МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КАМ’ЯНСЬКИЙ ЕНЕРГЕТИЧНИЙ ФАХОВИЙ КОЛЕДЖ

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни
«Тестування програмних систем і комплексів»
для здобувачів фахового молодшого бакалавра за освітньо-професійними програмами
«Комп’ютерні науки» та «Комп’ютерна інженерія»
із спеціальностей 122 – «Комп’ютерні науки» та 123 – «Комп’ютерна інженерія»
галузь знань 12 – «Інформаційні технології»

Укладач: Маріщук А.В.
викладач

Кам’янське
2024

Зміст

Тема №1. Поняття тестування ПЗ, відладка, тестування, фази, дефект

У вузькому розумінні ***тестування*** – це процес визначення відповідності об'єкта тестування заданим специфікаціям (характеристикам, функціям), який полягає в опрацюванні програмою послідовності різноманітних контрольних наборів тестів з відомими результатами. Тести підбираються так, щоб вони охопили найрізноманітніші типи можливих ситуацій. У контексті розробки, відповідно, тестування розуміється як процес визначення відповідності продукту початковим специфікаціям, які були задані технічним завданням.

Якщо розглядати тестування в широкому сенсі, то його можна характеризувати як процес експериментального аналізу функціональності деякої досліджуваної системи. При цьому постулюється, що всякий випадок тестування можна визначити як дослідження. Наприклад, тестування з метою виявлення структурних дефектів (дефектів конструкції) можна визначити, як дослідження системи на предмет наявності дефектів конструкції.

Усяке тестування має на увазі дві діючі особи: суб'єкт і об'єкт тестування. ***Суб'єктом тестування*** (тобто джерелом активності) виступає «тестувальник» (tester) – спеціально призначена особа, в обов'язки якої входить виконання тестування. Під ***об'єктом тестування*** розуміється досліджувана система. При цьому в ролі тестованої системи може виступати як весь продукт, так і окремі його модулі, складові частини.

При тестуванні програмного забезпечення, в загальному випадку, об'єктом тестування є версія розроблюваного програмного продукту.

Зрозуміло, що під версією мається на увазі не тільки «фінальна» (кінцева) версія, а й проміжні версії, одержувані в ході розробки.

При проведенні тестування завжди має місце «зовнішній вплив» на систему з боку тестувальника, який також називають «тестовим впливом». Тестування полягає у здійсненні тестувальником спрямованого впливу на систему, що тестується, який передбачає отримання деякої очікуваної реакції, яку називають «тестовою реакцією», поява якої має підтвердити відповідність системи конкретної окремо взятої специфікації. Отримання зворотньої (стосовно очікуваної) або неспецифічної реакції

дозволяє говорити про невідповідність системи специфікації і приймати рішення про повернення програмного продукту на доопрацювання з метою усунення дефекту.

Як і в моделюванні, у тестуванні поняття системи є ключовим, оскільки наше уявлення про досліджуваний об’єкт є первинним стосовно експерименту. Іншими словами характер експерименту залежить від наших знань про структурний склад і характер зв’язків між компонентами тестованої системи.

Нагадаємо, що під системою розуміється деякий образ, що узагальнено або повно описує досліджуваний нами об’єкт. Таким чином, можна виділити два крайніх випадки: у першому – ми володіємо абсолютно повною інформацією про систему, що тестується; у другому – тестована система виступає в ролі «чорного ящика», тобто ми володіємо лише знанням про входи системи і можемо фіксувати одержувані виходи (результати внутрішньої активності системи).

Весь процес тестування можна умовно поділити на тестові завдання, кожне з яких полягає в дослідженні якоїсь окремої риси випробуваної системи. Тестове завдання, як правило, складається з опису мети завдання, перерахування та вказівки порядку тестових впливів і формалізованої фіксації тестової реакції, а так само з висновку про те, чи відповідає система, що тестується, специфікації.

При наявності вичерпної інформації про систему ми маємо можливість складати тестові завдання, ґрунтуючись на знанні про внутрішню логіку функціонування системи. Наприклад, знаючи, як вхідні дані перетворюється
всередині системи, ми можемо подавати на входи системи некоректні дані з метою тестування її відмовостійкості. В описаному випадку, можливість формування некоректних даних залежить від знання про те, які дані очікуються системою.

У разі ж, коли ми володіємо неповною інформацією про систему, тестування змінює свою форму. Як правило, ми завжди маємо підстави що-небудь припускати про характер функціонування системи, що випробується.

У таких умовах, тестові завдання будуть формуватися на основі наших припущень.

Тест – контрольна задача для перевірки коректності функціонування ПЗ.

Основна ідея тестування – запустити ПЗ і спостерігати за його роботою та її наслідками. Якщо збій в роботі ПЗ відбувся, то аналізується звіт з метою виявлення

місцезнаходження помилки, яка його викликала. ***«Вдалим» тестом*** є той, при якому виконання програми закінчилось з помилкою.

У контексті розробки тестування розуміється як процес визначення відповідності продукту початковим специфікаціям, які були задані технічним завданням. Тобто, тестування – це процес керованого експериментування з продуктом за допомогою тестів з метою виявлення в ньому помилок та неточностей, що допущені розробниками ПЗ.

Тестування пронизує весь життєвий цикл ПЗ, починаючи від аналізу вимог, проектування і закінчуючи невизначено довгим етапом експлуатації.

Ці роботи безпосередньо пов’язані із завданнями управління вимогами та змінами, адже метою тестування є можливість переконатися у відповідності програм заявленим вимогам.

Тестування – процес також ітераційний. Після виявлення і виправлення кожної помилки обов’язково слід повторити тести, щоб переконатися у працездатності програми. Більш того, для ідентифікації причини виявленої проблеми може бути потрібне проведення спеціального додаткового тестування. При цьому завжди потрібно пам’ятати фундаментальний висновок, зроблений професором Едсгером Дейкстрой у 1972 р.: «Тестування програм може служити доказом наявності помилок, але ніколи не доведе їх відсутність!».

При тестуванні програмного забезпечення, в загальному випадку, об’єктом тестування є версія розроблюваного програмного продукту.

Зрозуміло, що під версією мається на увазі не тільки «фінальна» (кінцева) версія, а й проміжні версії, що одержані в ході розробки.

Нагадаємо, що під системою розуміється деякий образ, який узагальнено або повно описує досліджуваний нами об’єкт. Таким чином, можна виділити два крайніх випадки: у першому випадку ми володіємо абсолютно повною інформацією про систему, що тестується; у другому – система, що тестується, виступає в ролі «чорного ящика», тобто ми володіємо лише знанням про входи системи і можемо фіксувати одержувані виходи (результати внутрішньої активності системи).

При наявності вичерпної інформації про систему ми маємо можливість складати тестові завдання, ґрунтуючись на знанні про внутрішню логіку функціонування

системи. В описаному випадку, можливість формування некоректних даних залежить від знання про те, які дані очікуються системою.

Якщо ми володіємо неповною інформацією про будову системи, тестування змінює свою форму. Як правило, ми завжди маємо підстави що-небудь припускати про характер функціонування системи, що тестується. У таких умовах тестові завдання будуть формуватися на основі наших припущень.

У тих випадках, коли ми не володіємо жодною інформацією про систему, тестування приймає дослідний характер. У такому випадку наше завдання полягає в тому, щоб визначити, як виходи залежать від входів, або, іншими словами, як реакція системи залежать від зовнішніх впливів і, на підставі отриманої інформації визначити, чи відповідає система заданим специфікаціям.

1.1 Рівні тестування

Необхідно відзначити, що програмні продукти в силу різних причин мають модульну структуру і рівневу організацію, і, як правило, функціонують не відокремлено, а в рамках деякої інфраструктури (спільно з іншими додатками, програмними комплексами, а також в рамках програмних середовищ).

Вищевказане призводить до виникнення різних рівнів тестування:

1. ***Компонентне*** або ***модульне тестування*** перевіряє функціональність і шукає дефекти в частинах програми, які доступні і можуть бути протестовані окремо в силу своєї автономності (модулі програми, об’єкти, функції і т.д.). Ознакою, що утворює даний рівень, є те, що тестування окремих модулів можна виконувати на ранніх етапах розробки, поки робота над іншими модулями триває. Один з найбільш ефективних підходів до компонентного (модульного) тестування – це підготовка автоматизованих тестів до початку основного кодування ПЗ. Це називається розробкою від тестування або підходом тестування спочатку.

2. ***Інтеграційне тестування*** призначено для перевірки зв’язків між компонентами програми, а також дослідження взаємодії додатку з середовищем, в рамках якого воно буде виконуватися.

3. ***Системне тестування*** направлено на дослідження функціональних і нефункціональних особливостей системи в цілому. При цьому виявляються дефекти, такі як невірне використання ресурсів системи, непередбачені комбінації даних рівня користувача, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність використання і т.д.

4. ***Приймальне тестування*** проводиться на фінальному етапі розробки та спрямовано на з'ясування того, чи відповідає система приймально/здавальним критеріям до вимог замовника. Приймальне тестування виконується відповідно до Плану приймальних робіт. Рішення про проведення приймального тестування приймається, коли: продукт досяг необхідного рівня якості та замовник ознайомлений з Планом приймальних робіт або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні і т.д.

1.1.1 Методи приймального тестування

1. Тестування замовником самостійно. Це ризиковано в тому плані що у замовника може не бути творчих ресурсів, а завантаження по поточним завданням може розтягти процес приймання.

2. Тестування третьою стороною (аудит). Наймається спеціалізована компанія на тестування або підписується договір з конкурентом постачальника на надання послуг аудиту.

3. Спільне тестування за сценаріями із замовником. Постачальник допомагає готувати пакет матеріалів для приймального тестування, готує команду замовника до методичного приймального тестування, контролює хід приймального тестування і терміни його виконання.

Присутність інженера з тестування з боку виконавця допоможе краще зафіксувати розбіжності, зауваження та виявлені дефекти.

Фаза приймального тестування триває до тих пір, поки замовник не виносить рішення про відправлення програми на доопрацювання або реліз програми.

Незважаючи на те, що приймання знаходиться в кінці етапу (а в невеликих проектах і в кінці проекту) – готуватися до нього потрібно заздалегідь і перший прогін потрібно робити трохи раніше – щоб визначитися з повнотою і якістю робочого набору артефактів приймання, привчити до нього замовника, заздалегідь виявити можливі проблеми в приймальних тестах або в продукті.

1.2 Види тестування

Залежно від цілей тестування, виділяють три основних види тестування програмного забезпечення:

1. **Функціональні види тестування пов'язані з дослідженням зовнішньої поведінки системи, тобто виконуваних нею функцій.** До них належать:

− функціональне тестування, що спрямоване на перевірку коректності виконуваних системою функцій і може бути присутнім на всіх рівнях тестування;

− тестування безпеки, яке направлене на перевірку безпеки системи, а також оцінку цілісності підходу до захисту додатку від несанкціонованого доступу і захисту конфіденційних даних;

− тестування взаємодії, яке направлене на оцінку можливості додатка взаємодіяти із зовнішніми компонентами або системами, а також включає тестування сумісності та інтеграційне тестування.

2**. Нефункціональні види тестування, які спрямовані на перевірку всіх нефункціональних особливостей системи.** Сюди належать тести специфічних для програмних продуктів характеристик:

− тестування встановлення, яке направлене на перевірку процесу інсталяції системи, а також процесів настойки, вилучення і оновлення програмного забезпечення;

− тестування зручності використання, яке пов'язане з оцінкою ступенем зручності використання, а також зрозумілості та привабливості користувальницького інтерфейсу додатку;

− тестування на відмову і відновлення, яке пов'язане з оцінкою засобів забезпечення відмовостійкості та надійності системи;

− конфігураційне тестування, яке спрямоване на перевірку функціональності системи при всіх можливих конфігураціях програмного забезпечення і устаткування, що підтримується системою;

− тестування продуктивності (навантаження), яке складається з таких методів тестування як :

a) тестування навантаження, що полягає в дослідженні реакції системи на функціонування в умовах навантаження (мається на увазі навантаження в межах норми);

b) стресове тестування, що передбачає дослідження поведінки системи при функціонуванні в умовах перевантаження (навантаження, яке перевищує штатне);

c) тестування стабільності і надійності, що спрямоване на вивчення поведінки системи в умовах нормального навантаження при тривалому функціонуванні;

d) об’ємне тестування, яке використовується для оцінки поведінки системи за умови збільшення обсягу даних, що обробляються додатком.

2. Види тестування, які пов’язані із змінами:

– димове тестування, яке спрямоване на оглядову перевірку всіх компонентів програми на предмет працездатності, а також на виявлення грубих дефектів, наявність яких можна визначити, так би мовити, «неозброєним оком». Поняття димове тестування пішло з інженерного середовища. При введенні в експлуатацію нового обладнання вважалося, що тестування пройшло вдало, якщо з установки не пішов дим. В області ж тестування програмного забезпечення, воно спрямоване на поверхневу перевірку всіх модулів програми на предмет працездатності та наявність критичних і блокуючих дефектів. За результатами димового тестування робиться висновок про те, приймається чи ні встановлена версія програмного забезпечення на тестування, експлуатацію або на постачання замовнику. Димові тести повинні виконуватися на всьому проекті від початку до кінця. Вони не повинні бути вичерпними і всебічними, але повинні містити перевірку всіх основних функцій. Димове тестування має бути досить глибоким, щоб, у разі вдалого його проходження, можна було назвати проект стабільним і таким, що може піддаватися більш глибшому тестуванню;

– регресивне тестування (від лат. regressio – рух назад) – збірна назва для всіх видів тестування програмного забезпечення, спрямованих на виявлення помилок у вже

протестованих ділянках вихідного коду. Регресивне тестування в основному призначено для перевірки здійснених в системі змін, а також на підтвердження того, що функціональність, яка існувала до зміни, працює так же як і до змін. Регресивне тестування є невід'ємною частиною екстремального програмування. У цій методології проектна документація замінюється на розширюване, повторюване та автоматизоване тестування всього програмного пакету на кожній стадії процесу розробки програмного забезпечення;

– тестування збірки, яке направлене на перевірку відповідності версії програмного продукту, що випускається, критеріям якості, необхідним для початку тестування. За своєю метою є аналогом димового тестування, спрямованого на приймання нової версії в подальше тестування або експлуатацію. Вглиб воно може проникати далі, в залежності від вимог до якості випущеної версії;

– санітарне тестування чи інакше – перевірка узгодженості/справності полягає в проведенні тесту достатнього для підтвердження того, що певна окремо взята функція працює відповідно до заявлених специфікацій;

– альфа-тестування – імітація реальної роботи з системою штатних розробників, або реальна робота з системою потенційних користувачів/замовників. Найчастіше альфа-тестування проводиться на ранній стадії розробки продукту, але в деяких випадках може застосовуватися для закінченого продукту в якості внутрішнього приймального тестування. Іноді альфа-тестування виконується з або з використанням оточення, яке допомагає швидко виявляти знайдені помилки;

– бета-тестування – у деяких випадках виконується поширення попередньою версією (іноді з обмеженнями по функціональності або часу роботи) для деякої більшої групи осіб з тим, щоб переконатися, що продукт містить досить мало помилок. Іноді бета-тестування виконується для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів.

1.3 Необхідність тестування

На сьогоднішній день, до програмних продуктів пред'являються досить високі вимоги в області якості, в силу великої конкуренції і широкого спектру пропозицій у

багатьох сферах ринку програмного забезпечення. Якість визначають за двома складовими: специфікацією виробника і специфікацією споживача.

Специфікації виробника формують об'єктивні вимоги до програмного забезпечення, породжувані вільною конкуренцією. Іншими словами, висуваючи вимоги до кінцевого продукту, виробник виходить з того, що програмне забезпечення має бути конкурентоспроможним, але забезпечити до мінімально можливих витрат на розробку.

Специфікації споживача виражають загальні очікування кінцевих користувачів щодо розроблюваного продукту. Відповідність обом описаним специфікаціям і розуміється якістю програмного забезпечення. Очевидно, що не всі очікування можна виразити у формальному вигляді, а, отже, не завжди можливо виміряти, чи відповідає продукт очікуванням, чи ні. Тому прийнято говорити про вимірювані очікування, які знаходять вираження в специфікаціях – заздалегідь заданих формальних вимогах до кінцевого продукту.

Тестування дозволяє контролювати задані специфікації на всіх етапах розробки, і, отже, є частиною забезпечення якості кінцевого продукту. Таким чином, тестування слід розглядати, як необхідний і обов'язковий етап розробки програмного забезпечення.

1.4 Мета і задачі тестування

Загальна мета тестування – виявлення дефектів програмного забезпечення, є однією з цілей забезпечення якості в рамках уніфікованого процесу розробки програмного забезпечення. Однак тестування не обмежується одним виявленням дефекту, воно також має контролювати виправлення виявленого недоліку. Таким чином, можна визначити такі цілі тестування програмного забезпечення:

− виявлення дефектів на різних етапах життєвого циклу додатку (у процесі розробки, під час супроводження тощо);

− перевірка того, чи був виявлений дефект успішно усунутий; з'ясування того, що зміни, пов'язані з усуненням виявленого дефекту, не привнесли нових дефектів в систему. Виходячи з цілей, перед тестуванням ставляться такі завдання:

− виявлення дефектів моделі системи;

− виявлення дефектів кодування;

− виявлення помилок і недоліків взаємодії системи з оточенням, а також зовнішніми компонентами і системами;

− виявлення дефектів інтеграції програмного забезпечення; − виявлення недоліків продуктивності системи;

− виявлення нестійкості програмного забезпечення до перевантажень;

− виявлення нестійкості програмного забезпечення до введення помилкових даних і відмови при збільшенні обсягів даних, що оброблюються;

− виявлення вразливостей в системі безпеки програми і можливостей несанкціонованого доступу до конфіденційних даних;

− контроль виправлення виявлених у процесі тестування дефектів;

− виявлення регресії системи в процесі розробки. Тестування пронизує весь життєвий цикл ПЗ, починаючи від проектування і закінчуючи невизначено довгим етапом експлуатації, та безпосередньо пов’язане з управлінням вимогами і змінами, адже метою тестування якраз є можливість переконатися у відповідності програм заявленим вимогам. Як було зазначено вище, тестування підтверджує, що ПЗ працює відповідно до специфікації. Вважають, що програма працює коректно, якщо вона задовольняє таким критеріям:

− отримавши коректні дані програма надає правильний результат;

− отримавши некоректні дані програма відхиляє їх;

− програма не «зависає» і не «вилітає», приймаючи як коректні, так і некоректні дані;

− програма функціонує нормально стільки часу скільки потрібно;

− програма працює без збоїв і виконує всі необхідні функції в повному обсязі.

1.5 Базова термінологія тестування

1.5.1 Помилка, баг, дефект

Помилка (Error) – хибне значення величини на виході системи або підсистеми, що викликане несправностями або збоями, яке, в свою чергу, може викликати відмову.

З точки зору надійності ПЗ помилку можна розглядати, як упущення або неточність, що допущені проектувальниками ПЗ, програмістами, аналітиками та тестувальниками. Наприклад, проектувальник може неправильно зрозуміти завдання, а програміст – неправильно описати змінну тощо.

Несправність, дефект (Fault) – визнана або передбачувана причина помилки; наслідок відмови деякої системи, що обслуговувала або обслуговує в даний момент часу розглянуту систему. Дефекти також часто називають «***багами***» (від англ. bugs – жучки). Цей термін раніше використовувався, якщо вплив дефекту на роботу програми був незначний. Якщо ж помилка пов'язана із специфікаціями або архітектурою програми, то використовували слово «дефект».

Тепер термін ***«баг»*** – це сучасний сленговий вираз, що означає помилку проектування або розробки, що відбувається під час виконання програми. Існує кілька версій історії про те, хто першим використовував вираз «баг» у тому сенсі, в якому воно зараз вживається ІТ-спільнотою. Найбільш усталеною легендою про походження слова баг вважається історія про те, як в 1947 році вченими Гарвардського університету під час тестування обчислювальної машини Mark II був знайдений метелик, який застряг між контактами електромеханічного реле. Знайдена комаха була вклеєна в технічний журнал з написом: «Перший фактичний випадок виявлення жука (бага)» (англ. «First actual case of bug being found»). І саме цей вираз прийнято вважати першим вживанням слова баг стосовно до комп'ютерної техніки.

У тестуванні програмного забезпечення для позначення помилки прийнято користуватися терміном «дефект», однак сьогодні слово баг всюди використовується для позначення будь-якого роду помилок додатків на всіх етапах розробки.

Походження більшості помилок очевидно і, відповідно, їх виправлення не викликає складнощів. Але існують помилки, виявити які не так вже й просто (в силу різних причин). Виділяють такі «породи жуків», яких складно зловити:

1. «**Гейзінбаг**» (англ. heisenbug) – це програмна помилка, яка зникає або змінює свої характеристики при спробі її виявити. Як приклад гейзінбага можна навести помилки, які мають місце при звичайній компіляції, але пропадають в режимі налагодження (при компіляції за допомогою оптимізованого компілятора; іншими словами, при створенні дебаго-версії програми).

2. «**Борбаг**» (англ. bohrbug) – це така програмна помилка, поведінку якої визначено деякою кількістю певних (але, можливо, невідомих) умов. Баги такого типу не змінюють своєї поведінки і не зникають при спробі їх виявити, а також найчастіше зустрічаються серед складно-усунених. Однак вони можуть з'являтися тільки за певних умов (наприклад, якщо були введені деякі специфічні дані, або за певних налаштувань програми). Тому небезпека помилок даного типу полягає в тому, що вони можуть бути пропущені тестувальником, і про їх наявність стане відомо тільки через деякий час після початку експлуатації додатка кінцевим користувачем.

3. «**Шредінбаг**» (англ. schrodinbug) – це програмна помилка, яка ніяк не проявляється (або є невідомою), до тих пір, поки хто-небудь не виявить її, прочитавши вихідний код або використавши програми в незвичайних (не передбачених) умовах. Після виявлення шредінбага, як правило, незрозуміло, як програма функціонувала до цього моменту (або просто здавалося, що вона функціонує).

4. «**Статистичний баг**» (англ. statistical bug) – це така програмна помилка, яка може бути виявлена тільки при агрегації досить великої кількості результатів тестів. Іншими словами, окремі випробування (і навіть їх невелика кількість) не виявляють помилки – вона стає видна, тільки якщо розглядати велику кількість результатів одночасно. Даний тип помилок специфічний для програм, які виробляють випадковий або псевдовипадковий висновок. Прикладом може служити алгоритм випадковості, що виробляє нерівномірний висновок (тобто, більша частина вихідних значень зосереджена в якомусь окремому діапазоні). Дефект алгоритму буде не видно при малій кількості випробувань, але якщо провести достатню кількість випробувань і розглянути всі виходи разом, то помилковість алгоритму стане очевидна.

1.5.2 Збій, відмова, аварія

Збій (Malfunction) – перший прояв дефекту в роботі системи (проява несправності, зазвичай в роботі устаткування). Збої мають невелику тривалість в часі і можуть бути усунені без тривалих процедур відновлення. Як правило, збій викликає або короткочасну псування даних користувача без припинення роботи всієї системи в

цілому. Наслідки збою можуть бути істотними з точки зору користувача, особливо якщо дані є критично важливими, однак безперебійна робота системи не порушується.

Відмова (Failure) – це більш серйозний прояв дефекту в системі (або підсистеми), при якому вся система або її частина виходять з ладу, виходячи при цьому з працездатного стану, тобто стану в якому всі аспекти функціонування системи відповідають вимогам. Це така зміна робочих параметрів системи (стан), при якому знижується ефективність її функціонування нижче допустимого рівня, або повністю припиняється виконання функцій.

У разі відмови системи для її повернення до нормального функціонування потрібне втручання оператора. Для програмних систем причиною відмови може служити прихований дефект, що виявляються тільки з перебігом великого проміжку часу (переповнення внутрішнього лічильника часу, переповнення даних тощо).

Відмова може виникати внаслідок внутрішніх змін у системі (зміна параметрів всієї системи або її компонентів) або під впливом зовнішніх змін, по відношенню до системи, середовища. Відмова може бути раптовою або поступовою. При раптовій відмові характеристики системи змінюються стрибкоподібно, тоді як поступова відмова характеризується повільною, поступальною зміною параметрів системи, що створює труднощі в сенсі виявлення його причини.

Під час виконання програми або роботи всієї системи тестувальник, розробник або користувач можуть не отримати очікуваних результатів. У деяких випадках така поведінка – симптом помилки. Досвідчений розробник/тестувальник завжди зберігає базу даних помилок, з якими він стикався.

Некоректна поведінка може також означати неправильні значення вихідних даних, неправильний відгук пристрою або неправильне зображення на екрані. У процесі розробки відмови та баги зазвичай виявляються тестувальниками, а дефекти знаходяться і виправляються самими розробниками.

Несправність у коді не завжди веде до відмови. Насправді неправильна частина програми може функціонувати довгий час без прояву яких-небудь недоліків. Проте, за відповідних умов несправність може викликати відмову.

У сенсі допустимості відмов, всі системи можна розділити на дві групи:

− системи, які припускають можливість відмови при деяких передбачених умовах;

− системи, до яких пред’являються високі вимоги надійності, оскільки їх відмова може призвести до незворотних наслідків.

Наприклад, якщо відмовить текстовий редактор, то текст можна набрати заново, або відновити, але, якщо відмовить система навігації літака, то це може призвести до аварії. Звідси виникає поняття надійності, як властивості системи зберігати значення встановлених параметрів в заданих межах, що відповідають режимам і умовам функціонування.

У рамках забезпечення надійності застосовують ряд заходів, пов’язаних із забезпеченням відмовостійкості системи. Відмовостійкість визначається як властивість системи зберігати здатність коректно функціонувати після відмови.

Аварія – відмова системи, при якому система виходить з ладу таким чином, що відновлення її працездатного стану або неможливо, або займає значний час. У разі програмних систем можна уникнути виникнення аварійних ситуацій за допомогою повного дублювання системи як виконуваного програмного коду, так і даних.

Збої і відмови є причиною відмовних ситуацій, в яких працездатний стан системи порушується тимчасово. Аварії є причиною аварійних ситуацій, тобто ситуацій, в яких працездатний стан системи порушується назавжди або на тривалий термін.

1.6 Тестування, верифікація і валідація

Незважаючи на схожість, терміни «тестування», «верифікація» і «валідація» означають різні рівні перевірки коректності роботи програмної системи. Визначимо ці поняття (рис. 1.1).

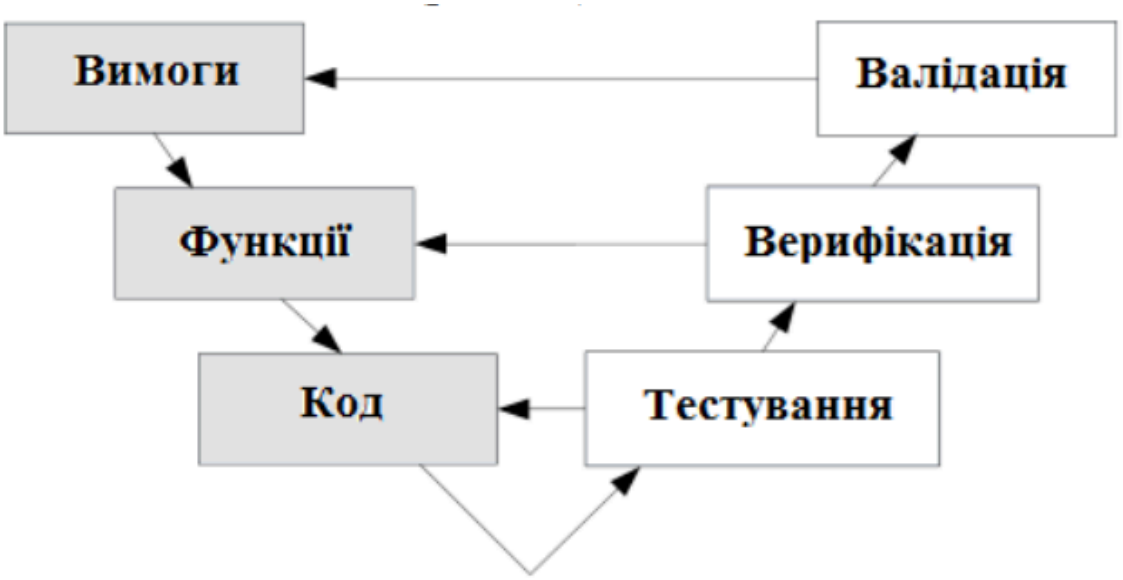

Рисунок 1.1 – Тестування, верифікація і валідація

Тестування програмного забезпечення – вид діяльності в процесі розробки, пов'язаний з виконанням процедур, спрямованих на виявлення (доказ наявності) помилок (невідповідностей, неповноти, двозначностей тощо) в поточному визначенні розроблюваної програмної системи. Процес тестування стосується в першу чергу перевірки коректності програмної реалізації системи, відповідності реалізації вимогам, тобто тестування – це кероване виконання програми з метою виявлення невідповідностей її поведінки та вимог.

Верифікація (лат. verus – «істинний» і facere – «роблю») – постійно виконуваний аналітичний процес перевірки того, що розробка перебуває на правильному шляху: кожен етап розробки є коректним, необхідним (не зайвим) і задовольняє потреби наступного етапу. Проводиться на всіх етапах розробки. На кожному етапі слід переконуватися, що зробили саме те, що планували, і це відповідає загальній логіці розробці – «Ми створюємо систему правильно».

Валідація (англ. Validation – перевірка правильності) – процес перевірки того, що реалізована система задовольняє пред'явленим вимогам і працює так, як передбачалося – «Ми створюємо правильну систему».

Верифікація – це процес оцінки системи/компоненту з метою визначити, чи задовольняють результати конкретної фази умовам, накладеним на початку цієї фази.

Верифікація дозволяє переконатися в коректності переходів між фазами:

1. Потреби користувачів.

2. Функції продукту.

3. Вимоги.

4. Архітектура.

5. Модель проектування.

6. Реалізація.

7. Планування тестів.

Верифікувати слід:

1. Описані функції системи дійсно відповідають потребам клієнта/користувачів.

2. Варіанти використання та вимоги, створені на основі функцій, підтримують ці функції.

3. Проектування, яке підтримує функціональну і нефункціональну поведінку системи.

4. Програмний код відповідає цілям і результатам проектування.

5. Тести, які повністю покривають варіанти використання і вимоги.

Верифікація – більше ніж діяльність групи із забезпечення якості. Потрібно дослідити вимоги і переконатися, що вони повно і ненадлишково відповідають потребам користувачів (функціям) верхнього рівня (п. 2). Далі переконатися, що при проектуванні використовувалися саме ці вимоги, а технічний проект вийшов повним і ненадлишковим.

Валідація – це процес оцінки системи/компоненту під час або після закінчення процесу розробки з метою визначити, чи задовольняє вона/він заданим вимогам. Для перевірки правильності проводяться приймально-здавальні випробування. Вони засновані на сценаріях тестування, які користувач погоджує, а потім виконує в середовищі використання системи.

Стандарт IEEE 829-2008 (Standard for Software and System Test Documentation) містить зразки 8 документів, якими слід керуватися при плануванні, організації та проведенні тестування:

1. План тестування. Керівний документ, що відображає:

− як проводитиметься тестування, включаючи конфігурації тестованої системи;

− хто буде тестувати;

− що буде тестуватися;

− скільки часу займе тестування;

− який рівень якості тестування необхідний.

2. Критерії успішності тестування.

3. Дані для тестування.

4. Сценарії тестування, включаючи передумови і кроки тестів.

5. Звіт про переходи між етапами тестування.

6. Протокол тестування.

7. Звіт про інциденти, включаючи очікуваний результат, фактичний результат, час, передбачувані причини інциденту і все, що може допомогти з розв'язанням ситуації. Інцидент – не обов'язково має на увазі помилку в системі: очікуваний результат міг бути невірним, перевірка могла проводитися невірно, вимога могла тлумачитися по іншому.

8. Звіт про тестування.

1.7 Контрольні питання до теми:

1. Поняття «Тестування».

2. Ідея тестування.

3. Рівні тестування.

4. Методи приймального тестування.

5. Види тестування.

6. Необхідність тестування.

7. Мета і задачі тестування.

8. Поняття «баг».

9. Поняття «дефект».

10. Поняття «Помилка».

11. Схема «Тестування-верифікація-валідація».

12. Поняття «Верифікація».

13. Поняття «Валідація».

14. Документи стандарту планування, організації та проведення тестування.

15. Поняття «Аварія».

16. Різниця між багом та дефектом.

17. Алгоритм плану тестування.

Тема №2. Моделі розробки ПЗ. Життєвий цикл тестування

Під **моделлю ЖЦ ПЗ** розуміється структура, що визначає послідовність виконання та взаємозв'язок процесів, дій, завдань, які виконуються протягом ЖЦ. Модель ЖЦ залежить від специфіки ІС та специфіки умов, в яких остання створюється та функціонує.

Знати і розуміти моделі розробки ПЗ необхідно для того, щоб вже з перших днів роботи розуміти, що відбувається навколо, що, навіщо і чому ви робите. Чим повніше ви будете представляти картину того, що відбувається на проекті, тим ясніше вам буде видно ваш власний внесок у загальну справу і сенс того, чим ви займаєтеся.

Ще одна важлива річ, яку слід розуміти, полягає в тому, що ніяка модель не є догмою або універсальним рішенням. Немає ідеальної моделі. Є та, яка гірше або краще підходить для конкретного проекту, конкретної команди, конкретних умов.

Існують три основні стратегії розробки ПЗ:

1. ***Одноразовий прохід*** (водоспадна стратегія) – лінійна послідовність етапів конструювання.

2. ***Інкрементна (чи ітеративна) стратегія***. На початку процесу визначаються усі призначені для користувача і системні вимоги, частина конструювання, що залишилася, виконується у вигляді послідовності версій (заплановане поліпшення продукту).

3. ***Еволюційна стратегія***. Система також будується у вигляді послідовності версій, але на початку процесу визначені не усі вимоги. Вимоги уточнюються в результаті розробки версій.

Розглянемо коротко декілька моделей життєвого циклу ПЗ. Вони усі базуються на розглянутому скелеті і так чи інакше його включають.

2.1 Класична або каскадна модель

Найбільш широко відомою і вживаною довгий час залишалася класична або так звана каскадна (70-85 р.р.) або водоспадна (waterfall) модель життєвого циклу. Вона

була уперше чітко сформульована в стандартах міністерства оборони США (автор Уїнстон Ройс, 1970).

Ця модель припускає послідовне виконання різних видів діяльності, починаючи з вироблення вимог і закінчуючи супроводом, з чітким визначенням меж між етапами, на яких набір документів, виробленою на попередній стадії, передається в якості вхідних даних для наступної.

Дуже часто класичний життєвий цикл називають каскадною або водоспадною моделлю, підкреслюючи, що розробка розглядається як послідовність етапів, причому перехід на наступний, ієрархічно нижній етап відбувається тільки після повного завершення робіт на поточному етапі (рис. 2.1).

Рисунок 2.1 – Каскадна модель ЖЦ ПЗ

Охарактеризуємо зміст основних етапів.

Стратегія. Визначення стратегії припускає обстеження системи. Основне завдання обстеження – оцінка реального об'єму проекту, його цілей і завдань, а також отримання визначень суті і функцій на високому рівні. На цьому етапі притягуються висококваліфіковані бізнес-аналітики, які мають постійний доступ до керівництва фірми; етап припускає тісну взаємодію з основними користувачами системи і бізнес-експертами. Основне завдання взаємодії – отримати якомога повнішу інформацію про систему (повне і однозначне розуміння вимог замовника) і передати

дану інформацію у формалізованому вигляді системним аналітикам для подальшого проведення етапу аналізу.

Результатом етапу визначення стратегії є документ, де чітко сформульовано: що отримає замовник, якщо погодиться фінансувати проект; коли він отримає готовий продукт (графік виконання робіт); скільки це коштуватиме (для великих проектів повинен бути складений графік фінансування на різних етапах робіт).

Виконана на даному етапі робота дозволяє відповісти на питання, чи варто продовжувати даний проект і які вимоги замовника можуть бути задоволені за тих або інших умов. Може так статися, що проект продовжувати не має сенсу, наприклад через те, що ті або інші вимоги не можуть бути задоволені за якимись об'єктивними причинами. Якщо ухвалюється рішення про продовження проекту, то для проведення наступного етапу аналізу вже є уявлення про об'єм проекту і кошторис витрат.

Аналіз. Етап аналізу припускає докладне дослідження бізнес-процесів (вимог і функцій, визначених на етапі вибору стратегії) і інформації, необхідної для їх виконання (сутності, їх атрибутів і зв'язків (відносин)). Вся інформація про систему, зібрана на етапі визначення стратегії, формалізується і уточнюється на етапі аналізу. Особливу увагу слід приділити повноті переданої інформації, аналізу інформації на предмет відсутності суперечностей, а також пошуку невживаною взагалі або інформації, що дублюється. Як правило, замовник не відразу формує вимоги до системи в цілому, а формулює вимоги до окремих її компонентів. Тому необхідно приділити увагу узгодженості цих компонентів. Фактично ці вимоги визначають повне завдання на розробку.

Проектування (моделювання). Моделювання присвячене виконанню двох дій – аналізу вимог і проектуванню. Результати цих дій моделі – зазвичай записуються на графічній мові моделювання, мові картинок.

Розгляд результатів аналізу – це процес передачі інформації від аналітиків проектувальникам. На практиці це інтеративний процес. У проектувальників неминуче виникатимуть питання до аналітиків, і навпаки. Інформація про систему постійно уточнюватиметься. При розробці схеми бази даних може змінитися інформаційна модель, отримана на етапі аналізу, наприклад, тому, що наявне проектне рішення

нестабільне або поволі працює при реалізації його за допомогою вибраної СУБД або через інші причини.

Робота проектувальників бази даних в значній мірі залежить від якості інформаційної моделі. Інформаційна модель не повинна містити ніяких незрозумілих конструкцій, які не можна реалізувати в рамках вибраної СУБД. Слід зазначити, що інформаційна модель створюється для того, щоб на її основі можна було побудувати модель даних, тобто повинна враховувати особливості реалізації вибраної СУБД.

Побудова логічної і фізичної моделей даних є основною частиною проектування бази даних. Отримана в процесі аналізу інформаційна модель спочатку перетвориться в логічну, а потім у фізичну модель даних. Після цього для розробників інформаційної системи створюється пробна база даних. З нею починають працювати розробники коду. У ідеалі до моменту початку розробки модель даних повинна бути стійка. Проектування бази даних не може бути відірване від проектування модулів і додатків, оскільки бізнес-правила можуть створювати об'єкти в базі даних, наприклад серверні обмеження.

Головна мета проектування полягає у відображенні функцій, отриманих на етапі аналізу, в модулі інформаційної системи. Визначення модулів розкриваються в технічній специфікації програм. При проектуванні модулів визначають розмітку меню, вид вікон, гарячі клавіші і пов'язані з ними виклики.

Тестування. Проектування процесу тестування, як правило, слідує за процесом функціонального проектування і проектування схеми бази даних. На цьому етапі можна використовувати складні схеми тестування, а можна обмежитися і простими. Коли генерація модуля завершена, виконують автономний тест, який переслідує дві основні цілі:

− виявлення відмов модуля (жорстких збоїв);

− відповідність модуля специфікації (наявність всіх необхідних функцій, відсутність зайвих функцій).

Після того, як автономний тест пройшов успішно, група модулів, що згенерували, проходить тести зв'язків, які повинні відстежити взаємний вплив модулів.

Далі група модулів тестується на надійність роботи, тобто проходять, по-перше, тести імітації відмов системи, а по-друге, тести напрацювання на відмову. Перша

група тестів показує, наскільки добре система відновлюється після збоїв програмного забезпечення, відмов апаратного забезпечення. Друга група тестів визначає ступінь стійкості системи при штатній роботі і дозволяє оцінити час безвідмовної роботи системи. У комплект тестів стійкості повинні входити тести, що імітують пікове навантаження на систему.

Потім весь комплект модулів проходить системний тест – тест внутрішнього приймання продукту, що показує рівень його якості. Сюди входять тести функціональності і тести надійності системи.

Останній тест інформаційної системи – приймально-здавальні випробування. Такий тест передбачає показ інформаційної системи 34 замовникові і повинен містити групу тестів, що моделюють реальні бізнес-процеси, щоб показати відповідність реалізації вимогам замовника.

Реалізація (розробка, кодування). На етапі розробки здійснюється тісна взаємодія проектувальників, розробників і груп тестерів. У разі інтенсивної розробки тестер фактично є членом групи розробки. Проектувальник на даному етапі виконує функції «ходячого довідника», оскільки постійно відповідає на питання розробників, що стосуються технічної специфікації. Найчастіше на етапі розробки міняються інтерфейси користувача. Це обумовлено у тому числі і тим, що модулі періодично демонструються замовникові. Істотно можуть мінятися і запити до даних. Взаємодія тестера і розробника без централізованої передачі частин проекту допустимо, але тільки у випадку, якщо необхідно терміново перевірити якусь правку. Дуже часто етап розробки і етап тестування взаємозв'язані і йдуть паралельно.

Введення в дію (упровадження, розгортання) – етап класичного життєвого циклу націлений на дві дії: постачання розробленого продукту замовникові і супровід процесу експлуатації цього продукту. Згідно із статистичними даними, 65% супроводи пов’язано з удосконаленням ПО, 18% відводиться на адаптацію і 17% пов’язано з виправленням помилок.

Експлуатація перекриває процес тестування, система вводиться в експлуатацію не повністю, а поступово. Введення в експлуатацію проходить три фази:

− первинне завантаження інформації;

− накопичення інформації;

− вихід на проектну потужність.

Первинне завантаження інформації ініціює досить вузький круг помилок – в основному це проблеми розузгодження даних при завантаженні і власні помилки завантажувачів, тобто те, що не було відстежене на тестових даних. Подібні помилки повинні бути виправлені щонайшвидше.

В період накопичення інформації виявиться найбільша кількість помилок, допущених при створенні ПЗ. Як правило, це помилки, пов'язані з багатокористувальницьким доступом. Часто на етапі тестування таким помилкам не приділяється належної уваги – мабуть, ізза складності моделювання і дорожнечі засобів автоматизації процесу тестування системи в умовах багатокористувальницького доступу. Деякі помилки виправити буде складно, оскільки вони є помилками проектування. Жоден хороший проект від них не застрахований. Це означає, що про всяк випадок треба резервувати час на локалізацію і виправлення таких помилок.

Друга категорія виправлень пов'язана з тим, що користувача не влаштовує інтерфейс. Тут не завжди потрібно виконувати абсолютно всі побажання користувача, інакше процес введення в експлуатацію не кінчиться ніколи.

Позитивні сторони застосування каскадного підходу полягають в наступному:

− дає план і часовий графік по усіх етапах проекту, упорядковує хід конструювання;

− на кожному етапі формується закінчений набір проектної документації, що відповідає критеріям повноти і узгодженості;

− виконувані в логічній послідовності етапи робіт дозволяють планувати терміни завершення усіх робіт і відповідні витрати.

Каскадний підхід добре зарекомендував себе при побудові ПЗ, для якого на самому початку розробки можна досить точно і повно сформулювати усі вимоги з тим, щоб надати розробникам свободу реалізувати їх якнайкраще з технічної точки зору.

Основними недоліками каскадного підходу є:

− істотне запізнювання з отриманням результатів, оскільки реальні проекти часто вимагають відхилення від стандартної послідовності кроків;

− вимога повного закінчення фази-діяльності, закріплення результатів у вигляді детального початкового документу (технічного завдання, проектної специфікації);

− узгодження результатів з користувачами робиться тільки в точках, що плануються після завершення кожного етапу робіт;

− користувачі і замовник не можуть ознайомитися з варіантами системи під час розробки, і бачать результат тільки в самому кінці;

− вимоги до ПЗ «заморожені» у вигляді технічного завдання на увесь час її створення, тому користувачі можуть внести свої зауваження тільки після того, як робота над системою буде повністю завершена.

Незважаючи на наполегливі рекомендації компаній – експертів в області проектування і розробки ПЗ за допомогою новітніх технологій, багато компаній продовжують використати каскадну модель на практиці замість якого-небудь варіанту ітераційної моделі.

Головні причини, по яких каскадна модель зберігає свою популярність, наступні: звичка, розробка невеликих проектів, ілюзія зниження ризиків учасників проекту (замовника і виконавця), проблеми впровадження при використанні ітераційної моделі.

2.2 Компонетні моделі

У більшості програмних проектів застосовується повторне використання деяких програмних модулів (компонентів). Це зазвичай трапляється там, де розробники проекту знають про раніше створені програмні продукти, у складі яких є компоненти, що приблизно задовольняють вимогам компонентів, що розробляються. Цей підхід заснований на наявності великої бази існуючих програмних компонентів, які можна інтегрувати в створювану нову систему. Часто такими компонентами є програмні продукти, що вільно продаються на ринку, які можна використати для виконання певних спеціальних функцій.

У цьому підході початковий етап специфікації вимог і етап атестації (тестування і введення в експлуатацію) такі ж, як і в інших моделях процесу створення ПЗ. А етапи, розташовані між ними, мають наступний сенс.

1. ***Аналіз компонентів.*** Маючи специфікацію вимог, на цьому етапі здійснюється пошук компонентів, які могли б задовольняти сформульованим вимогам.

2. ***Модифікація вимог.*** На цій стадії аналізуються вимоги з урахуванням інформації про компоненти, отриманої на попередньому етапі. Вимоги модифікуються так, щоб максимально використати можливості відібраних компонентів.

3. ***Проектування системи.*** На цьому етапі проектується структура системи або модифікується існуюча структура повторно використовуваної системи.

4. ***Розробка і зборка системи.*** Це етап безпосереднього створення системи. У рамках даного підходу збірка системи є швидше частиною розробки системи, чим окремим етапом.

Основні переваги описуваної моделі полягають в тому, що скорочується кількість компонентів, що безпосередньо розробляються, і зменшується загальна вартість створюваної системи.

2.3 Макетування (прототипування)

Досить часто замовник не може сформулювати детальні вимоги по введенню, обробці або виведенню даних для майбутнього програмного продукту. У цих випадках доцільно використати макетування.

Основна мета макетування: зняти невизначеності у вимогах замовника.

Макетування (прототипування) – це процес створення моделі необхідного програмного продукту. Модель може приймати одну з трьох форм:

1) паперовий макет або макет на основі ПК (зображує або малює людино-машинний діалог);

2) працюючий макет (прототип, версія програми), що виконує деяку частину необхідних функцій;

3) існуюча програма (версія), характеристики якої потім мають бути поліпшені.

Як показано на рис. 2.2, макетування ґрунтується на багаторазовому повторенні ітерацій, в яких беруть участь замовник і розробник.

Рисунок 2.2 – Макетування (прототипування)

2.4 Ітеративні (інкрементні) моделі

Ітеративна модель є класичним прикладом інкрементної стратегії розробки. Вона об’єднує елементи послідовної водоспадної моделі з ітераційним макетуванням. Кожна лінійна послідовність тут виробляє інкремент (версію) ПЗ, що поставляється.

Основний недолік каскадного підходу – відсутність гнучкості. Саме цей недолік долається каскадно-поворотним підходом, в якому дозволені повернення до попередніх стадій і перегляд або уточнення раніше прийнятих рішень. Каскадно-поворотний підхід відбиває ітеративний або ітераційний характер розробки ПЗ.

Ітеративні моделі припускають розбиття створюваної системи на набір кроків, які розробляються за допомогою декількох послідовних проходів усіх робіт або їх частини. При цьому велика частина або навіть повний цикл робіт проходиться на нього, потім оцінюються результати і на наступній ітерації розробляється наступний, який може залежати від першого, або доопрацьовується перший з додаванням нових функцій.

В результаті на кожній ітерації можна аналізувати проміжні результати робіт і реакцію на них усіх зацікавлених осіб, включаючи користувачів, і вносити зміни, що коригують, на наступних ітераціях. Кожна ітерація може містити повний набір видів діяльності від аналізу вимог, до введення в експлуатацію чергової частини ПЗ. Каскадна модель з можливістю повернення на попередній крок стає ітеративною (рис. 2.4).

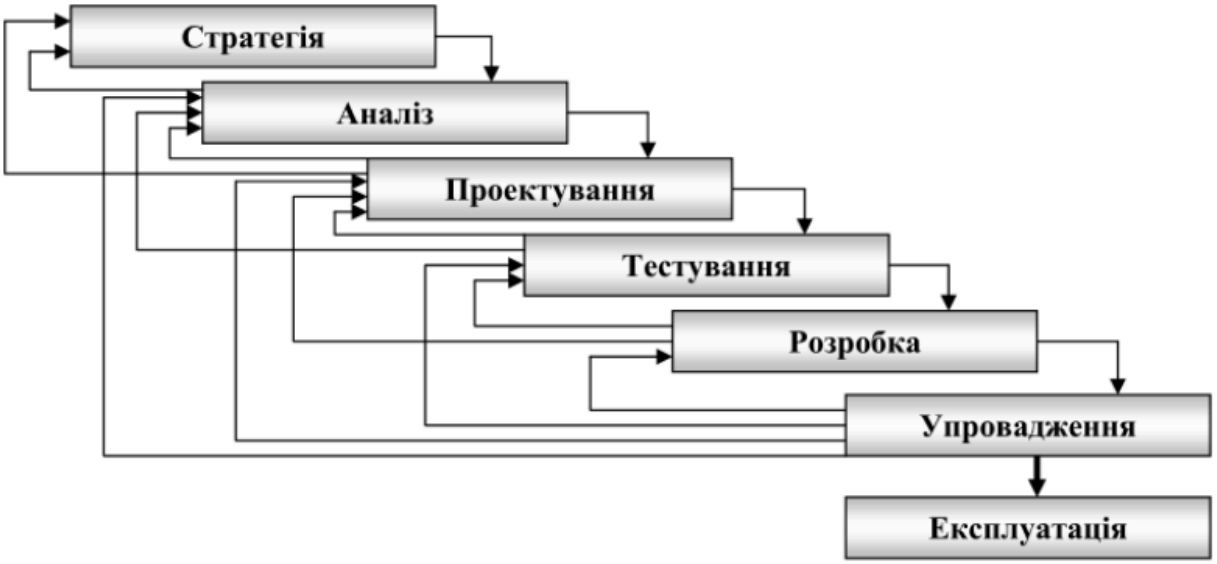

Рисунок 2.4 – Можливий хід робіт по ітеративній каскадній моделі

Процес ітераційної розробки має цілий ряд переваг.

Замовникові немає необхідності чекати повного завершення розробки системи, щоб отримати про неї представлення. Компоненти, отримані на перших кроках розробки, задовольняють найбільш критичним вимогам (оскільки зазвичай мають найбільший пріоритет) і їх можна оцінити на самій ранній стадії створення системи.

2.5 Спіральна модель

Розвитком ідеї ітерацій є спіральна ***модель життєвого циклу ПЗ, запропонована Барри Боэмом*** в 1988 році з метою скоротити можливий ризик розробки.

Спіральна модель використовує поняття **прототипу** – програми, що реалізовує часткову функціональність створюваного програмного продукту.

Створення прототипів здійснюється за декілька витків спіралі, кожен з яких складається з «аналізу ризику», «деякого процесу» і «верифікації».

Спіральна модель (рис. 2.5) – класичний приклад застосування еволюційної стратегії конструювання, була запропонована для подолання перерахованих проблем в попередніх моделях.

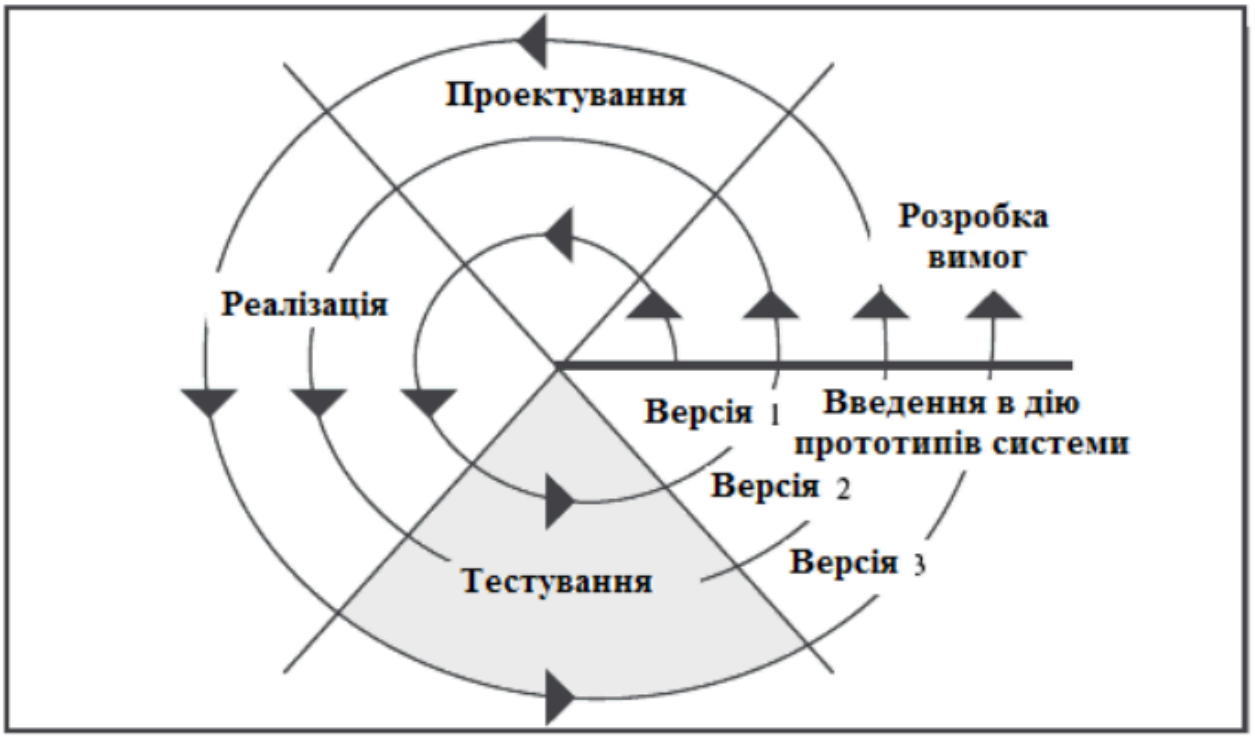

Рисунок 2.5 – Спіральна модель ЖЦ ПЗ

Спіральна модель пропонує кожну ітерацію розпочинати з виділення цілей і планування чергової ітерації, і робить упор на початкові етапи ЖЦ: аналіз і проектування. На цих етапах реалізовані технічні рішення перевіряються шляхом створення ***прототипів***. Кожен виток спіралі відповідає створенню фрагмента або версії ПЗ, на нім уточнюються цілі і характеристики проекту, визначається його якість і плануються роботи наступного витка спіралі. Таким чином поглиблюються і послідовно конкретизуються деталі проекту і в результаті вибирається обґрунтований варіант, який доводиться до реалізації.

Перевагами спіральної моделі є:

1. Неповне завершення робіт на кожному етапі дозволяє переходити на наступний етап, не чекаючи повного завершення роботи на поточному.

2. Кожен виток спіралі відповідає створенню фрагмента або версії ПЗ, на ньому уточнюються цілі і характеристики проекту.

3. Дозволяє явно враховувати ризик на кожному витку еволюції розробки.

4. Використовує моделювання для зменшення ризику і вдосконалення програмного виробу.

Серед проблем спірального циклу можна виділити наступні:

1. Визначення моменту переходу на наступний етап. Для її вирішення необхідно ввести тимчасові обмеження на кожного з етапів життєвого циклу.

2. Підвищені вимоги до замовника.

3. План складається на основі статистичних даних, отриманих в попередніх проектах, і особистого досвіду розробників.

У спіральній моделі немає фіксованих етапів, таких як розробка специфікації або проектування. Ця модель може включати будь-які інші моделі розробки систем на певному витку спіралі.

2.6 Життєвий цикл тестування

В етапах життєвого циклу розробки програмного продукту є етап, на якому виконується тестування частини або цілком усього продукту. Незалежно від того відповідно до якої методології ЖЦ ПЗ проводиться розробка, можна виділити загальні стадії (стани) та етапи, які притаманні кожній з методологій та є основою будь-якого процесу розробки. При цьому, тестування має свої цілі, завдання, роль, види, методи, критерії, свою методологію та технологію. На основі аналізу сучасних методологій і моделей якості, можна зробити висновок, що тестування має свій власний життєвий цикл – життєвий цикл тестування програм (ЖЦ ТП).

Для структуризації процесу тестування розглянемо основні етапи розробленого життєвого циклу тестування програмного забезпечення на базі каскадної моделі життєвого циклу розробки ПЗ.

1. **Аналіз вимог**. Для людини, що займається тестуванням програмного забезпечення, інформація про те, чого хоче клієнт, дуже важлива. Клієнтів буде більше, якщо вони отримуватимуть саме те, що хочуть. На фазі аналізу вимог тестувальник повинен отримати вимоги до тестованого продукту і сформувати з них «матрицю вимог», що дозволить в майбутньому стежити, щоб програма залишалася такою, якою хоче бачити клієнт.

2. **Аналіз дизайну проекту**. На етапі життєвого циклу тестувальники вирішуватимуть, який підхід до тестування буде використовуватися, рішення залежить

від «матриці вимог», яка створилася на минулій фазі життєвого циклу. Наприклад, компанія розробляє продукт – відеоредактор, основними вимогами виявилися швидкість конвертації відео і дружній інтерфейс. На стадії аналізу дизайну проекту було прийнято рішення розробити автоматичний тест для вимірювання швидкості конвертації відео, і знайти людину, яка буде дивитися за тим, щоб інтерфейс був дружнім.

3. **Планування тестування** – це етап, на якому проводитися планування тестів. Тестувальники планують, як вони перевірятимуть вимоги, пред'явлені до продукту, як будуть підібрані тестові дані (відео матеріал, і еталонний час роботи з ним). На цьому етапі проводитися розподіл завдань між тестувальниками.

4. **Розробка тестів**. Розробка тестів ведеться паралельно розробці програми. Як тільки розробляється частина програми, до неї розробляють тест, який забезпечить тестування цієї частини програми.

5. **Виконання тестів**. Виконання тестів проводитися постійно в процесі розробки продукту, для того щоб найраніше відстежити дефекти.

6. **Написання звітів**. Після виконання тестування тестувальники повинні написати звіти про знайдені дефекти і до тих частин програми, які працюють правильно. У будь-якому випадку повинна бути документація про всі частини продукту, не залежно від знайдених дефектів.

7. **Повторна перевірка дефектів**. Коли звіти були написані і передані розробникам, їх завдання виправити знайдені дефекти додатку. Після виправлення дефектів весь додаток повинен пройти повторну перевірку тестувальниками.

2.7 Контрольні питання до теми:

1. Поняття моделі ЖЦ.
2. Стратегії розробки ПЗ.
3. Класична (каскадна) модель.
4. Етапи класичної моделі.
5. Компонентні моделі.
6. Прототипування, його особливості.

7. Інкреметні моделі. Принцип. Переваги та недоліки.

8. Спіральна модель. Принцип. Переваги та недоліки.

9. Життєвий цикл тестування. Етапи ЖЦ ТП.

Тема №3. Характеристика використання, застосування переваг та недоліків функціонального та нефункціонального тестування

3.1 Функціональне тестування

Функціональне тестування — це тип тестування програмного забезпечення, під час якого система перевіряється на відповідність вимогам і специфікаціям. Функціональне тестування гарантує, що вимоги або специфікації належним чином задовольняються продуктом. Цей тип тестування, у першу чергу, стосується результату обробки. Він зосереджений на моделюванні фактичного використання системи, але не розробляє жодних припущень про структуру системи. За базу визначення цього типу тестування береться перевірка, що кожна функція програми працює відповідно до вимог і специфікацій. Цей вид не стосується вихідного коду програми. Кожна функціональність програми чи сайту перевіряється шляхом надання відповідних тестових вхідних даних, очікування вихідного результату та порівняння фактичного результату з очікуваним результатом.

Функціональне тестування — це загальний термін для багатьох типів тестів, призначених для перевірки всієї функціональності програмного забезпечення. Функціональне тестування підтверджує, що програмне забезпечення працює належним чином і не містить помилок. Щоб підтвердити це, тестувальник моделює реальний сценарій кінцевого користувача та порівнює результати тесту з очікуваними результатами, що зазначені у документі вимог. Функціональне тестування може виконуватися вручну або автоматизовано.

Здебільшого, функціональне тестування зосереджено на зовнішній поведінці програмного забезпечення, тому немає потреби у знаннях внутрішнього вихідного коду, і здебільшого воно вважається тестуванням чорного ящику. Цей тип тестування виконується перед нефункціональним тестуванням. Існує кілька типів функціонального тестування, наприклад: димове тестування (Smoke testing), санітарне тестування (Sanity testing), регресійне тестування (Regression testing) тощо.

3.2 Нефункціональне тестування

Нефункціональне тестування – це тип тестування програмного забезпечення, яке виконується для перевірки нефункціональних вимог продукту. Воно перевіряє, чи відповідає поведінка системи вимогам, чи ні. Цей тип тестування – широкий термін, який містить усі аспекти, не пов'язані з функціональністю системи. Серед аспектів, які вважаються нефункціональними, є продуктивність, безпека, зручність використання тощо. **Основна мета нефункціонального тестування** — відповісти на потреби клієнтів і зробити продукт зручним та безпечним для користувача. Нефункціональні тести зосереджуються на часі відгуку, надійності системи, питаннях безпеки тощо.

Нефункціональне тестування виконується переважно як автоматизоване тестування з використанням різних засобів автоматизації залежно від типу тесту. Типи тестування, які класифікуються як нефункціональне тестування, це навантажувальне тестування (Load testing), стрес-тестування (Stress testing), тестування доступності (Accessibility testing) тощо.

3.3 Різниця між функціональним та нефункціональним тестуванням

Табличне подання різниці між тестуваннями наведено на табл. 3.1.

Таблиця 3.1 – Різниця між функціональним та нефункціональним тестуваннями

Ознака	**Функціональне тестування**	**Нефункціональне тестування**
Визначення	Тип тестування, який підтверджує визначені характеристики продукту	Тип тестування, який підтверджує аспекти поведінки продукту
Що перевіряється	Код продукту у процесі роботи досягає поставлених вимогами результатів	Код продукту працює належним чином та вказаним способом
На чому базується	На специфікації та вимогах	На очікуваннях користувачів
Мета	Поліпшити поведінку продукту	Покращити продуктивність продукту

		Продовження таблиці 3.1
Цілі	Перелічені функції продукту працюють	Фактори швидкості, продуктивності та зручності використання працюють
Ручне/ автоматизоване	Може бути і ручним, і автоматизованим	Автоматизоване
Коли виконується	Перед нефункціональним	Після функціонального
Переваги	● Перевіряє, що програмне забезпечення працює належним чином ● Дозволяє уникати помилок та дефектів у програмному забезпеченні ● Допомагає задовольнити потреби бізнесу та користувачів ● Зроблено з точки зору реального кінцевого користувача	● Покращує продуктивність програмного забезпечення ● Забезпечує високий рівень безпеки ● Робить програмне забезпечення більш зручним для користувача ● Допомагає програмному забезпеченню задовольнити потреби клієнтів
Види тестування	● Модульне тестування (Unit testing) ● Інтеграційне тестування (Integration testing) ● Тестування системи (System testing) ● Димове тестування (Smoke testing) ● Санітарне тестування (Sanity testing) ● Регресійне тестування (Regression testing)	● Тестування продуктивності (Performance testing) ● Тестування безпеки (Security testing) ● Тестування навантаження (Load testing) ● Стрес-тестування (Stress testing) ● Об'ємне тестування (Volume testing) ● Тестування масштабованості (Scalability testing) ● Тестування сумісності (Compatibility testing) ● Юзабіліті тестування (Usability testing)

3.3.1 Розбір на прикладі інтернет-магазину

Розберемо кожен із видів на прикладі тестування сайту інтернет-магазину.

Функціональним тестуванням для нього буде перевірка того, чи виконує сайт ключові функції відповідно до вимог продукту: чи можна товар додати до кошика, чи вірно рахується кількість і вартість товару у кошику, чи валідуються поля на формі оформлення замовлення тощо.

Нефункціональне тестування сайту передбачатиме усі перевірки щодо швидкості завантаження та відгуку сторінок, тестування у різних браузерах, тестування візуальних складових на зручність для цільового користувача та багато іншого.

Табличне подання застосування двох видів тестування наведено на табл. 3.2.

Таблиця 3.2 – Застосування функціонального та нефункціонального тестування у контексті інтернет-магазину

Вид тестування	*Приклад застосування (у розрізі сайту інтернет-магазину)*
Функціональні види тестування	
Модульне тестування (Unit testing)	Тестування окремих компонентів (модулів) сайту, зазвичай з позиції аналізу коду продукту. Наприклад, перевірка коду функції додавання товару до списку бажань
Інтеграційне тестування (Integration testing)	Перевірка різних компонентів/модулів сайту як єдиного цілого (у взаємозв’язку). Наприклад, перевірка інтерфейсу при авторизації, додаванні товару до списку бажань тощо
Тестування системи (System testing)	Тестування сайту як комплексної системи з урахуванням особливостей взаємодії з іншим програмним/апаратним забезпеченням, необхідним для роботи. Наприклад, виконання комплексу тестів з реєстрації, повторного входу, додавання товару до списку бажань, до кошика, оформлення й підтвердження замовлення
Димове тестування (Smoke testing)	Перевірка після доопрацювань/змін критично важливих функцій, чи вони працюють відповідно до очікувань. Наприклад, чи успішно здійснюється реєстрація на сайті після зміни способу реєстрації з електронної пошти на СМС
Регресійне тестування (Regression testing)	Тестування функціонала з метою переконатися, що зміни в коді не завдали побічних ефектів на функціонал, що існує. Наприклад, чи додавання значення віку при реєстрації не вплинуло на можливість відновити пароль на сайті

	Продовження таблиці 3.2
Санітарне тестування (Sanity testing)	Детальна перевірка функціонала з незначними змінами. Наприклад, чи успішно здійснюється реєстрація на сайті після додавання до форми реєстрації поля з зазначенням віку
Нефункціональні види тестування	
Тестування продуктивності (Performance testing)	Тестування сайту при очікуваному або більш високому навантаженні (наприклад, при планових 100 одночасних користувачів, при незначному збільшенні - 120 одночасних користувачів)
Тестування безпеки (Security testing)	Дослідження, що спрямовані на пошук багів, пов’язаних зі збереженням даних користувача. Наприклад, чи можливо авторизуватися до особистого кабінету покупця з невірним паролем
Тестування навантаження (Load testing)	Тестування сайту за нормальних умов та нормальної завантаженості. Наприклад, чи допустимий час відгуку сторінки після успішного оформлення замовлення
Стрес-тестування (Stress testing)	Перевірка реакції сайту на більше, ніж очікувано, робоче навантаження (наприклад, 150 чи навіть 1000 одночасних користувачів) з метою виявлення витоків пам’яті та перевірки надійності сайту
Об'ємне тестування (Volume testing)	Тестування сайту при збільшеній кількості оброблюваних даних. Наприклад, при надвеликій кількості товарів у кошику (500 різних позицій) або ж надмірній кількості товарів у списку бажань
Тестування масштабованості (Scalability testing)	Перевірка здатності сайту до модернізації з метою задоволення зростаючого навантаження. Наприклад, при передбаченому зростанні кількості відвідувачів до 500 напередодні новорічних свят, слід визначити, які показники продуктивності можна поліпшити, щоб сайт і надалі працював належним чином
Тестування сумісності (Compatibility testing)	Перевірка роботи сайту у різному оточенні, наприклад, у різних браузерах чи у браузерах різних операційних систем
Юзабіліті тестування (Usability testing)	Тестування зручності використання сайту. Наприклад, чи іконка кошика розміщена справа в шапці сайту, як звикли користувачі, чи не надто багато кроків у процесі оформлення замовлення

Кожен із видів тестування є важливим та відіграє чи не найголовнішу роль у процесі перевірки продукту. Знання відмінностей функціональних та нефункціональних перевірок допоможе у плануванні та реалізації процесів тестування.

3.4 Контрольні питання до теми:

1. Функціональне тестування. Перевага. Використання. Недоліки.
2. Нефункціональне тестування. Перевага. Використання. Недоліки.
3. Різниця між функціональним та нефункціональним тестуванням.
4. Застосування на практиці функціонального та нефункціонального тестування.

Тема №4. Профілювання ПЗ

Профілювання – збір та аналіз інформації про виконання програми з метою оптимізації її роботи, застосовується в процесі розробки програмного забезпечення. Профілювання — форма аналізу динамічних показників програми, в протилежність статичному аналізу коду. Звичайна **задача аналізу продуктивності** — визначити частини програми, які слід оптимізувати для покращення використання пам'яті або підвищення швидкості. Профілювання виконується за допомогою спеціальних програмних засобів, що називаються **профайлерами**.

4.1 Використання профайлерів

Інструменти програмного аналізу критично важливі для розуміння поведінки програми. Комп'ютерним архітекторам потрібні такі інструменти, аби оцінити, як хороші програми виконуватимуться на новій архітектурі. Авторам програмного забезпечення також потрібні інструменти, аби проаналізувати їх програми і ідентифікувати критичні частини коду. Автори компіляторів часто використовують такі інструменти, аби з'ясувати, як добре виконується їх планування інструкцій або алгоритм передбачення, що відгалужується.

Профайлер — інструмент аналізу продуктивності, який збирає дані для профілювання, особливо кількість викликів і тривалість виконання функцій. Вихідний результат — потік записаних подій (a trace) або статистичний короткий звіт спостережуваних подій (a profile). Профайлери використовують широку різноманітність методів, аби зібрати дані, у тому числі апаратні переривання, апаратну підтримку, пастки операційної системи. Використання профайлерів потрібне в процесі планування продуктивності.

Оскільки підсумовування в профайлі пов'язано з позицією вихідного коду, розмір вихідних даних лінійно залежить від розміру коду програми та може залежати від часу її виконання. Для однопотокових програм профайл надає достатньо інформації для оптимізації, але проблеми продуктивності в багатопотокових програмах через очікувальні повідомлення або проблеми синхронізації часто залежать від взаємозв'язку

часу виникнення подій, тому вимагають розширеного запису профайлу, щоб зрозуміти проблему.

4.1.1 Типи профайлерів, основаних на виводі

1. Flat профайлер:

Flat-профайлери обчислюють середній час виклику функцій і не переривають виклики, засновані на callee або контексті.

2. Call-Graph профайлер:

Call Graph профайлери показують часи виклику і частоти функцій, а також ланцюги викликів, заснованих на callee. Проте не зберігають контекст.

4.2 Методи збору даних

4.2.1 Профайлери, засновані на подіях

Мови програмування, перелічені тут, мають профайлери, засновані на подіях:

1. .NET: Може прикріпити профілювального агента як сервер COM до CLR. Подібно до Java, час виконання потім забезпечує різні повторні виклики в агентові, для перехоплення подій подібно до методу JIT/ введення / вихід, створення об'єктів і т.п. Особливо потужний в цьому агент профілювання може переписати код цільової програми довільним способом.

2. Java: JVM-Tools Interface (колись JVM Profiling Interface) JVM API забезпечує пастки до профайлеру, для заманювання в пастку подій, таких як викликів, завантаження класу, вивантаження, вхід-вихід потоку.

3. Python: профілювання Python включає модуль профілювання, хотшот (який є заснованим на call-graph), і використовуючи 'Sys.setprofile()' модуль до подій-пасток подібно до c_{call,return,exception}, python_{call,return,exception}.

4. Ruby: Ruby також використовує подібний до Python інтерфейс для профілювання. Є flat-профайлер в profile.rb, модуль, і ruby-prof C-розширення.

4.2.2 Статистичні профайлери

Деякі профайлери оперують здійсненням вибірки. Профайлер, що здійснює вибірку, досліджує лічильник команд цільової програми з регулярними проміжками, використовуючи переривання операційної системи. Профайлери, що здійснюють вибірку, зазвичай менш точні і специфічні, але дозволяють цільовій програмі працювати майже на повну швидкість.

Деякі профайлери надають цільовій програмі додаткові команди збирати необхідну інформацію. Це може призводити до змін у виконанні програми, викликаючи неакуратні результати і помилки. Це може бути дуже специфічним, але уповільнює цільову програму, оскільки більше специфічної інформації збирається.

Результативні дані не остаточна істина, а статистична апроксимація. Фактична кількість помилок зазвичай більша, ніж один вибраний період вибірки. Фактично, якщо значення - n разів періоду вибірки, очікувана помилка в ньому - корінь квадратний з вибраних періодів.

Деякі з найбільш використовуваних статистичних профайлерів це GNU's gprof, Oprofile, AMD's CodeAnalyst та SGI's Pixie.

4.3 Діалог з користувачем

Хороше програмне забезпечення має бути швидким. Користувач не повинен гадати, чи то він кнопку не натиснув, чи то мишка не спрацювала, чи то програма все-таки щось робить.

Якщо потрібно виконати тривалу операцію, то необхідно запускати її в окремому потоці, тоді як в основному повідомляйте користувача про початок виконання операції, про хід виконання і реагуйте на його дії.

З одного боку, програмне забезпечення, принаймні, його інтерфейсна частина, повинна постійно вести активний діалог з користувачем. Хороше ПЗ не «зависає» без відповідного повідомлення і не робить того, чого користувач не замовляв.

З іншого боку, коли якась часта операція супроводжується питанням, на яке постійно потрібно відповідати, – це теж дратує користувача.

Так само варто відмітити, що діалог повинен здійснюватися грамотно і зрозумілою мовою. Повідомлення, що виводяться програмою, мають бути максимально повними і зрозумілими, але і не містити зайвої інформації. Не потрібно виводити зайвих повідомлень.

Програма веде діалог з користувачем засобами інтерфейсу. А створення якісного інтерфейсу і побудова хорошого діалогу – це окрема наука (юзабіліті, зручність використання), яка мало перетинається з програмуванням. Тому призначений для користувача інтерфейс хорошого ПЗ розробляється не програмістами, а дизайнерами призначеного для користувача інтерфейсу. Звичайно, можливо, що хороший програміст також є і хорошим дизайнером, але у більшості випадків це не так.

4.4 Система довідки

Про всяк випадок відмітимо, що хороше ПЗ супроводжується хорошою документацією і посібником користувача. Але тут мова піде не про це. У цьому підрозділі говоритися про те, що кожен елемент (блок, вікно, відеокадр і тому подібне) повинен мати кнопку «Довідка».

При натисненні цієї кнопки користувач повинен отримати вичерпну інформацію про призначення елементу в цілому і в кожній його частині окремо, про те як користуватися тим або іншим елементом.

Довідка форми введення повинна містити приклади даних, що вводяться. При тому не лише коректних, але і не коректних. Довідка форми виведення повинна описувати усі дані, що виводяться. Природно, що довідка має бути написана не менш хорошою мовою, ніж діалоги.

4.5 Обробка помилок

Помилки можна розділити на три види: помилки користувача, помилки програміста і помилки устаткування. Усі три види помилок хороше ПЗ повинно грамотно обробляти.

Помилки користувача найчастіше зводяться до різних друкарських помилок і нерозуміння призначення елементів управління. Такі помилки не повинні ставати

причинами програмних помилок: дані, що вводяться, повинні перевірятися, і якщо виявлена помилка, то має бути надана можливість для її виправлення. При тому усе це повинно бути добре прокоментовано. Не можна просто припиняти виконання операції, користувач повинен зрозуміти, чому операція не виконана. Намагайтеся давати користувачеві рекомендації щодо виправлення помилок.

Обробка помилок програміста включає їх виявлення (перехоплення), спробу виправлення і відновлення роботи програми. Якщо помилка виправлена, то не потрібно повідомляти про неї користувачеві. Якщо виправити не вдалося, то необхідно повернути програму в коректний стан і повідомити користувача, в який стан програма повернена. Якщо помилка критична, то необхідно акуратно завершити програму, повідомивши користувача про причини завершення і дати йому можливість зберегти якомога більше даних.

Помилки програміста включають і непередбачену поведінку програми. Тому в точках галуження необхідно враховувати непередбачені варіанти.

Наприклад, у блоці switch завжди має бути мітка default, яка зупиняє виконання операції з відповідними повідомленнями користувача.

Помилки устаткування зазвичай виправити програмними засобами неможливо. Тому кращою стратегією буде повідомлення про неї користувачеві, відмінити операцію, якщо програма може продовжити роботу або коректне завершення в іншому випадку.

4.6 Юніт-тести

Хороше ПЗ на передбачених вхідних даних, завжди працює правильно. Але це не означає, що потрібно писати програми без помилок – так не буває. Але потрібно бути упевненим, що передбачена поведінка працює без помилок.

Якщо функція учора працювала коректно, то це зовсім не означає, що вона і сьогодні працює коректно. У складних застосуваннях, із складними зв'язками буває дуже важко передбачити наслідки тієї або іншої зміни.

Для того щоб не гадати «А чи все правильно працює?», існують юніт-тести. Вони дозволяють відносно швидко (на багато швидше, ніж вручну) перевірити

працездатність усієї передбаченої функціональності і переконатися, що, виправивши одну помилку, було не зроблено іншу.

Краще всього, якщо час виконання дозволяє, при тестуванні поточної роботи виконувати тести усієї системи або модуля, щоб відразу виявити помилки, що з'явилися в інших ділянках коду. Якщо ж час не дозволяє, то юніт-тести необхідно запускати як мінімум один раз в день і обов'язково перед інтеграцією (з фіксацією в систему контролю версій).

4.7 Субкультура

Практично кожна область програмування має свої культурні особливості. Наприклад, існують такі поняття як «Культура паралельного програмування», «Культура мережевого програмування» тощо. Необхідно вивчати культуру тієї області, в якій працюєте. Це заощадить вам багато часу і сил.

Тут же можна сказати і про різні мови. Наприклад, в C і C++ широко розвинена культура роботи з пам'яттю і покажчиками, тоді як в Java і C#, за рахунок «збирача сміття», значення навичок роботи з пам'яттю сильно ослаблене.

Існують також мови, які підштовхують до безкультурності або, навпаки, до підвищення культури програмування. Це залежить від завдань, які зазвичай розв’язуються з використанням тієї або іншої мови програмування.

До розв’язання складних завдань залучаються висококваліфіковані фахівці, які задають високу культуру програмування. Відповідно, якщо мова використовується для розв’язання складних завдань, то і колектив розробників, зазвичай, складається з висококваліфікованих фахівців, які підвищують загальну культуру всього колективу розробників.

4.8 Контрольні питання до теми:

1. Поняття профайлера.
2.Пояснення задачі аналізу продуктивності.
3.Дати визначення поняттю профайлер.

4. Області використання профайлерів.

5. Поділ на типи профайлерів.

6. Flat-профайлер.

7. Call-Graph профайлер.

8. Методи збору даних.

9. Мови програмування, які відносяться до методів збору, що основані на подіях.

10. Необхідність діалогу з користувачем.

11. Необхідність обробки помилок.

12. Призначення системи довідки.

13. Юніт-тести.

14. Субкультури програмування.

Тема №5. Основні поняття та аспекти визначення якості ПЗ

5.1. Аспекти визначення якості та її атрибути

Якість ПЗ — набір характеристик продукту або сервісу, які характеризують його здатність задовольнити встановленим або передбачуваним потребам замовника. Поняття якості має різні інтерпретації в залежності від конкретної системи і вимог до програмного продукту. Крім того, в різних джерелах моделі якості відрізняються. Кожна модель має різне число рівнів і загальне число характеристик якості.

Область знань «Якість ПЗ (Software Quality)» складається з наступних розділів:

- концепція якості ПЗ (Software Quality Concepts);
- визначення та планування якості (Definition & Planning for Quality);
- діяльності і техніки гарантії якості і V & V (Activities and Techniques for Software Quality • Assurance, Validation-V & Verification - V);
- вимірювання в аналізі якості ПЗ (Measurement in Software Quality Analysis).

Під час вивчення даної галузі знань детально розглядаються проблеми якості ПЗ та шляхів його досягнення в процесі проектної діяльності груп розробників.

Концепція якості ПЗ включає зовнішні і внутрішні характеристики якості, їх мет рики, а також моделі якості, визначені на множині зовнішніх і внутрішніх характерист ик, які визначені у стандартах якості — це шість характеристик і для кожного з них 4 —5 атрибутів.

До характеристикам якості відносяться:

- Функціональність.
- Надійність.
- Зручності використання.
- Ефективність.
- Супровід.
- Переносимість.

Базова модель якості включає ці характеристики і відноситься до будь-якого типу програмних продуктів. Під час розробки вимог замовник формулює ті вимоги до якості, які найбільше підходять для програмного продукту, що замовляється.

Визначення і планування якості ПЗ ґрунтується на положеннях стандартів у цій галузі, складанні планів графіків робіт та процедури перевірки та ін. План

забезпечення якості включає набір дій для перевірки процесів забезпечення якості (верифікація, валідація тощо) і формування документа щодо управління якістю. Управління якістю застосовується до процесів, продуктами і ресурсами, а також включає вимоги до процесів та їх результатів.

Планування якості включає:

- Визначення продукту в термінах заданих характеристик якості.
- Планування процесів для отримання необхідної якості.
- Вибір методів оцінки планованих характеристик та встановленню відповідності продукту сформульованим вимогам.

Щоб зрозуміти широту визначення якості програмного забезпечення, потрібно відповісти на питання, що часто виникає: що таке якість? Після того, поняття якості розуміється легше, і легше зрозуміти різні структури якості наявні на ринку. Як випливає, і перш ніж ми приступимо до поняття якості, ми будемо витрачати час, щоб розібратися в питанні: що таке якість. Оскільки багато видних дослідників та авторів дали відповідь на це питання, ми не маємо амбіції введення ще однієї відповіді, але ми будемо, розглядати як відповіли деякі з найбільш видних дослідників з управління якістю. На основі цього ми можемо визначити, що існують два основних табори, визначення якості програмного забезпечення:

1. Відповідність специфікації: якість, що визначається як властивості продуктів і послуг, на основі вимірювальних характеристик, що задовольняють фіксованій специфікації — тобто відповідність в заздалегідь визначиній специфікації.

2. Задоволення потреб: якість, що визначається незалежно від будь-яких вимірних характеристик. Тобто, якість визначається як продукти чи послуги, що дають можливість задовольнити очікування клієнтів — явні чи ні.

Протягом багатьох років окремі автори й цілі організації визначали термін "якість" по—різному.

Уотс Хемпфрі (Watts Hamphrey, автор концепції моделі оцінки зрілості CMM, а також PSP і TSP — People Software Process і Team Software Process) описує якість як "досягнення відмінного рівня придатності до використання".

Уолтер Едвардс Демінг в "Out of the crisis: quality, productivity and competitive position ", говорить:

Труднощі у визначенні якості є переклад майбутніх потреб користувачів у вимірні характеристики, так що продукт може бути розроблений і розповсюджений, щоб дати задоволення з приводу ціни, яку користувач буде платити. Це не легко, і як тільки один стає досить успішним у цих зусиллях, виявляється, що потреби споживачів змінилися, з'явилися конкуренти в т.д.

Одна із сильних сторін Демінга в тому, що якість має бути визначено з точки зору задоволеності клієнтів — що набагато ширше поняття, ніж "відповідність специфікації" визначення якості (наприклад, з точки зору "задоволення потреб клієнтів"). Демінг пропонує, що якість має бути визначено тільки з точки зору агента — судді якості.

Філософія якості Демінга наголошує, що задовольнити і перевищити вимоги замовників є завданням, що всі в організації повинні виконати. Крім того, система управління якості працює з тим, що кожен несе відповідальність за якість його продукції з його внутрішніми клієнтами.

Компанія IBM, у свою чергу, ввела в обіг фразу "якість, керована ринковими потребами " ("market-driven quality ").

Критерій Белдріджа (Baldrige) для організаційного якості (NIST — National Institu te of Standards and Technology, "Baldrige National Quality Program") використовує схожу фразу — "якість, що задається споживачем" ("customer—driven quality").

Найчастіше, поняття якості використовується відповідно з визначенням системи менеджменту якості ISO 9001 як "ступінь відповідності властивих характеристик вимогам".

Фейгенбаум визначає, що ім'я та термін контролю якості, практично є синонімом через його глибокий вплив на концепцію тотального контролю якості (а також у зв'язку з його авторством концепції). У "Total quality control" Арманд Фейгенбаум Валле пояснює свій погляд на якість наступним текстом:

Якість визначається клієнтом, а не інженером, це не визначення маркетингу, ні загального визначення управління. В його основу покладено на фактичному досвіді клієнта з продуктом або послугою, вимірюваний проти його вимоги, — очевидні або неявні, свідомо чи просто відчув, технічно або повністю суб'єктивно — і завжди представляє рухомі мішені в умовах конкурентного ринку.

Якість товарів і послуг може бути визначена як: загальна складова продукту і експлуатаційні характеристики маркетингу, інжинірингу, виробництва та обслуговування, продуктів і послуг у використанні, що буде відповідати очікуванням замовника.

У визначенні Фейгенбаума якість як "задоволення потреб клієнтів" не викликає сумнівів. Насправді, він йде дуже широко у своєму визначенні якості, підкресливши важливість задоволення клієнтів у реальних і очікуваних потребах. Ясно, що у визначенні якості Фейгенбаум охоплює не тільки управління продуктами і послугами, а й сподівань клієнтів.

В " Jurans's Quality Control Handbook" Джозеф М. Джуран передбачає два значення для якості:

Слово якість має кілька значень. Два цих значення домінують у використанні цього слова: 1) Якість складається з тих властивостей продукту, які відповідають потребам клієнтів і тим самим забезпечують успіх продукту. 2) Якість складається через відсутність недоліків. Тим не менш, виходячи з цього найзручніше дати коротке визначення поняття якості, як "придатність для використання".

Таким чином, прийнятна якість може розглядатися як кількісно виражений компроміс між замовником і виконавцем щодо характеристик продукту, створюваного виконавцем в інтересах вирішення завдань замовника з урахуванням інших обмежень проекту (зокрема, вартістю).

Якість ПЗ є відносним поняттям, яке має сенс тільки під час врахування реальних умов його застосування, тому вимоги, що пред'являються до якості, ставляться відповідно до умов і конкретної області їх застосування.

Якість ПЗ характеризується трьома головними аспектами:

- якість програмного продукту;
- якість процесів ЖЦ;
- якість супроводу або впровадження (Рис. 5.1).

Аспект, пов'язаний з процесами ЖЦ, характеризується ступенем формалізації, достовірністю і якістю самих процесів ведення розробки ПЗ, а також верифікацією та валідацією отриманих проміжних результатів на процесах ЖЦ, що зменшують кількість помилок у ПЗ і тим самим сприяють підвищення якості готового продукту.

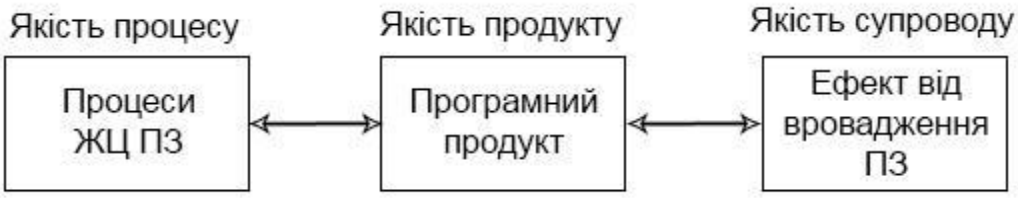

Рисунок 5.1 - Основні аспекти якості ПЗ

Якість продукту цілком і повністю визначається процесами ЖЦ. Ефект від впровадження отриманого програмного продукту в значній мірі залежить від якості супроводу і знань обслуговуючого персоналу.

5.2 Кодекс етики програмної інженерії

Своєю появою інженерія ПЗ зобов'язана діяльності потужних професійних об'єднань — The Assotiation for Computer Machmery (ACM) і Institute of Electrical and El ectrorncs Engmeers Computer Sodety (IEEE Computer Sotiety). Спільними зусиллями цих двох об'єднань розроблений кодекс етики програмної інженерії. Він фокусує мораль, правила і норми поведінки професіоналів, їх зобов'язання і відповідальність по відношенню до суспільства і один до іншого.

Етика інженерної діяльності в програмуванні відрізняється від етики прикладних досліджень, де дослідники працюють у прикладній науці, спрямовують свої зусилля на реалізацію можливостей і відповідають певним вимогам. Інженерна діяльність у програмну інженерію, ще включає:

- технічні вміння;
- відповідальність перед користувачами;
- вміння керувати і приводити до вдалого завершення великі програмні проекти.

Інакше кажучи, інженери повинні добре знати, що є ризик зробити реалізацію швидко або високоякісно. Кодекс складається з преамбули і восьми принципів, яких повинні дотримуватися професіонали з інженерії ПЗ.

У преамбулі професіонал визначається як фахівець, який приймає безпосередньо участь у діяльності з аналізу, специфікації, проектування, розробки, сертифікації, супроводження та тестування програмних систем. Сформульовані принципи забезпечують здоров'я, безпеку та добробут суспільства як головний фактор, який необхідно брати до уваги під час прийняття рішень у професійній діяльності інженерії ПЗ.

У кодексі задекларовано вісім принципів, які стосуються відповідно:

1) узгодження професійної діяльності з інтересами суспільства;
2) взаємовідносини між клієнтом, роботодавцем і виконавцем розробки;
3) досягнення відповідності якості продукту кращим професійним стандартам;
4) дотримання чесності і незалежності при професійних оцінках;
5) дотримання етичних норм у менеджменті і в супроводі розробок;
6) підтримка становлення професії у відповідності з кодексом етики;
7) дотримання етичних норм у взаєминах між колегами;
8) удосконалення кваліфікації розробників.

Кожен з наведених принципів має детальні пояснення щодо різних спектрів його дотримання.

5.3 Значення і вартість якості

Поняття "якість", насправді, не настільки очевидно і просто, як це може здатися на перший погляд. Для будь-якого інженерного продукту існує безліч інтерпретацій якості, залежно від конкретної «системи координат». Безліч цих точок зору необхідно обговорити і визначити на етапі вироблення вимог до програмного продукту. Характеристики якості можуть вимагатися в тому або іншому ступені, можуть бути відсутніми або можуть ставити певні вимоги, все це може бути результатом певного компромісу. Вартість якості (cost of quality) може бути диференційована на:

- вартість попередження <дефектів> (prevention cost);
- вартість оцінки (appraisal cost);
- вартість внутрішніх збоїв (internal failure cost);

- вартість зовнішніх збоїв (external failure cost);

Рушійною силою програмних проектів є бажання створити програмне забезпечення, що володіє певною цінністю. Цінність програмного забезпечення може виражатися у формі вартості, а може і ні. Замовник, звичайно, має своє уявлення про максимальні вартісні вкладення, повернення яких очікується в разі досягнення основних цілей створення програмного забезпечення. Замовник може, також, мати певні очікування щодо якості ПЗ.

Іноді, замовники не замислюються про питання якості і пов'язаної з ними вартості. Чи є характеристики якості чисто декоративними або, все ж таки, це невід'ємна частина програмного забезпечення? Відповідь, ймовірно, знаходиться десь посередині, як майже завжди буває в таких випадках, і є предметом обговорення ступеня залучення замовника в процес прийняття рішень і повного розуміння замовником вартості та вигоди, пов'язаної з досягненням того чи іншого рівня якості. В ідеальному випадку, більшість такого роду рішень повинно прийматися процесі роботи з вимогами, однак ці питання можуть підніматися протягом усього життєвого циклу програмного забезпечення. Не існує якихось "стандартних" правил того, як саме необхідно приймати такі рішення. Однак, інженери повинні бути здатні представити різні альтернативи (у досягненні різного рівня якості) і їх вартість.

5.4 Контрольні питання до теми:

1. Описати область знань «якість програмного забезпечення», виділити її основні проблеми.
2. Надати визначення поняттю якості та її атрибутам.
3. Описати основні принципи кодексу етики в інженерії програмного забезпечення.
4. Парадигма «вбудови» якості в інженерії програмного забезпечення (QFD).
5. Сертифікація програмного забезпечення в Україні.
6. Значення і вартість якості.

Тема №6. Застосування модульного тестування. Процес та виклики

6.1 Що таке модульне тестування?

Модульне тестування — це метод ізоляції та тестування окремих одиниць коду для визначення ефективності кожного компонента. Замість тестування програмного забезпечення цей метод розбиває його на менші частини, щоб переконатися в коректності окремих компонентів.

6.2 Навіщо нам потрібні модульні тести?

Оскільки модульні тести зазвичай проводяться на етапі розробки, вони дозволяють командам виявити та виправити проблеми перед випуском програмного забезпечення. Модульні тести попереджають розробників про потенційні помилки або прогалини, які можуть викликати проблеми в майбутньому, і покращують загальну якість і продуктивність.

Модульне тестування залишається дещо суперечливою темою в галузі. Групи контролю якості чемпіон тестування програмного забезпечення у той час як кодери застерігають від надмірного використання, і лише деякі команди приходять до консенсусу. Розуміння більшої картини може допомогти вам пробратися через аргументи та прийняти найкраще рішення для вашого бізнесу.

6.3 Що слід тестувати в модульному тестуванні (а що не слід)?

Модульне тестування — це інструмент, який має час і місце, як і будь-який інший інструмент у вашому арсеналі для підвищення ефективності програмного забезпечення та економічності. Він може багато чого досягти, але може бути не найкращим вибором у кожній ситуації.

Є явні переваги використання модульного тестування в таких сценаріях:

Пройдіть тест-драйв, щоб переконатися, що код працює, перш ніж розгортати його.

Перевірте роботу, щоб перевірити функцію коду та виявити потенційні дефекти.

Задокументуйте процес, щоб підтримувати найкращі практики та відстежувати прогрес.

Може виникнути спокуса розширити використання модульного тестування, але його обмеження також можуть створити проблеми, якщо ви використовуєте його в певних ситуаціях. Наприклад, виконання модульного тестування на компонентах, які працюють із системами сторонніх виробників, може не дати послідовних або надійних результатів. Завдання надто складне, щоб розбити його на більш дрібні компоненти, не втративши нічого.

Модульне тестування також створює проблеми зі складними системами, як-от ШІ та Роботизована автоматизація процесів (RPA).. Хоча ви можете виконувати модульні тести в цих сценаріях, це масштабне завдання, і доступні кращі інструменти.

6.4 Переваги модульного тестування

Важливо відзначити, що модульне тестування зазвичай відбувається на початку процесу розробки як проактивний захід або перед впровадженням нового коду в існуючу систему. Включення модульного тестування програмного забезпечення у ваш існуючий план тестування може принести користь вашому проекту очікуваними та неочікуваними способами.

1. Економія часу та грошей

Мабуть, найціннішою причиною для включення модульного тестування є вплив на часові рамки випуску та кінцевий результат. Хоча це додає додаткові кроки до процесу розробки, модульне тестування не є таким трудомістким або дорогим, як пошук незначних дефектів у вашому готовому продукті через кілька місяців після доставки.

Оскільки модульне тестування шукає дефекти та потенційні проблеми, перевіряючи код на різні умови, це дозволяє швидше та легше виправляти код.

Налаштування коду в міру розвитку проекту є ефективним і більш ефективним використанням людських і фінансових ресурсів.

Пошук і виявлення потенційних дефектів за допомогою модульного тестування на ранній стадії процесу є одним із найбільш практичних кроків, які ви можете зробити. Це дешевше та простіше вирішити існуючі та потенційні проблеми до того, як доставити продукт вашому клієнту.

2. Покращує якість

Модульне тестування також покращує якість продукту, вирішуючи проблеми до того, як вони створять проблеми. Ви можете постачати продукт вищої якості, знаючи, що він пройшов низку випробувань аж до найменшого рівня.

Це також дозволяє командам перевіряти продуктивність, наголошуючи на програмному забезпеченні протягом усього процесу розробки, щоб переконатися в його готовності. Ваша команда може експериментувати з різними сценаріями, включаючи екстремальні умови, щоб визначити, як відреагує програмне забезпечення.

Успішне тестування дозволяє командам усунути будь-які недоліки та створити надійніший і складніший продукт.

3. Надає документацію

Модульне тестування передбачає запис, який документує весь процес і функції кожного компонента. Він надає схему та огляд усієї системи та демонструє можливості та ідеальні варіанти використання програмного забезпечення, одночасно пропонуючи уявлення про неналежне використання.

4. Підвищує загальну ефективність

Ізолюючи різні частини програмного забезпечення, модульне тестування може перевірити ефективність окремих компонентів. Якщо менші компоненти добре працюють самі по собі, це робить всю систему надійнішою.

Крім того, тестування ізольованих компонентів дозволяє розробникам виявляти та виправляти проблеми, перш ніж вони зможуть вплинути на інші компоненти.

6.5 Проблеми та обмеження модульного тестування

Жодна система не є ідеальною, і методи модульного тестування не є винятком. Фахівці галузі не погоджуються щодо важливості модульного тестування, оскільки з цим процесом пов’язані деякі помітні обмеження.

1. Потрібно більше коду

Хоча модульне тестування може врятувати вас у довгостроковій перспективі, воно потребує значного кодування для перевірки компонентів. Таким чином, одна з найкращих практик модульного тестування полягає в тому, щоб мати принаймні три модульних тести, щоб гарантувати, що у вас завжди буде тай-брейк.

2. Не звертається до кожної ситуації

Модульне тестування не є ідеальним для всіх можливостей, особливо для тестування інтерфейсу користувача. Він також не може виявити кожну помилку, оскільки неможливо передбачити кожну потенційну ситуацію.

3. Утруднює зміни

Зміцнення окремих компонентів створює потужнішу програму. Що станеться, коли вам потрібно змінити або оновити цю програму? Більш складно змінити систему, яка настільки ізольована від помилок, не порушуючи загальну функцію.

6.6 Типи модульного тестування

Модульне тестування зазвичай виконується автоматизованим інструментом модульного тестування, але також можна застосувати ручний підхід. Обидва методи мають переваги та недоліки, які слід враховувати, хоча автоматизоване модульне тестування є найпопулярнішим і найважливішим кроком для компаній, які використовують гіперавтоматизація.

1. Ручне модульне тестування

Ручне модульне тестування покладається на тестувальників, які можуть зрозуміти складні функції та функції. Оскільки люди можуть мислити нестандартно, вони можуть виявляти проблеми за межами коду та моделювати досвід користувача.

Недоліком є те, що ручне модульне тестування коштує дорого, оскільки вам потрібно платити кваліфікованим програмістам. Це забирає багато часу та складно,

тому що команди повинні ізолювати окремі компоненти та запускати кілька тестів на кожному з них.

2. Автоматизоване модульне тестування

Автоматизоване модульне тестування використовує програми та код для виконання тестів. Як інші автоматизація тестування програмного забезпечення, модульне тестування програмного забезпечення працює швидше та обмежує вплив на інші компоненти. Крім того, ви можете написати тест один раз і використовувати його кілька разів.

На жаль, для створення необхідного коду та його підтримки потрібен час. Автоматизоване модульне тестування все ще має деякі обмеження, оскільки воно не може виловити кожну помилку.

6.7 Характеристики хорошого модульного тесту

Модульне тестування вимагає тонкого балансу, щоб збільшити переваги та подолати обмеження. Найкраще модульне тестування має чотири характеристики, які створюють цей баланс.

1. Ізольований

Кожен модульний тест повинен бути автономним, тобто він може існувати незалежно від інших факторів. Якщо робота тесту залежить від інших програм або систем, це може змінити результати.

2. Швидко

Розгляньте обсяг коду, який потрібно перевірити, і скільки часу знадобиться для виконання достатньої кількості тестів для отримання задовільних результатів. Хороший модульний тест повинен займати лише мілісекунди для завершення тестування. Крім того, створення модульного тесту не повинно тривати довше, ніж створення компонентів, які ви збираєтеся протестувати.

3. Послідовний

Модульні тести мають щоразу повертати ідентичні результати. Якщо ви не можете повторити тест кілька разів і отримати однакові результати, він ненадійний.

4. Самоперевірка

Ручні та автоматизовані модульні тести повинні мати можливість виявляти результати автоматично без втручання людини. Вашій команді не потрібно переглядати результати, щоб визначити, чи це так чи ні.

Прорізаючи жаргон: модульні тести проти інтеграційних тестів

Тестування програмного забезпечення таке ж складне, як і програми, які воно тестує, а це означає, що різні терміни та типи досягають різних результатів. Розуміння різниці між модульними тестами та інтеграційними тестами є необхідним для визначення найкращого способу реалізації кожного з них.

1. Що таке інтеграційні тести?

Інтеграційне тестування визначає, як різні компоненти працюють разом у програмі. Він визначає будь-які проблеми між компонентами, коли вони об’єднуються для виконання завдань. Деякі проблеми можуть підтримувати програмне забезпечення, але це тестування шукає ті, які погіршують загальну продуктивність.

2. Модульні тести проти інтеграційних тестів

Модульне тестування та інтеграційне тестування є подібними концепціями, які стосуються різних елементів. Замість того, щоб розглядати окремі функції найменшого блоку, інтеграційне тестування дивиться на те, як компоненти працюють разом.

Інтеграційне тестування також шукає дефекти та побічні ефекти на ранніх стадіях процесу та виявляє проблеми, які не очевидні на перший погляд. Однак інтеграційне тестування стосується кількох компонентів, оскільки вони взаємодіють один з одним, а не окремих функцій.

6.8 Методи модульного тестування

Три методи модульного тестування стосуються різних рівнів системи. Ці типи можуть охоплювати як ручне, так і автоматичне тестування.

1. Техніки функціонального модульного тестування

Функціональні методи модульного тестування, відомі як тестування чорної скриньки, стосуються функціональності кожного компонента. Він оцінює валідність інтерфейсу користувача, введення та виведення, встановлюючи межі та еквівалентності.

2. Методи тестування структурних одиниць

Структурні методи або тестування білої скриньки перевіряють компоненти, які відповідають встановленим функціональним вимогам, і відображають їхні шляхи. Наприклад, це може передбачати встановлення серії умов, щоб побачити, яким шляхом слідує код у програмі на основі вхідних даних.

3. Методи модульного тестування на основі помилок

Методи, засновані на помилках, працюють найкраще, якщо оригінальний програміст займається тестуванням, оскільки він знайомий з його роботою. Також відоме як тестування сірого ящика, воно використовує тестові випадки та виконує оцінку ризиків для виявлення дефектів.

6.9 Застосування модульного тестування

Як зазначалося, програми модульного тестування майже нескінченні, але для деяких цілей вони служать краще, ніж для інших.

1. Екстремальне програмування

Екстремальне програмування це одна з ідеологій розробки програмного забезпечення, яка прагне створити програмне забезпечення найвищої якості. Ця методологія значною мірою покладається на фреймворки модульного тестування програмного забезпечення для проведення комплексного тестування. Екстремальні програмісти часто використовують автоматизовані інструменти тестування для покращення загальної якості та оперативності при адаптації до мінливих потреб клієнтів.

Одним із керівних принципів є тестування всього, що потенційно може вийти з ладу, включаючи найменші компоненти. Отже, модульне тестування є потужним інструментом для екстремальних програмістів.

2. Модульне тестування на рівні мови

Деякі мови вроджено сумісні з модульним тестуванням. Наприклад, такі мови, як Python і Apex, безпосередньо підтримують модульне тестування через структуру коду, тобто для включення модульних тестів потрібні обмежені коригування. Інші мови потребують незначних модифікацій і спеціальних фреймворків, як-от модульне тестування PHP.

3. Інфраструктури модульного тестування

Модульне тестування відкриває двері для продуктів сторонніх розробників, які можна встановити для виконання тестів у вашій існуючій системі. багато Інструменти автоматизованого модульного тестування сумісні з кількома мовами, щоб спростити процес тестування та дозволити користувачам перевіряти їхнє раніше розроблене програмне забезпечення.

6.10 Як написати тестовий приклад для модульного тестування

Написання тестових випадків модульного тестування може бути складним залежно від компонента, який ви тестуєте; написання модульного тесту має бути зосереджено на тих же трьох пунктах. Зауважте, що можуть бути невеликі відмінності між ручним і автоматичним тестуванням, але процес, по суті, однаковий.

1. Перевірте правильну відповідь

Почніть з тесту, який перевіряє оптимальну відповідь, щоб переконатися, що він розпізнає, що має статися. Цей крок також встановлює базову лінію.

2. Тест відповіді на неправильний вхід

Встановіть тест, щоб перевірити відповідь на недійсний вхід. Створіть базову лінію для відповіді компонента на недійсні дані.

3. Виконайте кілька дій

Повторно перевіряйте компонент, використовуючи дійсні та недійсні відповіді, щоб визначити, як компонент реагує. Потім відстежте відповіді, щоб знайти будь-які дефекти.

6.11 Як ми виконуємо модульне тестування?

Модульне тестування передбачає написання коду для перевірки певного компонента в програмному забезпеченні. Ручне тестування зазвичай вимагає більше кроків і не є особливо поширеним, тому давайте подивимося на процес за допомогою інструментів автоматизації модульного тестування.

Одним із найпопулярніших інструментів на ринку є ZAPTEST API Studio. За допомогою ZAPTEST користувачі можуть автоматизувати тестування REST; МИЛО; і openAPI з використанням повної параметризації та простих у використанні утиліт кореляції та керування даними. ZAPTEST також надає можливість об'єднати тестування API та інтерфейсу користувача в безпроблемний процес.

1. Визначте розділ коду для перевірки та визначте метод

Розробники можуть написати та додати код до програми, щоб перевірити функцію компонента та видалити тестовий код пізніше. І навпаки, можна виділити компонент і скопіювати його в тестову систему. Останній дозволяє користувачам виявити будь-які непотрібні посилання на інші компоненти під час тесту.

2. Ініціювати тестові випадки

Розробник використовує тестові випадки, розроблені програмістом, щоб перевірити функціональність компонента. Цей процес зазвичай відбувається в системі автоматизованого тестування, яка позначає будь-які дефекти під час тестування та може попередити команду про помилку.

3. Перегляд і переробка

Після завершення тестування команда може переглянути дані, щоб визначити будь-які дефекти чи помилки. Потім команда вносить виправлення та оновлює компонент перед повторним тестуванням.

Команди можуть переглядати тестові приклади стільки разів, скільки потрібно для досягнення бажаних результатів. Можна зупинити модульний тест, тобто компонент або тестовий приклад вийшли з такої серйозної невдачі, що не варто продовжувати.

6.12 Приклади модульних тестів

Існують сотні прикладів модульного тестування, які стосуються різних компонентів і проблем. Ось кілька базових прикладів модульного тестування, які демонструють реальні додатки.

1. Модульне тестування API

Сучасні системи покладаються на різні програми, які спілкуються одна з одною, часто покладаючись на інтерфейси, відомі як API. Наприклад, розробники можуть підвищити ефективність, протестувавши кінцеві точки за допомогою модульного тестування REST API.

2. Автомобільна промисловість

Автомобільна промисловість пропонує широкі можливості для прикладів модульного тестування, тому розгляньте широкі наслідки. Наші транспортні засоби більше, ніж будь-коли, покладаються на код і можуть створювати небезпечні ситуації, якщо є навіть незначний дефект. Інструменти модульного тестування можуть виділити код ще до того, як автомобіль покине завод, щоб визначити, чи він чистий, і зменшити ймовірність несправностей на дорозі.

6.13 Найкращі методи модульного тестування

Незалежно від того, чи хочете ви провести модульне тестування на REST API або визначити, як банківська програма реагує на різні вхідні дані в одному обліковому записі, ці найкращі практики допоможуть утримувати модульне тестування на правильному шляху.

1. Напишіть і дотримуйтесь плану модульного тестування

Одним із найважливіших елементів модульного тестування є дотримання плану, який детально визначає розмір, обсяг і цілі. Визначте обсяг вашого модульного тесту та те, що вам потрібно протестувати, визначте тестові випадки та виберіть відповідні інструменти чи програмне забезпечення.

Просто створити план модульного тестування недостатньо; ваша команда повинна дотримуватися плану від початку до кінця. Пропуск кроків або відхилення від плану може призвести до плутанини та створити непотрібну роботу.

2. Розглянемо мову

Переконайтеся, що ваш код розмовляє тією ж мовою, що й програма або програма, яку ви тестуєте. Модульне тестування PHP відрізняється від модульного тестування C#, хоча загальна структура виглядає схожою.

3. Реінтеграційне та регресійне тестування

Якщо ви скопіювали код і протестували його в системі тестування, а не в програмі, регресійне тестування має вирішальне значення. Переробка будь-якого коду може змінити функціональність програми, тому повторно інтегруйте пристрій, а потім виконайте регресійне тестування, щоб переконатися, що він працює належним чином.

6.14 Хто має бути залучений до модульних тестів?

Хоча багато людей роблять внесок у розробку програмного забезпечення та додатків, не всі мають час, навички чи знання для участі в модульному тестуванні. Тому обмежте команду кількома кваліфікованими особами або командами.

1. Розробники програмного забезпечення проводять модульне тестування

Розробники несуть основний тягар відповідальності за модульне тестування, тому що вони знають свій код і те, як він має працювати. Розробники пишуть тестові випадки, реалізують тест і зазвичай мають найкраще уявлення про те, яке програмне забезпечення для модульного тестування використовувати.

2. Група забезпечення якості

Команда контролю якості знає, як має працювати програмне забезпечення та як виявляти дефекти. Вони дивляться на програмне забезпечення з іншої точки зору та забезпечують його належне функціонування в рамках більшої системи.

6.15 Контрольний список модульного тестування

Цей контрольний список модульного тестування є рекомендацією, яка допоможе вашій команді йти на шляху досягнення цілей.

1. Виберіть правильні інструменти модульного тестування

Вибір правильних інструментів автоматизації модульного тестування є важливим. Переконайтеся, що програмне забезпечення для модульного тестування сумісне з мовою вашої програми та може досягати цілей вашої команди.

2. Налаштуйтеся на успіх

Створіть докладні назви для тестового проекту, щоб майбутні команди знали, що було зроблено, і могли легко ідентифікувати тест. Визначте код, який ви збираєтеся протестувати, і переконайтеся, що він повністю незалежний.

3. Тестовий код Індивідуально

Тестуйте лише один компонент за раз, щоб залишатися послідовним і ефективним, а також уникати збігів або непорозумінь між членами команди.

4. Відтворити дефекти

Якщо ви виявите дефект, перевірте ще раз, щоб переконатися, що та сама дія повертає дефект знову. Виправте дефект, якщо його можна відтворити.

6.16 Висновок

Модульне тестування — це спосіб підвищити ефективність програмного забезпечення та додатків шляхом перевірки правильності найменших компонентів. Це ще одна можливість удосконалити існуюче програмне забезпечення та підвищити ефективність.

Для тих, хто цікавиться програмною автоматизацією та робото технічними засобами автоматизації процесів модульне тестування відіграє допоміжну роль на шляху до гіперавтоматизації. Оскільки він розбиває програми на найдрібніші компоненти, він може ідентифікувати раніше непомічені дефекти та запобігати майбутнім проблемам, перш ніж вони переростуть у проблеми та затримають виробництво.

Як і інші інструменти автоматизації, важливо розумно використовувати модульне тестування та дотримуватися найкращих практик галузі.

Поширені запитання

Модульне тестування — це потужна можливість для компаній покращити програмне забезпечення та програми.

6.17 Що таке модульне тестування в C#?

Модульне тестування в C# передбачає виділення сегментів коду, які представляють найменші компоненти, і перевірку їх правильності за допомогою засобів автоматизації модульного тестування.

6.18 Що таке модульне тестування в Java?

Модульне тестування в Java вимагає інфраструктури для перевірки поведінки бітів коду перед його використанням у виробництві.

6.19 Що таке модульне тестування в розробці програмного забезпечення?

Модульне тестування в розробці програмного забезпечення виділяє найменший тестований компонент у програмі та перевіряє його валідність і продуктивність.

Тема №7. Інтеграційне тестування

Тема №8. Теми до семінару

Тема №9. Екзаменаційні питання

1. Види тестування ПЗ в залежності від його цілей.
2. Функціональне тестування, його визначення та сфери застосування.
3. Найпоширеніші види функціональних тестів.
4. Тест-дизайн. Техніки тест-дизайну.
5. Тест-кейси.
6. Шаблон та атрибути тест-кейсів.
7. Баг-трекінгові системи.
8. Баг-репорт.
9. Застосування програми ListBoxer у тестуванні.
10. Нефункціональне тестування, його визначення та сфери застосування.
11. Тестування зручності використання.
12. Способи конфігураційного тестування.
13. Принципи тестування на відмову та відновлення.
14. Методи тестування властивостей якості ПЗ.
15. Огляд піраміди тестування Майка Кона.
16. Нефункціональні види тестування ПЗ.
17. Цілі нефункціонального тестування.
18. Різниця функціонального та нефункціонального тестування.
19. Алгоритм налагоджування програмного продукту за допомогою Visual Studio.
20. Сучасні види дебаггерів, їх функціонал.
21. Степпінг для тестувань: «Крок з заходом», «Крок з обходом», «Крок з виходом», «Виконати до поточної позиції». Точки зупинки.
22. Поняття налагоджування ПЗ. Етапи процесу налагоджування ПЗ.
23. Типи помилок, що вимагають налагоджування.
24. Основні стратегії налагоджування ПЗ.
25. Принцип модульного тестуванні у Visual Studio.
26. Програмні інструменти налагоджування ПЗ.
27. Основні класи деббагерів.
28. Системи контролю версій програмного продукту.

29. Локальні системи контролю версій, переваги та недоліки.
30. Централізовані системи контролю версій, переваги та недоліки.
31. Децентралізовані системи контролю версій, переваги та недоліки.
32. Принцип системи контролю версій Tortoise SVN.
33. Приклади сучасних систем контролю версій ПЗ.
34. Основні різновиди архітектур систем контролю версій ПЗ.
35. Призначення систем контролю версій ПЗ.
36. Контроль та управління версіями ПЗ.
37. Основні можливості сучасних систем контролю версій ПЗ.
38. Характеристики, недоліки і переваги систем контролю версій ПЗ.
39. Алгоритм модульного тестування програмного продукту.
40. Заглушки у модульному тестуванні.
41. Написання юніт-тестів у модульному тестуванні.
42. Основні стратегії налагоджування ПЗ.
43. Поняття модульного тестування.
44. Ключові поняття юніт-тестування.
45. Тестове покриття програми.
46. Технологія розроблення ПЗ через тестування.
47. Тестовий сценарій. Фікстури.
48. Моки у модульному тестуванні.
49. Інструменти та бібліотеки модульного тестування.
50. Профілювання ПЗ. Способи та методи.
51. Поняття та аспекти визначення якості ПЗ. Життєвий цикл ПЗ.
52. Інтеграційне тестування. Переваги та недоліки.
53. Різниця інтеграційного та модульного тестування.
54. Переваги і недоліки модульного тестування.
55. Дефект, тестування, відладка, фаза.
56. Тестування безпеки.
57. Тестування установки.
58. Моделі розробки ПЗ.
59. Процеси та виклики у інтеграційному тестуванні.

60. Види тестування ПЗ залежно від цілей.