**Лекція 16.** **Механізми та методи тестування.** **Загальні питання організації тестування.**

**Підготовка тестування програмного продукту**

Одним з інструментів вимірювання якості розроблюваного програмного забезпечення (ПЗ) є тестування, який по суті є процесом технічного дослідження на вимогу замовників для вияву інформації про якість продукту відносно контексту, в якому він має використовуватись. До цього процесу входить виконання програми з метою знайдення помилок, що було розглянуто на попередніх заняттях. Щодо інших аспектів оцінювання якості ПЗ таких як надійність, супроводжуваність, практичність, ефективність, мобільність, функціональність за стандартом ISO 9126, а також заданих замовником, то не їх завжди можна оцінити через метрики. Що дає можливість визначити, що якість - це суб'єктивне поняття. Тому тестування не може повністю забезпечити коректність ПЗ з точки зору заданих визначень якості. Воно тільки порівнює стан і поведінку продукту зі специфікацією. При цьому треба розрізняти тестування програмного забезпечення і забезпечення якості програмного забезпечення, до якого належать усі складові ділового процесу, а не тільки тестування.

Тестування виконується протягом всього життєвого циклу ПЗ, починаючи від визначення вимог і закінчуючи етапом експлуатації, на кожному перевіряючи відповідність програм визначеним вимогам. Як ітераційний процес тестування не може вважатися остаточно завершеним або, як висловився професор Едсжер Дейкстра у 1972 році: "Тестування програм може служити доказом наявності помилок, але ніколи не доведе їхню відсутність!" Таким чином, при підготовці тестування стоять два завдання.

*1. П*ідготувати такий набір тестів і застосувати до них ПЗ, щоб виявити в нім по можливості більше число помилок. Проте чим довше продовжується процес тестування (і відладки в цілому), тим більшою стає вартість ПЗ.

2. *В*изначити момент закінчення відладки ПЗ (або окремою його компоненти). Ознакою можливості закінчення відладки є повнота обхвату пропущеними через ПЗ тестами (тобто тестами, до яких застосоване ПЗ) безлічі різних ситуацій, що виникають при виконанні програм ПЗ, і відносний рідкісний прояв помилок в ПЗ на останньому відрізку процесу тестування. Останнє визначається відповідно до необхідного ступеня надійності ПЗ, вказаної в специфікації його якості.

Для оптимізації набору тестів, тобто для підготовки такого набору тестів, який дозволяв би при заданому їх числі (або при заданому інтервалі часі, відведеному на тестування) виявляти більше число помилок, необхідно, по-перше, заздалегідь планувати цей набір і, по-друге, використовувати раціональну стратегію планування (проектування ) тестів.

Проектування тестів бажано починати відразу ж після завершення етапу зовнішнього опису ПЗ. Можливі різні підходи до вироблення стратегії проектування тестів, які можна умовно графічно розмістити (див. рис. 1) між наступними двома крайніми підходами. Лівий крайній підхід (тестування по відношенню до специфікації) полягає в тому, що тести проектуються тільки на підставі вивчення специфікацій ПЗ (зовнішнього опису, опису архітектури і специфікації модулів). Будова модулів при цьому ніяк не враховується, тобто вони розглядаються як чорні ящики. Фактично такий підхід вимагає повного перебору всіх наборів вхідних даних, оскільки при використанні як тести тільки частині цих наборів деякі ділянки програм ПЗ можуть не працювати ні на якому тесті і, значить, помилки, що містяться в них, не виявлятимуться. Проте тестування ПЗ повним безліччю наборів вхідних даних практично нездійсненно. Правий крайній підхід (тестування по відношенню до текстів програм ) полягає в тому, що тести проектуються на підставі вивчення текстів програм з метою протестувати всі шляхи виконання кожній програм ПЗ. Якщо взяти до уваги наявність в програмах циклів із змінним числом повторень, то різних шляхів виконання програм ПЗ може опинитися також надзвичайно багато, так що їх тестування також буде практично нездійсненно.



Рис. 1. Різні підходи до проектування тестів.

Оптимальна стратегія проектування тестів розташована усередині інтервалу між цими крайніми підходами, але ближче до лівого краю. Вона включає проектування значної частини тестів по специфікаціях, виходячи з принципів: на кожну використовувану функцію або можливість - хоч би один тест, на кожну область і на кожну межу зміни якої-небудь вхідної величини - хоч би один тест, на кожен особливий випадок або на кожну виняткову ситуацію, вказані в специфікаціях, - хоч би один тест. Але вона вимагає також проектування деяких тестів і по текстах програм, виходячи з принципу (як мінімум): кожна команда кожної програми ПЗ повинна пропрацювати хоч би на одному тесті.

Оптимальну стратегію проектування тестів можна конкретизувати за принципом: для кожного програмного документа (включаючи тексти програм), що входить до складу ПЗ, повинні проектуватися свої тести з метою виявлення в ньому помилок. Майерс навіть визначає різні види тестування залежно від вигляду програмного документа, на підставі якого будуються тести. До таких документів відносяться визначення вимог до ПЗ, специфікація якості ПЗ, функціональна специфікація ПЗ, опис архітектури ПЗ і тексти програм ПЗ [7].

Згідно Майерсу [7] тестування повинне бути організоване за принципами:

1) кожен тест повинен включати опис очікуваних результатів роботи програми, щоб можна було швидко з'ясувати наявність або відсутність помилок в ній;

2) бажано, щоб окрім автора програми тестування проводили і інші спеціалісти, оскільки виявлення недоліків в своїй діяльності протирічить людській психології. Однак, відладка програми найефективніше виконується саме автором програми;

3) організація-розробник ПЗ не повинна "одноосібно" тестувати програмний продукт, потрібно організовувати бета-тестування з залученням інших організацій;

4) результати кожного тесту повинні бути документовані і детально вивчені, щоб не пропустити малопомітну на перший погляд помилку в програмі;

5) тести для неправильних (непередбачуваних) даних повинні підбиратися так само ретельно, як і для правильних (передбачуваних) вхідних даних;

6) при аналізі результатів кожного тесту необхідно перевіряти, чи не робить програма того, що вона не повинна робити;

7) потрібно зберігати використані тести (для підвищення ефективності повторного тестування програми після її модифікації або установки у замовника);

8) тестування не повинно плануватися виходячи з припущення, що в програмі не будуть виявлені помилки (зокрема, потрібно виділяти на тестування достатньо часових і матеріальних ресурсів);

9) потрібно враховувати так званий "принцип накопичення помилок": вірогідність наявності не виявлених помилок в деякій частині програми прямо пропорційна числу помилок, вже виявлених в цій частині;

10) потрібно завжди пам'ятати, що тестування - творчий процес, і не відноситися до нього як до рутинного заняття.

Процес тестування об'єднує різні способи тестування в сплановану послідовність кроків, які приводять до успішної побудови програмної системи (ПС). Методика тестування ПС може бути представлена у вигляді спіралі, що розгортається (рис. 2).

На початку здійснюється *тестування елементів (модулів), яке* перевіряє результати етапу *кодування* ПС. На другому кроці виконується *тестування інтеграції,* орієнтоване на виявлення помилок етапу *проектування* ПС. На третьому обороті спіралі проводиться *тестування*



Рис.2. Спіраль процесу тестування ПС

Кожен крок процесу тестування характеризується таким чином.

1. *Тестування елементів.* Мета — індивідуальна перевірка кожного модуля. Використовуються способи тестування «білого ящика».

2. *Тестування інтеграції.* Мета — тестування збірки модулів в програмну систему. В основному застосовують способи тестування «чорного ящика».

3. *Тестування правильності.* Мета — перевірити реалізацію в програмній системі всіх функціональних і поведінкових вимог, а також вимоги ефективності. Використовуються виключно способи тестування «чорного ящика».

4. *Системне тестування.* Мета — перевірка правильності об'єднання і взаємодії всіх елементів комп'ютерної системи, реалізації всіх системних функцій.

Організація процесу тестування у вигляді еволюційної спіралі, що розгортається, забезпечує максимальну ефективність пошуку помилок. Проте виникає питання — коли закінчувати тестування? На практицізазвичай застосовують статистичний критерій: «Можна з 95%-ю впевненістю сказати, що проведено достатнє тестування, якщо вірогідність безвідмовної роботи ПП протягом 1000 годин складає щонайменше 0,995».

На кожному кроці тестування застосовуються різні види тестування, залежно від типу ПС та її обсягів.

**Ознаки класифікації видів тестування**

Класифікацію видів тестування зазвичай виконують за такими ознаками:

За об'єктом тестування:

* функціональне тестування (functional testing);
* тестування продуктивності (performance testing):
  + навантажувальне тестування (load testing);
  + стрес-тестування (stress testing);
  + тестування стабільності (stability / endurance / soak testing);
* тестування зручності використання (usability testing);
* тестування інтерфейсу користувача (ui testing);
* тестування безпеки (security testing);
* тестування локалізації (localization testing);
* тестування сумісності (compatibility testing).

За знанням системи:

* тестування чорного ящика (black box);
* тестування білого ящика (white box);
* тестування сірого ящика (gray box).

За ступенем автоматизації:

* ручне тестування (manual testing);
* автоматизоване тестування (automated testing);
* напівавтоматизоване тестування (semiautomated testing).

За ступенем ізольованості компонентів:

* компонентне (модульне) тестування (component / unit testing);
* інтеграційне тестування (integration testing);
* системне тестування (system / end-to-end testing).

За часом проведення тестування:

* Альфа-тестування (alpha testing):
  + тестування при прийманні (smoke testing);
  + тестування нової функціональності (new feature testing);
  + регресійне тестування (regression testing);
  + тестування при здачі (acceptance testing);
* Бета-тестування (beta testing).

За ознакою позитивності сценаріїв:

* позитивне тестування (positive testing);
* негативне тестування (negative testing).

За ступенем підготовленості до тестування:

* тестування за документацією (formal testing);
* ед хок (інтуїтивне) тестування (ad hoc testing).

**Види тестування за цілями**

За цілями всі ці види можна поділити на типи:

* Функціональні.
* Нефункціональні.
* Пов'язані зі змінами.

***Функціональні тести*** базуються на функціях та особливостях, а також на взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: компонентному або модульному (Component/Unit testing), інтеграційному (Integration testing), системному (System testing) і приймальному (Acceptance testing). Функціональні види тестування розглядають зовнішню поведінку системи. Одні з найпоширеніших видів функціональних тестів:

* Функціональне тестування (Functional testing).
* Тестування безпеки (Security and Access Control Testing) - стратегія тестування, що використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту програми, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних. Тестування безпеки може виконуватися як автоматизовано так і в ручну, включаючи перевірку як позитивних, так і негативних тестових випадків.
* Тестування взаємодії (Interoperability Testing) - це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності (compatibility testing) та інтеграційне тестування (integration testing).

**Нефункціональне тестування** описує тести, необхідні для визначення характеристик ПЗ, які можуть бути виміряні різними величинами. У цілому, це тестування того, "Як" система працює. Основні види нефункціональних тестів:

тестування продуктивності;

тестування установки ;

тестування зручності користування;

тестування на відмову і відновлення;

конфігураційне тестування.

***Тестування продуктивності***:

* + тестування навантаження (Performance and Load Testing) - визначення масштабованості додатків під навантаженням. Дії: вимір часу виконання вибраних операцій за певних показників інтенсивні виконання цих операцій; визначення кількості користувачів, що одночасно працюють з додатком; визначення меж прийнятної продуктивності при збільшенні навантаження (при збільшенні інтенсивності виконання цих операцій); дослідження продуктивності при високих, граничних, стресових навантаженнях;
  + стресове тестування (Stress Testing) дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу та оцінити здатність системи до відновлення, тобто до повернення до нормального стану після припинення впливу стресу. Стресом у даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійна зміна конфігурації сервера. Також одним із завдань при стресовому тестуванні може бути оцінка падіння продуктивності, і тут цілі стресового тестування перетинаються з цілями тестування продуктивності;
  + тестування стабільності або надійності (Stability / Reliability Testing) - перевірка працездатності програми при тривалому (багатогодинному) тестуванні з середнім рівнем навантаження. Час виконання операцій може грати в даному виді тестування другорядну роль. При цьому на перше місце виходить відсутність витоків пам'яті, перезапусків серверів під навантаженням та інші аспекти, які впливають саме на стабільність роботи;
  + об'ємне тестування (Volume Testing) - отримання оцінки продуктивності при збільшенні обсягів даних у базі даних програми, при цьому відбувається: вимір часу виконання вибраних операцій за певних показників інтенсивності виконання цих операцій; визначення кількості користувачів, що одночасно працюють з додатком.

***Тестування установки*** (Installation testing) спрямоване на перевірку успішної інсталяції та настройки, а також оновлення або видалення ПЗ. На даний момент найбільш поширена установка ПЗ за допомогою інсталяторів (які самі по собі так само потребують належного тестування). У реальних умовах інсталяторів може не бути, і в цьому випадку замовнику/користувачу доведеться самостійно виконувати установку ПЗ, використовуючи документацію у вигляді інструкцій або readme файлів, де крок за кроком описано всі необхідні дії та перевірки.

***Тестування зручності користування*** (Usability Testing) - це метод тестування, спрямований на встановлення ступеня зручності використання, навченості, зрозумілості та привабливості для користувачів розроблюваного продукту в контексті заданих умов. Цей тип тестування дає оцінку рівня зручності використання програми за такими показниками:

* + продуктивність, ефективність (efficiency) - скільки часу і кроків знадобиться користувачеві для завершення основних завдань програми, наприклад, розміщення новини, реєстрації, покупки тощо (менше - краще);
  + правильність (accuracy) - скільки помилок зробив користувач під час роботи з додатком (менше - краще);
  + активізація в пам'яті (recall) - як багато користувач пам'ятає про роботу програми після припинення роботи з нею на тривалий період часу (повторне виконання операцій після перерви має проходити швидше ніж у нового користувача);
  + емоційна реакція (emotional response) - як користувач почувається після завершення завдання - розгублений, знаходиться у стані стресу? Чи порекомендує користувач систему своїм друзям (позитивна реакція - краще)?

***Тестування на відмову і відновлення*** (Failover and Recovery Testing) перевіряє тестований продукт з точки зору здатності протистояти та успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками ПЗ, відмовами обладнання або проблемами зв'язку (наприклад, відмова мережі). Метою даного виду тестування є перевірка систем відновлення (або систем, дублюючих основний функціонал), які, у разі виникнення збоїв, забезпечать збереження і цілісність даних тестованого продукту.

***Конфігураційне тестування*** (Configuration Testing) - вид традиційного тестування продуктивності. У цьому випадку замість того, щоб тестувати продуктивність системи з точки зору навантаження, тестується ефект впливу на продуктивність змін у конфігурації. Прикладом такого тестування можуть бути експерименти з різними методами балансування навантаження. Конфігураційне тестування також може бути поєднане з навантажувальним, стрес- або тестуванням стабільності.

**Тестування, пов'язане зі змінами**

Після проведення необхідних змін, таких як виправлення бага/дефекту, ПЗ повинне бути перетестовано для підтвердження того факту, що проблему було дійсно вирішено. Для підтвердження працездатності програми або правильності здійсненого виправлення дефекту виконуються види тестування, які необхідно проводити після установки ПЗ:

* ***Димове тестування*** (Smoke Testing). Поняття димове тестування пішло з інженерного середовища. При введенні в експлуатацію нового "заліза" вважалося, що тестування пройшло вдало, якщо з установки не пішов дим. В області ж тестування ПЗ, воно спрямоване на поверхневу перевірку всіх модулів програми на предмет працездатності та наявність швидко встановлюваних критичних і блокуючих дефектів. За результатами димового тестування робиться висновок про те, приймається чи ні встановлена версія ПЗ на тестування, експлуатацію або на постачання замовнику.
* ***Регресійне тестування*** (Regression Testing) - вид тестування спрямований на перевірку змін, зроблених у додатку або навколишньому середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб-сервер або сервер додатка), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше.
* ***Тестування збірки*** (Build Verification Test) спрямоване на визначення відповідності випущеної версії критеріям якості 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*** або перевірка узгодженості / справності (Sanity Testing) – вузько направлене тестування достатнє для доказу того, що конкретна функція працює згідно заявлених у специфікації вимог.

**Відмінність санітарного тестування від димового** (Sanity vs Smoke testing)  
У деяких джерелах помилково вважають, що санітарне та димове тестування – це одне і теж. Вважається, що ці види тестування мають “вектори руху”, які спрямовані в різні боки. На відміну від димового (Smoke testing), санітарне тестування (Sanity testing) направлено вглиб функції, що перевіряється, в той час як димове направлено вшир, для покриття тестами якомога більшого функціоналу в найкоротші терміни.

**Кросбраузерність** (Cross-browser) – властивість сайту відображатися і працювати у всіх популярних браузерах ідентично. Під ідентичністю розуміється відсутність недоліків верстки і здатність відображати матеріал з однаковим ступенем читабельності.

Тестування сайту на кроссбраузерность необхідно проводити:

* У різних браузерах (сімейство Mozila, Internet Explorer, Opera, Safari, Мобільні браузери)
* При різних розширеннях екрану (зазвичай 640 \* 480, 800 \* 600, 1024 \* 768, 1280 \* 800, 1280 \* 1024, 1366 \* 768)
* В різних операційних системах (Mac OS, Linux, Win)

За рівнями тестування поділяється на ***модульне тестування*** (яке розглядалось в попередній лекції) та інтеграційне тестування. Якщо при модульному тестуванні (юніт-тестування), яке часто виконує розробник ПЗ тестується мінімально можливий для тестування компонент, наприклад, окремий клас або функція.

При ***інтеграційному тестуванні*** тестуються інтерфейси між компонентами, підсистемами. За наявності резерву часу на даній стадії тестування ведеться ітераційно, з поступовим підключенням наступних підсистем та за такими рівнями:

* + компонентний інтеграційний рівень (Component Integration testing), коли перевіряється взаємодія між компонентами системи після проведення компонентного тестування;
  + системний інтеграційний рівень (System Integration Testing), коли перевіряється взаємодія між різними системами після проведення системного тестування.

Підходи до інтеграційного тестування:

* + знизу вгору (Bottom Up Integration). Усі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для проведення інтеграційного тестування. Даний підхід вважається корисним, якщо всі або практично всі модулі розроблюваного рівня готові. Також даний підхід допомагає визначити за результатами тестування рівень готовності додатків;
  + згори вниз (Top Down Integration). У першу чергу тестуються компоненти верхнього рівня ієрархії об'єктів з використанням заглушок замість компонентів більш низького рівня;
  + великий вибух ("Big Bang" Integration). Усі або практично усі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини, і потім проводиться інтеграційне тестування. Такий підхід дуже хороший для збереження часу. Проте, якщо тест-кейси (артефакти, що описують сукупність кроків, конкретних умов та параметрів, необхідних для перевірки реалізації функції, що тестується чи її частини) та їх результати записані не вірно, то сам процес інтеграції дуже ускладниться, що стане перепоною для команди тестування при досягненні основної мети інтеграційного тестування.

**Системне тестування –** тестування інтегрованої системи на її відповідність вимогам. Виконується альфа- та бета-тестування.

* + ***Альфа-тестування*** - імітація реальної роботи з системою штатними розробниками, або реальна робота з системою потенційними користувачами / замовником. Найчастіше альфа-тестування проводиться на ранній стадії розробки продукту, але у деяких випадках може застосовуватися для закінченого продукту в якості внутрішнього приймального тестування. Іноді альфа-тестування виконується під відлагоджувачем або з використанням оточення, яке допомагає швидко виявляти знайдені помилки. Виявлені помилки можуть бути передані тестувальникам для додаткового дослідження в оточенні, подібному тому, в якому буде використовуватися програма.
  + ***Бета-тестування*** - у деяких випадках виконується поширення версії з обмеженнями (за функціональністю або часом роботи) для певної групи осіб, з тим щоб переконатися, що продукт містить достатньо мало помилок. Іноді бета-тестування виконується для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів.

Часто для вільного/відкритого ПЗ стадія альфа-тестування характеризує функціональне наповнення коду, а бета-тестування - стадію виправлення помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам.

Тестування "білого ящика" і "чорного ящика" розглядалось в попередній лекції.

Під час тестування "***білого ящика"*** (англ. white-box testing, або прозорого ящика), розробник тесту має доступ до вихідного коду програм і може писати код, який пов'язаний з бібліотеками ПЗ, що тестується. Це типовий підхід для модульного тестування (unit testing), при якому тестуються тільки окремі частини системи. Воно забезпечує те, що компоненти конструкції - працездатні й стійкі до певного ступеня. Під час тестування білого ящика використовуються метрики покриття коду.

При тестуванні "***чорного ящика***" тестувальник має доступ до ПЗ тільки через ті ж інтерфейси, що й замовник або користувач, або через зовнішні інтерфейси, що дозволяють іншому комп'ютеру або іншому процесу підключитися до системи для тестування. Наприклад, модуль, що тестується, може віртуально натискати клавіші або кнопки миші за допомогою механізму взаємодії процесів, із упевненістю в тому, чи все йде правильно, що ці події викликають той же відгук, що й реальні натискання клавіш і кнопок миші. Як правило, тестування "чорного ящика" ведеться з використанням специфікацій чи інших документів, що описують вимоги до системи. В даному виді тестування критерій покриття складається з покриття структури вхідних даних, покриття вимог і покриття моделі (у тестуванні на основі моделей).

При тестуванні "***сірого ящика***" розробник тесту має доступ до вихідного коду, але при безпосередньому виконанні тестів доступ до коду, як правило, не потрібний.

**Статичне і динамічне тестування**.

Техніки тестування білого ящика і тестування чорного ящика - припускають, що код виконується, і різниця полягає лише в тій інформації, якою володіє тестувальник. В обох випадках це ***динамічне тестування***.

При ***статичному тестуванні*** програмний код не виконується - аналіз програми відбувається на основі вихідного коду, який вираховується вручну, або аналізується спеціальними інструментами. Також до статичного тестування відносять тестування вимог, специфікацій, документації.

**Тестові скрипти.**

Тестувальники використовують тестові скрипти на різних рівнях: як у модульному, так і в інтеграційному і системному тестуванні. Тестові скрипти, як правило, пишуться для перевірки компонентів, в яких найбільш висока ймовірність появи відмов або вчасно не знайдена помилка може бути дорогою.

**Покриття коду.**

Покриття коду, за своєю суттю, є тестуванням методом білого ящика. Тестоване ПЗ збирається зі спеціальними настройками або бібліотеками та/або запускається в особливому оточенні, в результаті чого для кожної використовуваної (виконуваної) функції програми визначається місцезнаходження цієї функції у вихідному коді. Цей процес дозволяє розробникам і фахівцям із забезпечення якості визначити частини системи, які, при нормальній роботі, використовуються дуже рідко або ніколи не використовуються (такі як код обробки помилок тощо). Це дозволяє зорієнтувати тестувальників на тестування найбільш важливих режимів. Як правило, інструменти та бібліотеки, які використовуються для отримання покриття коду, вимагають значних витрат продуктивності та/або пам'яті, неприпустимих при нормальному функціонуванні ПЗ. Тому вони можуть використовуватися тільки в лабораторних умовах.

***Приймальне тестування*** (Acceptance Testing) - формальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою: визначення чи задовольняє система приймальним критеріям; винесення рішення замовником або іншою уповноваженою особою приймається додаток чи ні. Приймальне тестування виконується на підставі набору типових тестових випадків і сценаріїв, розроблених на відповідно до вимог до даного додатку. Рішення про проведення приймального тестування приймається тоді, коли: продукт досяг необхідного рівня якості; замовник ознайомлений з Планом приймальних Робіт (Product Acceptance Plan) або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні тощо. Фаза приймального тестування триває до тих пір, поки замовник не виносить рішення про відправлення програми на доопрацювання або готовності додатка.

Склад і зміст документації, супутньої процесу тестування, визначається стандартом IEEE 829-1998 Standard for Software Test Documentation.

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Література*

1. Тамре Л. Введение в тестирование программного обеспечения / Л. Тамре. –М.: Издательский дом «Вильямс», 2003. –368 с.
2. Сайкс Д. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Д.Сайкс, Д.Макгрегор. –К.: Диасофт, 2002. –432 с.
3. Фолк Д. Тестирование программного обеспечения / Д.Фолк, Е.К. Нгуен, С.Канер. – К.: Диасофт, 2003. –400 с.
4. Блэк Р. Ключевые процессы тестирования / Р.Блэк. – М.: Лори, 2006. –544 с.
5. Винниченко И.В. Автоматизация процессов тестирования / И.В.Винниченко. – СПб.: Питер, 2005. –208 с.
6. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Л.Криспин, Д.Грегори. –М.: Издательский дом «Вильямс», 2010. –464 с.
7. Майерс Г. Искусство тестирования программ / Пер. с англ. под ред. Б. А.Позина. – М.: Финансы и статистика, 1982. – 176 с.
8. Ален И., Голуб. С и С++. Правила программирования. – М.: БИНОМ, 1996. – 272ºс.
9. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1973. – 247º с.
10. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 275 с.
11. Йодан Э.. Структурное проектирование и конструирование програм. – М.: Мир, 1979. – 415 с.

*Контрольні запитання для самоперевірки*.

1. Що таке тестування?
2. Що таке відмовостійкість?
3. Які види тестування ви знаєте?
4. В чому полягає суть тестування?
5. В яких моделях немає тестування?
6. Чому при тестуванні необхідно бажати, щоб програма дала збій?
7. Поясніть, чому тестування є деструктивним процесом
8. Що найчастіше тестується в програмних засобах?
9. За якими ознаками прийнято робити класифікацію видів тестування?
10. Які види тестування, з наведених вище, потрібні для ПЗ Вашого курсового проекту?

**ДОДАТКОВІ ВІДОМОСТІ**

**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.

[](http://qalearning.com.ua/wp-content/uploads/2015/01/2.png)

Приклад 2:

Припустимо, ми тестуємо Інтернет-магазин, який продає олівці. У замовленні необхідно вказати кількість олівців (максимум для замовлення – 1000 штук). Залежно від замовленої кількості олівців змінюється вартість:

1 – 100 – 10 грн. за олівець;

101 – 200 – 9 грн. за олівець;

201 – 300 – 8 грн. за олівець і т.д.

З кожною новою сотнею, ціна зменшується на гривню.

Якщо тестувати «в лоб», то, щоб перевірити всі можливі варіанти обробки замовленої кількості олівців, потрібно написати дуже багато тестів (згадуємо, що можна замовити аж 1000 штук), а потім ще все це і протестувати. Спробуємо застосувати розбиття на класи еквівалентності. Очевидно, що наші вхідні дані можна розділити на наступні класи еквівалентності:

Невалідне значення:> 1000 штук;

Невалідне значення: <= 0;

Валідне значення: від 1 до 100;

Валідне значення: від 101 до 200;

Валідне значення: від 201 до 300;

Валідне значення: від 301 до 400;

Валідне значення: від 401 до 500;

Валідне значення: від 501 до 600;

Валідне значення: від 601 до 700;

Валідне значення: від 701 до 800;

Валідне значення: від 801 до 900;

Валідне значення: від 901 до 1000.

На основі цих класів ми і складемо тестові сценарії. Отже, якщо взяти по одному представнику з кожного класу, то отримуємо 12 тестів.

**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.

Тестовим прикладом, що створює нештатний стан середовища для функції Set\_CRC, може служити виділення всієї вільної пам’яті перед викликом функції. Якщо Set\_CRC використовує динамічну пам’ять, то в ній повинні бути перевірки на можливість виділити пам’ять, в іншому випадку виконання функції викличе її аварійне завершення:

*record\_type test\_value9;*

*int i;*

*int \*heap;*

*heap = malloc(\_MAXMEM);*

*test\_value9.A = false;*

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

*test\_value9.B[i] = i;*

*for (i=0;i<5;i++)*

*test\_value9.C[i] = i+5;*

*test\_value9.D[0] = i+8;*

*test\_value9.CRC = 0;*

*Set\_CRC(test\_value9);*

*free(heap);*

*printf(“%d\n”, test\_value9.CRC);*

**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.

Наприклад, якщо в функцію,

*char sum(char a, char b)*

*{*

*return a+b;*

*}*

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

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

*void abs\_array(char array[], char size)*

*{*

*for (int i=1;i<=size;i++)*

*{*

*array[i] = abs(array[i]);*

*}*

*return;*

*}*

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

**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.