**Лабораторна робота №9-2**. **Процес тестування програмного забезпечення. Підготовка тестів для перевірки ПЗ (формування формальних специфікацій функціональних елементів).**

**Мета:** Навчитися формувати формальні специфікації функціональних елементів

**Методичні рекомендації до виконання лабораторної роботи.**

1. Опрацювати теоретичні відомості.
2. Відповідно до проведеного в ЛР №08-3 розподілу функцій власного проекту на основні та другорядні із визначенням критерію віднесення та області можливої нестійкості продукту сформувати формальну специфікації тесту для функціональних елементів, заповнивши таблицю .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Функція** | | | | | Альтернативи | Очікуваний результат  тестування |
| Стійкість | О/Д | Назва | Параметр | Категорія |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  |  |  |  |  |

В графі "О/Д" записуємо "О" для основних, "Д" для другорядних.

В графі " Стійкість " визначаємо область можливої нестійкості функцій, або встановлюємо позначку "стійка".

Графи 4-6 заповнюються відповідно до прикладу, наданому в Практичній роботі №2.

Графа 7 містить опис очікуваного результату проходження тесту.

Повинно бути **не менше** одної основної стійкої функції, одної основної нестійкої функції, одної другорядної стійкої функції, одної другорядної нестійкої функції.

1. Робота повинна бути виконана згідно критеріїв оформлення документації та повинна містити

* Назва лабораторної роботи.
* Прізвище, група
* Назва проекту.
* Результати роботи оформлюються у вигляді таблиці:

По закінченню практичну роботу потрібно здати на перевірку викладачеві, надіславши електронною поштою на адресу [**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com) . Якщо викладач знаходить помилки чи неточності, він може повернути роботу на доопрацювання.

Файл з роботою повинен мати назву в такому форматі:

**QA<Номер групи><Номер лекції / практичної / лабораторної>[-<Номер завдання>][літера позначення типу роботи L – лекція, P –практична, R – лабораторна]<Прізвище англійською>**.. Наприклад, **QA4101Р**buts.doc.

Не копіюйте фрагментів з різних інформаційних джерел, подумайте і викладіть свою точку зору. При наявності робіт-"близнюків" відповідь буде зараховуватися першому за часом надсилання.

Тему в заголовку листа записати

**QA <Номер групи>-ЛР<Номер лабораторної>-<Прізвище >**

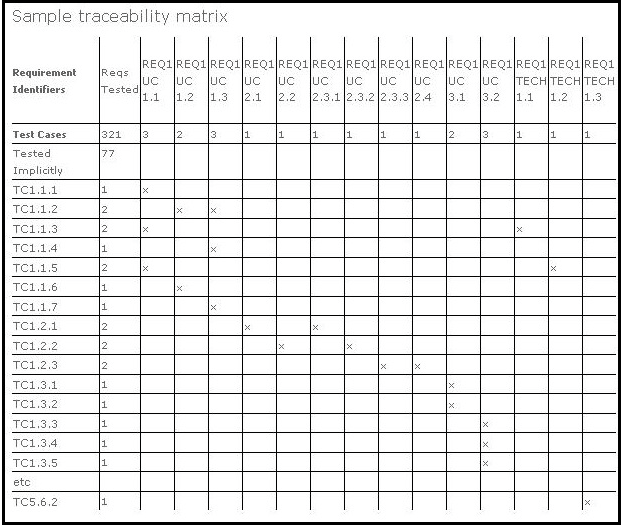
**Строк виконання цієї роботи ІПЗ-41 – 31.03.2024**

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

**Traceability matrix**

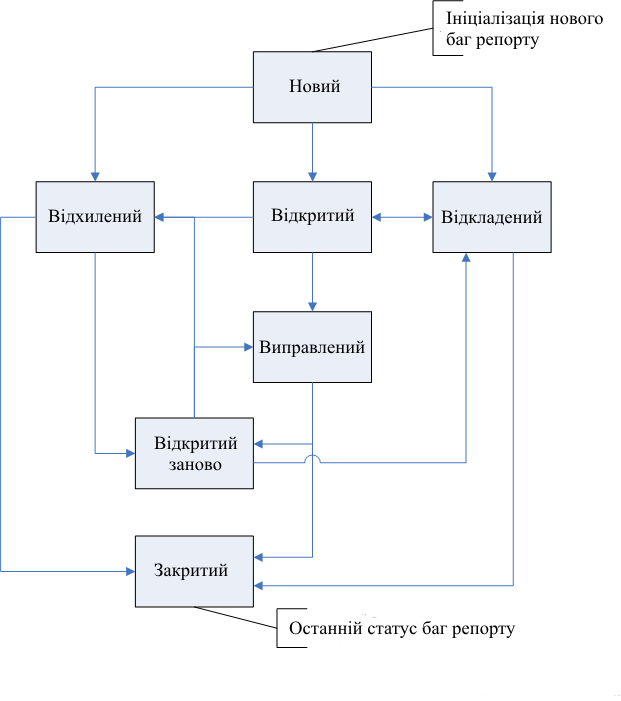
Матриця відповідності вимог – це двовимірна таблиця, яка містить відповідність функціональних вимог (functional requirements) продукту і підготовлених тестових сценаріїв (test cases). У заголовках колонок таблиці розташовані вимоги, а в заголовках рядків – тестові сценарії. На перетині – позначка, що означає, що вимогу поточної колонки покрито тестовим сценарієм поточного рядка.

Матриця відповідності вимог (МВВ) використовується QA-інженерами для валідації покриття продукту тестами. МВВ є невід’ємною частиною тест-плану.

  
https://en.wikipedia.org/wiki/Traceability\_matrix

**Життєвий цикл бага**

Блок-схема, що показує основні статуси і можливі переходи від статусу до статусу в процесі його існування:

  
Опис даної схеми.

Припустимо ви знайшли баг і зареєстрували його в баг трекінг системі. Відповідно до нашої блок-схемі він отримає статус “Новий”. Тестувальник, відповідальний за валідацію нових баг репортів, або координатор проекту (в залежності від розподілу ролей у команді) може перевести його в один з наступних статусів:

· “Відхилений”, якщо даний баг невалідний або повторний, або ж його просто не змогли відтворити

· “Відкладений”, якщо даний баг не потрібно виправляти в даній ітерації

· “Відкритий”, якщо виправити бага необхідно

Розглянемо тепер по порядку кожен з варіантів.

1. Відхилений. У цьому випадку ви можете або посперечатися про долю вашого багрепорта, змінивши статус на “Відкритий заново” або закрити його – статус “Закритий”

2. Відкладений. Баг репорт в статусі “Відкладений” можна перевести в статус “Відкритий”, коли буде потрібно виправлення або в статус “Закритий”, якщо вже не буде потрібно.

3. Відкритий. Саме в такому стані розробник отримує баг репорт для виправлення. Він може відхилити (подальші дії дивіться в пункті 1) або виправити баг. Баг репорт в статусі “Виправлений” перекладається на тестувальника для перевірки. У разі якщо проблема все ще відтворюється, виставляється статус “Відкритий заново” і баг репорт направляється назад на доопрацювання до розробника. Якщо ж виправлення було успішним, то баг репорт переводиться в статус “Закритий”.

\* \* \*

Необхідно зазначити, що дана схема сильно спрощена. Для більшої наочності і, можливо, зручності роботи на проекті, ви можете додати додаткові статуси і переходи, тим більше, що сучасні баг трекінгові системи дозволяють це робити. Правда майте на увазі, що заплутані схеми переходів і зайві статуси можуть значно ускладнити життя.

Примітка 1: в деяких системах баг трекінгу створений баг репорт відразу отримує статус “Відкритий” без додаткової валідації

Примітка 2: багато баг трекінгових систем дозволяють перевідкривати закриті баги, проте особисто я проти такої практики, тому й не описував подібний перехід у вище поданому життєвому циклі

Примітка 3: Розглянутий вище життєвий цикл заснований на тому, що в команді є хтось, відповідальний за призначення баг репортів. У випадку, якщо такої ролі на проекті немає, то баги призначаються розробниками самостійно, і тоді для уникнення плутанини, є сенс ввести ще один проміжний статус “У розробці” (In progress), що показує, що даний баг репорт вже призначений і знаходиться на стадії виправлення.

**Принципи тестування**

Тестування програмного забезпечення – креативна та інтелектуальна робота. Розробка правильних та ефективних тестів – досить непросте заняття. Принципи тестування, приведені нижче, були розроблені в останні 40 років та є загальною інструкцією для тестування в цілому.

1. **Тестування показує наявність дефектів**

Тестування може показати наявність дефектів в програмі, але не довести їх відсутність. Тим не менш, важливо складати тест-кейси, які будуть знаходити якомога більше багів. Таким чином, при належому тестовому покритті, тестування дозволяє знизити вірогідність наявності дефектів в програмному забезпеченні. В той же час, навіть якщо дефекти не були знайдені в процесі тестування, не можна стверджувати, що їх немає.

1. **Вичерпне тестування неможливе**

Неможливо провести вичерпне тестування, яке би покривало всі комбінації користувацького вводу та станів системи, за виключенням найбільш примітивних випадків. Замість цього необхідно використовувати аналіз ризиків та розташування пріоритетів, що дозволить більш ефективно розподілити зусилля по забезпеченню якості ПЗ.

1. **Раннє тестування**

Тестування повинне починатися якомога раніше в життєвому циклі розробки програмного забезпечення, і його зусилля повинні бути сконцентровані на визначених цілях.

1. **Скупчення дефектів**

Різні модулі системи можуть містити різну кількість дефектів – тобто, щільність скупчення дефектів в різних елементах програми може відрізнятися. Зусилля по тестуванню повинні розподілятися пропорційно фактичної щільності дефектів. В основному, більшу частину дефектів знаходять в обмеженій кількості модулів. Це прояв принципу Парето: 80% проблем містяться в 20% модулів.

1. **Парадокс пестициду**

Проганяючи одні й ті ж тести знову та знову, Ви зіткнетеся з тим, що вони знаходять все менше нових помилок. Оскільки система еволюціонує, багато із раніше знайдених дефектів виправляють і старі тест-кейси більше не спрацьовують.

Щоб подолати цей парадокс, необхідно періодично вносити зміни в набори тестів, що використовуються, рецензувати та коригувати їх з тим, щоб вони відповідали новому стану системи та дозволяли знаходити якомога більшу кількість дефектів.

1. **Тестування залежить від контенту**

Вибір методології, техніки та типу тестування буде напряму залежати від природи самої програми. Наприклад, програмне забезпечення для медичних цілей потребує більш строгої та ретельної перевірки, ніж, скажімо, комп’ютерна гра. З тих же міркувань, сайт із великою відвідуваністю повинен пройти через серйозне тестування продуктивності, щоб показати можливості роботи в умовах великого навантаження.

1. **Міф про відсутність помилок**

Той факт, що тестування не виявило дефектів, ще не значить, що програма готова до релізу. Знаходження та виправлення дефектів будуть не важливі, якщо система виявиться незручною у використанні, та не буде задовольняти очікуванням та вимогам користувача.

*І ще декілька важливих принципів:*

– тестування повинне проводитися незалежними спеціалістами;

– залучайте кращих професіоналів;

– тестуйте як позитивні, так і негативні сценарії;

– не допускайте змін в програмі в процесі тестування;

– вказуйте очікуваний результат виконання тестів.

## **Таблиці рішень та їх застосування в тестуванні**

Таблиці рішень використовуються для дослідження взаємодії між комбінаціями умов. Вони надають чіткий метод перевірки всіх відповідних комбінацій, щоб гарантувати, що продукт обробляє всі можливі умови, взаємозв'язки і обмеження. Таблиці рішень є ефективним методом тестування програмного забезпечення, що використовується для аналізу реакції системи на різні вхідні дані.

**Тестування за допомогою таблиць прийняття рішень** – це одна з технік тест-дизайну методом чорного ящика, яка відноситься до динамічного аналізу. Вона також відома як таблиця причинно-наслідкових зв'язків. А схематична демонстрація логіки відома як причинно-наслідковий графік. Це візуальне уявлення використовується для отримання таблиці рішень.

Інші методи досліджень, наприклад, перевірка на еквівалентність або аналіз граничних значень, часто застосовуються тільки для конкретних вхідних даних. Техніка таблиці рішень використовується у тому випадку, коли для різних вихідних використовується комбінація вхідних даних. Основною метою є перевірка бізнес-логіки і тестового покриття.

**Таблиця рішень, як правило, має 4 складові:**

1. Умови – список можливих умов.
2. Варіанти виконання дій – це комбінація зі списку виконаних або невиконаних умов.
3. Дії – це список всіляких дій.
4. Необхідність дій – вказівка ​​потрібно чи не потрібно виконувати відповідну дію для кожної з комбінацій умов.

**Приклад застосування таблиць рішень.**

Умова дуже проста: якщо користувач вводить правильне ім'я користувача і пароль, він буде авторизований і перенаправлений на домашню сторінку. Якщо будь-які з даних, що вводяться є неправильними, то з'явиться повідомлення про помилку.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Умова | 1 | 2 | 3 | 4 |
| Ім'я користувача **(T/F)** | **F** | **T** | **F** | **T** |
| Пароль **(T/F)** | **F** | **F** | **T** | **T** |
| Результат **(E/H)** | **E** | **E** | **E** | **H** |

Умовні позначення:

**T** – Правильне ім’я користувача/пароль.

**F** – Неправильне ім’я користувача/пароль

**E** –Відображається повідомлення про помилку.

**H** – Користувач авторизований/Відображається домашня сторінка.

Розглянемо окремо кожен із стовпчиків:

**Стовпчик 1.** Ім’я користувача і пароль були неправильні. Відображається повідомлення про помилку.

**Стовпчик 2.** Ім’я користувача було правильним, але пароль був неправильним. Відображається повідомлення про помилку.

**Стовпчик 3.** Неправильне ім’я користувача, але правильний пароль. Відображається повідомлення про помилку.

**Стовпчик 4.** Ім’я користувача і пароль були правильними, авторизація пройшла успішно, відображається домашня сторінка.

Для того щоб конвертувати це у тестовий приклад, можна створити декілька тест-кейсів.

**Тест-кейс 1.** Ввести валідні ім’я користувача та пароль, натиснути кнопку «Увійти».

*Очікуваний результат*: користувач авторизований і перенаправлений на домашню сторінку.

**Тест-кейс 2.** Ввести валідне ім'я користувача та невалідний пароль, натиснути на кнопку входу.

*Очікуваний результат*: відображається повідомлення про помилку.

**Тест-кейс 3.** Ввести невалідне ім’я та валідний пароль, натиснути кнопку «Увійти».

*Очікуваний результат*: відображається повідомлення про помилку.

**Тест-кейс 4.** Ввести невалідне ім’я користувача та пароль, натиснути на кнопку входу.

*Очікуваний результат*: відображається повідомлення про помилку.

Необхідно звернути увагу на той факт, що всі кейси, крім першого, перевіряють одне і те ж правило. Таким чином, була створена найпростіша таблиця рішень. За необхідності її можна розширити.

Ще один приклад застосування рішень на прикладі працездатності принтера

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Умови | Принтер не друкує | Y | Y | Y | Y | N | N | N | N |
| Мерехтить червоний індикатор | Y | Y | N | N | Y | Y | N | N |
| Принтер не розпізнається | Y | N | Y | N | Y | N | Y | N |
| Дії | Перевірте дріт живлення |  |  | X |  |  |  |  |  |
| Перевірте дріт принтера | X |  | X |  |  |  |  |  |
| Переконайтесь що ПЗ принтера встановлено | X |  | X |  | X |  | X |  |
| Перевірити/замінити чорнила | X | X |  |  | X | X |  |  |
| Перевірте, чи не застряг папір в лотку |  | X |  | X |  |  |  |  |

Умовні позначення: Y – так; N – ні.

Цей приклад показує простоту, з якою таблиця рішень демонструє можливі комбінації умов і дій. До того ж її легко модифікувати в разі зміни вихідних даних (наприклад, при включенні індикатора іншого кольору в принтері).

**Переваги використання таблиці рішень під час тестування програмного забезпечення:**

1. Навіть найскладніша бізнес-логіка, використовуючи цей метод, може бути легко перетворена в тестові сценарії і випадки.
2. Ітеративність роботи. Таблиця, створена на першому етапі, використовується в якості вхідної таблиці для всіх наступних. Ітерація виконується тільки в тому випадку, якщо вихідні дані не є задовільними.
3. Проста і зрозуміла техніка. Кожен може використовувати цей метод для розробки тестових сценаріїв і випадків.
4. Забезпечення повного охоплення тестових випадків, що істотно допомагає скоротити обсяг роботи.
5. Гарантія розгляду всіх можливих комбінацій умов і значень.
6. Надання більш компактної документації.
7. Легка зміна даних.

**Недоліки таблиць рішень під час тестування програмного забезпечення.**

1. Труднощі в масштабуванні. Потрібно «розділяти» великі таблиці на більш дрібні, щоб запобігти надмірності.
2. Збільшення кількості вхідних даних робить таблицю складнішою.

Таблиці рішень є хорошим способом опису вимог, коли існує кілька бізнес-правил, що взаємодіють один з одним. Використовуючи цей метод, фахівцю стає простіше написати вимоги, які включають в себе всі доступні умови. Застосування цієї техніки допомагає краще проаналізувати тестований продукт, систематизувати всі знання по ньому. В результаті стає легше писати повні тестові приклади, охопити всі можливі комбінації.

**Тестові дані**

Усяке ПЗ обробляє певні дані, які можна визначити, як відомості про деяку подію, факт тощо, що представлені у формальному вигляді і підлягають подальшому аналізу. Аналіз і перетворення даних і являють собою основні завдання ПЗ (важливо пам’ятати різницю між даними та інформацією, яку можна отримати на підставі даних). Таким чином, для проведення тестування ПЗ необхідні дані, які б симулювали реальний потік, оброблюваний програмою.

Існує два типи тестових даних: *реальні та синтезовані*. Реальні дані можна отримати двома способами: перший полягає в тому, щоб тестувати ПЗ в реальних умовах (що практично неможливо); другий – отримувати реальні дані у кінцевого користувача (що можливо досить рідко) або з існуючої системи (у цьому випадку можуть виникнути проблеми з конфіденційністю). Тому, найчастіше для тестування використовують синтезовані дані.

Основною умовою синтезу даних є їх правдоподібність. Синтезувати дані можна на підставі зразка (невеликої кількості реальних даних), статистичної інформації про те, які дані є типовими для конкретного тестового випадку, а також іншого набору умов, що обмежують спектр допустимих варіацій вхідних даних. На базі наявної інформації будується генератор тестових даних, який формує необхідний для тестування потік.

**Тестова ситуація**

Тестова ситуація – це базисне поняття тестування. *Під тестовою ситуацією розуміється деякий певний стан системи, що тестується, якому відповідає строго певний набір її параметрів.* Тестову ситуацію можна визначити також як сукупність зовнішніх і внутрішніх факторів, що дозволяють розрізнити умови виконання програми. Якщо один з параметрів змінюється – зміниться і тестова ситуація. Виходячи з цього, повний набір всіх можливих тестових ситуацій визначає тестове покриття.

**Типи тестових прикладів**

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

**Допустимі дані**

Найчастіше дефекти в програмних системах проявляються при обробці нестандартних даних, що не передбачені вимогами – при введенні невірних символів, порожніх рядків, дуже великій швидкості введення інформації. Перед пошуком таких дефектів необхідно упевнитися в тому, що програма коректно обробляє вірні дані, передбачені специфікацією, тобто перевірити роботу основних алгоритмів.

Приклад 1. Для функції обчислення контрольної суми допустимими вхідними даними буде довільний запис, що містить дані у всіх полях, крім поля контрольної суми CRC. Сценарієм буде виклик функції розрахунку Set\_CRC, яка обчислює контрольну суму передану через структуру з усіма потрібними для розрахунку даними, а очікуваним вихідним значенням – коректне значення поля CRC, що розраховане за алгоритмом.

Зазвичай для перевірки допустимих даних достатньо одного тестового прикладу. Але функціональні вимоги можуть визначати різні групи допустимих даних, які можуть об’єднуватися в класи еквівалентності. У цьому випадку необхідно визначати як мінімум один тестовий приклад для одного класу еквівалентності.

**Класи еквівалентності (Equivalence class):**

Підхід полягає в наступному: вхідні / вихідні дані розбиваються на класи еквівалентності, за принципом, що програма веде себе однаково з кожним представником окремого класу. Таким чином, немає необхідності тестувати всі можливі вхідні дані, необхідно перевірити по окремо взятому представнику класу.

Клас еквівалентності – це набір значень змінної, який вважається еквівалентним.

Тестові сценарії еквівалентні, якщо:

* Вони тестують одне і те ж;
* Якщо один з них знаходить помилку, то й інші виявлять її;
* Якщо один з них не знаходить помилку, то й інші не виявлять її.

**Еквівалентне розбиття:** Розробка тестів методом чорного ящика, в якому тестові сценарії створюються для перевірки елементів еквівалентної області. Як правило, тестові сценарії розробляються для покриття кожної області як мінімум один раз.

**Граничні дані**

Окремий вид допустимих даних, передача яких в систему може розкрити дефект, – граничні дані, тобто, наприклад, числа, значення яких є граничними для їх типу, рядки граничної або нульової довжини тощо. Зазвичай за допомогою тестування граничних умов виявляються проблеми з арифметичним порівнянням чисел або з ітераторами циклів. Для тестування функції Set\_CRC з прикладу 1 на граничних умовах можна визначити два тестових приклади з мінімальними і максимальними значеннями полів у записі.

**Відсутність даних**

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

**Повторне введення даних**

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

**Невірні дані**

При перевірці поведінки системи необхідно не забувати перевіряти систему при передачі їй даних, що не передбачені вимогами – занадто довгих або занадто коротких рядків, невірних символів, чисел за межами діапазону тощо. Невірні дані, як і допустимі, також можна розділяти на різні класи еквівалентності. Прикладом невірних даних для функції Set\_CRC може служити запис з іншою структурою, переданій у функцію через приведення типів. Якщо розрахунок контрольної суми використовує імена полів запису, то контрольна сума може виявитися обчисленою невірно або може відбутися перезапис областей пам’яті, що не призначені для зберігання даних.

**Реініціалізація системи**

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

С**тійкість системи**

Під стійкістю системи розуміють її здатність витримувати нештатне навантаження, яке явно не передбачене вимогами. Наприклад, чи збереже система працездатність після 10 тисяч викликів.

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

**Позаштатні стани середовища виконання**

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

**Граничні умови**

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

* значення всередині діапазону;
* мінімальне значення;
* максимальне значення.

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

* значення всередині діапазону;
* мінімальне значення;
* мінімальне значення + 1;
* максимальне значення;
* максимальне значення – 1.

Такий спосіб перевірки називається перевіркою на граничних значеннях. Така перевірка дозволяє виявляти проблеми, що пов’язані з виходом за межі діапазону.

**Вимоги до критерію тестування**

Він повинен бути:

* **достатнім**, тобто, коли деяка кінцева множина тестів достатня для тестування конкретно обраної програми;
* **повним**, тобто у разі помилки повинен існувати тест з множини тестів, що задовольняють критерію, який виявляє помилку;
* **надійним**, тобто будь – які дві множини тестів, що задовольняють йому, одночасно повинні розкривати або не розкривати помилки програми;
* таким, що **легко перевіряється**, наприклад, обчислюваним на тестах.

Для нетривіальних класів програм у загальному випадку не існує повного і надійного критерію, що залежить від програм або специфікацій. Тому прагнуть до ідеального загального через реальні часткові критерії, які можна розбити на такі класи.

1. Структурні критерії, що використовують інформацію про структуру програми (критерії так званого «білого ящика» ).

2. Функціональні критерії, що формулюються в описі вимог до програмного виробу (критерії так званого «чорного ящика» )

3. Критерії стохастичного тестування, що формулюються в термінах перевірки наявності заданих властивостей у тестованому додатку, засобами перевірки деякої статистичної гіпотези.

4. Мутаційні критерії, що орієнтовані на перевірку властивостей програмного виробу на основі підходу Монте–Карло.

**Особливості застосування структурних і функціональних критеріїв**

***Структурні критерії (клас I).***

Структурні критерії використовують модель програми у вигляді «білого ящика», що припускає знання початкового тексту програми або специфікації програми у вигляді потокового графа управління. Структурна інформація зрозуміла і доступна розробникам підсистем і модулів додатку, тому даний клас критеріїв часто використовується на етапах модульного і інтеграційного тестування (Unit testing, Integration testing).

Структурні критерії базуються на основних елементах управляючого графа програми, операторах, гілках і шляхах.

Умовою критерію **тестування команд** є те, що набір тестів у сукупності повинен забезпечити проходження кожної команди не менше одного разу. Це слабкий критерій, він, як правило, використовується у великих програмних системах, де інші критерії застосувати неможливо.

Умовою критерію **тестування гілок** є те, що набір тестів у сукупності повинен забезпечити проходження кожної гілки не менше одного разу. Це досить сильний і при цьому економічний критерій, оскільки множина гілок у тестованому додатку і не така вже велика. Даний критерій часто використовується в системах автоматизації тестування.

Умовою критерію **тестування шляхів** є те, що набір тестів в сукупності повинен забезпечити проходження кожного шляху не менше одного разу. Якщо програма містить цикл (особливо з неявно заданим числом ітерацій), то число ітерацій обмежується константою (часто – 2, або числом класів вихідних шляхів).

**Функціональні критерії (клас II).**

**Функціональний критерій** – найважливіший для програмної індустрії критерій тестування. Він забезпечує, перш за все, контроль ступеня виконання вимог замовника в програмному продукті. Оскільки вимоги формулюються до продукту в цілому, вони відображають взаємодію тестованого додатка з оточенням. При функціональному тестуванні переважно використовується модель «чорного ящика». Проблема функціонального тестування – це, перш за все, трудомісткість, оскільки документи, які фіксують вимоги до програмного виробу (Software requirement specification, Functional specification тощо), як правило, достатньо об’ємні, проте, відповідна перевірка повинна бути всеосяжною.

Основні види функціональних критеріїв.

**Тестування пунктів специфікації –** набір тестів у сукупності повинен забезпечити перевірку кожного тестованого пункту не менше одного разу.

Специфікація вимог може містити сотні і тисячі пунктів вимог до програмного продукту і кожна з цих вимог при тестуванні повинна бути перевірена відповідно до критерію не менше ніж одним тестом

**Тестування класів вхідних даних** – набір тестів у сукупності повинен забезпечити перевірку представника кожного класу вхідних даних не менше одного разу.

**Тестування пунктів специфікації –** набір тестів у сукупності повинен забезпечити перевірку кожного тестованого пункту не менше одного разу.

Специфікація вимог може містити сотні і тисячі пунктів вимог до програмного продукту і кожна з цих вимог при тестуванні повинна бути перевірена відповідно до критерію не менше ніж одним тестом

**Тестування класів вхідних даних** – набір тестів у сукупності повинен забезпечити перевірку представника кожного класу вхідних даних не менше одного разу.

Граматика повинна бути достатньо простою, щоб трудомісткість розробки відповідного набору тестів була реальною (вписувалася в терміни і штат фахівців, виділених для реалізації фази тестування).

**Тестування класів вихідних даних** – набір тестів в сукупності повинен забезпечити перевірку представника кожного вихідного класу, за умови, що вихідні результати наперед класифікуються, причому окремі класи результатів враховують, зокрема, обмеження на ресурси або на якийсь час (time out).

**Тестування функцій** – набір тестів у сукупності повинен забезпечити перевірку кожної дії, що реалізовується тестованим модулем, не менше одного разу.

Функціональний критерій дуже популярний на практиці, але не забезпечує покриття частини функціональності тестованого компонента, яка пов’язана із структурними і поведінковими властивостями, опис яких не зосереджений в окремих функціях (тобто опис розосереджений по компоненту).

Критерій тестування функцій об’єднує частково особливості структурних і функціональних критеріїв. Він базується на моделі «напівпрозорого ящика», де явно вказані не тільки входи і виходи тестованого компонента, але також склад і структура використовуваних методів (функцій, процедур) і класів.

Комбіновані критерії для програм і специфікацій – набір тестів в сукупності повинен забезпечити перевірку всіх комбінацій несуперечливих умов програм і специфікацій не менше одного разу.

При цьому всі комбінації несуперечливих умов треба підтвердити, а умови суперечностей слід виявити і ліквідовувати.

**Застосування методів стохастичного тестування**

***Стохастичні критерії (клас III).***

Стохастичне тестування застосовується при тестуванні складних програмних комплексів, коли набір детермінованих тестів (X, Y) має велику потужність. У випадках, коли подібний набір неможливо розробити і виконати на фазі тестування, можна застосувати таку методику.

Розробити програми-імітатори випадкових послідовностей вхідних сигналів {*x*}.

Обчислити незалежним способом значення {*y*} для відповідних вхідних сигналів {*x*} і отримати тестовий набір (X, Y).

Протестувати додаток на тестовому наборі (X, Y), використовуючи два способи контролю результатів:

**Детермінований контроль** – перевірка відповідності обчисленого значення yвих ∈{*y*} значенню *у*, отриманому в результаті прогону тесту на наборі {*x*} – випадковій послідовності вхідних сигналів, що згенерована імітатором.

**Стохастичний контроль** – перевірка відповідності множині значень {yвих}, отриманих у результаті прогону тестів на наборі вхідних значень {*x*}, наперед відомому розподілі результатів F(Y).

У цьому випадку множина Y невідома (її обчислення неможливе), але відомий закон розподілу даної множини.

***Критерії стохастичного тестування.***

**Статистичні методи закінчення тестування** – стохастичні методи ухвалення рішень про збіг гіпотез про розподіл випадкових величин. До них належить широко відомий критерій Стьюдента (**St**), метод Хі–квадрат (χ2) та ін.

**Метод оцінки швидкості виявлення помилок** – заснований на моделі швидкості виявлення помилок, згідно з якою тестування припиняється, якщо оцінений інтервал часу між поточною помилкою і наступною дуже великий для фази тестування додатку

**Мутаційний критерій і техніка роботи з ним**

***Мутаційний критерій (клас IV).***

Професійні програмісти пишуть відразу майже правильні програми, що відрізняються від правильних дрібними помилками, наприклад – перестановка місцями максимальних значень індексів в описі масивів, помилки в знаках арифметичних операцій, заниження або завищення межі циклу на 1 тощо. Розглянемо підхід, що дозволяє на основі дрібних помилок оцінити загальне число помилок, що залишилися в програмі.

Підхід базується на таких поняттях:

* **мутації** – дрібні помилки в програмі;
* **мутанти** – програми, що відрізняються одна від одної мутаціями.

Метод мутаційного тестування полягає в тому, що в програми А та Б, що розробляються, вносять мутації, тобто штучно створюють програми-мутанти А1, Б1... Потім програми А та Б і їх мутанти тестуються на одному і тому ж наборі тестів (X,Y).

Якщо на наборі (X,Y) підтверджується правильність програми P і, крім того, виявляються всі внесені до програм-мутантів помилки, то набір тестів (X,Y) відповідає мутаційному критерію, а тестована програма оголошується правильною.

Якщо деякі мутанти не виявили всіх мутацій, то треба розширювати набір тестів (X,Y) і продовжувати тестування.