системи зазвичай відбувається за ієрархічним принципом. Проте на відміну від низхідного або висхідного тестування напрямок проходу за ієрархією в цьому підході не задано. Особливості різних видів тестування подано у табл. 6.

Таблиця 6

Характеристики різних видів інтеграційного тестування

Особливість	Тестування			Інтеграція		
	висхідне	низхідне	монолітне	пізня	постійна	регулярна
Час інтеграції	Пізно (після тестування модулів)	Рано (паралельно з розробленням)	Пізно (після розроблення всіх модулів)		Рано (паралельно з розробленням)	
Частота інтеграції	Рідко	Часто	Рідко	Рідко	Часто	Часто
Драйвери	Так	Ні	Ні	Ні	Так	Так
Заглушки	Так	Так	Ні	Ні	Ні	Так

Час інтеграції характеризує момент, коли проводиться інтеграційне тестування, частота інтеграції – наскільки часто при розробленні виконується інтеграція. Необхідність у драйверах і заглушках наведено в останніх двох рядках таблиці.

2.14.3. Планування інтеграційного тестування

Процес організації і планування інтеграційного тестування багато в чому схожий з процесом організації модульного тестування. Проте інтеграційне тестування має деякі організаційні особливості, які перелічено нижче.

На етапі планування розробляється концепція і стратегія інтеграції – документ, де описується загальний підхід до визначення послідовності, у якій мають інтегруватися модулі. Зазвичай концепція ґрунтується на одному з видів інтеграції, розглянутих вище (наприклад, на низхідній), але враховуються особливості конкретної системи (наприклад, спочатку мають інтегруватися компоненти роботи з базою даних, потім такі, що призначені для користувача інтерфейсу, потім інтерфейсні компоненти і компоненти роботи з БД, які інтегруються разом).

Складається інтеграційний тест-план, наприклад, кластерного типу, у якому для кожного кластера з інтегрованих модулів визначаються:

– кластери, від яких залежить цей кластер;
– кластери, які слід попередньо протестувати;
– опис функціональності кластера, що тестується;
– список модулів в кластері;
– опис тестів для перевірки кластера.

Планування інтеграційного тестування має бути синхронізоване із загальним планом проекту причому в кластерах, що виділяються для інтеграційного тестування, і в термінах їх тестування має бути враховано пріоритети важливості частин системи. Часто розгляд пріоритетів пов'язаний з тим, що системи розробляються в кілька етапів, на кожному з яких в експлуатацію вводиться тільки частина нової системи. Інтеграційне тестування в цьому випадку має укладатися в загальний план-графік проекту і при цьому враховуватися витрати ресурсів на тестування інтеграції з уже працюючими частинами системи.

2.15. Системне тестування

2.15.1. Мета і завдання системного тестування

Після завершення інтеграційного тестування всі інтерфейси і функціональність модулів системи мають бути узгодженими. Починаючи з цього моменту, можна переходити до тестування системи в цілому як

єдиного об'єкта тестування – до системного тестування. На рівні інтеграційного тестування тестувальника цікавлять в основному структурні аспекти системи, на рівні системного тестування – функціональні. Зазвичай для системного тестування використовується підхід «чорна скриня», при цьому як вхідні використовуються реальні дані, з якими працює система, або схожі з ними.

Системне тестування – один із найскладніших видів тестування. На цьому етапі проводиться не тільки функціональне тестування, але й оцінювання характеристик якості системи: її стійкість, надійність, безпека й продуктивність. На цьому етапі виявляється багато проблем зовнішніх інтерфейсів системи, пов'язаних з неправильною взаємодією з іншими системами, апаратним забезпеченням, неправильним розподілом пам'яті, відсутністю коректного звільнення ресурсів тощо.

Після завершення системного тестування розроблення переходить у фазу приймально-здавальних випробувань (для програмних систем, що розробляються на замовлення) або у фазу α- і β-тестування (для програмних систем загального застосування).

Оскільки системне тестування потребує значних ресурсів, його проводить окремий колектив тестувальників, а системне тестування часто виконується організацією, що не пов'язана з колективом розробників і тестувальників, які виконували роботи на попередніх етапах тестування. При цьому необхідно зазначити, що при розробленні деяких типів ПЗ (наприклад, авіаційного бортового) вимога незалежного тестування на всіх етапах розроблення є обов'язковою.

Системне тестування складається з кількох фаз, на кожній з яких перевіряється один із аспектів функціональності системи, тобто проводиться один із типів системного тестування. Усі ці фази можуть бути одночасними або послідовними.

2.15.2. Види системного тестування

Існують такі види системного тестування:
– функціональне тестування;
– тестування продуктивності;
– тестування навантаження або стресове тестування;
– тестування конфігурації;
– тестування безпеки;
– тестування надійності й відновлення після збоїв;
– тестування зручності використання.

Під час системного тестування проводяться далеко не всі види тестування з перелічених – конкретний їх набір залежить від системи, що тестується.

Системне тестування ґрунтується на двох класах вимог: функціональних і нефункціональних. Функціональні вимоги явно описують, що система має робити і як перетворювати вхідні значеня на вихідні. Нефункціональні вимоги визначають властивості системи, які напряму не пов'язані з її функціональністю. Прикладом таких властивостей може бути час відгуку на запит користувача (наприклад, не більше двох секунд), час безперебійної роботи (наприклад, не менше 10 000 годин між двома збоями), кількість помилок, які допускає недосвідчений користувач за перший тиждень роботи (не більше 100) тощо.

Розглянемо кожний вид тестування докладніше.

Функціональне тестування призначено для доказу того, що вся система в цілому поводиться відповідно до очікувань користувача, формалізованих у вигляді системних вимог. Під час цього тестування перевіряються всі функції системи для її користувачів (як людей, так і програмних систем). Система при функціональному тестуванні розглядається як «чорна скриня», тому в цьому випадку корисно використовувати класи еквівалентності вхідних даних. Критерієм повноти тестування є повнота покриття тестами системних функціональних вимог (або системних тестових вимог) і повнота тестування класів еквівалентності, а саме:

– усі функціональні вимоги мають бути протестованими;

– усі класи допустимих вхідних даних мають бути коректно обробленими;

– усі класи неприпустимих вхідних даних мають бути відкинутими системою, при цьому не повинна порушуватися стабільність її роботи;

– у тестах повинні генеруватися всі можливі класи вихідних даних системи;

– під час тестування система має побувати в усіх своїх внутрішніх станах і пройти по всіх можливих переходах між станами.

Результати системного тестування протоколюються й аналізуються аналогічно тому, як це робиться для модульного й інтеграційного тестувань. Основна складність тут полягає в локалізації дефектів у програмному коді системи й визначенні залежності одних дефектів від інших (ефект «парної кількості помилок»).

Тестування продуктивності. Цей вид тестування спрямовано на визначення того, чи забезпечує система належний рівень продуктивності при обробленні призначених для користувача запитів. Тестування продуктивності виконується при різних рівнях навантаження на систему, на різних конфігураціях обладнання. Вирізняють три основні чинники, що впливають на продуктивність системи: кількість потоків (наприклад, призначених для користувача сесій), що підтримуються системою, кількість вільних системних або апаратних ресурсів.

Тестування продуктивності дає змогу виявляти проблемні місця в системі в умовах підвищеного навантаження або браку системних ресурсів. У цьому випадку за висновками тестування проводять доробку системи, змінюють алгоритми виділення і розподілу ресурсів системи.

Усі вимоги, що належать до продуктивності системи, необхідно чітко визначити й до них обов'язково занести числові оцінки параметрів продуктивності. Наприклад, вимога «Система повинна мати прийнятний час відгуку на запит користувача» є непридатною для тестування. Навпаки, вимогу «Час відгуку на запит користувача не повинен перевищувати дві секунди» можна протестувати.

Те саме стосується й результатів тестування продуктивності. У звітах тестування зберігають такі показники, як завантаження апаратури й системного ПЗ (кількість циклів процесора, виділеної пам'яті, вільних системних ресурсів тощо). Важливими також є швидкісні характеристики системи, що тестується (кількість оброблених запитів за одиницю часу, часові інтервали між початком оброблення кожного наступного запиту, рівномірність часу відгуку в різні моменти часу тощо).

Для проведення тестування продуктивності необхідно мати генератор запитів, який подає на вхід системи потік даних, що є типовим для сеансу роботи з нею. Тестове оточення повинне мати крім програмної компоненти ще й апаратну, а тестовий стенд – давати змогу моделювати різний рівень доступних ресурсів.

Стресове тестування має багато загального з тестуванням продуктивності, проте його задача – не визначити продуктивність системи, а оцінити продуктивність і стійкість системи у разі, коли для своєї роботи система виділяє максимально доступну кількість ресурсів або працює в умовах критичного їх браку. Основна мета стресового тестування – вивести систему з ладу, визначити ті умови, за яких вона не зможе далі нормально функціонувати. Для проведення стресового тестування використовуються ті самі інструменти, що й для тестування продуктивності. Проте, наприклад, генератор навантаження при стресовому тестуванні має генерувати запити користувачів з максимально можливою швидкістю або генерувати дані запитів так, щоб вони були максимально можливими за об'ємом оброблення.

Стресове тестування є дуже важливим при тестуванні web-систем і систем з відкритим доступом, рівень навантаження на які часто дуже складно прогнозувати.

Тестування конфігурації. Більшість програмних систем масового призначення використовують різне обладнання. Незважаючи на те, що сьогодні особливості реалізації периферійних пристроїв приховуються драйверами операційних систем, які мають уніфікований для прикладних систем інтерфейс, проблеми сумісності (як програмної, так і апаратної) усе одно є.

Під час тестування конфігурації перевіряється коректність роботи програмної системи на всьому підтримуваному апаратному забезпеченні й спільно з іншими програмними системами, а також стабільність її роботи при «гарячій» заміні будь-якого підтримуваного пристрою на аналогічний. При цьому система не повинна давати збоїв ані в момент заміни пристрою, ані після початку роботи з новим пристроєм.

Необхідно також перевіряти, чи коректно система обробляє проблеми, які виникають в устаткуванні, як штатні (наприклад, сигнал кінця паперу в принтері), так і нештатні (збій живлення).

Тестування безпеки. Якщо програмну систему призначено для зберігання або оброблення даних, вміст яких є деякою таємницею (особистою, комерційною, державною тощо), то до властивостей системи, що забезпечують збереження цієї таємниці, будуть ставитися підвищені вимоги. Ці вимоги слід перевіряти шляхом тестування безпеки системи. Під час цього тестування перевіряється, чи не втрачається, не ушкоджується інформація, чи можна її підмінити або отримати несанкціонований доступ до неї, у тому числі з допомогою використання вразливих місць у самій програмній системі.

У вітчизняній практиці прийнято проводити сертифікацію програмних систем, призначених для зберігання даних службового користування, таєимних, цілком таємних, цілком таємних особливої важливості або керування критично важливими об’єктами, аварії на яких можуть призвести до значних втрат або жертв.

Незважаючи на те, що сертифікація – це процес, який виконується після верифікації, необхідно при тестуванні безпеки системи перевіряти такі вимоги:

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

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

– маркування й захист інформації, що передається іншим системам, – збереження рівня таємності навіть поза системою;

– ідентифікація й аутентифікація – надання доступу тільки санкціонованим користувачам і відмова всім іншим;

– реєстрація (аудит подій) – реєстрація в спеціальному журналі всіх подій системи, пов'язаних з безпекою, для подальшого аналізування;

– гарантії проектування й архітектури – систему необхідно спроектувати так, щоб була гарантія захищеності інформації;

– тестування – всі функції із забезпечення безпеки слід протестувати в усіх режимах;

– цілісність і відновлення засобів захисту – у системі повинні бути засоби контролю коректності всіх правил розмежування доступу й системи безпеки в цілому, а також засоби їх відновлення при збою;

– документація розробника, адміністратора й користувача – усі засоби системи із забезпечення безпеки необхідно описувати у відповідних інструкціях.

При розробленні й верифікації програмної системи, що потребує подальшої сертифікації, під час тестування безпеки необхідно перевіряти всі перелічені властивості.

Тестування надійності й відновлення після збоїв. Щоб система в будь-якій ситуації працювала коректно, необхідно упевнитися в тому, що вона відновлює свою функціональність і продовжує коректно працювати після будь-якої проблеми, яка перервала її роботу. При тестуванні відновлення системи після збоїв імітуються збої устаткування, що оточуює програмний продукт. При аналізуванні функціональності системи необхідно звертати увагу на два фактори – мінімізацію втрат даних унаслідок збою і мінімізацію часу між збоєм і продовженням нормального функціонування системи.

Тестування зручності використання. Окрема група нефункціональних вимог – це вимоги до зручності використання інтерфейсу системи, призначеного для користувача.

Після виконання всіх розглянутих видів тестування робиться висновок про функціональність і властивості системи, після чого проблемні місця системи доопрацьовуються до реалізації необхідної функціональності або досягнення системою необхідних властивостей.

2.16. Тестування інтерфейсу користувача

2.16.1. Мета і задачі тестування інтерфейсу користувача

Частина програмної системи, що забезпечує роботу інтерфейсу з користувачем, – один із найбільш нетривіальних об'єктів для верифікації.

З одного боку, інтерфейс, призначений для користувача, – це частина програмної системи. На інтерфейс користувача пишуться функциональні й низькорівневі вимоги, на основі яких складаються тестові вимоги й тестові плани. При цьому вимоги зазвичай визначають реакцію системи на кожну дію користувача (з допомогою клавіатури, миші або іншого пристрою) і вид інформаційних повідомлень системи, що виводяться на екран, друківний пристрій тощо. При верифікації таких вимог йдеться про перевірку функціональної повноти інтерфесу, призначеного для користувача, – наскільки реалізовані функції відповідають вимогам, чи коректно виводиться інформація на екран.

З іншого боку, інтерфейс користувача – це «обличчя» системи, і від того, як його продумано, залежить ефективність роботи з системою. Чинники, що впливають на ефективність роботи, менше піддаються формалізації у вигляді конкретних вимог до окремих елементів, проте

мають бути враховані у вигляді загальних рекомендацій і принципів інтерфейсу. Перевірка інтерфейсу на ефективність людино-машинної взаємодії отримала назву перевірки зручності використання (usability) або практичності.

2.16.2. Функціональне тестування інтерфейсу користувача

Функціональне тестування інтерфейсу користувача складається з таких фаз:

– аналіз вимог до інтерфейсу користувача;

– розроблення тестових вимог і планів перевірки інтерфейсу;

– виконання тестів і накопичення інформації про їх виконання;

– визначення повноти покриття вимогами інтерфейсу користувача;

– складання звітів про проблеми у разі розбіжності реальної функціональності системи і вимог або відсутності вимог на окремі інтерфейсні елементи.

Усі ці фази такі ж самі, як і при тестуванні будь-якого іншого компонента програмної системи. Відмінність полягає у трактуванні деяких термінів щодо їх застосування до інтерфейсу користувача і в особливостях автоматизованого накопичення інформації на кожній фазі.

Так, тестові плани для перевірки інтерфейсу користувача зазвичай є сценаріями, що описують дії користувача під час роботи з системою. Сценарії можуть бути записані як на природній, так і на формальній мові якої-небудь системи автоматизації тестування інтерфейсу користувача. Виконання тестів при цьому проводиться або оператором у ручному режимі, або системою, що емулює поведінку оператора.

Зазвичай аналізуються форми і їхні елементи (у разі графічного інтерфейсу) або тексти (у разі текстового інтерфейсу), що виводяться на екран, а не перевіряються значення тих або інших змінних програмної системи.

Під повнотою покриття інтерфейсу користувача розуміється те, що внаслідок виконання всіх тестів кожний інтерфейсний елемент було задіяно хоча б один раз у всіх доступних режимах.

Звіти про проблеми інтерфейсу користувача можуть містити описи як невідповідностей вимог і реальної функціональності системи, так і проблем у вимогах інтерфейсу користувача. Основне джерело проблем у цих вимогах – їхня тестопридатність, що спричинено розпливчастістю формулювань і неконкретністю.

2.16.3. Перевірка вимог до інтерфейсу користувача

Класифікація вимог до інтерфейсу

Вимоги до інтерфейсу користувача можна поділити на дві групи:

– вимоги до зовнішнього вигляду інтерфейсу і форм взаємодії з користувачем;

– вимоги щодо інтерфейсного доступу до внутрішньої функціональності системи.

Іншими словами, у першій групі вимог описується взаємодія підсистеми інтерфейсу з користувачем, а в другій – із внутрішньою логікою системи.

До першої групи можна віднести такі типи вимог.

Вимоги до розміщення елементів керування. За цими вимогами можна визначати загальні принципи розміщення елементів інтерфейсу або деталізувати вимоги до розміщення конкретних елементів. Наприклад, загальні вимоги з розміщення елементів на графічній екранній формі можуть мати такий вигляд:

> *Кожне програмне вікно слід поділити на три частини: рядок меню, робочу область і статусний рядок. Рядок меню має бути горизонтальним і притиснутим до верхньої частини вікна, статусний рядок – горизонтальним і притиснутим до нижньої частини вікна, робоча область має знаходитися між рядком меню й статусним рядком і займати всю площу вікна, що залишилася.*

При тестуванні цієї вимоги досить визначити, що в кожному вікні системи дійсно є три частини, які розташовані й притиснуті згідно з вимогами навіть при зміненні розмірів вікна, його згортанні/розгортанні, переміщенні по екрану, перекритті іншими вікнами.

Приклад вимог з розміщення конкретного елемента:

> *Кнопка «Почати друк» має знаходитися безпосередньо під рядком меню в лівій частині робочої області вікна.*

Під час тестування такої вимоги необхідно визначити, чи зберігається розташування елемента при зміненні розміру вікна, а також при використанні елемента (у цьому випадку – при натисненні).

Вимоги до змісту й оформлення повідомлень. За цими вимогами визначається текст повідомлення, що виводиться системою, його шрифтове й колірне оформлення. Часто в таких вимогах визначається, у яких випадках виводиться те або інше повідомлення.

Наприклад, для тестування вимоги

> *Повідомлення «Неможливо відкрити файл» має виводитися в статусний рядок притиснутим до лівого краю, виділеним червоним кольором напівжирним шрифтом у разі неможливості відкриття файла».*

необхідно перевірити, що при виникненні цієї ситуації повідомлення дійсно виводиться згідно з вимогами.

Проте у разі тестування вимоги вигляду

Повідомлення про помилки має виводитися в статусний рядок притиснутим до лівого краю і виділеним червоним кольором напівжирним шрифтом.

необхідно перевіряти формати всіх можливих повідомлень про помилки програми в усіх можливих помилкових ситуаціях. Таким чином, можна бачити, що при тестуванні інтерфейсу, призначеного для користувача, не завжди можна однозначно визначити кількість тестових завдань, які знадобляться для тестування вимоги. Ця проблема виникає через те, що вимоги до призначеного для користувача інтерфейсу часто здаються дуже очевидними для їх точного формулювання. Ця неконкретність і спричиняє велику кількість тестів для кожної вимоги.

Вимоги до форматів. За цією групою вимог визначається, у якому вигляді інформація надходить від користувача до системи. При цьому окрім власне вимог, що визначають коректний формат, до цієї групи належать вимоги, які визначають реакцію системи на некоректні дані. Для перевірки таких вимог необхідно перевіряти дані у коректному й некоректному форматі. Бажано при цьому поділяти різні варіанти на класи еквівалентності (щонайменше на два – коректні й некоректні).

До другої групи належать такі типи вимог.

Вимоги до реакції системи на введення користувача. За цим типом вимог визначається зв'язок внутрішньої логіки системи й інтерфейсних елементів, наприклад:

При натисненні кнопки «Скидання» значення таймера синхронізації передачі має скидатися до нуля.

Для перевірки такої вимоги у тесті слід зімітувати натиснення на кнопку «Скидання», після чого провести перевірку значення таймера. Проте деякі вимоги визначають як реакції системи не те, як змінюється її внутрішній стан, а як змінюється інтерфейс. Наприклад, у вимозі

При натисненні кнопки «Відкладене скидання» має виводитися вікно «Уведення значення часу для відкладеного скидання».

як реакції на використання одного інтерфейсного елемента визначається виникнення іншого інтерфейсного елемента. Такі вимоги перевіряються з допомогою імітації введення користувача й аналізування інтерфейсних елементів.

Вимоги до тривалості відгуку на команди користувача. Як окремий тип вимог можна виділити вимоги до тривалості відгуку системи на різні операції. Це пов'язано з тим, що підсвідомо користувач сприймає операції понад одну секунду як тривалі. Якщо в цей момент система не повідомляє користувачу про те, що вона виконує яку-небудь операцію, то

користувач почне вважати, що система зависла або працює в неправильному режимі. У зв'язку з цим або всі граничні тривалості відгуку треба вказати у вимогах і документації, призначеній для користувача, або під час тривалих операцій мають виводитися інформаційні повідомлення (наприклад, індикатор прогресу). Значення граничного часу і рівномірність збільшення значень індикатора прогресу слід перевіряти відповідними тестами.

Тестопридатність вимог до інтерфейсу користувача

Деякі вимоги до інтерфейсу можуть виявитися тестонепридатними або їх тестування буде значно утрудненим. До таких вимог належать насамперед вимоги, які описують суб'єктивні характеристики інтерфейсу, що не можна точно визначити або виміряти при виконанні тестів. Аналізуючи вимоги, необхідно чітко уявляти, який елемент інтерфейсу і яким чином буде перевірятися, як його характеристика буде вимірюватися під час тестування.

Прикладом тестонепридатної вимоги є класична вимога

Інтерфейс користувача має бути інтуїтивно зрозумілим.

Без визначення точних критеріїв інтуїтивної зрозумілості перевірка такої вимоги є неможливою. При цьому необхідно розуміти, що критерій у цьому випадку може бути двох видів: детермінованим або ймовірнісним. Прикладом детермінованого критерію є доповнення

Інтерфейс вважається інтуїтивно зрозумілим, коли будь-яка функція системи є доступною з допомогою не більше п’яти клацань миші по інтерфейсних елементах.

Така вимога піддається як ручному, так і автоматизованому тестуванню. Більш того, результат тестування не буде залежати від суб'єктивної думки тестувальника (поняття про інтуїтивну зрозумілість у всіх є різним).

Прикладом імовірнісного критерію може бути таке доповнення:

Інтерфейс вважається інтуїтивно зрозумілим, якщо користувач звертається до інструкції не частіше, ніж раз за п'ять хвилин на етапі навчання, і не частіше, ніж раз на дві години на етапі активного користування системою. Значення необхідно отримати на репрезентативній вибірці користувачів не менш ніж 1 000 осіб.

Перевірка вимоги з таким доповненням не є задачею класичної верифікації і вналежить швидше до перевірки зручності використання інтерфейсу. Проте тут також вводиться точний критерій, при використанні якого результати тестування можна відтворити.

Повнота покриття інтерфейсу користувача

При визначенні поняття покриття інтерфейсу можна ввести такі рівні покриття:

– функціональне – покриття вимог інтерфейсу;

– структурне – для забезпечення повного структурного покриття кожний інтерфейсний елемент необхідно використати в тестах хоча б один раз;

– структурне з урахуванням стану елементів інтерфейсу – для забезпечення цього рівня покриття необхідно не тільки використовувати кожний елемент інтерфейсу, але й привести його у всі можливі стани (наприклад, для чек-боксів – відмічений/невідмічений, для полів уведення – порожнє/заповнене не повністю/заповнене повністю тощо);

– структурне з урахуванням стану елементів інтерфейсу й внутрішнього стану системи – функціональність деяких інтерфейсних елементів може змінюватися залежно від внутрішнього стану системи. Кожний з таких елементів слід перевірити. Наприклад, система може мати два режими роботи – для досвідченого і для недосвідченого користувача, у якому натиснення кожного елемента супроводжується підказкою. У цьому випадку потрібно в кожному з режимів перевірити, що підказки виникають тільки в режимі для початківців.

При визначенні ступеня покриття необхідно враховувати, що реакція на деякі інтерфейсні елементи визначається не програмною системою, а на рівні операційної системи або середовища виконання. Так, наприклад, реакція на використання більшості інтерфейсних елементів стандартного діалогового вікна відкриття файла визначається операційною системою і може не тестуватися.

Якщо рівень покриття інтерфейсних елементів тестами є недостатнім, то це свідчить про необхідність уточнення вимог до інтерфейсу або до зниження ступеня деталізації тестування.

2.16.4. Методи тестування інтерфейсу

Функціональне тестування інтерфейсу користувача може проводитися різними методами: як вручну при безпосередній участі оператора, так і з допомогою різного інструментарію, що автоматизує виконання тестів. Розглянемо ці методи більш детально.

Ручне тестування інтерфейсу проводить тестувальник-оператор, який керується в своїй роботі описом тестів у вигляді набору сценаріїв. Кожний сценарій містить перелік послідовності дій, які має виконати оператор, і опис важливих для аналізу результатів тестування відповідних реакцій системи, що відображаються інтерфейсом. Типовою формою запису сценарію для проведення ручного тестування є таблиця, у якій в

одному стовпці описано дії (кроки) сценарію, у другому – очікувану реакцію системи, а в третьому – відповідність очікуваної реакції системи реальній і перелік розбіжностей.

Ручне тестування інтерфейсу є зручним у тому сенсі, що контроль коректності інтерфейсу проводить людина, тобто основний «споживач» цієї частини програмної системи. До того ж під час чисто косметичних змін в інтерфейсах системи, не відображених у вимогах (наприклад, при переміщенні кнопок керування на 10 пікселів ліворуч), аналізування успішності проходження тесту буде виконуватися не за формальними ознаками, а згідно з людським сприйняттям.

Ручне тестування має також істотний недолік – для його проведення потрібні значні людські й часові ресурси. Особливо сильно цей недолік виявляється при проведенні регресійного (і взагалі будь-якого повторного) тестування – на кожній ітерації повторного тестування інтерфейсу потрібна участь тестувальника-оператора. У зв'язку з цим отримали розповсюдження засоби автоматизації тестування інтерфейсу, що значно зменшують навантаження на тестувальника-оператора.

Сценарії на формальних мовах. Звичайний спосіб автоматизації тестування інтерфейсу – використання програмних інструментів, що емулюють поведінку тестувальника-оператора при ручному тестуванні. Такі інструменти використовують як вхідну інформацію сценарії тестових завдань, які записано на деякій формальній мові, оператори якої відповідають діям користувача – уведенню команд, переміщенню курсора, активізації пунктів меню та інших інтерфейсних елементів.

При виконанні автоматизованого тесту інструмент тестування імітує дії користувача, які описано в сценарії, і аналізує інтерфейсну реакцію системи. При цьому для визначення очікуваного стану інтерфейсу можуть використовуватися різні методи: або аналіз знімків екрану й порівняння їх з еталонними, або доступ до цих інтерфейсних елементів засобами операційної системи (наприклад, доступ до всіх кнопок вікна за їхніми дескрипторами й отримання значень тексту).

І при переданні інформації в інтерфейс, і при отриманні інформації для аналізу можуть використовуватися два способи доступу до елементів інтерфейсу:

– позиційний, при якому доступ до елемента здійснюється з допомогою завдання його абсолютних (щодо екрану) або відносних (щодо вікна) координат і розмірів;

– за ідентифікатором, при якому доступ до елемента здійснюється з допомогою його унікального ідентифікатора у межах вікна.

При внесенні змін в інтерфейс з використанням першого методу внаслідок проведення регресійного тестування буде виявлено велику кількість пройдених тестів – досить змінити місцеположення одного ключового інтерфейсного елемента, як всі сценарії почнуть працювати

неправильно. За таким методом автоматизації тестування необхідно змінювати значну частину сценаріїв у системі тестів при кожному зміненні інтерфейсу системи і його можно застосовувати для систем з інтерфейсом, змінення яких є малоймовірним.

Другий метод автоматизації тестування є більш стійким до змінення розташування інтерфейсних елементів, але може потребувати змінення тестів у разі змінення логіки роботи інтерфейсних елементів. Наприклад, у першій версії системи при натисненні на кнопку «Передати дані» передання даних починалося відразу і виводилося вікно з індикатором прогресу. Сценарій тесту в цьому випадку містить імітацію натиснення на кнопку і звернення до індикатора прогресу для отримання значення прогресу у відсотках. Якщо в другій версії системи після натиснення на кнопку «Передати дані» спочатку виводиться вікно «Ви впевнені?» з кнопками «Так» і «Ні», то для перевірки роботи індикатора прогресу в тестовий сценарій необхідно додати імітацію натиснення кнопки «Так».

Навіть при використанні ідентифікаторів без модифікації тестового завданя тест не буде коректно виконуватися. Проте цей спосіб звернення до інтерфейсних елементів є придатним для тестування програмних систем, у тому числі інтерфейсів, що часто змінюються.

2.16.5. Тестування інтерфейсу на зручність використання

Зручність використання інтерфейсу користувача (usability) – це показник якості, що визначає кількість зусиль, необхідних для вивчення принципів роботи з програмною системою з допомогою цього інтерфейсу, підготовки вхідних та інтерпретацій вихідних даних. Інакше кажучи, зручність використання визначає ступінь простоти доступу користувача до функцій системи, що надаються через людино-машинний інтерфейс.

Тестування інтерфейсу на зручність використання, взагалі кажучи, не належить до класичних методів тестування програмних систем. Фахівець із тестування інтерфейсу має поєднувати знання в області як програмної інженерії, так і фізіології, психології й ергономіки.

На зручність використання інтерфейсу впливають:

– легкість навчання – як швидко людина вчиться використовувати систему;

– ефективність навчання – як швидко людина працює після навчання;

– запам'ятовуваність навчання – як легко запам'ятовується все, чому людина навчилася;

– помилки – як часто людина припускається помилок у роботі;

– загальна задоволеність – чи є загальне враження від роботи з системою позитивним.

Усі ці чинники, незважаючи на свою неформальність, можна виміряти. Для цього вибирається група типових користувачів системи і

вимірюються показники їхньої роботи (наприклад, кількість допущених помилок), а також їм пропонується виказати власні враження від роботи із системою шляхом заповнення відповідних анкет.

Вирізняють такі етапи тестування зручності використання інтерфейсу:

1. Дослідне – проводиться після формулювання вимог до системи й розроблення прототипу інтерфейсу. Основна мета на цьому етапі – проведення високорівневого обстеження інтерфейсу і з'ясування можливості вирішення з достатнім ступенем ефективності завдань користувача.

2.Оцінювальне – проводиться після розроблення низькорівневих вимог і прототипу інтерфейсу, що деталізується, заглиблює дослідне тестування і має ту саму мету. На цьому етапі проводяться кількісні вимірювання характеристик інтерфейсу: кількість звернень до системи допомоги відносно загальної кількості операцій, кількість помилкових операцій, час усунення наслідків помилкових операцій тощо.

3. Валідаційне – проводиться ближче до етапу завершення розроблення. На цьому етапі проводиться аналіз відповідності інтерфейсу програмної системи стандартам, що регламентують питання зручності інтерфейсу, загальне тестування всіх компонентів інтерфейсу з погляду користувача. Під компонентами інтерфейсу розуміється як його програмна реалізація, так і система допомоги й інструкція користувача. На цьому етапі також перевіряється відсутність дефектів незручностей при використанні інтерфейсу, виявлених на попередніх етапах.

4. Порівняльне – цей вид тестування можна проводити на будь-якому етапі розроблення інтерфейсу. Під час порівняльного тестування порівнюються два або більше варіантів реалізації інтерфейсу, призначеного для користувача.

Зазвичай при тестуванні інтерфейсу на зручність використовують деякі евристичні критерії і характеристики, які замінюють точні оцінювання в класичному тестуванні програмних систем.

Наприклад, існує десять евристичних характеристик, які повинні перевірятися при тестуванні інтерфейсу на зручність.

Спостережуваність стану системи. Система завжди має оповіщати користувача про те, що вона в певний момент робить, причому через розумні проміжки часу.

Співвідношення з реальним світом. Термінологія, яку використано в інтерфейсі системи, має співвідноситися з призначеним для користувача світом, тобто це має бути термінологія проблемної області користувача, а не технічна.

Призначене для користувача керування і свобода дій. Користувачі часто вибирають окремі інтерфейсні елементи і застосовують функції системи помилково. У цьому випадку необхідно надавати точно

визначений «аварійний вихід», з допомогою якого можна повернутися до попереднього нормального стану. До таких «аварійних виходів» належать, наприклад, функції відкоту та зворотного відкоту.

Цілісність і стандарти. Для позначення одних і тих самих об'єктів, ситуацій і дій мають використовуватися однакові слова в усіх частинах інтерфейсу. Більш того, у термінології повідомлень в інтерфейсі мають враховуватися угоди конкретної платформи.

Допомога користувачам в розпізнаванні, діагностиці й усуненні помилок. Повідомлення про помилки має бути написано на звичайній мові, а не замінюватися кодами помилок. У повідомленнях про помилки має бути точно визначено суть проблеми, що виникла, і запропоновано її конструктивне рішення.

Запобігання помилкам – продуманий дизайн інтерфейсу, що запобігає появі помилок користувача, завжди краще добре продуманих повідомлень про помилки. При проектуванні інтерфейсу необхідно або повністю усунути елементи, у яких можуть виникати помилки користувача, або перевіряти дані, що ввів користувач у цих елементах, і повідомляти його про потенційно можливе виникнення проблеми.

Розпізнавання, а не згадування – при створенні інтерфейсу необхідно мінімізувати навантаження на пам'ять користувача, роблячи об'єкти, дії та опції ясними, доступними і явно видимими. Користувач не повинен запам'ятовувати інформацію при переході від одного діалогового вікна до іншого. В усіх необхідних місцях мають бути доступними контекстні інструкції щодо використання інтерфейсу.

Гнучкість і ефективність використання – в інтерфейсі має бути передбачено «гарячі клавіші» (не обов'язкові до використання користувачем-початківцем), які часто значно прискорюють роботу досвідченого користувача. Іншими словами, система має надавати два способи роботи – для новачків і досвідчених користувачів. Бажано при цьому давати можливість користувачу автоматизувати дії, що часто повторюються.

Естетичний і мінімально необхідний дизайн. Вікна не повинні містити інформацію, що не стосується справи або мало використовується. Кожний інтерфейсний елемент, що містить непотрібну інформацію, є інформаційним шумом і відволікає користувача від дійсно корисних інтерфейсних елементів.

Допомога і документація. Незважаючи на те, що в ідеальному випадку краще, коли системою можна користуватися без документації, вона все одно є необхідною як у вигляді системи допомоги, так і, можливо, надрукованої інструкції користувача. Інформація в документації має бути структурованою так, щоб користувач міг легко знайти потрібний розділ, де описано завдання, що він вирішує. Кожний такий розділ, орієнтований на

конкретне завдання, повинен крім загальної інформації мати покрокову інструкцію щодо виконання завдань і не бути занадто довгим.

Усі ці евристики можна задіяти при тестуванні інтерфейсу на зручність. Очевидно, що при тестуванні на зручність мало застосовуються способи автоматизації тестування з допомогою сценаріїв і такі інші методи. Один із найбільш ефективних методів перевірки інтерфейсу на зручність – використання формальної інспекції. Питання в бланку інспекції можуть бути як загальними (наприклад, можна використати перелічені вище 10 евристик), так і конкретними.

2.17. Приймально-здавальні та сертифікаційні випробування

При розробленні масового ПЗ після системного тестування програмний продукт проходить етапи α- і β-тестування, під час яких роботу системи перевіряють потенційні користувачі (або спеціально призначені фокус-групи користувачів, або всі, хто забажає). На цьому етапі до програмної системи вносяться останні незначні зміни, що суттєво не впливають на роботу системи. Після завершення цієї стадії система надходить до продажу кінцевим користувачам.

При розробленні ПЗ на замовлення фази α- і β-тестування замінюють приймально-здавальні випробування. Під час цих випробувань замовник упевнюється, що система працює відповідно до його потреб (як зафіксованими в технічному завданні на систему, так і не зафіксованими). Замовник може проводити такі випробування самостійно, виконуючи наперед підготовлені тести системи, або спільно з представниками колективу розробників. У цьому випадку тести також готуються розробниками, наприклад, на основі тестів, що використовувалися на етапі системного тестування.

Після завершення приймально-здавальних випробувань або підписується акт приймання, або замовник ставить додаткові вимоги до системи, які необхідно виправити до строку приймання системи. Після усунення всіх недоліків системи приймально-здавальні випробування повторюються (можливо, за скороченою програмою). Після успішного підписання акту система надходить до експлуатації замовнику.

Існує спеціальний вид програмних систем, до властивостей яких ставляться особливі вимоги. Прикладом таких систем можуть бути бортові авіаційні програмні системи, у яких особлива увага приділяється питанням безпеки, надійності й відмовостійкості. Незважаючи на те, що велику частину таких систем можна віднести до категорії ПЗ, що створюється на замовлення, для отримання дозволу на установлення системи на борт необхідно отримати сертифікат на льотну придатність.

Таким чином, після проведення системного тестування й приймально-здавальних випробувань проводяться сертифікаційні

випробування. Сертифікація ПЗ – процес установлення й офіційного визнання того, що розроблення ПЗ проводилося відповідно до певних вимог. Під час сертифікації відбувається взаємодія заявника, сертифікуючого й наглядового органів.

Заявник – це організація, що подає заявку до відповідного сертифікуючого органу на отримання сертифіката (відповідності, якості, придатності тощо) виробу.

Сертифікуючий орган – організація, що розглядає заявку заявника про проведення сертифікації ПЗ і або самостійно, або шляхом формування спеціальної комісії проводить процедури, які спрямовано на проведення процесу сертифікації ПЗ заявника.

Наглядовий орган – комісія фахівців, що спостерігають за процесами розроблення заявником інформаційної системи, яка передається до розгляду до сертифікуючого органу, і визначають відповідність цього процесу певним вимогам.

Основним об'єктом перевірки у сертифікаційних випробуваннях є процес розроблення програмної системи регламенту й рекомендацій стандарту, на відповідність якому проводиться сертифікація. Така відповідність визначається з допомогою аналізування життєвого циклу системи й документів, що створюються на ключових етапах. Весь процес аналізу і ті властивості системи, які піддаються сертифікації, описуються в плані сертифікаційних випробувань, що затверджується спільно заявником і сертифікуючим органом.

У разі сертифікації бортової системи за планом додатково визначається рівень впливу відмови програмної системи на безпеку польоту (рівень відмовобезпеки), за яким буде проводитися сертифікація. Будь-які питання, що виникають у сертифікуючого органу щодо змісту плану сертифікаційних випробувань, слід розв’язати до початку самих випробувань.

План сертифікаційних випробувань (план програмних аспектів сертифікації) повинен містити:

– *огляд системи* – у цьому розділі описується система, у тому числі її функції і їхнє розміщення в програмному й апаратному забезпеченні, архітектура, процесор (процесори), що використовується, апаратно-програмний інтерфейс і особливості відмовобезпеки;

– *огляд ПЗ* – у цьому розділі коротко описуються функції ПЗ з акцентом на концепцію забезпечення відмовобезпеки і поділ на відособлені частини, наприклад розподіл ресурсів, резервування, несиметрично резервоване ПЗ, стійкість до відмов тощо;

– *сертифікаційні міркування* – у цьому розділі міститься зведення сертифікаційного базису, у тому числі засоби підтвердження відповідності, як це визначено у програмних аспектах сертифікації, також заявляється запропонований рівень (рівні) ПЗ і наводяться підтвердження правильності

цього рівня, які отримано під час оцінювання відмовобезпеки системи, а також потенційний внесок ПЗ у відмовні ситуації;

– *життєвий цикл ПЗ* – у цьому розділі визначається життєвий цикл ПЗ, який буде використано, а також зведення його процесів, детальна інформація про які визначається у відповідних планах ПЗ. У зведенні роз'яснюється, як будуть задовольнятися цілі кожного процесу життєвого циклу, визначаються організації, що будуть залучатися, організаційна відповідальність, а також відповідальність за процеси життєвого циклу системи і за процес підтримки контактів під час сертифікації;

– *дані життєвого циклу ПЗ* – у цьому розділі визначаються дані життєвого циклу, які будуть контролюватися, а також описується зв'язок даних між собою або з іншими даними, що визначають систему, дані життєвого циклу ПЗ, які надаються сертифікуючим органом, а також форма даних, і засоби, з допомогою яких дані життєвого циклу ПЗ можна зробити доступними для сертифікуючих органів;

– *план-графік* – у цьому розділі описуються засоби, які заявник буде використовувати, щоб забезпечити для сертифікуючих органів осяжність діяльності в процесах життєвого циклу ПЗ і, отже, можливість планування перевірок;

– *додаткові міркування* – у цьому розділі описуються особливості, що можуть вплинути на процес сертифікації, наприклад альтернативні методи підтвердження відповідності; кваліфікація інструментальних засобів; раніше розроблене ПЗ; варіантне ПЗ, яке можна вибрати за бажанням; ПЗ, яке буде доступним для модифікації користувачем; готове ПЗ інших розробників, що використовується без модифікацій; ПЗ, яке завантажується в польових умовах; несиметрично резервоване ПЗ або історія експлуатації продукту.

Під час сертифікаційних випробувань заявник надає свідчення того, що процеси життєвого циклу ПЗ задовольняють планам ПЗ. Заявник організовує доступ сертифікуючого органу до даних життєвого циклу ПЗ. При цьому мінімальний перелік цих даних містить:

– *план сертифікаційних випробувань* (план програмних аспектів сертифікації);

– *індекс конфігурації ПЗ* – документ, який має однозначно ідентифікувати кожний компонент проекту (у тому числі вимоги, програмний, об'єктний і виконуваний код), середовище реалізації системи, інструкції з компіляції системи, апаратне і програмне забезпечення для роботи системи, апаратне і прорамне забезпечення для проведення сертифікації;

– *підсумковий висновок про ПЗ*, що є основним документом для демонстрації відповідності ПЗ Плану програмних аспектів сертифікації.

Підсумковий висновок повинен містити такі підрозділи:

– *огляд системи* – цей розділ містить огляд системи, у тому числі

опис її функцій і їх розміщення в апаратному і програмному забезпеченні, архітектуру, використовуваний процесор (процесори), апаратно-програмний інтерфейс, засоби забезпечення відмовобезпеки. У цьому розділі також описуються всі відмінності системи, раніше описані в плані програмних аспектів сертифікації;

– *огляд ПЗ*, де стисло описуються функції ПЗ; тут особлива увага надається концепції відмовобезпеки, що використовується, і поділу на відособлені частини, а також роз'яснюються відмінності від огляду ПЗ, раніше внесеного до плану програмних аспектів сертифікації;

– *сертифікаційні міркування*, які повторно формулюють сертифікаційні міркування, наведені у плані програмних аспектів сертифікації, а також описують будь-які відмінності від раніше наведених міркувань;

– *характеристики ПЗ,* де констатуються дані про розмір виконуваного коду, запаси часу і пам'яті, обмеження ресурсів, а також описуються засоби для вимірювання кожної характеристики;

– *життєвий цикл ПЗ* – тут підсумовується реальний життєвий цикл (цикли) ПЗ і роз'яснюються відмінності від життєвого циклу ПЗ і процесів життєвого циклу, раніше запропонованих в плані програмних аспектів сертифікації;

– *дані життєвого циклу ПЗ* – тут дається посилання на дані процесів розроблення ПЗ і забезпечення цілісності, у ньому описується зв'язок даних між собою і з іншими даними, що визначають систему, і засоби, з допомогою яких до даних життєвого циклу ПЗ може бути забезпечено доступ сертифікуючих органів; у цьому розділі також описуються будь-які відмінності від опису даних життєвого циклу, раніше занесених до плану програмних аспектів сертифікації;

– *додаткові міркування* – у цьому розділі підсумовуються питання, які можуть привернути увагу сертифікуючих органів, і даються посилання на створені документи або спеціальні умови, що стосуються цих питань;

– *ідентифікація ПЗ* – у цьому розділі ідентифікується конфігурація ПЗ за номенклатурним номером або версією;

– *історія змін* – цей розділ містить зведення змін ПЗ, особливу увагу приділяється змінам, зробленим для виправлення помилок, що впливають на відмовобезпеку, а також ідентифікацію змін в процесах життєвого циклу ПЗ з часу попередньої сертифікації;

– *статус ПЗ* – цей розділ містить зведення повідомлень про проблеми, які не було вирішено на момент сертифікації, у тому числі заяви про функціональні обмеження.

– *заява про відповідність* – цей розділ містить заяву про відповідність ПЗ цьому документу, зведення методів, використаних для демонстрації відповідності з визначенням критеріїв, специфікованих в планах ПЗ, у ньому наводяться додаткові критерії відносно планів ПЗ, стандартів і цього

документа, а також використані правила і відхилення від планів і стандартів.

Повний перелік даних ЖЦ для сертифікації ПЗ є таким:

– план програмних аспектів сертифікації;
– план розроблення ПЗ;
– план верифікації ПЗ;
– план керування конфігурацією ПЗ;
– план гарантії якості ПЗ;
– стандарти на вимоги до ПЗ;
– стандарти проектування ПЗ;
– стандарти на код ПЗ;
– дані вимог на ПЗ;
– опис проекту;
– програмний код;
– виконуваний об'єктний код;
– тести й тестові процедури верифікації ПЗ;
– звіт за результатами верифікації ПЗ;
– індекс конфігурації навколишнього середовища ЖЦ ПЗ;
– індекс конфігурації ПЗ;
– повідомлення про проблеми;
– документи з керування конфігурацією ПЗ;
– документи з гарантії якості ПЗ;
– підсумковий висновок щодо ПЗ.

Сертифікуючий орган встановлює так званий сертифікаційний базис для системи під час консультацій із заявником, у якому визначено конкретні правила разом з будь-якими спеціальними умовами, що можуть доповнювати опубліковані правила сертифікації, які регламентовано у стандартах.

Установка базису для ПЗ проводиться після підсумкового висновку про ПЗ і отримання свідоцтва відповідності.

Сертифікуючий орган оцінює план програмних аспектів сертифікації на повноту й узгодженість з критеріями оцінювання відмовобезпеки системи й іншими даними життєвого циклу ПЗ. Якщо всі дані життєвого циклу є правильними (це є доказом того, що під час проекту було активовано всі необхідні процеси розроблення й верифікації), то сертифікуючий орган видає ствердне рішення про видачу сертифіката.

Існують два типи сертифікатів ПЗ.

Сертифікат якості – свідоцтво, що підтверджує якість фактично поставленого товару і його відповідність умовам договору. У сертифікаті якості дається характеристика товару або підтверджується відповідність товару певним стандартам або технічним вимогам замовника. Сертифікат якості видається компетентними організаціями і спеціальними лабораторіями в країнах як експорту, так і імпорту. Сторони договору

купівлі-продажу можуть домовитися про надання сертифікатів різних контрольних і перевірних установ.

Сертифікат відповідності – документ, що є результатом дій третьої організації, який доводить, що забезпечується необхідна впевненість у тому, що належним чином ідентифікована продукція, процес або послуга відповідають конкретному стандарту або іншому нормативному документу.

Сертифікат на льотну придатність поєднує в собі властивості обох типів сертифікатів. З одного боку, він підтверджує, що розроблена система має певний рівень якості реалізації, а з іншого – що процеси з її розроблення відповідають міжнародному галузевому стандарту.

2.18. Формальні інспекціїї

2.18.1. Мета і задачі формальних інспекцій

Не завжди можна розробити автоматичні або хоча б точно формалізовані ручні тести для перевірки функціональності ПС. У деяких випадках виконання програмного коду є неможливим в умовах, які створюються тестовим оточенням. Така ситуація є можливою у вбудованих системах, якщо програмний код призначено для оброблення виняткових ситуацій, які створюються тільки на реальному обладнанні.

У тих випадках, коли верифікується не програмний код, а проектна документація на систему, яку не можна «виконати» або створити для неї окремі тести, зазвичай застосовують метод експертних досліджень програмного коду або документації на коректність чи несуперечність.

Такі експертні дослідження називають інспекціями, або переглядами. Існує два типи інспекцій – неформальні й формальні.

При неформальній інспекції автор документа або частини програмної системи передає його експерту, який після ознайомлення з документом передає автору список зауважень, що необхідно виправити. Сам факт проведення інспекції і зауваження зазвичай ніде окремо не зберігаються, стан виправлень за зауваженнями ніде не відстежується.

Формальна інспекція є точно керованим процесом, структура якого визначається відповідним стандартом проекту. Таким чином, усі формальні інспекції мають однакову структуру й однакові вихідні документи, які потім використовуються під час розроблення.

Факт початку формальної інспекції точно фіксується в загальній базі даних проекту. Фіксуються також документи, що інспектуються, списки зауважень, відстежуються внесені згідно із зауваженнями зміни. Цим формальна інспекція схожа на автоматизоване тестування – списки зауважень багато в чому збігаються зі звітами про виконання тестів.

Під час формальної інспекції групою фахівців здійснюється незалежна перевірка відповідності інспектованих документів початковим

документам. Незалежність перевірки забезпечується тим, що її здійснюють інспектори, які не брали участі в розробленні інспектованого документа. Входами процесу формальної інспекції є інспектовані й початкові документи, а виходами – матеріали інспекції, що містять список знайдених невідповідностей і рішення про змінення статусу інспектованих документів.

2.18.2. Етапи формальної інспекції та функції її учасників

Формальна інспекція поділяється на п'ять фаз: ініціалізація, планування, експертиза, обговорення, завершення. У деяких випадках підготовку й обговорення доцільно розглядати не як послідовні етапи, а як паралельні підпроцеси. Зокрема, така ситуація може мати місце при використанні автоматизованої системи підтримки проведення формальних інспекцій. Процедура формальної інспекції проекту має точно описувати порядок проведення формальних інспекцій в цьому проекті.

Після усунення знайдених під час формальної інспекції невідповідностей процес формальної інспекції повторюється, можливо, в іншій формі і з іншим складом учасників. Процедура формальної інспекції має регламентувати можливі форми проведення повторної інспекції залежно від об'єму й характеру змін, внесених в об'єкт інспекції. Зазвичай допускається спрощення процесу повторної інспекції (проведення інспекції одним інспектором, відсутність фази обговорення) при внесенні в об'єкт інспекції незначних змін щодо версії, що раніше інспектувалася.

Ініціалізація формальної інспекції

Керівник проекту або його заступник переглядає з бази, що зберігає всі дані проекту (наприклад, із системи конфігураційного керування), список об'єктів, готових до інспекції, вибирає об'єкт інспекції, потім призначає учасників формальної інспекції: автора, ведучого й одного або кількох інспекторів. Ведучий також може виконувати функції інспектора; решта учасників виконує тільки одну функцію. Ведучим або інспектором не можуть бути співробітники, що брали участь в розробленні об'єкта інспекції.

Зазвичай автором є один із розробників об'єкта інспекції, але можливі ситуації, коли розробника переведено до іншого проекту або він знаходиться у відпустці. Тоді функції автора може виконувати співробітник, який буде виправляти знайдені невідповідності в інспектованих документах. Для інспекції документів, розроблених замовником, автор може не призначатися.

Рекомендується призначати не менше двох інспекторів. Кількість інспекторів можна збільшити, якщо інспектуються документи, які мають особливу складність або новизну понять, а також якщо інспекторами є

співробітники з невеликим досвідом. Проте загальна кількість учасників інспекції не повинна бути більше п'яти.

У деяких випадках процедура формальної інспекції може бути проведена одиним інспектором, наприклад, коли об'єкт інспекції є особливо простим і його оцінювані характеристики – тривіальні. Прикладом може бути пакет результатів структурного покриття, який одержано після виконання раніше проінспектованих тестів і для якого перевіряється тільки склад пакета і узгодженість версій.

Якщо необхідна повторна інспекція за укороченою формою, ведучий самостійно ініціює процес без участі керівника проекту. Процедура формальної інспекції дає змогу ведучому самостійно ініціювати процес повторної інспекції (у тому самому складі учасників), навіть коли його проводять у повній формі, якщо це необхідно за специфікою проекту.

Планування формальної інспекції

Після ініціалізації формальної інспекції ведучий перевіряє, чи розміщено інспектовані документи в базі даних проекту, а їхній статус відповідає готовності до формальної інспекції. Якщо це не так, інспекція відкладається. Потім він повинен змінити статус інспектованих документів так, щоб відзначити факт початку інспекції і обмежити доступ до інспектованої документації. Під час інспекції змінити документи неможливо, а відповідний статус зберігається до закінчення інспекції (далі будемо називати цей статус Review). Після цього ведучий повинен скопіювати з бази даних проекту бланк інспекції і занести до нього ідентифікатори інспектованих і початкових документів, номера їхніх версій, список учасників із зазначенням функцій і дату фактичного початку процесу інспекції, тобто того моменту, коли інспектовані документи було переведено в стан Review.

Ведучий повинен розрахувати час, необхідний інспекторам для підготовки, і тривалість обговорення. Час, що відводиться на етап підготовки, не може бути менше однієї години. Ведучій також повинен визначити дату, час і місце обговорення, якщо воно буде проходити як нарада. При цьому може знадобитися узгодження з іншими учасниками інспекції. Якщо обговорення буде тривати більше двох годин, то необхідно запланувати декілька нарад, кожна з яких буде тривати не більше двох годин.

При проведенні процедури формальної інспекції може допускатися повторна інспекція без нарад, якщо результатом попередньої інспекції було рішення про проведення повторної інспекції в укороченій формі. Так само допускається не проводити наради, якщо результати формальної інспекції зберігаються в електронному вигляді. У цьому випадку у процедурі формальної інспекції проекту має регламентуватися взаємодія

учасників формальної інспекції. Крім того, у процедурі формальної інспекції проекту мають визначатися механізм підготовки, проведення обговорення й прийняття рішення.

Ведучий визначає час і місце наради, сповіщає учасників інспекції про це і розсилає їм підготовлений бланк інспекції.

У процедурі формальної інспекції проекту може передбачатися використання бланка, заповненого під час попередньої інспекції, за умови, що проводиться повторна інспекція в скороченій формі і при цьому ведучий є єдиним інспектором.

Підготовка формальної інспекції

Отримавши лист або призначення з бланком інспекції, інспектори повинні, використовуючи зазначені в бланку ідентифікатори і номери версій, витягнути з бази даних проекту необхідні документи, при цьому переконатися, що всі документи перебувають у відповідному стані.

Під час підготовки інспектори детально вивчають інспектовані документи, керуючись списком контрольних запитань. Знайдені невідповідності слід точно локалізувати, сформулювати й записати.

При проведенні повторної інспекції в скороченій формі допускається обмежитися аналізуванням змін відносно тієї версії об'єкта інспекції, яка інспектувалася раніше. Якщо при цьому виявляється, що є зміни, не пов'язані із зафіксованими зауваженнями, то процес інспекції припиняється і призначається нова інспекція в повній формі. Виключенням з цього правила може бути випадок, коли такі зміни полягають у виправленні тривіальних помилок, що не стосується суті інспектованих документів, таких, наприклад, як друкарські помилки в коментарях, що не впливають на значення фрази.

Якщо інспектор, який проводить повторну інспекцію в укороченій формі, вважає, що обсяг змін дуже великий або зміни дуже складні, то він має право перервати процес інспекції, сповістивши керівника проекту про призначення нової інспекції в повній формі.

Автор, якщо він не є розробником об'єкту інспекції, повинен під час підготовки детально з ним ознайомитися, щоб бути готовим відповідати на запитання інспекторів під час обговорення, а після завершення інспекції – усунути знайдені невідповідності.

Обговорення об’єкта інспекції

Обговорення проводиться у формі однієї або кількох нарад, кожна з яких триває не більше двох годин. За один день рекомендується проводити не більше однієї наради. Якщо обговорення не укладається в заплановану кількість нарад, то призначаються додаткові наради, для проведення яких необхідна присутність ведучого, хоча б одного з

інспекторів і зазвичай автора. Проте ведучий може на свій розсуд провести нараду без автора, якщо той хворіє або з якої-небудь іншої причини не може бути присутнім на нараді, за умови, що жоден з інспекторів не знайшов невідповідностей або їхні зауваження є очевидними й не потребують роз'яснень з боку автора (або з автором встановлено телефонний зв'язок). Якщо нараду було розпочато без автора, а надалі виникла необхідність у його присутності, то ведучий повинен припинити й відкласти нараду.

Нарада відкладається, якщо жоден з інспекторів не підготувався до обговорення. Ведучий також може на свій розсуд відкласти нараду, якщо не підготувався або немає хоча б одного з інспекторів.

Під час обговорення ведучий синхронізує роботу учасників, зачитуючи інспектовний документ або послідовно називаючи розділи чи абзаци тексту або елементи діаграм, або якимось іншим способом забезпечує синхронний перегляд документа всіма учасниками. Під час зачитування документа інспектори переривають ведучого в тих місцях, до яких є зауваження. У разі відсутності розбіжностей ведучий фіксує невідповідність і продовжує зачитувати документ. При інспекції документів невеликого обсягу ведучий на свій розсуд може не синхронізувати перегляд документа всіма учасниками, а просто опитувати учасників про зауваження. Такі опитування можна об'єднувати із заповненням списку контрольних запитань.

Якщо думки учасників щодо якогось зауваження не збігаються, то ведучий послідовно надає слово всім, хто хоче висловитися, причому автор користується правом позачергового надання слова. Якщо внаслідок дискусії змінилося формулювання зауваження, то ведучий записує його, потім зачитує і, якщо всі учасники з ним згодні, продовжує зачитувати документ.

Результатом дискусії може також бути визнання відсутності проблеми. У цьому випадку ведучий переконується, що всі з цим згодні, і продовжує зачитувати документ.

Учасники мають прагнути виявити проблеми, але не шукати їх вирішення. Досягнення консенсусу зі спірних питань також не є ціллю дискусії. Якщо є розбіжність в думках, то необхідно зафіксувати всі альтернативні думки. Ведучий повинен припинити дискусію, якщо оцінює її як непродуктивну.

Усі учасники зобов'язані шанобливо ставитися до опонентів, не перебивати того, хто говорить, і висловлюватися тоді, коли ведучий надасть їм слово. Не допускаються паралельні обговорення вузьким складом – кожний учасник зобов'язаний адресувати свої вислови всім учасникам зборів, а не сусіду.

Необхідно також уникати критики й оцінювання кваліфікації колег. Метою інспекції є підвищення якості інспектованих документів, а не

оцінювання кваліфікації автора або інших учасників інспекції. Ведучий формальної інспекції не має переваги перед іншими учасниками обговорення, він лише організовує цей процес і фіксує його результати в бланку інспекції.

Під час обговорення необхідно в бланку інспекції проставити відповіді на контрольні запитання і зафіксувати зауваження. Для цього ведучий послідовно зачитує контрольні запитання. У разі відсутності в усіх інспекторів зауважень, що порушують сформульовану в запитанні властивість, проти запитання ставиться мітка (галочка або хрестик) у графі «Yes» або «Так»; інакше мітка ставиться в графі «No» або «Ні», а в графі «Зауваження» (або аналогічній) перелічуються номери відповідних зауважень, записаних в спеціальній таблиці, яку поміщено в кінці бланка інспекції.

Мітка в графі «N/A» («Непридатний») ставиться тільки тоді, коли сформульовану у відповідному запитанні властивість не можна оцінити для певного об'єкта інспекції; у цьому випадку в графі «Зауваження» записується обґрунтовування неможливості оцінити цю властивість.

Якщо під час повторної інспекції використовується бланк від попередньої інспекції, у якому вже проставлено відповіді на контрольні запитання, то ці відповіді не виправляють, а в таблиці для зауважень проти зафіксованих раніше невідповідностей роблять мітки про їх усунення. У разі виявлення нових невідповідностей зауваження записують до таблиці після попередніх, а наступна повторна інспекція обов'язково призначається в повній формі.

Наприкінці обговорення учасники приймають рішення про можливість прийняття об'єкта інспекції в наявній версії, або про необхідність внесення виправлень і проведення повторної інспекції в повній або укороченій формі. Об'єкт інспекції можна прийняти в наявній версії тільки за умови відсутності невідповідностей. Рішення про проведення повторної інспекції в укороченій формі приймається тільки в тому випадку, коли всі учасники з цим згодні. Якщо хоча б один з учасників наполягає на повній формі повторної інспекції, то її треба проводити в повній формі. Думка ведучого враховується нарівні з думками інших учасників. Прийняте рішення фіксується ведучим на бланку інспекції і завіряється підписами всіх учасників, а також представника служби якості, якщо він був на зборах.

Теоретично можливою є ситуація, коли автор не згоден ані з одним із зафіксованих зауважень, а інспектори наполягають, що невідповідності мають місце. У такому разі неможливо прийняти рішення про змінення статусу інспектованих документів, тому інспекцію необхідно відкласти, а вирішення проблеми винести за межі процесу формальної інспекції.

На бланку інспекції також фіксується тривалість нарад і час, витрачений на підготовку кожним із учасників.

У процедурі формальної інспекції проекту може допускатися скасування обговорення й наради, якщо в жодного з інспекторів немає зауважень. Це є можливим або при проведенні інспекції одним інспектором, що одночасно є ведучим, або при використанні автоматизованої системи підтримки проведення формальних інспекцій, з допомогою якої ведучий отримує від кожного з інспекторів інформацію про невідповідності і підтвердження ними завершення вивчення об'єкта інспекції. У цьому випадку ведучий заповнює бланк інспекції самостійно.

Завершення формальної інспекції

Наприкінці наради, після закінчення обговорення, інспектори здають ведучому свої робочі матеріали, які містять інспектовані документи з позначками та бланки інспекції. Ведучий складає ці матеріали в прозору теку разом з примірником бланка інспекції, заповненим під час обговорення, причому титульний аркуш бланка інспекції має лежати зверху, щоб можна було за ним ідентифікувати теки.

Після наради ведучий змінює статус інспектованих документів в базі даних проекту відповідно до прийнятого рішення – їм присвоюється статус або «Прийнято», або «Переробити».

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

2.18.3. Документування формальної інспекції

Якщо підприємство веде декілька проектів з розроблення програмних систем, то процедура формальної інспекції регламентується за стандартом підприємства. Це дає змогу співробітникам, що беруть участь у формальних інспекціях, легко пристосовуватися при переході з проекту в проект.

Проте кожний проект може мати свою специфіку. Через це рекомендується розробляти для кожного проекту свою процедуру формальної інспекції, у якій має бути уточнено й доповнено чинний стандарт з урахуванням специфіки цього проекту і яка не повинна суперечити вимогам справжнього стандарту.

Процедура формальної інспекції проекту має точно описувати порядок проведення формальних інспекцій у цьому проекті.

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

У процедурі формальної інспекції проекту слід навести всі ідентифікатори станів інспектованих документів у базі даних проекту, з

якими доведеться мати справу учасникам формальної інспекції, щонайменше, такі:

– готовність документа до проведення інспекції;

– проходження фаз планування, підготовки й обговорення;

– необхідність перероблення документа;

– підтвердження відповідності початковим документам.

З допомогою формальної інспекції проекту має регламентуватися мінімальна кількість інспекторів для кожного типу об'єктів інспекції, якщо в проекті інспектуються документи різного рівня складності, які потребують участі різної кількості інспекторів.

За процедурою формальної інспекції проекту мають регламентуватися можливі форми проведення повторної інспекції залежно від об'єму і характеру змін, внесених в об'єкт інспекції. Зазвичай допускається спрощення процесу повторної інспекції (виконання інспекції одним інспектором, відсутність фази обговорення) при внесенні в об'єкт інспекції незначних змін. У процедурі формальної інспекції проекту може передбачатися використання бланка від попередньої інспекції, якщо проводиться повторна інспекція в укороченій формі. Під час процедури формальної інспекції проекту ведучий може самостійно ініціювати процес повторної інспекції (у тому ж складі учасників), навіть коли її проводять у повній формі, якщо це зумовлено специфікою проекту.

Бланк інспекції – основний документ, що заповнюється під час проведення інспекцій. Зазвичай його розробляють разом зі стандартами проекту. Для кожного типу об'єктів інспекції в проекті має бути розроблено свій бланк інспекції.

Бланк інспекції складається з трьох основних частин:

– титульний аркуш;

– список контрольних запитань;

– список невідповідностей.

Крім того, рекомендується на всіх сторінках бланка, окрім першої, розміщувати колонтитул, який містить щонайменше номер бланка інспекції.

Титульний аркуш призначено для ідентифікації формальної інспекції та запису рішення і зазвичай містить такі елементи:

– слова «формальна інспекція»;

– ідентифікатор проекту;

– ідентифікатор типу об'єкта інспекції, наприклад, «Тест», «Стандарт проекту»;

– ідентифікатор версії бланка інспекції;

– ідентифікатор конфігураційної бази даних;

– місце для запису ідентифікаторів кожного з інспектованих документів;

– місце для запису ідентифікаторів версій кожного з інспектованих документів;
– місце для запису ідентифікаторів кожного з документів;
– місце для запису ідентифікаторів версій кожного з документів;
– місце для запису дати початку інспекції;
– місце для запису фактичних дати й часу початку наради;
– місце (таблиця) для запису прізвищ учасників інспекції із зазначенням їхніх функцій і з місцями для підпису і запису часу, що витрачено на підготовку;
– місце для запису тривалості наради;
– місце для фіксації прийнятого рішення.

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

Якщо процедурою формальної інспекції проекту передбачено можливість проведення повторної інспекції з використанням бланка від попередньої інспекції, то титульний аркуш має містити також такі поля:
– місце для запису дати проведення повторної інспекції;
– місце для запису ідентифікаторів версій кожного документа, що повторно інспектується;
– місце для запису прізвища ведучого повторної інспекції;
– місце для запису часу, що витратив ведучий на проведення повторної інспекції;
– місце для фіксації прийнятого рішення;
– місце для підпису ведучого.

Усі перелічені елементи слід розташовувати на одній сторінці.

Список контрольних запитань

Список контрольних запитань необхідно оформляти у вигляді таблиці, що має такі стовпці:
– порядковий номер;
– запитання;
– позитивна відповідь («Yes» або «Так»);
– негативна відповідь («No» або «ні»);
– відповідь «N/A» або «Непридатний»;
– посилання на невідповідність.

Контрольні запитання слід формулювати так, щоб позитивна відповідь означала відсутність невідповідностей. Формулювання мають бути зрозумілими, чіткими й однозначними.

Список невідповідностей

Список невідповідностей необхідно оформляти у вигляді незаповненої таблиці з трьома колонками:

– порядковий номер;

– опис невідповідності;

– відмітки про виправлення.

Колонтитул повинен містити:

– ідентифікатор проекту;

– ідентифікатор версії бланка інспекції;

– місце для запису ідентифікаторів хоча б одного з інспектованих документів;

– місце для запису ідентифікаторів версій хоча б одного з інспектованих документів.

2.18.4. Життєвий цикл документа, що інспектується

Під час формальної інспекції аналізують два типи документів:

– документи проекту;

– допоміжні документи (звіт про проведену інспекцію, список контрольних запитань, список виявлених проблем).

Допоміжні документи складають під час інспекції, і їх можна змінювати протягом процесу інспекції. Титульний аркуш створюється на стадії ініціалізації. Список виявлених проблем складають на стадії підготовки, список контрольних запитань – на стадії обговорення. Після завершення процесу інспекції допоміжні документи переносять в архів і їх більше не можна змінювати. Допоміжні документи зберігаються відповідно до встановлених для них термінів.

Під час формальної інспекції стан інспектованого документа послідовно змінюється декілька разів. Під час розроблення (до початку формальної інспекції) документ має стан Active (Активний), і автор може його читати і модифікувати. Після того як автор вирішив, що закінчив роботу над документом, він переводить документ у стан Ready (Готовий). Це означає, що документ готовий до формальної інспекції. Коли документ має стан Ready, автор вже не може його змінювати. Наступним станом документа є Review (Формальна інспекція). Цього стану документ набуває на стадії ініціалізації формальної інспекції. Перехід документа в стан Review здійснює ведучий, доступ до документа можливий тільки для читання всіма учасниками формальної інспекції. Якщо документ пройшов формальну інспекцію (не було знайдено проблем), то він набуває стану Approved (Затверджений). Перехід документа в стан Approved здійснює ведучий, і документ є доступним тільки для читання всім учасникам формальної інспекції та проекту. Якщо ж після формальної інспекції в

цільовому документі потрібні виправлення, документ переводиться в стан Update (Перероблення), і автор має доступ до документа як для читання, так і для його модифікації. Після перероблення документа автор переводить його у стан Ready, і процес переходу зі стану в стан повторюється доти, доки документ не буде переведено в стан Approved. Якщо в інспектований документ не потребується вносити значних змін, то після того як ведучий переконається в тому, що необхідні виправлення було зроблено, цільовий документ можна переводити в стан Approved.

Інспектований документ підлягає виправленню після завершення процесу інспекції. Після виправлення цільовий документ може пройти повторну інспекцію. Таким чином, цільовий документ може пройти декілька послідовних інспекцій (зробити декілька витків життєвого циклу документів під час формальної інспекції).

2.18.5. Формальні інспекції програмного коду

Процес формальної інспекції програмного коду підпорядковується всім правилам, визначеним для абстрактної формальної інспекції, проте має деякі особливості, пов'язані передусім із структурою інспектованого програмного коду, а також з тим, що зазвичай інспектуються ділянки коду, які неможливо перевірити з допомогою автоматизованого тестування, що базується на тестах.

Особливості перегляду програмного коду

Перед початком перегляду коду рекомендується визначити пункти вимог, на відповідність яким перевіряється код, а також записати обґрунтування того, чому ці вимоги не можна перевірити в автоматичному режимі. Після цього можна переходити до перегляду власне коду. Усі помітки, які доведеться вносити під час інспекції в код, необхідно робити не у файлі, що зберігається в базі даних проекту, а в його копії, яку потім буде підшито до матеріалів інспекції. Копія може мати той самий форматі, що й програмний файл, бути надрукованою на папері, або мати один із електронних форматів, у яких допускаються коментування, наприклад DOC, PDF.

З допомогою таблиць трасувань у коді визначаються інспектовані функції або методи, що відповідають необхідним вимогам. Ділянки коду виділяють і позначають міткою або номером відповідної вимоги. Якщо ділянка коду відповідає вимогам, то необхідно позначити її або кольором виділення, або відповідною текстовою міткою. Якщо ділянка коду має проблеми, то цей факт необхідно відобразити або кольором виділення, або посиланням на відповідний пункт списку зауважень у бланку інспекції.

У разі відсутності таблиць трасувань рекомендується робити посилання, що пояснюють, чому саме ця ділянка коду реалізує зазначені вимоги. Такі посилання допоможуть на етапі обговорення документа.

Перевірка стилю кодування. Окремим об'єктом перевірки при формальній інспекції програмного коду є стиль кодування. У більшості проектів існують стандарти, у яких описано правила оформлення програмних кодів і файлів даних. Неправильний стиль кодування не впливає на функціональність програми в цілому, але значно утруднює супровід і підтримку змін під час подальшого розвитку системи. Тому відхилення від стилю кодування в інспектованих ділянках коду також мають позначатися в тексті й списку зауважень.

У деяких випадках проводять інспекції, які цілком направлено на перевірку стилю кодування.

Перевірка надійності коду. У деяких випадках рекомендується перевіряти наявність ділянок, що гарантують робастність, навіть якщо вимоги прямо не визначають необхідності в обробленні неприпустимих значень. У випадку, коли потенційно є можливою некоректна робота програми за відсутності обробників неправильних значень, рекомендується навести це в списку зауважень.

Особливості проведення нарад

Розподіл функціональних обов'язків. У складі інспекторів бажано мати хоча б одного фахівця, що уявляє собі особливості виконання інспектованого коду в реальній системи. Це є надто важливим при тестуванні вбудованих систем, яке має проводитися на емуляторах. Під час наради такий фахівець може допомагати ведучому визначати послідовність розгляду зауважень у разі їх великої кількості.

Керування нарадами. При проведенні наради недоцільно зачитувати текст інспектованого файла, як це зазвичай рекомендується. Ведучому краще обмежитися переліком імен функцій і методів або (у випадку, якщо під час інспекції перевіряється відповідність коду вимогам) переліком номерів чи ідентифікаторів вимог. Інспектори за наявності зауважень щодо функції або вимоги піднімають руку і зачитують зауваження.

Особливості завершення і повторного проведення інспекції

Документування наради. Для полегшення праці автора інспектованого документа кожне зауваження, визнане істотним, рекомендується точно трасувати на рядки коду й вимоги.

Контроль за внесенням змін. При повторній інспекції програмних кодів рекомендується використовувати спеціалізовані інструментальні засоби для порівняння файлів. Змінення за підсумками інспекції слід

вносити тільки в ті ділянки, до яких були зауваження. У разі наявності інших змін ведучий має право призначити нову інспекцію в повній формі.

2.18.6. Формальні інспекції проектної документації

Процес формальної інспекції проектної документації підпорядковується правилам, визначеним для абстрактної формальної інспекції, проте має деякі особливості, позв'язані насамперед з тим, що в проектній документації перевіряється її несуперечність і повнота. Точне визначення цих термінів в певному контексті є неможливим, тому під несуперечністю будемо розуміти відсутність в проектній документації вимог з протилежним значенням, коли можливим буде декілька абсолютно різних варіантів реалізації програмної системи, під повнотою – достатність вимог для однозначної реалізації функціональності системи.

При інспекції вимог до системи зазвичай розглядається як зовнішня, так і внутрішня інформація, що стосується певного документа. Під зовнішньою інформацією розуміється передусім суть технічних рішень, прийнятих при розробленні системи, ті принципи, які відрізняють її від інших систем. При цьому перевіряється узгодженість вимог з цими принципами. Під внутрішньою інформацією розуміється перш за все внутрішня цілісність і несуперечність документа – властивості, які дають змогу розробляти програмний код, позбавлений двозначностей і нестабільних ділянок. Головним запитанням, на яке має відповісти інспектор при перевірці внутрішніх властивостей документа, є таке: «Чи визначено вимоги так, щоб колектив розробників міг працювати з ним?» або «Ці вимоги недвозначні, повні і їх можна виразити?». Процес інспекції може допомогти відповісти на це запитання, а список контрольних запитань, який можна використати при проведенні інспекцій, є таким:

1. Чи є кожна вимога абсолютно недвозначною? (Якщо вимогу прочитати кілька разів, роблячи наголос спочатку на першому слові, потім – на другому, потім – на третьому і т.д., то чи буде при цьому змінюватися значення цієї вимоги?)

2. Чи існує для кожної зі встановлених вимог деякий компетентний фахівець, який зможе сказати після завершення розроблення, чи виконано цю вимогу чи ні?

3. Чи визначено метод вирішення цієї проблеми в документації, що стосується вимог?

4. Чи існують вимоги, які не встановлено?

5. Чи бракує яких-небудь вимог?

6. Чи існують серед заданих такі вимоги, які не є необхідними?

7. Чи існує правило, за яким у певній ситуації можна віддати перевагу одній із суперечливих вимог?

Первинна інспекція проектної документації зазвичай проводиться тоді, коли саму програмну систему ще не написано. Проте при проведенні інспекції змін у вимогах до системи, що вже працює (наприклад, при оновленні її версії), може потребуватися комплексна одночасна інспекція документації і створеного на її основі програмного коду. При цьому може виникнути ситуація, коли зміни, які необхідно внести в документацію за результатами інспекції, потребують відповідного змінення в програмному коді. Вирішити цю проблему можна, використовуючи таблиці трасувань.

Дещо інша ситуація виникає у разі, коли комплексна інспекція проводиться не після змінення вимог, а після завершення всього ланцюжка змін після змінення функціональних вимог, архітектури, низькорівневих вимог і програмного коду. У цьому випадку за наявності суперечливих вимог необхідно виявити всі частини програмної системи, з допомогою яких реалізують ці вимоги. Якщо ж розроблення цих частин виконувалося різними людьми, могло різнитися й трактування суперечливих вимог. Тоді ліквідація суперечності може спричинити «хвилю змін» у проектній документації і програмних кодах системи. Для того щоб уникнути «хвиль змін» після завершення інспекцій, рекомендується проводити її поетапно до початку наступного етапу життєвого циклу або до розроблення документів наступного рівня деталізації.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Харченко, В.С. Анализ рисков аварий для ракетно-космической техники: эволюция причин и тенденций [Текст] / В.С. Харченко, В.В. Скляр, О.М. Тарасюк // Радіоелектронні і комп'ютерні системи. – 2003. – №. 3. – С. 135–149.

2. Липаев, В.В. Обеспечение качества программных средств. Методы и стандарты [Текст] / В.В. Липаев. – М. : СИНТЕГ, 2001. – 380 с.

3. ДСТУ ISO/IEC TR 9126-2:2008 Програмна інженерія. Якість продукту. – К. : Держстандарт України, 1995. – Ч. 2 : Зовнішні метрики. – 91 с.

4. Липаев, В.В. Функциональная безопасность программных средств [Текст] / В.В. Липаев. – М. : РФФИ ; СИНТЕГ, 2003. – 348 с.

5. Майерс, Г. Надежность программного обеспечения [Текст] : пер. с англ. / Г. Майерс. – М. : Мир, 1980. – 360 с.

6. Тейер, Т. Надежность программного обеспечения [Текст] : пер. с англ. / Т. Тейер, М. Липов, Э. Нельсон. – М. : Мир, 1981. – 323 с.

7. Смит, Д. Функциональная безопасность. Простое руководство по применению стандарта МЭК 61508 и связанных с ним стандартов [Текст] / Д. Смит, К. Симпсон – М. : Изд. дом «Технологии», 2004. – 208 с.

8. ДСТУ 3918–1999 (ISO/IEC 12207–1995). Інформаційні технології. Процеси ЖЦ програмного забезпечення. – К. : Держстандарт України, 1995. – 57 с.

9. Нормативная база программной инженерии в разработке систем с интенсивным использованием программного обеспечения [Текст] : учеб. пособие / Б.М. Конорев, Л.Ф. Пудовкина, И.Б. Сироджа, О.Е. Федорович. – Х. : Нац. аэрокосм. ун-т им. Н.Е. Жуковского «Харьк. авиац. ин-т», 2001. – 162 с.

10. Липаев, В.В. Выбор и оценивание характеристик качества программных средств [Текст] / В.В. Липаев. – М. : СИНТЕГ, 2001. – 228 с.

11. Липаев, В.В. Методы обеспечения качества крупномасштабных программных средств [Текст] / В.В. Липаев – М. : РФФИ ; СИНТЕГ, 2003. – 520 с.

12. Холстед, М.Х. Начала науки о программах [Текст] : пер. с англ. / М.Х. Холстед. – М. : Финансы и статистика, 1981. – 128 с.

13. Lyu, M.R. Handbook of Software Reliability Engineering. Computing [Текст] / M.R. Lyu. – New York, McGraw-Hill,1996. – 805 p.

14. Андерсон, Р. Доказательство правильности программ [Текст] :

пер. с англ. / Р. Андерсон. – М. : Мир, 1982. – 380 с.

15. Abrial, J.R. The B-Book: Assigning Programs to Meanings [Text] / J.R. Abrial // Cambridge University Press, 1996. – 180 p.

16. Одинцов, И. Профессиональное программирование. Системный подход [Текст] / И. Одинцов. – СПб. : БХВ-Петербург, 2002. – 512 с.

17. Харченко, В.С. Теоретические основы дефектоустойчивых цифровых систем с версионной избыточностью [Текст] / В.С. Харченко. – К. : МО Украины, 1996. – 503 с.

18. Дастин, Э. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация [Текст] : пер. с англ. / Э. Дастин, Д. Рэшка, Д. Пол. – М. : ЛОРИ, 2003. – 567 с.

19. Динамическая отработка программного обеспечения бортовых цифровых вычислительных машин систем управления объектов ракетно-космической техники [Текст] / Я.Е. Айзенберг, А.В. Бек, Ю.М. Златкин и др. // Космическая наука и технология. – 1997. – Т. 3, № 1/2. – С. 61–74.

20. Липаев, В.В. Стандартизация верификации программных средств [Текст] / В.В. Липаев // Информационные технологии. – 2001. – № 3. – С. 2–6.

21. Липаев, В.В. Верификация и тестирование сложных программных средств [Текст] / В.В. Липаев // Информационные технологии. – 2004. – № 7. – С. 42–47.

22. Hayhurst, K.J. A Practical Tutorial on Modified Condition/Decision Coverage [Text] / K.J. Hayhurst // NASA Langley Research Center, Hapton, Virginia, 2001. – 85 p.

23. IEEE 829–2008. IEEE Standard for Software and System Test Documentation. IEEE Computer Society. Institute of Electrical and Electronics Engineers. 3 Park Avenue, New York, NY 10016-5997, USA, 18 July 2008. – 118 p.

ЗМІСТ

Перелік скорочень 3
1. Якість програмного забезпечення 4
1.1. Класифікація проблем, які виникають під час експлуатації програмних систем 4
1.1.1. Збої програмного забезпечення 7
1.1.2. Відмови програмного забезпечення 8
1.2. Стандартизація процесів забезпечення якості 10
1.3. Загальні положення стандартів серії ISO 12
1.4. Застосування ISO 9001 при розробленні програмних систем 16
1.5. Система якості 19
1.6. Функціональний склад колективу розробників 22
1.7. Моделі життєвого циклу програмного забезпечення 24
1.8. Життєвий цикл розроблення програмного забезпечення з підвищеними вимогами до безпеки 31
1.9. Процеси життєвого циклу програмного забезпечення загального використання 40
1.9.1. Процес придбання програмного забезпечення 42
1.9.2. Процес розроблення програмного забезпечення 44
1.9.3. Процес експлуатації програмного забезпечення 50
1.9.4. Процес супроводу програмного забезпечення 50
1.9.5. Документація життєвого циклу 51
1.9.6. Керування конфігурацією 52
1.9.7. Оцінювання процесів розроблення програмних систем 54
1.9.8. Вимірювання процесу 56
1.9.9. Оцінювання процесів 57
1.10. Показники якості програмного забезпечення 60
1.11. Моделі та метрики оцінювання якості ПЗ 65
1.11.1. Метрики складності потоку керування 66
1.11.2. Метрики складності потоку даних 69
1.11.3. Метрики стилістики і зрозумілості програми 71
1.11.4. Метрика Холстеда 72
1.11.5. Об'єктно-орієнтовані метрики 75
1.11.6. Метрики надійності 77
1.12. Методи гарантування якості програмного забезпечення 78
2. Тестування програмного забезпечення 81
2.1. Місце верифікації у життєвому циклі 81
2.1.1. Мета і завдання верифікації 82
2.1.2. Особливості тестування, верифікації і валідації ПЗ 84
2.2. Типи процесів верифікації 85
2.3. Інструментальні засоби автоматизованої верифікації 87
2.4. Тестування програмного коду 91
2.4.1. Задачі і мета тестування програмного коду 91
2.4.2. Методи тестування програмного забезпечення 93

2.4.3. Тестувальне оточення ... 95
2.4.4. Тестові класи ... 97
2.4.5. Генератори тестових сигналів ... 98
2.4.6. Тести ... 99
2.5. Тестові плани ... 109
2.6. Оцінювання якості коду, що тестується ... 111
2.6.1. Покриття програмного коду ... 112
2.6.2. Аналіз покриття ... 116
2.7. Повторюваність тестування ... 118
2.7.1. Мета і задачі забезпечення повторюваності тестування ... 118
2.7.2. Передумови для виконання тестів ... 120
2.8. Документування верифікації і тестування ... 121
2.8.1. Проектна документація і її призначення ... 121
2.8.2. Стратегія і плани верифікації ... 123
2.8.3. Тестові вимоги ... 125
2.8.4. Тестові плани ... 127
2.8.5. Тестові сценарії ... 129
2.8.6. Тестові таблиці ... 129
2.8.7. Кінцеві автомати ... 130
2.9. Генератори тестів ... 132
2.10. Звіти про проходження тестів ... 133
2.10.1. Звіти автоматичного і ручного тестування ... 135
2.10.2. Звіти про покриття програмного коду ... 136
2.10.3. Форми звітів про покриття ... 136
2.10.4. Покриття на рівні програмних і машинних кодів ... 140
2.11. Звіти про проблеми ... 141
2.11.1. Зв'язок звітів з іншою проектною документацією ... 141
2.11.2. Структура звіту про проблему ... 142
2.12. Таблиці трасування ... 143
2.12.1. Зв'язок таблиць трасування з іншою документацією ... 143
2.12.2. Можливі форми таблиць трасування ... 144
2.13. Модульне тестування ... 144
2.13.1. Мета і задачі модульного тестування ... 145
2.13.2. Поняття модуля і його границь. Тестування класів ... 146
2.13.3. Визначення ступеня повноти тестування класу ... 147
2.13.4. Особливості тестового оточення ... 149
2.13.5. Організація модульного тестування ... 149
2.14. Інтеграційне тестування ... 153
2.14.1. Мета і задачі інтеграційного тестування ... 153
2.14.2. Організація інтеграційного тестування ... 154
2.14.3. Планування інтеграційного тестування ... 159
2.15. Системне тестування ... 159
2.15.1. Мета і завдання системного тестування ... 159

2.15.2. Види системного тестування 160
2.16. Тестування інтерфейсу користувача 164
2.16.1. Мета і задачі тестування інтерфейсу користувача 164
2.16.2. Функціональне тестування інтерфейсу користувача 165
2.16.3. Перевірка вимог до інтерфейсу користувача 166
2.16.4. Методи тестування інтерфейсу 169
2.16.5. Тестування інтерфейсу на зручність використання................ 171
2.17. Приймально-здавальні та сертифікаційні випробування.............. 174
2.18. Формальні інспекціїї 179
2.18.1. Мета і задачі формальних інспекцій.. 179
2.18.2. Етапи формальної інспекції та функції її учасників 180
2.18.3. Документування формальної інспекції 185
2.18.4. Життєвий цикл документа, що інспектується.......................... 188
2.18.5. Формальні інспекції програмного коду 189
2.18.6. Формальні інспекції проектної документації 191
Бібліографічний список 193

Навчальне видання

Манжос Юрій Семенович

ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТУВАННЯ

Редактор О.Ф. Серьожкіна

Зв.план, 2012
Підписано до видання 29.12.2012
Ум. друк. арк. 11,1. Обл.-вид.арк. 12,5. Електронний ресурс

Національний аерокосмічний університет ім. М.Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http:\\ www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб’єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001