**Зміст**

|  |  |  |
| --- | --- | --- |
| **ВСТУП………**……..…………………………………………………………………..….………………….. | | 2 |
| **Лекція 1. Тестування - спосіб забезпечення якості……………………………..** | | 5 |
| **Лекція 2. Рівні і види тестування………….............................................................** | | 10 |
| **Лекція 3. Agile (Гнучка модель). Scrum. Selenium …...........................................** | | 21 |
| **Лекція 4. UX/UI, Usability…………………………………………………………...** | | 25 |
| **Лекція 5. Типи тестування ………………………………………………………...** | | 46 |
| **Лекція 6. Аналіз вимог (Requirement Analysis) ………………………………….** | | 54 |
| **Лекція 7. Техніки тест дизайну (Test Design Techniques)……………………….** | | 58 |
| **Лекція 8. Тест-кейс (Test Case)…………………………………………………….** | | 64 |
| **Лекція 9. Fundamental Test Process………………………………………………..** | | 67 |
| **Список використаних джерел….……….………………………………………….** | | 71 |
|  |  |  |
|  |  |  |
|  |  |  |

# **Вступ**

Тестування програмного забезпечення (ПЗ)– це процес дослідження ПЗ з метою отримання інформації про якість програмного продукту, а саме відповідність специфікації, технічному завданню, або вимогам замовника ПЗ.

Практичний підхід до тестування ПЗ особливу увагу приділяє процесам тестування на фоні стрімкого прискорення процесу розробки ПЗ.

Цей підхід орієнтований на використання спеціалістами з тестування ПЗ тестових робіт. Швидкість і ефективність розробки ПЗ залежить від того наскільки процес тестування вписується в загальний життєвий цикл розробки ПЗ і від ефективності використання технології тестування.

Тестування - це одна з технік контролю якості, що включає в себе діяльність з планування робіт (Test Management), проектуванню тестів (Test Design), виконанню тестування (Test Execution) і аналізу отриманих результатів (Test Analysis).

Необхідними умовами для тестування є наявність :

* об'єкта тестування, доступного для проведення досліджень;
* виконавця(ів) (залежно від виду проведених досліджень їм може бути як людина, так і машина або комбінація людина + машина).

Достатніми умовами для тестування є наявність:

* об'єкта тестування, доступного для проведення досліджень;
* виконавця(ів) (залежно від виду діяльності на різних фазах їм може бути як людина, так і машина або комбінація людина + машина);
* плану тестування;
* тест кейсів / тестів;
* звіту, що підтверджує виконання задач і досягнення цілей, по тестуванню об'єкта.

В конспекті лекцій з дисципліни „Якість програмного забезпечення та тестування” викладені основні принципи, технології тестування, вимоги до документів з тестування згідно відповідних стандартів. Основна увага приділяється налаштуванню процесу тестування, щоб як можна скоріше досягнути мети виходу на ринок якісного програмного продукту.

**Лекція 1. Тестування - спосіб забезпечення якості**

**План лекції:**

1. **Поняття якості програмного продукту.**
2. **Основні визначення.**
3. **Тестування в життєвому циклі розробки ПЗ.**
4. **Еволюція поняття якості ПЗ.**
5. **Модель якості ПЗ ISO 9126.**

Якість програмного продукту характеризується набором властивостей, що визначають, наскільки продукт "хороший" з точки зору зацікавлених сторін, таких як замовник продукту, спонсор, кінцевий користувач, розробники і тестувальники продукту, інженери підтримки, співробітники відділів маркетингу, навчання і продажів. Кожен з учасників може мати різне уявлення про продукт і про те, наскільки він хороший чи поганий, тобто про те, наскільки висока якість продукту. Таким чином, постановка задачі забезпечення якості продукту виливається у завдання визначення зацікавлених осіб, їх критеріїв якості і потім знаходження оптимального рішення, що задовольняє цим критеріям. Тестування є одним з найбільш усталених способів забезпечення якості розробки програмного забезпечення і входить в набір ефективних засобів сучасної системи забезпечення якості програмного продукту.

Аналізуючи інформацію, можна зустріти зовсім протилежні думки про значення тестування в нових процесах розробки, і його важливість для системи забезпечення якості в цілому. Одні говорять, що в сучасних умовах тестування є здебільшого непотрібним, що нові методи проектування й розробки ПЗ в поєднанні з методами забезпечення якості на ранніх етапах розробки самі по собі гарантують одержання якісного продукту, що традиційне тестування є неефективною тратою коштів, що не дає результату для підвищення якості продукту, і водночас відкладає вихід продукту на ринок. Інші вважають, що нові методи розробки ПЗ й управління проектами вимагають нових методів тестування, що в нових умовах виникають нові ризики й нові види помилок у ПЗ, і тому значення тестування зростає, що ефективність новітніх методів забезпечення якості часто переоцінюється, і вказують при цьому на зазначене користувачами загальне зниження надійності комерційного ПЗ в останні роки. Більшість керівників і розробників починають розуміти важливість процесу тестування для підвищення якості програмних систем. Тестування із прикладного процесу з невисоким пріоритетом переходить у розряд особливо важливих процесів, чий життєвий цикл починається паралельно з розробкою програмних систем.

Конспект лекцій присвячений обговоренню способів вирішення завдання контролю якості розробки програмного забезпечення з позицій тестування. У цій області поряд з вирішенням наукових і технічних проблем важлива роль належить проблемі підготовки кадрів, здатних вирішувати завдання тестування і автоматизації тестування в умовах виробництва програмного продукту.

**Тестування** – це процес керованого експериментування з продуктом за допомогою тестів з метою виявлення в ньому помилок, тобто виявлення неточностей допущених розробниками ПЗ.

**Тест** – контрольна задача для перевірки коректності функціонування системи та/або її ПЗ. Основна ідея тестування – запустити ПЗ і спостерігати за його роботою та її наслідками. Якщо збій в роботі ПЗ відбувся, тоді аналізується звіт з метою виявлення місцезнаходження помилки, яка його викликала.

Вдалим тестом є той, при якому виконання програми закінчилось з помилкою і навпаки.

**Тестування виконує дві основні задачі:**

1) демонстрація якості функціонування ПЗ;

2) знаходження і усунення помилок в ПЗ.

Головною метою тестування є збільшення ймовірності того, що ПЗ, яке тестується, буде відповідати своїм специфікаціям. Тестування - процес ітераційний. Після виявлення і виправлення кожної помилки обов'язково слідує повторення тестів, що має на меті перевірку працездатності програми. Більш того, для ідентифікації причини виявленої проблеми може бути потрібно проведення спеціального додаткового тестування. При цьому завжди потрібно пам'ятати фундаментальний висновок, зроблений професором Едсжером Дейкстрой в 1972 г: "Тестування програм може служити доказом наявності помилок, але ніколи не доведе їх відсутність!" [1]. Тестування може виконуватися вручну або автоматично. Автоматичне тестування, якщо воно виконано коректно, забезпечує більш швидке, якісне і ефективне тестування, що приводить до зменшення кількості тестів, скороченню часу тестування, підвищення стійкості системи.

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 [2]. Як було зазначено вище, тестування підтверджує, що ПЗ працює згідно специфікації.

Вважають, що програма працює коректно, якщо вона задовольняє наступним критеріям:

1) отримавши коректні дані, програма надає правильний результат;

2) отримавши некоректні дані програма відхиляє їх;

3) програма не зависає і не вилітає, приймаючи як коректні, так і некоректні дані;

4) програма функціонує нормально стільки часу скільки потрібно;

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

**Основні визначення**

В цьому розділі викладені основні визначення, відповідно до стандартів, які використовуються як в наукових, так і в прикладних сферах стосовно ПЗ та його якості [3, 4, 5].

**Помилка (Еrror)** – хибне значення величини на виході системи (або підсистеми), викликане несправностями або збоями, яке, в свою чергу, може викликати відмову. З точки зору надійності ПЗ помилку можна розглядати, як упущення або неточність, допущені проектувальниками ПЗ, програмістами, аналітиками та тестерами. Наприклад, проектувальник може неправильно зрозуміти завдання, а програміст – неправильно описати змінну та інш.

**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 використовують, якщо вплив дефекту на роботу програми невеликий. Якщо ж помилка пов’язана із специфікаціями або архітектурою програми, то використовують слово дефект.

**Збій** (**Malfunction)** – прояв несправності, зазвичай в роботі устаткування.

**Відмова (Failure)** - порушення нормального функціонування системи, повна або часткова втрата працездатності системи (або підсистеми).

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

Некоректна поведінка може також означати неправильні значення вихідних, неправильний відгук пристрою або неправильне зображення на екрані. В процесі розробки відмови та баги зазвичай виявляються тестерами, а дефекти знаходяться і виправляються самими розробками. Якщо відмова виникла у користувача, звіт про помилку прямує розробнику з метою її усунення.

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

1. вхідні дані програми повинні використовувати частину коду, яка містить несправність;
2. наслідком несправності має стати некоректний внутрішній стан програми;
3. некоректний внутрішній стан програми повинен впливати на вихідні дані так, щоб результат помилки можна було спостерігати.

Якщо при наявності несправності у коді ПЗ відповідає умовам зазначеним вище, то воно вважається **придатним для тестування (testable).**

**Тестування (testing) – це:**

- Процес використання ПЗ при умові аналізу або запису отримуваних результатів з метою перевірки (оцінки) деяких властивостей тестованого об'єкту. (The process of operating а system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component).

- Процес аналізу ПЗ з метою виявлення відмінностей між створеним станом ПЗ та зазначеним у специфікації (що свідчить про прояв помилки) при експериментальній перевірці відповідного пункту вимог. (The process of analyzing а software item to detect the differences between existing and required conditions (that is, bugs))

- Контрольоване виконання програми на множині тестових даних і аналіз результатів цього виконання для пошуку помилок.

**Тест (Test)** – контрольна задача для перевірки коректності функціонування ПЗ. Групу взаємозв’язаних тестів називають **комплектом тестів** (test suite).

**Контрольний приклад (test case).** Контрольний приклад в сенсі тестування – записи, що відносяться до тесту, а саме, звіти, що містять:

1. Вхідні дані тесту – інформація, яку програма отримує із зовнішнього джерела, як то пристрій, інша програма або людина.

2. Умови виконання – вимоги для проведення тесту, наприклад, певний стан бази даних або конфігурація пристрою.

3. Очікувані вихідні дані – передбачуваний результат роботи коду.

Даний перелік визначає мінімальну необхідну інформацію для контрольного прикладу відповідно до стандартів. Компанія- розробник може визначити додаткову інформацію необхідною для внесення в звіт з метою її повторного використання в майбутньому або надання докладніших даних іншим тестерам і розробникам. Наприклад, мета тесту може бути включена в запис для більшої зрозумілості результатів тесту. До визначення точного формату звіта контрольного приклада слід залучати розробників, тестерів та/або відділ забезпечення якості ПЗ.

**Якість (Quality)** – ступінь відповідності системи, компоненту або процесу заданим вимогам, потребам або очікуванням користувача. З метою визначення добротності системи, компоненту або процесу використовують так звані атрибути якості – характеристики, що відображають дану властивість.

**Метрика (Metrics)** – кількісна міра ступеня наявності атрибута системи, компоненту або процесу.

Приведемо декілька прикладів атрибутів якості з коротким описом [6]:

- властивість, що відповідає за безперервність коректного протікання процесу, відноситься до **надійності** (reliability);

-властивість, що відповідає за постійну готовність системи, відноситься до **готовності** (availability);

- властивість, що відповідає за відсутність катастрофічних наслідків в системному середовищі, відноситься до **безпеки** (safety);

- властивість, що відповідає за запобігання несанкціонованому доступу до інформації, відноситься до **конфіденційності** (confidentiality);

- властивість, що відповідає за відсутність появи в системі невідповідних змін інформації, відноситься до **цілісності** (integrity);

- властивість, що відповідає за здатність системи піддаватися ремонту і розвитку, відноситься до **ремонтопридатність** (maintainability).

- властивість, що відповідає за здатність – рівень зусиль, необхідних для навчання, роботи, підготовки вхідних і обробки вихідних даних ПЗ, відноситься до **зручності роботи** (usability).

**Перевірка на валідність (Validation)** - процес, що дозволяє визначити, наскільки точно з позицій потенційного користувача деяка модель представляє задану суть реального миру.

**Верифікація (Verification)** - процес, який дозволяє визначити, що розроблене програмне забезпечення точно реалізує концептуальний опис даної системи. Або, як ще кажуть, процес перевірки відповідності системи заданими стандартами. Верифікація успішна, якщо отримані дані збігаються з очікуваними, заздалегідь визначеними як правильні. Зазначимо, що вона може бути неформальною, тобто тестер визначає успішність на основі своїх знань.

Верифікація та перевірка на валідність часто використовуються разом, але мають різні визначення. Різниця в них важлива для тестування ПЗ. Верифікація – це підтвердження того, що продукт відповідає специфікаціям, а перевірка на валідність – вимогам користувача.

**Методи забезпечення якості** є техніками, що гарантують досягнення певних показників якості при їх застосуванні.

**Методи контролю якості** дозволяють переконатися, що певні характеристики якості ПЗ досягнуті. Самі по собі вони не можуть допомогти їх досягненню, вони лише дають змогу визначити, чи вдалося отримати в результаті те, що хотілося, чи ні, а також знайти помилки, дефекти і відхилення від вимог.

Методи контролю якості ПЗ можна класифікувати таким чином:

- Методи та техніки, пов'язані з аналізом властивостей ПЗ під час його роботи. Це, перш за все, всі види тестування, а також вимірювання кількісних показників якості, які можна визначити за наслідками роботи ПЗ, — ефективність за часом й іншими ресурсами, надійність, доступність та ін.

- Методи та техніки визначення показників якості на основі симуляції роботи ПЗ за допомогою моделей різного роду. До цього вигляду відносяться перевірка на моделях (model checking), а також прототипіювання (макетування), що використовується для оцінки якості прийнятих рішень.

- Методи та техніки, націлені на виявлення порушень формалізованих правил побудови початкового коду ПЗ, проектних моделей та документації. До методів такого роду відноситься інспекція коду, що полягає в цілеспрямованому пошуку певних дефектів і порушень вимог в коді на основі набору шаблонів; автоматизовані методи пошуку помилок в коді, не засновані на його виконанні; методи перевірки документації на узгодженість і відповідність стандартам.

- Методи та техніки звичайного або формалізованого аналізу проектної документації і початкового коду для виявлення їх властивостей. До цієї групи відносяться численні методи аналізу архітектури ПЗ, методи формального доказу властивостей ПЗ і формального аналізу ефективності вживаних алгоритмів.

**Тестування в життєвому циклі розробки ПЗ.**

Нині використовують ітеративні процеси розробки ПЗ. Одним із прикладів такого підходу є RUP. При використанні ітеративного підходу тестування перестає бути процесом, що запускається після того, як програмісти написали весь необхідний код. Робота над тестами починається із самого першого етапу виявлення вимог до майбутнього продукту й тісно інтегрується з поточними завданнями, що спонукає до створення нових вимог для тестувальників. Їхня роль не зводиться просто до виявлення помилок якомога пов6 ніше і якомога раніше. Вони повинні брати участь у загальному процесі виявлення й усунення найбільш істотних ризиків проекту. Для цього на кожну ітерацію визначається мета тестування й методи її досягнення. Наприкінці кожної ітерації визначається, наскільки ця мета досягнута, чи потрібні додаткові випробування, і чи не потрібно змінити принципи й інструменти проведення тестів [7]. У свою чергу, кожний виявлений дефект повинен пройти через свій власний життєвий цикл. Дефект заноситься в базу дефектів. Аналітик визначає, чи не є він повтором внесеного раніше дефекту. Чи дійсно він є дефектом? Керівник затверджує виконавця, який приступає до усунення дефекту відповідно до пріоритету призначеного дефекту. Тестувальник повторює виконання тесту й переконується (або не переконується) в усуненні дефекту. Суворе дотримання життєвого циклу дефекту дозволяє істотно поліпшити керування проектом, а також якість створеного продукту [8].

**Еволюція поняття якості ПЗ.**

Що таке якісне програмне забезпечення? Якщо запитати про це досить широку групу людей, що мають справу з розробкою, продажем і використанням ПЗ, можна отримати такі відповіді:

— легко використовувати;

— висока продуктивність;

— немає помилок;

— можна використати на різних платформах;

— може працювати 24 години на добу й 7 днів на тиждень;

— легко додавати нові можливості;

— задовольняє потреби користувачів.

Всі ці відповіді виділяють характеристики, важливі для конкретного користувача, розробника ПЗ або групи таких осіб. Однак для підвищення ступеня задоволення всіх користувачів ПЗ, для досягнення ним стабільного становища на ринку й підвищення потенціалу розвитку важливо враховувати всі характеристики. Таким чином, якість ПЗ може бути описана великою кількістю різнорідних характеристик. Виходить, поняття якості програми багатопланове й може бути виражене адекватно тільки деякою структурованою системою характеристик або атрибутів. Така система характеристик називається моделлю якості [9].

**Якість ПЗ за Макколом**. Першою широко відомою моделлю якості ПЗ стала запропонована в 1977 р. Макколом й іншими модель. У ній характеристики якості розділені на три групи:

— фактори (factors), що описують ПЗ з позицій користувача та заданих вимог;

— критерії (criteria), що описують ПЗ з позицій розробника й задаються як цілі;

— метрики (metrics), що використовуються для кількісного опису й вимірювання якості.

Фактори якості, яких було виділено 11, групуються в три групи відповідно до роду роботи людей з ПЗ. Отримана структура зображується у вигляді трикутника Маккола. Критерії якості — це числові рівні факторів, поставлені як цілі при розробці. Об’єктивно оцінити або виміряти фактори якості досить важко. Тому Маккол використав метрики якості, які дозволяють легше вимірювати й оцінювати. Оцінки за шкалою набувають значень від 0 до 10.

Метрики якості:

— зручність перевірки на відповідність стандартам (auditability);

— точність управління й обчислень (accuracy);

— ступінь стандартності інтерфейсів (communication commonality);

— функціональна повнота (completeness);

— однорідність використовуваних правил проектування й документації (consistency);

— ступінь стандартності форматів даних (data commonality);

— стійкість до помилок (error tolerance);

— ефективність роботи (execution efficiency);

— розширюваність (expandability);

— широта сфери потенційного використання (generality);

— незалежність від апаратної платформи (hardware independence);

— повнота протоколювання помилок й інших подій (instrumentation);

— модульність (modularity);

— зручність роботи (operability);

— захищеність (security);

— самодокументованість (selfdocumentation);

— простота роботи (simplicity);

— незалежність від програмної платформи (software system independence);

* можливість порівняння проекту з вимогами (traceability);
* зручність навчання (training).



Рис. 1 – Трикутник Маккола

Кожна метрика впливає на оцінку декількох факторів якості. Числове вираження фактора являє собою лінійну комбінацію значень метрик, що впливають на нього. Коефіцієнти вираження визначаються по різному відповідно до різних організацій, команд розробки, видів ПЗ [9].

**Якість ПЗ за Боемом.** В 1978 р. Боем запропонував свою модель, власне кажучи, вона являла собою розширену модель Маккола. Атрибути якості класифікуються за способами використання ПЗ (primary use). Визначено 19 проміжних атрибутів (intermidiate construct), що включають всі 11 факторів якості за Макколом. Проміжні атрибути розділяються на примітивні (primitive construct), які у свою чергу можуть бути оцінені на основі метрик. На додаток до факторів Маккола атрибути якості за Боемом включають таке: ясність (clarity), зручність внесення змін (modifiability), документованість (documentation), здатність до відновлення функцій (resilience), зрозумілість (understandability), адекватність (validity), функціональність (functionality), універсальність (generality), економічну ефективність (economy) [9].

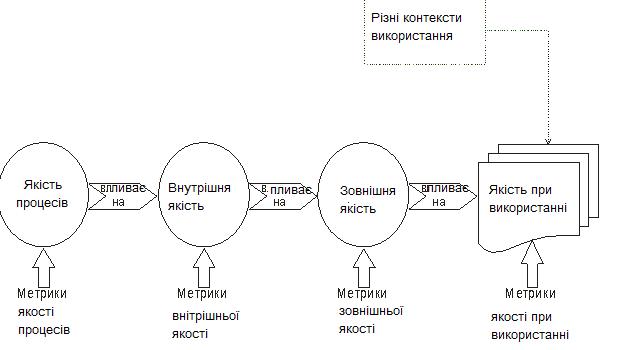
**Модель якості ПЗ ISO 9126.** У 1991 р. стандартною була прийнята модель якості ПЗ ISO 9126. Ця модель не є прямим розширенням раніше запропонованих. У ній оцінка якості ПЗ заснована на таких характеристиках:

— мета (goals) — те, що ми хочемо бачити в ПЗ;

— атрибути (attributes) — властивості ПЗ, що показують наближення до мети;

— метрики (metrics) — кількісні характеристики ступеня наявності атрибутів.

Виділено шість цілей: функціональність (functionality), надійність (reliability), практичність або зручність використання (usability), ефективність (efficiency), супроводжуваність (maintainability), мобільність (portability). В 2001р. цей стандарт був переглянутий і розширений. У нього було додано шість додаткових атрибутів якості: привабливість як атрибут практичності й ступінь відповідності стандартам. Повний список атрибутів якості ПЗ за стандартом ISO 9126: функціональність; надійність; практичність, зручність використання; ефективність; супроводжуваність; мобільність.

Рис. 2 - Представлення якості в стандарті ISO 9126

Розрізняються поняття **внутрішньої якості**, пов'язаної з характеристиками програмного забезпечення самого по собі, без урахування його поведінки; **зовнішнього якості**, що характеризує програмне забезпечення з точки зору його поведінки; і якості програмного забезпечення **при використанні** в різних контекстах - тієї якості, яку відчуває користувач при конкретних сценаріях роботи програмного забезпечення.

Для всіх цих аспектів якості введені метрики, що дозволяють оцінити їх. Крім того, для створення добротного програмного забезпечення істотною є якість технологічних процесів його розробки. Взаємини між цими аспектами якості за схемою, прийнятою ISO 9126, показано на рисунку.

У сфері управління якістю відомі три гуру Джозеф Джуран, Філіп Кросбі і Едвард Демінг. У кожного з них є своє визначення якості, але у всіх у них є спільне - це відношення клієнта до якості продукту.

Джозеф Джуран свого часу (1988р), ввів визначення якості як придатність до використання ("Фітнес для використання") - якість для замовника.

Другий елемент у визначенні Джураном якості полягає в продукції, вільної від дефектів. За Джураном, ці недоліки доставляють неприємності клієнтам і, отже, вони стають незадоволеними.

Визначення Джураном поняття якість відображає його тверду спрямованість на те, щоб задовольнити очікування клієнта.

Філіп Кросбі, визначає якість як відповідність вимогам («Вимоги сумісності», 1979). За Кросбі, якість або є, або її немає. Немає такого явища, як різні рівні якості.

Ми можемо визначити два види якості:

**Зовнішня якість** - якість для замовника (це зручність у використанні, відсутність помилок, хороша продуктивність і т.п.)

**Внутрішньо якість** - це якість для розробників програмного продукту (відповідність вимогам, зручна архітектура, простота зміни і т.п.).

Стандарт ISO 9126 пропонує використовувати для опису внутрішнього і зовнішнього якості програмного забезпечення багаторівневу модель. На верхньому рівні виділено 6 основних характеристик якості програмного забезпечення. Кожна характеристика описується за допомогою декількох атрибутів, що входять до неї. Для кожного атрибута визначається набір метрик, що дозволяють його оцінити. Безліч характеристик і атрибутів якості згідно ISO 9126 показано на рис. 6.



Рис. 3 - Характеристики і атрибути якості програмного забезпечення згідно ISO 9126.

Визначення цих характеристик і атрибутів подано в стандарті ISO 9126: 2001.

Перераховані характеристики і атрибути якості програмного забезпечення дозволяють систематично описувати вимоги до нього, визначаючи, які властивості програмного забезпечення по даній характеристиці хочуть бачити зацікавлені сторони.

Наведені атрибути якості визначені в стандартах, але це не означає, що вони цілком вичерпують поняття якості програмного забезпечення. Так, в стандарті ISO 9126 відсутні характеристики, пов'язані з мобільністю програмного забезпечення (мобільність), тобто здатністю програми працювати при фізичних переміщеннях машини, на якій вона працює.

Замість надійності багато дослідників воліють розглядати більш загальне поняття добротності (dependability), що описує здатність ПЗ підтримувати визначені показники якості за основними характеристиками (функціональності, продуктивності, зручності використання) із заданими ймовірностями виходу за їх рамки та визначеним максимальним збитком від можливих порушень. Крім того, активно досліджуються поняття зручності використання, безпеці й захищеності ПЗ, - вони здаються більшості фахівців набагато більш складними, ніж це описується даним стандартом.

**Питання для самоконтролю:**

1. Що таке якість програмного продукту?
2. Яке місце тестування в життєвому циклі розробки ПЗ.
3. Що входить в модель якості ПЗ ISO 9126?
4. Що таке внутрішня і зовнішня якість?
5. Назвіть основні характеристики і атрибути якості?

**Лекція 2. Рівні і види тестування**

**План лекції:**

1. **Фази, стадії та види тестування.**
2. **Техніка тестування.**
3. **Аксіоми тестування.**
4. **Місце тестування в циклі розробки програмного забезпечення.**
5. **Принципи тестування програмного забезпечення.**

В процесі розробки ПЗ тестування ПЗ зазвичай відбувається на декількох рівнях інтеграції: поблочне тестування, перевірка взаємодії (інтеграційне тестування) та системне тестування.

Відповідно до етапів розробки ПЗ прийнято виділяти три фази тестування: модульне, інтеграційне і системне.

**Модульне тестування**, тестування модуля, або автономне тестування (module testing, unit testing) — контроль окремого програмного модуля, зазвичай в ізольованому середовищі (тобто ізольовано від решти всіх модулів). Під модулем розуміється логічно замкнутий фрагмент програми, який може бути викликаний через його інтерфейс. Модуль перевіряється на відповідність своїм специфікаціям і внутрішню логіку.

**Інтеграційне тестування** або тестування взаємодій (integration testing) — контроль взаємодії між частинами системи (модулями, компонентами, підсистемами).

**Системне тестування або комплексне тестування** (system testing) — контроль та/або випробування всього програмного забезпечення, як повної системи, в цільовому середовищі, тобто підтвердження того, що доступ до всіх компонентів системи і взаємодія з ними несуперечливі і передбачені згідно специфікацій системи.

Вочевидь, що фази не є взаємозамінними і, наприклад, проведення модульного тестування не гарантує правильності інтеграційного тестування, бо правильність функціонування окремих компонент не гарантує правильності їх взаємодії, як між собою, так і з системою в цілому.

**Альфа- і бета- тестування.**

Як правило, перед тим, як вважатися завершеним, програмне забезпечення проходить дві стадії тестування. Перша стадія називається альфа-тестуванням.

**Альфа-тестування** (alpha testing) — використання майже готової версії продукту (як правило, програмного або апаратного забезпечення) штатними програмістами (розробниками і тестерами) з метою виявлення помилок в його роботі для їх подальшого усунення перед бета-тестуванням. По закінченню тестування альфи, розробка входить в стадію бети.

**Бета-тестування** (beta testing) — інтенсивне використання ПЗ з метою виявлення максимальної кількості помилок в його роботі для їх подальшого усунення перед остаточним виходом (випуском) продукту на ринок, до масового споживача.

На відміну від альфа-тестування, що проводиться силами штатних розробників або тестувальників, бета-тестування припускає залучення добровольців з числа звичайних майбутніх користувачів продукту, яким розсилається згадана попередня версія продукту (так звана бета-версія). Такими добровольцями (їх називають бета- тестерами) зазвичай керує цікавість до нового продукту — цікавість, задля задоволення якої вони цілком згодні миритися з можливістю випробувати наслідки ще незнайдених (а тому і невиправлених) помилок.

Крім того, бета-тестування може використовуватися як частина стратегії просування продукту на ринок (наприклад, безкоштовна роздача бета-версій дозволяє привернути широку увагу споживачів до остаточної дорогої версії продукту), а також для отримання попередніх відгуків про нього від широкого круга майбутніх користувачів.

**Тестування, що засноване на вимогах** (Requirement based testing) – тестування кожного припущення з певного документу (документи технічної підтримки, посібники та інша документація користувача).

**Регресійне тестування** (regression testing, від лати. regressio — рух назад) — спільна назва для всіх видів тестування програмного забезпечення, метою яких є виявлення помилок у вже протестованих ділянках початкового коду. Такі помилки — коли після внесення змін до програми перестає працювати те, що повинне було продовжувати працювати — називають регресійними помилками (regression bugs).

Зазвичай використовувані методи регресійного тестування включають повторні прогони попередніх тестів, а також перевірки, чи не потрапили регресійні помилки в чергову версію в результаті злиття коду. З досвіду розробки ПЗ відомо, що повторна поява одних і тих самих помилок — випадок доволі поширений. Іноді це відбувається із-за слабкої техніки управління версіями або унаслідок людської помилки при роботі з системою управління версіями. Але настільки ж часто вирішення проблеми буває таким, що «недовго живе»: після наступної зміни в програмі воно перестає працювати. І нарешті, при переписуванні якої-небудь частини коду, часто спливають ті ж помилки, що були в попередній реалізації.

Тому найкращім рішенням є створення тесту на помилку, при її виявленні, з метою його використання при подальших змінах програми. Хоча регресійне тестування може бути виконане і вручну, але найчастіше це робиться за допомогою спеціалізованих програм, що дозволяють виконувати всі регресійні тести автоматично. У деяких проектах навіть використовуються інструменти для автоматичного прогону регресійних тестів через заданий інтервал часу. Зазвичай це виконується після кожної вдалої компіляції (у невеликих проектах) або щоночі, або кожного тижня.

У термінології тестування поняття «тестування білого ящика» і «тестування чорного ящика» відносяться до того, чи має розробник тестів доступ до початкового коду тестованого ПЗ, або ж тестування виконується через призначений для користувача інтерфейс або прикладний програмний інтерфейс, наданий тестованим модулем.

При тестуванні **чорного ящика** (black-box testing), тестер має доступ до ПЗ тільки через ті самі інтерфейси, що і замовник або користувач, або через зовнішні інтерфейси, що дозволяють іншому комп'ютеру або іншому процесу підключитися до системи для тестування.

Можна виділити наступні види тестування чорного ящика:

**Тестування за класами еквівалентності** (Equivalence class testing). Клас еквівалентності - це множина значень змінної, що, за припущенням, є еквівалентними. Контрольні приклади є еквівалентними, якщо виконується наступна сукупність вимог:

1) вони всі перевіряють один об’єкт;

2) якщо один с них „спіймає‖ дефект, то інший також;

3) якщо один з них не виявляє дефект, то інший, скоріше за все, також цього не зробить.

Якщо такий клас еквівалентності виявлено, то треба використовувати для тестування лише один з його членів.

Способи вибору таких представників також визначають певний вид тестування:

1) **Граничне тестування** (Boundary testing). Граничні значення це найбільші та найменші значення класів еквівалентності. При тестуванні граничних значень тестуються також значення менше за мале, та більше за велике.

2) **Тестування кращих представників** (Best representative testing). Тестування значень, що найбільш вірогідно призведуть до виявлення дефекту. Значення завжди можна змінити різними шляхами. Треба покрити всі можливі варіанти.

При **тестуванні білого ящика** (white-box testing, також говорять — прозорого ящика), розробник тесту має доступ до початкового коду і може писати код, який пов'язаний з бібліотеками тестованого ПЗ. Це типово для модульного тестування (англ. unit testing), при якому тестуються тільки окремі частини системи. Воно забезпечує те, що компоненти конструкції — працездатні і стійкі, до певного ступеня.

При тестуванні білого ящика розрізняють наступні види:

**Тестування логіки** (Logic testing). Багато програм мають логіку типу „якщо, то”, тестування логіки тестує можливі сценарії та комбінації.

**Тестування станів** (State-bases testing). Робота кожної програми - це переходи з одного стану в інший. Тестування станів має на меті уважну перевірку коректності роботи програми у кожному її стані.

**Тестування покриття операторів та гілок** (Statement and branch coverage). Сто відсоткове покриття має місце коли покриті всі рядки та всі твердження коду. В іншому випадку береться до уваги покриття у процентному відношенні.

**Тестування шляхів** (Path testing). Тестування набору шляхів (під-шляхів), що проходить програма.

**Техніка тестування**

Існує багато прийомів тестування ПЗ. Деякі з них перевершують інші, деякі можна використовувати сукупно з іншими для кращих результатів. Нижче представлений список основних прийомів тестування:

- **Ручне тестування** – тести виконуються людьми з наперед складеними, або визначеними для кожного випробування тестовими даними.

- **Автоматизоване тестування** – тести виконуються спеціальними інструментами або самостійними процесами і можуть повторюватися багато раз. Тестові дані попередньо вводяться або генеруються.

**- Регресивне тестування** – тести, зазвичай автоматизовані, мають на меті виявлення негативного впливу змін в програмі на функції, що пройшли попередні перевірки.

- **Димове тестування** – тести, спрямовані на швидку перевірку базової функціональності, з метою виявлення, чи новий білд (версія програми чи її певного модуля) вартий тестування.

- **Дослідницьке тестування** – тести, що виконуються за відсутністю специфікацій. Тестер розроблює власну систему тестування, яка базується на накопиченому їм досвіді та оцінює ризики, створюючи сценарії тестування.

- **Мавпяче тестування** - тести, що не мають під собою певної системи, «швидка атака» програми тестером.

- **Стрес-тестування** – тести призначені для перевірки стійкості програми до надмірного навантаження при нестачі ресурсів

- **Тестування навантаження** – тести виконуються при різних рівнях навантаження з метою перевірити поведінку програми та виявити максимально дозволений рівень.

- **Тестування продуктивності** – тести виконуються для порівняння поточної продуктивності з розрахунковою.

- **Тестування інсталяції** – тести полягають у встановленні програми на різних платформах-комбінаціях та перевірки: чи всі файли було переписано, чи працює програма коректно.

- **Тестування довгим використанням** – тести виконуються довготривало, з метою виявлення такого роду помилок, що неможливо виявити при короткому використанні (наприклад, помилки при роботі з динамічним розподілом пам'яті).

**Аксіоми тестування**

З моменту виявлення першого багу, тестування програмного забезпечення пройшло великий шлях. Як всякий новий практичний напрям, воно динамічно розвивалося, не уникнувши і тупикових гілок, невдалих спроб адаптації та перенесення методологій, стандартів і концепцій з вже існуючих областей. Додатковою особливістю цього процесу стала залежність тестування від власне програмного забезпечення, чиї технології, методи та інструменти самі переживають період стрімкого й інтенсивного вдосконалення. Також слід відзначити, що не маючи за спиною багатого досвіду теоретичних досліджень, система забезпечення якості ПЗ, а услід за нею і тестування, протягом довгого часу обростали всілякими міфами і попадали під вплив різних ідейних течій. Мета даного розділу – розвіяти деякі з ілюзій, пов’язаних з тестуванням.

**Неможливо повністю протестувати програму**

Початківець у сфері тестування може вважати, що можна обробити ПЗ повністю протестувавши його, знайшовши всі помилки, і підсумувавши, що ПЗ ідеальне. Нажаль, це неможливо, навіть для найпростіших програм, через наступні чотири ключові причини:

1. Кількість можливих вхідних даних дуже велика.

2. Кількість можливих результатів дуже велика.

3. Кількість проходів по ПЗ дуже велика.

4. Специфікація ПЗ суб'єктивна. Можна сказати, що помилка – це зовсім не помилка, так і було задумано.

Підсумувавши ці причини, отримуємо набір умов тестування, який занадто великий, щоб його здолати. Ілюстративним прикладом може служити калькулятор Wіndows.

Приклад: Нехай поставлене завдання на тестування калькулятору Mіcrosoft Wіndows. Почнемо з операції додавання. Крок перший: 1+0=? Одержуєте відповідь 1. Коректно Крок другий: 1+1=? Одержали 2. Коректно Скільки таких кроків потрібно? Калькулятор приймає 32- х розрядне число, отже, необхідно перевірити всі можливі варіанти до 1+99999999999999999999999999999999 =. Закінчивши цю послідовність, треба перейти до 2+0=, 2+1=, 2+2= та інш. Останнім варіантом буде: 99999999999999999999999999999999+999999999999999999999999999 99999= Далі необхідно перевірити коректність додавання дробі: 1.0+0.1,1.0+0.2, та інш.

Після перевірки того, що коректні дані додаються правильно, залишається ввести некоректні дані. Це дасть змогу переконатися, що вони грамотно оброблюються. Нагадаємо, що можна використовувати не лише цифри на екрані, але й набирати символи з клавіатури. Важливо перевірити можливість введення 1+a, z+1, 1a1+2?>2,.... Скільки буде таких варіантів?

Виправлення введення також повинно бути протестовано. Калькулятор Wіndows дозволяє використовувати Backspace і Delete, отже, належить перевірити і їх.

Kbackspace>2+2 повинно дорівнювати 4.

Всі вищезазначені перевірки мають бути повторені з використанням клавіші Backspace для кожного введення, для кожних двох введень і так далі.

Після закінчення цього етапу можна перейти до додавання трьох чисел, потім чотирьох і т. д.

Це приклади лише для операції додавання, а залишаються ще різниця, множення, ділення, відсоток, корінь квадратний та інш. Вочевидь, що через кількість контрольних прикладів неможливо завершити повне тестування навіть такої програми як калькулятор. Перейдемо до розгляду наступної аксіоми.

**Тестування – це процес, що містить ризики**.

Попередній приклад показав, що перебрати всі варіанти неможливо, треба чимось нехтувати. Якщо приймається рішення не тестувати всі можливі сценарії, то вибирається деякий ризик. У прикладі з калькулятором, що буде, якщо вирішити не перевіряти 1024+1024=2048? Існує ймовірність того, що програміст зробив помилку, яка впливає саме на цю ситуацію. Якщо не протестувати її, користувач рано чи пізно з нею стикнеться. Ця помилка може коштувати дуже дорого, адже вона буде знайдена, коли ПЗ вже знаходиться в експлуатації, а тому на вилучення та заміну версії, що містить баг, мають бути витрачені значні зусилля.

Таким чином, маємо протиріччя – неможливо протестувати все, проте якщо не протестувати, то виникає ймовірність пропущення помилок. Перед розробником стоять суперечливі цілі - продукт повинен бути реалізований, отже, необхідно закінчити тестування, але якщо закінчити його занадто швидко, то залишаться не протестовані частини.

Тестеру необхідно навчитися скорочувати величезну область всіх можливих тестів до керованого набору та приймати, враховуючий ризик, розумні рішення: що важливо для тестування, а що ні.

На рис. 2 проілюстровано залежність між обсягом тестування та якістю проекту у сенсі кількості знайдених помилок. Якщо спробувати протестувати абсолютно все, ціна різко зросте, а кількість пропущених помилок спаде до нуля.



Рис. 4 - Кожен проект має оптимальний обсяг тестування

Якщо сильно скоротити тестування або приймати хибні рішення відносно того, що саме тестувати, ціна зменшиться, але залишиться велика кількість помилок . Мета - це знайти оптимальний обсяг тестувань.

**Тестування не може показати, що помилок немає.**

Ця аксіома була висунута професором Е. Дейкстрой в 1972 [1]. Проілюструємо її. Уявіть роботу винищувача, що шукає в будинку жуків. Він перевіряє будинок і знаходить їх - можливо живих, мертвих або навіть гніздо. Можна спокійно стверджувати, що в будинку є жуки. Потім винищувач відвідує інший будинок, та не знаходить в ньому присутності жуків. Він дивиться у всіх передбачуваних місцях, але не виявляє ніяких ознак навали. Можливо, буде знайдено кілька мертвих жуків або старих гнізд, але нема нічого, що могло б свідчити про присутність живих жуків. Чи можна однозначно, впевнено заявити, що в будинку жуків немає? Все, що можна сказати – це те, що пошук не виявив жуків. І поки не буде розібрано будинок на частині аж до фундаменту, не можна бути впевненими в тому, що жуків просто випадково не помітили.

Тестування ПЗ дуже схоже на пошук жуків. Воно може показати наявність жуків, але не може показати, що їх немає. Можна провести тестування, знайти й доповісти про помилки, але не можна стверджувати, що їх більше немає. Можна тільки продовжити тестування й, по можливості, знайти інші помилки.

**Чим більше помилок знаходить тестер, тим більше їх існує**

Існує багато подібностей між реальними жуками й несправностями (bugs) у ПЗ. І ті, і інші мають тенденцію з'являтися групами - якщо була помічена одна, імовірно поблизу є ще. Часто тестер досить довго не може нічого знайти. Потім він раптом знаходить одну помилку, потім дуже швидко наступну й наступну. От кілька причин цього:

- Як і в усіх нас, у програмістів бувають невдалі дні. Код, написаний в один день, може бути ідеальним; код написаний в 30 іншій - недбалим. Одна помилка може бути маячком, що показує - тут поруч є ще.

- Програмісти часто роблять ті самі помилки. В усіх є звички. Програміст, схильний до певних помилок, буде часто повторювати їх.

- Деякі помилки дійсно тільки вершина айсбергу. Дуже часто проект або архітектура ПЗ мають фундаментальні проблеми. Тестер знаходить помилки, які на перший погляд здаються не зв'язаними, але, в остаточному підсумку, указують на одну серйозну первинну причину.

Важливо відзначити, що зворотне твердження до "помилки випливають за помилками" також правдиво. Якщо тестер, попри всі зусилля, не зміг знайти помилки, це, з великою ймовірністю, значить, що ПЗ було акуратно написане, у ньому міститься всього кілька помилок, які можна знайти.

**Парадокс пестицидів**

В 1990 році Борис Бейзер у своїй книзі "Техніки тестування програмного забезпечення" запропонував термін парадокс пестицидів - чим більше тестер тестує ПЗ, тим більше воно стає невразливим до тестувань [11]. Аналогічне відбувається з комахами при використанні пестицидів (Рис. 3). Якщо використовувати ті самі пестициди, у комах виробляється захист, і пестициди більше не діють.

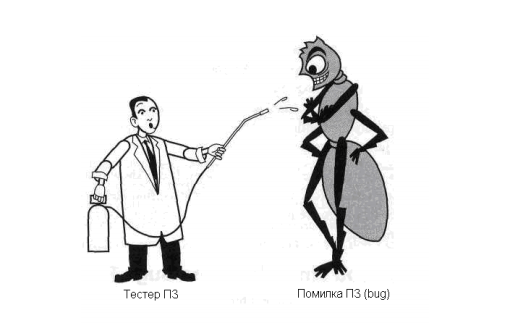


Рис. 5 - ПЗ під дією однакових повторюваних тестів стає стійким до них.

Для подолання парадокса пестицидів, тестери ПЗ повинні писати нові, різноманітні тести, що дозволить знаходити більше помилок.

**Не всі знайдені помилки будуть виправлені**

Одна із реалій тестування ПЗ полягає в тому, що, навіть після наполегливої роботи, не всі помилки будуть виправлені. Це не означає, що тестер не досяг своєї мети, або що команда в цілому випустила поганий продукт. Іноді треба знаходити компроміс та йти на ризик, приймаючи рішення щодо того виправляти помилку, чи ні. Наведемо деякі причини, з яких помилка може бути не виправленою:

- **Недостатньо часу.** У кожному проекті завжди є багато специфікацій та нюансів (іноді занадто багато, що б написати до них код і протестувати його у зазначений час) і не досить можливостей, що б закінчити їх усі.

- **Це насправді не помилка.** Часто від розробників можна почути: "Це не помилка, це - властивість!" Це незвично, але помилки тестування або зміни у специфікації можуть призвести до того, що несправності будуть залишені, як властивості.

- **Занадто ризиковано виправляти.** Нажаль, саме так буває дуже часто. ПЗ іноді схоже на спагетті: виправлення однієї помилки може спричинити виникнення нових. Під тиском реалізації та щільного графіка може бути занадто ризиковано змінювати ПЗ. Тому краще залишити відому помилку, чим ризикувати створити нову, невідому.

- **Це просто не варто виправляти.** Можливо, це звучить грубо, але це реальність. Помилки, які виникають нерегулярно або в мало використовуваних функціях можуть бути опущені. Причина цього – бізнес рішення, що базуються на ризику.

Для прийняття рішень звичайно необхідні тестери ПЗ, керівник проекту та програмісти. Кожний піклується про своє бачення майбутнього помилки, має свої дані та думку - чому треба або не треба її виправляти.

Що відбувається, коли приймається неправильне рішення?

Можна згадати рішення Іntel Pentіum, коли тестери Іntel знайшли помилку при обробці чисел з плаваючою точкою раніше, ніж чип був випущений, але команда проектувальників вважала, що вона є незначною і її не обов'язково виправляти. У них був дуже тісний графік, тому вони вирішили продовжувати проект за планом, а знайдену помилку виправити в наступних версіях. На жаль, помилка була виявлена користувачами, і компанія понесла великі фінансові збитки.

**Іноді складно сказати чи є помилка помилкою**

Якщо в ПЗ є проблема, але її не було знайдено ні програмістами, тестерами, ні навіть користувачами – чи помилка це?

Якщо задати це питання групі тестерів, то можна побачити вражаючу дискусію. У кожного з них буде своя власна думка й кожний зможе добре та яскраво її аргументувати. Проблема в тому, що певної відповіді на це питання не існує. Вона полягає у тому, що на певний час краще для тестерів та команди розроблювачів.

Твердження, що програмне забезпечення робить або не робить "чогось" припускає, що воно було запущено, "щось" протестовано або ж недолік "чогось" був очевидний. Пізніше не можливо буде доповісти про те, чого не видно та не можна відтворити. Визнавайте помилку помилкою, тільки якщо вона спостерігається.

Про це можна подумати й по-іншому. Зовсім не дивно, що дві людини мають зовсім різні думки про якість ПЗ. Один може сказати, що програма містить жахливо багато помилок, інший - що вона ідеальна. Як обоє можуть бути праві? Відповідь проста - один користується програмою так, що з'являються помилки, а інший – що ні.

**Специфікація розробки ніколи не завершується**

Спочатку необхідно розібратися з терміном: специфікація розробки.

Специфікація розробки, іноді називана просто специфікацією (англ. product specіfіcatіon, product spec, spec) – це домовленість між членами команди розроблювачів ПЗ. Вона визначає продукт, що вони створюють, деталізує, яким він буде, як буде працювати, що буде робити й чого не буде. Ця домовленість може варіюватися: від простої усної згоди до формалізованого записаного документа.

У розробників ПЗ є проблема. Індустрія розвивається так швидко, що найпередовіші розробки минулого року в цьому вже є застарілими. ПЗ стає більшим, складнішим й містить у собі все більше функцій, отже, як результат, подовжуються й подовжуються списки для його розробки.

Не існує іншого шляху реагування на настільки швидкі зміни. Припустимо, що продукт має закриту, закінчену і не підлягаючу зміні специфікацію. На середині дворічного циклу виробництва продукту А, головний супротивник випускає дуже схожий продукт В з декількома корисними властивостями, яких немає в А. Що робити далі с продуктом А? Продовжувати роботу за специфікацією й випустити через рік другосортний продукт? Або перегрупувати команду, переписати специфікацію, і працювати над виправленою розробкою? У більшості випадків, розумно останнє.

Тестер ПЗ повинен засвоїти, що специфікація буде мінятися. Властивості й функції будуть додаватися, не заважаючи на те, що спочатку вони не мали бути протестовані. Вони будуть змінюватися або взагалі видалятися, хоча вже були протестовані та позбавлені частини помилок.

**Тестери ПЗ не самі популярні члени команди розробників**

Мета тестера ПЗ – якомога раніше знаходити помилки і робити так, щоб вони були виправлені. Тобто робота тестера полягає в тому, що він змушений перевіряти та контролювати своїх колег, знаходити їхні проблеми і оголошувати їх. От кілька варіантів, як підтримувати мир із членами команди:

- **Знаходити помилки як можна раніше.** Набагато меншим збитком для всіх і набагато більшим плюсом тестеру буде, якщо він знайде серйозну помилку за три місяці, а не за день, до випуску програми.

- **Стримувати захват.** Добре, якщо тестер дуже полюбляє свою роботу та приходить у захват, коли знаходить серйозну помилку. Але, якщо він увірветься до кімнати програмістів з яскравою посмішкою й скажіть їм, що знайшов в їхній частині коду найжахливішу помилку із всіх, вони не будуть дуже щасливі.

- **Приносити не тільки погані новини.** Якщо завжди казати погане, то згодом люди почнуть уникати тестера, щоб не отримувати поганих звісток. Тому, якщо він знайшов шматочок коду без помилок, то буде краще сказати усім про це. Зрідка можна заходити до програмістів взагалі просто задля того, щоб побалакати.

**Місце тестування в циклі розробки програмного забезпечення**

В попередніх розділах йшлося про помилки програмного забезпечення (ПЗ). Дуже часто називаючи усі проблеми ПЗ «помилками» (bugs), тепер треба розкрити суть цього поняття.

Повернемося до визначення некоректної роботи ПЗ:

1) ПЗ не робить чогось, що, відповідно до специфікації, воно повинне робити.

2) ПЗ робить щось, що, відповідно до специфікації, воно не повинне робити.

3) ПЗ робить щось, що не згадується в специфікації.

4) ПЗ не робить чогось, що не згадується в його специфікації, але повинне.

5) ПЗ складно зрозуміти, важко використовувати, воно повільне, або - на думку тестерів ПЗ - буде сприйнято кінцевими користувачами, як явно не правильне.

Проілюструємо роботу цих правил на прикладі калькулятора:

Нехай у специфікації калькулятора зазначено, що він має правильно виконувати додавання, віднімання, множення та ділення. Якщо тестер, одержавши калькулятор, натискає кнопку «+», і нічого не відбувається, то це помилка відповідно до правила №1. Якщо надана відповідь буде неправильною, то це теж буде помилка №1.

Специфікація може містити вимогу, що калькулятор ніколи не повинен зависати. Якщо тестер щось увів, і калькулятор перестав реагувати на всі подальші дії, це помилка відповідно до пункту №2.

Припустимо, тестер одержав калькулятор на тестування й виявив, що крім додавання, віднімання, множення й ділення, він також містить квадратний корінь. Ніде в специфікації цього не було - честолюбний програміст просто додав його, оскільки вирішив, що це буде корисна функція. Це не функція - це скоріш за все помилка відповідно до правила №3.

Правило №4, можливо, звучить дивно з подвійним запереченням, але його завдання – вловити нюанси, які були упущені в специфікації. Тестер почав тестувати калькулятор і виявив, що коли батарея сідає, він перестає рахувати правильно. Ніхто ніколи не замислювався, як повинен поводитися калькулятор у таких випадках. Поганим рішенням була б розробка ПЗ з розрахунком на те, що батарея повинна завжди бути повністю зарядженою. Тестер очікував, що зможе продовжити свою роботу, доти батарея не сяде остаточно, або хоча б доти там не залишиться зовсім трошки заряду. Калькулятор рахує не правильно з батареєю, що розряджається, але ніде не записано, як він повинен поводитися в такій ситуації. Це помилка відповідно до правила №4.

Правило №5 дуже узагальнене. Тестер – перша людина, що користується програмою. Якщо це будете не він, то першими користувачами стануть покупці. Якщо тестер знайде щось, що здасться йому не правильним, не важливо чому, - це помилка. У випадку з калькулятором, можливо, він знайде, що кнопки замалі, через що натискати їх незручно. Можливо, при яскравому освітленні складно розібрати, що на екрані. Все це помилки відносно правила №5.

Зауважимо, що кожна людина, яка використовує ПЗ, буде мати різні сподівання і свою думку що до того, як воно повинно працювати. Це неможливо - написати ПЗ, яке всі користувачі назвуть ідеальним. Тестер ПЗ повинен завжди пам'ятати про це, застосовуючи правило №5 у своїй роботі.

**Чому виникають помилки?**

Треба зауважити, що більшість помилок виникають не через помилки програмістів. Аналіз багатої кількості розробок виявив: головна причина виникнення помилок у ПЗ – це специфікація (Рис. 4).

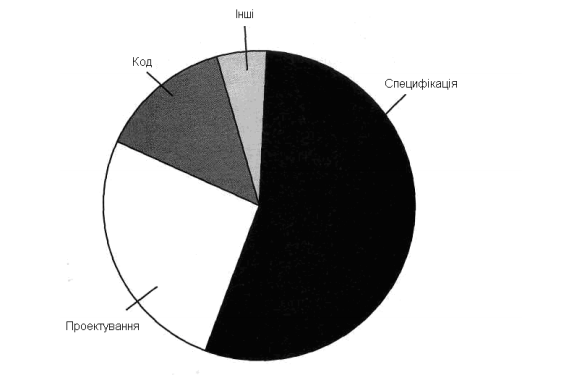


Рис. 6 - Помилки виникають на багатьох етапах розробки, але найбільше – на етапі специфікації.

Є кілька причин, які роблять специфікацію основним "постачальником" помилок. Інколи її просто не існує, інколи – вона неповна, або занадто часто змінюється. Написання та аналіз специфікацій необхідна та найголовніша частина циклу розробки ПЗ. Якщо цей етап пропущено чи здійснено некоректно, помилки будуть виникати.

Наступним джерелом виникнення помилок є проектування. Неправильно розроблена архітектура ПЗ через поспіх, непродуманість, відсутність досвіду, непослідовність та ін. може мати величезні наслідки

Помилки в коді виникають через складність ПЗ, бідність документації, високу щільність графіка, або випадково. Важливо відзначити, що у більшості випадків помилки, які виникли на поверхні, указують на проблеми специфікації та проектування.

Остання категорія містить у собі все, що залишилось. Деякі помилки можуть виявитися помилками, що з якихось причин були прийняті за них, але ними не були. Можуть бути повторювані помилки, скопійовані із одного джерела. Деякі помилки можуть вказувати на неправильне тестування. В остаточному підсумку, ці помилки зазвичай становлять настільки малий відсоток від інших, що не варто особливо хвилюватися з цього приводу. 3.2

**Ціна помилок**

ПЗ не виникає миттєво – в його основі лежать планування й методики процесу розробки. Починаючи з виникнення ідеї, далі в процесі планування, програмування й тестування та закінчуючи використанням цього програмного забезпечення користувачами, потенційно можуть бути знайдені помилки.

На рис. 5 проілюстровано, скільки коштує виправити помилку, знайдену на різних етапах.

Ціна логарифмічна - тому, із часом вона зростає десятикратно. Помилка, виправлена на самому початку, тобто на етапі специфікації, нічого не коштує, або 10 центів, як у прикладі. Та ж помилка, не знайдена до написання коду й тестування ПЗ, може коштувати від $1 до $10. А якщо її знайдуть користувачі, то вартість виправлення легко перевищить $100.

Наглядним прикладом є випадок з компанією Дісней, а саме її грою «Король лев»[10].

Восени 1994 року, компанія Дісней реалізувала на компакт- дисках свою першу мультимедійну гру для дітей «Король Лев, анімована збірка розповідей» (The Lion King Animated Storybook). Не дивлячись на те, що багато інших компаній займалися продажем дитячих програм роками, це була перша розробка Діснея, що поступала на ринок, тому вона була відмінно представлена і розрекламована. Продажі були величезні – це була «гра, яку варто купити для дітей на свята». Проте те, що трапилося пізніше, було колосальним розгромом. 26 грудня, тобто на наступний день після Різдва, телефони підтримки покупців Діснея захлинулися від дзвінків розлючених батьків з дітьми, які не могли змусити програму працювати.

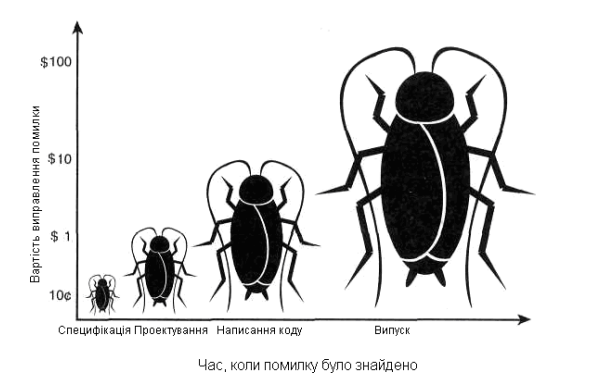


Рис. 7 - Ціна виправлення помилки.

Виявилось, що Дісней не протестував належним чином свою програму на різних версіях ПК, представлених на ринку. Програма працювала всього в декількох системах – подібних тій, яку розробники Діснея використовували для створення гри, - але не в найбільш поширених, які були у більшості покупців.

Якби на етапі аналізу вимог було досліджено, які ПК користуються попитом, та вписано в специфікацію, що гра повинна розроблятися і тестуватися на їх конфігурації, помилки б не виникло. Якщо це не було зроблено на етапі специфікації, тестери мусили б зібрати всі популярні моделі ПК і протестувати на них гру. Тоді вони б знайшли помилку, але вона, безумовно, коштувала б дорожче, тому що ПЗ було вже написано, налагоджене й протестовано. Команда розроблювачів також могла зробити бета тест і розіслати першу версію ПЗ невеликій кількості користувачів. Ці користувачі, обрані представляти величезний ринок, із задоволенням знайшли б помилку. Як виявилося, незважаючи ні на що, помилка лишилась непоміченою, а багато тисяч дисків були створені й продані. Компанія Дісней, в решті решт, сплачувала витрати на телефонну підтримку користувачів, повернення продукту, заміну дисків та інш. Отже, можна побачити, як легко втратити весь дохід від розробки, якщо серйозна помилка дійде до користувачів.

**Принципи тестування програмного забезпечення**

Принципи і правила грають важливу роль у всіх інженерних дисциплінах. Принципи тестування важливі для спеціалістів/ інженерів-тестерів, оскільки вони забезпечують основу для надбання знань і навичок в цій сфері.

Слово «принцип» має декілька значень:

- загальний або фундаментальний закон, доктрина або допущення;

- правило або кодекс поведінки;

- закони або факти, які складають основу роботи штучного пристрою.

Застосовуючи ці визначення до сфери розробки ПЗ, можна сказати, що її принципи лежать в законах, правилах і доктринах, що відносяться до систем ПЗ, способів їх побудов та їх поведінці. У сфері ПЗ до принципів також можуть бути віднесені правила і кодекси поведінки професіоналів, що проектують, розробляють, тестують та управляють системами ПЗ. Тестування, як етап розробки ПЗ, також має особливий набор принципів необхідних для тестера. Вони вказують, як перевіряти системи ПЗ, і визначають правила поведінки тестера-професіонала. Гленфорд Майєрс в своїй книзі «Мистецтво тестування ПЗ»[12] виділив список обов'язкових принципів. Вони описані нижче. Принципи, що представлені в таблиці 1 праворуч, взяті з оригінального списку Майєрса. Ліворуч представлені видозмінені правила відповідно до еволюції тестування від мистецтва до одного з процесів інженерної дисципліни, що безпосередньо відноситься до якості [10]. Треба зауважити, що описані нижче принципи відносяться лише до тестування, заснованого на виконанні ПЗ. Принципи, які відносяться до переглядів, сертифікації, доказу коректності в рамках тесту не розглядаються. Більшість з приведених принципів може здаватися очевидною, тим не менше, часто про них забувають.

Таблиця 1. Принципи тестування програмного забезпечення

|  |  |
| --- | --- |
| Нові принципи | Оригінальні принципи |
| Принцип 1: Тестування – це процес випробування програми за допомогою вибраного набору варіантів тесту з метою: 1) виявити помилки, 2) оцінити якість. | Принцип 1: Необхідна складова контрольного прикладу – визначення очікуваного результату та вихідних даних. |
| Принцип 2: Якщо мета тесту – визначити помилки, то якісний варіант тесту той, що найімовірніше виявить незнайдену досі помилку. | Принцип 2: Програмісту слід уникати спроб тестувати свою власну програму. |
| Принцип 3: Результати тесту повинні бути ретельно перевірені. | Принцип 3: Компанії, яка займається розробкою коду, не слід тестувати свої власні програми. |
| Принцип 4: Контрольний приклад повинен містити очікуваний результат та вихідні дані. | Принцип 4: Треба ретельно досліджувати результати кожного тесту. |
| Принцип 5: Тести повинні розроблятися як для коректних вхідних даних, так і для некоректних. | Принцип 5: Тести повинні писатися як для некоректних і неочікуваних вхідних даних, так і для коректних та очікуваних |
| Принцип 6: Існування більшого числа помилок пропорційне кількості знайдених помилок в компоненті. | Принцип 6: Дослідження програми на правильність функціонування – лише половина роботи. Друга половина – виявити зайву функціональність програми. |
| Принцип 7: Тестування повинне проводитися групою людей, незалежною від розробників | Принцип 7: Краще уникати одноразових варіантів тестів, якщо сама програма не одноразового використання. |
| Принцип 8: Тести повинні передбачати багаторазове виконання. | Принцип 8: Не слід планувати тестування, негласно вважаючи, що помилок немає. |
| Принцип 9: Тестування слід планувати. | Принцип 9: Існування більшого числа дефектів пропорційно кількості знайдених дефектів в компоненті. |
| Принцип 10: Слід включати процес тестування в життєвий цикл ПЗ. | Принцип 10: Тестування, це завдання, що вимагає творчості та розумової праці. |
| Принцип 11: Тестування, це завдання, що вимагає творчості та розумової праці. |  |

Проаналізуємо наведені принципи:

Принцип 1. Тестування – це процес випробування програми за допомогою вибраного набору варіантів тесту з метою 1) виявити помилки, 2) оцінити якість.

В оригіналі: Принцип 6: Дослідження програми на правильність функціонування – лише половина роботи. Друга половина – виявити зайву функціональність програми.

Цей очевидний принцип – одна з помилок, що найбільш часто зустрічаються, в тестуванні. Знову таки, це наслідок людської психології. Якщо очікуваний результат варіанту тесту не представлений, існує ймовірність, що правдоподібний, проте некоректний результат вважатимуть правильним через принцип «очі бачать те, що хочуть». Іншими словами, незважаючи на чітке визначення тестування, підсвідомо тестер бажає побачити правильний результат. Один із способів боротьби з цим – уважно перевіряти вихідні дані, заздалегідь точно сформулювавши очікуваний результат. Таким чином, варіант тесту повинен складатися з двох частин:

- Опис вхідних даних програми.

- Точний опис правильних вихідних даних при зазначених вхідних.

Розробники ПЗ набули значних навичок в запобіганні та знищенні дефектів та помилок. Тим не менш, останні трапляються і негативно впливають на якість програм. Для виявлення наявності дефектів до того, як програма буде випущена на ринок, необхідні тестери. Цей принцип підтверджує той факт, що тестування засноване на виконанні програми. Це і визначає відмінність тестування від відладки, метою якої є знаходження місця дефекту і його усунення. Термін «дефекти» використовується в цьому і наступних принципах для позначення будь-яких відхилень в програмі, які негативно впливають на її функціональність, продуктивність, надійність і/або інші атрибути її якості.

Тестування може бути розглянуте як «динамічний процес виконання програми із цінними вхідними даними». Такий підхід, разом з визначенням тестування, наданим вище, припускає, що тестування не лише виявляє дефекти, але і використовується для оцінки якості ПЗ. У такому разі тестер виконує програму, використовуючи контрольні приклади, для оцінки таких характеристик, як надійність, зручність використання, рівень продуктивності та інші. Результати тесту дозволяють порівняти поточну якість з рівнем, описаним в документації. Неможливість досягти необхідної якості або будь-які відхилення обов'язково повинні бути розглянуті.

Перейдемо до розгляду наступної пари принципів.

Принцип 2. Якщо мета тесту – визначити дефекти, то якісний варіант тесту той, що найімовірніше виявить незнайдений досі дефект.

В оригіналі: Принцип 8: Не слід планувати тестування, негласно вважаючи, що помилок немає.

Принцип 2 підтримує акуратну розробку тесту та задає критерій оцінки контрольного прикладу і ефективності тесту в тих випадках, коли мета – виявлення дефектів. Тестеру потрібно визначити ціль кожного контрольного прикладу, тобто виділити тип шуканого дефекту. Таким чином, тестер виконує перевірки аналогічно вченому, який проводить експеримент. У вченого існує гіпотеза, яку він повинен підтвердити або спростувати експериментальним шляхом. У тестера гіпотеза пов'язана з можливою наявністю конкретних типів дефекту. Мета тесту – довести/спростувати тезу, тобто визначити наявність/відсутність конкретного дефекту. На основі тези обираються вхідні і визначаються правильні вихідні дані тесту, після чого він проводиться. Результати аналізуються для доказу/спростування тези. Потрібно розуміти, що на процес тестування витрачається багато ресурсів, включаючи ресурси для розробки контрольних прикладів, їх виконання, запису і аналізу результатів. За допомогою ретельної розробки тесту відповідно до принципу 2 тестер може добитися мінімальної витрати ресурсів.

Розглянемо наступний принцип.

Принцип 3. Результати тесту повинні бути ретельно перевірені.

В оригіналі:

Принцип 4: Треба ретельно досліджувати результати кожного тесту.

Тестеру необхідно ретельно досліджувати й аналізувати результати тесту. Інакше можливі несприятливі наслідки. Наприклад:

- *Може бути пропущено критичну помилку, при цьому тест вважатиметься пройденим, хоча програма його провалила. Тестування може продовжитися з урахуванням неправильного результату, і, навіть, якщо дефект буде виявлено на пізнішій стадії тесту, його усунення виявиться важчим та більш ресурсоємним.*

*- Виникне припущення про критичну помилку, якої не існує. У такому разі тест вважатиметься проваленим. Значну кількість часу і ресурсів буде витрачено на виявлення неіснуючого дефекту. І лише повторна перевірка покаже, що помилки не було.*

*- Вихідні дані тесту на якість будуть інтерпретовані невірно, що може призвести до непотрібної переробки або пропуску критичної помилки.*

Отже, без відповідної уваги до результатів процес тестування вимагатиме більше фінансових та часових ресурсів.

Наступний принцип формулюється наступним чином:

Принцип 4. Контрольний приклад повинен містити очікуваний результат та вихідні дані.

В оригіналі: Принцип 1. Необхідна складова контрольного прикладу – визначення очікуваного результату та вихідних даних.

Зазвичай зрозуміло, що вхідні дані – це складова частина контрольного прикладу. Проте, доки відсутній докладний опис очікуваних вихідних даних або результатів, наприклад, значення конкретної змінної або підсвічування потрібної кнопки на панелі, контрольний приклад не може вважатися повноцінним. Очікувані вихідні дані дозволяють тестеру визначити:

1) чи виявлено дефект;

2) чи пройдено/провалено тест.

Дуже важливо чітко описувати вихідні дані, щоб не витрачати час на уточнення дрібниць. Визначення конкретних вхідних і вихідних даних повинно бути частиною процесу розробки тесту. В разі тестування на оцінку якості, має сенс описати цільовий рівень якості кількісними даними в спеціальному документі з вимогами. Це дасть тестерам змогу порівнювати поточну якість програми з тим рівнем, що вимагається.

Наступний принцип:

Принцип 5. Тести повинні розроблятися як для коректних вхідних даних, так і для некоректних.

В оригіналі: Принцип 5: Тести повинні писатися як для некоректних і неочікуваних вхідних даних, так і для коректних та очікуваних.

Тестер не повинен вважати, що програма на тесті повинна забезпечуватися лише коректними вхідними даними. З деяких причин це не так. Наприклад, користувачі можуть не бути обізнані про те, які дані можна вводити. До того ж, вони часто роблять друкарські помилки навіть за наявності повних/коректних вхідних даних. Некоректні вхідні дані також можуть бути наслідком несправності ПЗ чи системи в цілому. Використання контрольних прикладів із некоректними вхідними даними корисно для виявлення дефектів, оскільки у такому разі код виконується нестандартним способом і примушує програму поводитися несподівано. Некоректні вхідні дані також допомагають розробникам і тестерам оцінити надійність програми, її здатність відновлюватися (у разі помилкового введення).

Зауважимо, що представлена пара принципів підтверджує необхідність залучення до тестування незалежної групи тестерів, яка згадується у принципі 7, бо розробник програмного компоненту може бути схильний вибирати лише коректні вхідні дані для демонстрації працездатності програми. Незалежний тестер схильний обирати також і некоректні вхідні дані.

Перейдемо до розгляду наступної групи принципів:

Принцип 6. Існування більшого числа дефектів пропорційне кількості знайдених дефектів в компоненті.

В оригіналі:

Принцип 9: Існування більшого числа дефектів пропорційне кількості знайдених дефектів в компоненті.

Принцип стверджує, що чим більше дефектів вже знайдено в компоненті, тим імовірніше виявлення додаткових дефектів при подальшому тестуванні. Наприклад, якщо тестери знайшли 20 дефектів в компоненті А і 3 в компоненті В, імовірність існування додаткових дефектів в А більше, ніж у В. Дефекти часто виявляються в класах і в неякісно написаному коді з високим рівнем складності. За наявності таких компонентів розробники і тестери повинні вирішити, чи кинути поточну версію і переробити її або виділяти більше ресурсів на тестування доки компонент не відповідатиме вимогам якості. Такі випадки критичні для важливих компонентів, наприклад тих, що відповідають за безпеку.

Наступні принципи:

Принцип 7. Тестування повинне проводитися групою людей, незалежною від розробників.

В оригіналі цьому принципу відповідало два:

Принцип 2. Програмісту слід уникати спроб тестувати свою власну програму.

Принцип 3: Компанії, що займається розробкою коду, не слід тестувати свої власні програми.

Кожен письменник знає – або повинен знати – що намагатися виправляти або коректувати власну роботу – погана ідея. Ви пам’ятаєте, що мали на увазі в тій або іншій частині і можете не помітити, що вона містить ще щось. І вже точно ви не хочете шукати в своїй роботі помилки. Цей підхід стосується і авторів програм.

Більш того, тестування має на меті іншу задачу ніж написання коду – в його основі лежить «деструктивний» підхід: треба довести, що програма не працює. А програмісту, який написав програму, дуже складно переключитися на її випробування, що носять «руйнівний» характер.

Багато господарів знають, що видалення шпалер зі стін (руйнівний процес) – нелегка справа, яка перетворюється на муку, якщо шпалери були поклеєні власноруч. Аналогічно, більшість програмістів не в змозі ефективно тестувати власні програми, оскільки вони не можуть зосередитися на спробах знайти помилки. До того ж, програміст може підсвідомо уникати знаходження помилок через страх перед покаранням з боку керівництва, клієнта або власника програми, що розробляється. Додатково з цими психологічними перешкодами існує і ще одна значна проблема: програма може містити помилки через нерозуміння програмістом постановки задачі або специфікацій. В такому разі нерозуміння, ймовірно, пошириться і на тести. Це не означає, що програмісту заборонено тестувати свою програму. Проте процес буде ефективнішим і успішнішим, якщо ним займеться стороння людина.

Зауважимо, що ці аргументи не стосуються відладки (виправленню відомих помилок); її ефективніше проводити програмісту-автору, бо розбиратися в чужому коді значно складніше.

Потреба організації в незалежній групі, що тестує, може бути задоволена декількома способами. Група, що тестує, може бути виділена в повністю самостійний відділ організації. Альтернативою 51 цьому може стати група тестерів з групи забезпечення якості ПЗ (Software Quality Assurance Group) або група розробників, що спеціалізується на тестуванні, але в останньому випадку вона повинна бути максимально об'єктивною. Первинні обов'язки члена будь-якої з цих груп – тестування, а не розробка.

Нарешті, незалежність групи, що тестує, не має на увазі ворожих відносин з програмістами. Тестерам не варто грати в «квача» з розробниками. Вони повинні співпрацювати щоб забезпечити покупцю продукт найвищої якості.

Для того, щоб мати змогу перейти від ручного тестування до автоматичного була сформульована наступна пара принципів.

Принцип 8. Тести повинні передбачати багаторазове виконання. В оригіналі:

Принцип 7: Краще уникати одноразових варіантів тестів, якщо сама програма не одноразового використання.

Представлений вище Принцип 2 закликав тестера розглядати свою роботу, як роботу ученого-експериментатора. Принцип 8 вимагає точного запису умов тесту, всіх важливих подій, що виникли під час тесту, і уважного аналізу результату. Ця інформація безцінна для розробників при поверненні коду на відладку, бо вона дає змогу відтворити умови тесту. Також вона корисна для повторних тестів після виправлення дефекту. Багаторазове виконання тесту відбувається і при регресивному тестуванні (поворотне тестування зміненої програми) нових версій ПЗ. Вчені знають, що їх експерименти повторюватимуть інші люди, це повинні знати і тестери.

Наступний принцип може здаватися очевидним, проте про нього часто забувають.

Принцип 9. Тестування слід планувати.

План треба розробляти для кожного етапу тестування, а мета повинна бути описана у відповідній його частині. Вона має бути задана настільки чітко, наскільки це можливо. Плани з описаною в них метою необхідні для адекватного розподілу часу і ресурсів на тести, а також для спостереження і управління процесом тестування. Планування тесту повинно бути внесено в життєвий цикл програми (Принцип 10) і скоординовано із плануванням проекту. Тестери не зможуть працювати над компонентом в заздалегідь обумовлений день, якщо розробники на той час його не доробили. Тому повинні бути обумовлені ризики тестування. Наприклад, імовірність затримки надання програмних компонентів, наявність компонентів складних для тесту, потреба в додатковому навчанні використанню нових інструментів. Шаблон плану тесту повинен бути доступний керівнику для грамотної розробки плану відповідно до політики і стандартів організації. Уважне планування тестів виключає некорисні одноразові тести та неефективні, незаплановані цикли «тест – виправлення – повторний тест», які часто затримують терміни виходу програми.

Принцип 10. Слід включати процес тестування в життєвий цикл ПЗ.

Якщо розпочинати тестування лише після того як програмний код вже повністю написано, то, з точки зору часових та фінансових витрат, воно не буде достатньо ефективним. Планування процесу тестування, як стверджує принцип 10, повинно бути включено в життєвий цикл програми, починаючи з етапу аналізу вимог і проводитися надалі, паралельно розробці. Додатково до планування, можна проводити різні тести на ранніх етапах, використовуючи прототипи, наприклад, перевірки на зручність використання. Ці процеси можуть тривати до самого випуску програми. Можливо використання різних моделей впровадження тестування в життєвий цикл ПЗ.

Останній принцип є узагальнюючим і відображає підхід до роботи тестера

Принцип 11 (10). Тестування, це завдання, що вимагає творчості та розумової праці.

Труднощі роботи тестера, якщо їх деталізувати, полягають в наступному:

- Тестеру необхідно мати вичерпні знання в області програмної інженерії.

- Тестеру необхідно мати досвід і системні знання у області специфіки, проектування і розробки ПЗ.

- Тестер повинен справлятися з багатьма дрібними задачами.

- Тестер повинен розбиратися у видах помилок і знати, де вони найчастіше можуть зустрічатися.

- Тестер повинен міркувати як вчений, висловлюючи припущення про наявність конкретних типів дефектів.

- Тестер повинен добре орієнтуватися в характерних для тестованого ПЗ дефектах. Це можливо при якісній освіті і досвіді тестування.

- Тестер повинен створювати і документувати контрольні приклади. Для розробки тестів необхідно вибирати вхідні дані з дуже широкого спектру. Вибрані дані повинні найбільшою імовірністю виявити дефект (Принцип 2). Необхідна обізнаність в цій області.

- Тестер повинен розробляти і зберігати процедури тесту для його виконання.

- Тестер повинен планувати оптимальну витрату ресурсів.

- Тестер повинен проводити всі випробування і протоколювати результати.

- Тестер повинен аналізувати результати і вирішувати, вважати тест успішним чи ні. Це вимагає розуміння та відстежування величезної кількості детальної інформації. Тестеру також потрібно збирати і аналізувати пов'язані з тестами вимірювання.

- Тестер повинен вчитися використовувати різні інструменти тестування, включаючи новітні розробки.

- Тестеру необхідно співпрацювати з інженерами, дизайнерами і розробникам. Також слід підтримувати контакт із клієнтами і користувачами.

- Тестер повинен підтримувати свої знання на високому рівні, регулярно підвищуючи свою кваліфікацію.

**Питання для самоконтролю:**

1. **Назвіть фази, стадії та види тестування.**
2. **Які техніки тестування.**
3. **Поясніть аксіоми тестування.**
4. **Яке місце тестування в циклі розробки програмного забезпечення.**
5. **Поясніть принципи тестування програмного забезпечення.**

**Лекція 3. Agile (Гнучка модель). Scrum. Selenium**

**Agile** – сімейство процесів розробки, а не єдиний підхід у розробці програмного забезпечення. Agile не включає практик, а визначає цінності і принципи, якими керуються успішні команди. Включає в себе наступні методології: наступних методологій Extreme programming, Scrum, DSDM, Adaptive software development, Crystal Clear, Feature-driven development, Pragmatic Programming.

**Основні ідеї:**

1. Особистості та їх взаємодія є важливішим, ніж процеси та інструменти;
2. Працююче програмне забезпечення є важливішим, ніж повна документація;
3. Співпраця з замовником важливіша, ніж контрактні зобов’язання;
4. Реакція на зміни є важливішим, ніж дотримання плану.

**Принципи:**

* задоволення клієнта за рахунок ранньої та безперебійної поставки цінного програмного забезпечення;
* можливість змін вимог навіть наприкінці розробки (це може підвищити конкурентоспроможність отриманого продукту);
* часта поставка робочого програмного забезпечення (кожен місяць або тиждень або ще частіше);
* тісне, щоденне спілкування замовника з розробниками протягом всього проекту;
* проектом займаються мотивовані особистості, які забезпечені потрібними умовами роботи, підтримкою і довірою;
* рекомендований метод передачі інформації – особиста розмова (обличчям до обличчя);
* працююче програмне забезпечення – кращий вимірювач прогресу;
* спонсори, розробники і користувачі повинні мати можливість підтримувати постійний темп на невизначений термін;
* постійна увага поліпшенню технічної майстерності і зручному дизайну;
* простота – мистецтво не робити зайвої роботи;
* кращі технічні вимоги, дизайн та архітектура виходять у самоорганізованої команди;  
  постійна адаптація до мінливих обставин.

**Скрам** (Scrum) – одна з найпопулярніших методологій гнучкої розробки. Scrum простий у використанні і робить акцент на якісному контролі процесу розробки.

У методології Scrum всього три ролі:

* Scrum Master
* Product Owner
* Team

**Скрам Майстер** (Scrum Master) – відповідає за успіх у проекті. По суті, Скрам Майстер є інтерфейсом між менеджментом і командою (менеджер проекту або тімліда). Важливо підкреслити, що Скрам Майстер не роздає завдання членам команди.

У Agile команда є самоорганізованою.

Основні обов’язки Скрам Майстра:

* Створює атмосферу довіри
* Бере участь у мітингах в якості фасилітатора
* Усуває перешкоди
* Робить проблеми і відкриті питання видимими
* Відповідає за дотримання практик і процесу в команді
* Скрам Майстер веде Daily Scrum Meeting і відстежує прогрес команди за допомогою Sprint Backlog, відзначаючи статус всіх завдань у спринті. ScrumMaster може також допомагати Product Owner створювати Backlog для команди.

**Product Backlog** – це пріоритезований список наявних на даний момент бізнес вимог і технічних вимог до системи. Product Backlog включає в себе use cases, defects, enhancements, technologies, stories, features, issues, і т.д .

Product backlog також включає завдання, важливі для команди, наприклад «провести тренінг», «додати всім пам’яті».

Product Backlog постійно переглядається і доповнюється – в нього включаються нові вимоги, видаляються непотрібні, переглядаються пріоритети. За Product Backlog відповідає Product Owner. Він також працює спільно з командою для того, щоб отримати наближену оцінку на виконання елементів Product Backlog для того, щоб більш точно розставляти пріоритети відповідно до необхідним часом на виконання.

**Product Owner** – це людина, яка відповідає за розробку продукту (представник замовника). Product Owner – це єдина точка прийняття остаточних рішень для команди в проекті.

Обов’язки Product Owner такі:

* Управляє очікуваннями замовників і всіх зацікавлених осіб
* Координує і пріорітизує Product backlog
* Надає зрозумілі вимоги команді
* Взаємодіє з командою і замовником
* Відповідає за приймання коду в кінці кожної ітерації
* Product Owner ставить завдання команді, але він не має права ставити завдання • конкретного члена проектної команди протягом спринту

**Команда** (Team). Команда бере на себе зобов’язання щодо виконання обсягу робіт на спринт перед Product Owner. Робота команди оцінюється як робота єдиної групи. У Scrum внесок окремих членів проектної команди не оцінюється, оскільки це розвалює самоорганізацію команди.  
Обов’язки команди такі:

* Відповідає за оцінку елементів backlog
* Приймає рішення по дизайну
* Розробляє софт і надає його замовнику
* Відстежує власний прогрес (разом зі Скрам Майстром)
* Відповідає за результат перед Product Owner

**Selenium** – це інструмент для тестування Web-додатків.  
Офіційний сайт: <http://seleniumhq.org>.

Selenium на даний момент є найпопулярнішим іструментом для автоматизації тестування web-додатків, оскільки він: безкоштовний, гнучкий, працює безпосередньо через браузер, доступний в різних мовах програмування. Важливим є те що тести Selenium виконуються безпосередньо в браузері як це роблять звичайні користувачі.

2 головні переваги Selenium:

Створення тестових сценаріїв Selenium відтворює дії користувача, тобто додаток тестується з точки зору кінцевого користувача.  
Можливість запуску тестів в різних браузерах, що полегшує визначення несумісності браузера.

Selenium працює з:

Браузери: Internet Explorer, Mozilla Firefox, Safari, Chrome

Операційні системи: Microsoft Windows, Mac OS, Linux

**Питання для самоконтролю:**

* + 1. Якщо ви Scrum майстер на невеликому проекті. Що до ваших обов’язків відноситься.
    2. Яка з технік, вважається перевагою програмування Agile технології розробки?
    3. Філософія керування проектом по Agile це:
    4. **Що зі списку не є Agile методологією?**
* Rapid
* eXtreame програмування
* Testing drive development
* Scrum
* Kanban

1. **Основні принципи гнучкої розробки ПЗ викладені в Agile Manifesto. Виберіть пункт, який не входить в маніфест:**

* Люди та співпраця важливіші за процеси та інструменти.
* Працюючий продукт важливіший за вичерпну документацію.
* Співпраця із замовником важливіша за обговорення умов контракту.
* Готовність до змін важливіша за дотримання плану.
* Вичерпна документація кожної фази розробки найважливіша.

1. **Що не є артефактом у Scrum?**

* Власний список побажань Product Ownerа, відсортований за пріоритетами (owner backlog)
* Список задач, що залишилися у межах одного спринту (burndown chart)
* Список задач для одного спринту, що відсортовані за пріоритетом (sprint backlog)
* Список високорівневих вимог до продукту, що відсортовані за пріоритетом (product backlog)

1. **У процесі роботи з вимогами до програмної системи у Product backlog визначається**

* Пріоритетність вимог за бажанням команди
* Логічна послідовність вимог
* Пріоритетність вимог за вимогами замовника
* Вимоги до архітектури системи
* Бізнес вимоги до системи

1. **При плануванні тривалості кожного спринту в Agile використовується:**

* Метод Ігри у покер
* Метод Ігри у преферанс
* Тривалість визначається тільки замовником
* Тривалість визначається тільки Scrum мастером
* Тривалість визначається тільки менеджером проекту

1. **Якщо до кінця спринту команді не вдається завершити усі user story, заплановані на цей спринт, команда повинна**

* Збільшити тривалість спринту
* Перенести частку **user story** назад в **Product backlog**
* Залишити все без жодних змін
* Збільшити кількість годин роботи команди для завершення спринту у строк
* Додати нових членів у команду розробників.

**Лекція 4. UX/UI, Usability**

**План лекції:**

1. **Інтерфейс користувача (UI)**
2. **Елементи управління інтерфейсу користувача**
3. **Досвід взаємодії (UX)**
4. **Зручність використання**
5. **Директива інтерфейсу користувача**

Згідно Вікіпедії, UI це різновид інтерфейсів, в якому одна сторона представлена ​​людиною (користувачем), інша - машиною / пристроєм. Являє собою сукупність засобів і методів, за допомогою яких користувач взаємодіє з різними, найчастіше складними, машинами, пристроями та апаратурою.

Іншими словами, призначений для користувача інтерфейс є системою, за допомогою якої люди (користувачі) взаємодіють з машиною. Інтерфейс включає в себе два компоненти: обладнання (фізичний компонент) і програмне забезпечення (логічний компонент). Існують інтерфейси для різних систем. UI є засобом для виконання двох типів взаємодії:

- Введення (Input), що дозволяє користувачам управляти системою;

- Вивід (Output), що дозволяє системі демонструвати ефект від вироблених користувачем маніпуляцій.

Основною метою при розробці інтерфейсу для взаємодії людини з машиною є створення UI, який дозволить легко (інтуїтивно), ефективно, і з задоволенням (user friendly) домагатися бажаного результату під час роботи з машиною. Ідеальним результатом вважається той, при якому оператор (або користувач) виробляє мінімальні зусилля для введення, отримуючи від машини бажаний вивід, при цьому машина мінімізує вивід зайвих даних.

У комп'ютерну еру термін «призначений для користувача інтерфейс» як правило, передбачає графічний інтерфейс користувача, проте спочатку мова йшла про панелі управління в промислових галузях.

Кращий інтерфейс користувача – це такий інтерфейс, якому користувач не повинен приділяти багато уваги, майже не помічати його. Користувач просто працює, замість того, щоб розмірковувати, яку кнопку натиснути або де клацнути мишею. Такий інтерфейс називають прозорим – користувач ніби дивиться крізь нього на свою роботу.

Основні типи інтерфейсу користувача:

* 1. **Інтерфейс командного рядка** (*command-line interface*, CLI);
  2. **Графі́чний інтерфейс користувача** (*GUI, Graphical user interface*) — тип інтерфейсу, який дозволяє користувачам взаємодіяти з електронними пристроями через вікна, графічні зображень меню та візуальні вказівники;
  3. **SILK (speech, image, language, knowledge) -** інтерфейс найбільш наближений до звичайної людської форми спілкування;
  4. **Сенсорний інтерфейс користувача** **(*Touch user interface*)**  це графічний інтерфейс, який використовує тачпад або тачскрін дисплей в якості комбінованого пристрою вводу і виводу.
  5. **Апаратний інтерфейс** (**Hardware interfaces** ) - пристрій, що перетворює сигнали і передає їх від одного компонента обладнання до іншого, визначається набором електричних зв'язків і характеристиками сигналів.
  6. **Веб-інтерфейс (WUI).**



Рис.8 - **Характеристики хорошого інтерфейсу**

Працюючи над досягненням однієї з цих характеристик, ви можете створити проблеми для досягнення іншої.

* стараючись зробити інтерфейс більш зрозумілим, ви можете додати багато описів і пояснень, що в кінцевому підсумку зробить інтерфейс ще більш громіздким і незручним.
* урізуючи матеріал для досягнення мінімалізму, може зробити речі незрозумілими пересічному користувачеві.

Для досягнення балансу необхідна майстерність і багато часу, і пам'ятайте, що ваші дизайнерські рішення, швидше за все, будуть різними в різних проектах.

Що актуально для одного, для іншого може бути не допустимо.

**Досвід користування (User Experience (UX)** - це те, що людина відчуває при використанні продукту, системи або сервісу.

Згідно зі стандартом  ISO 9241-210, досвід користування — це «відчуття і реакції людини, що виникли внаслідок використання або уявного використання продукту, системи чи сервісу».

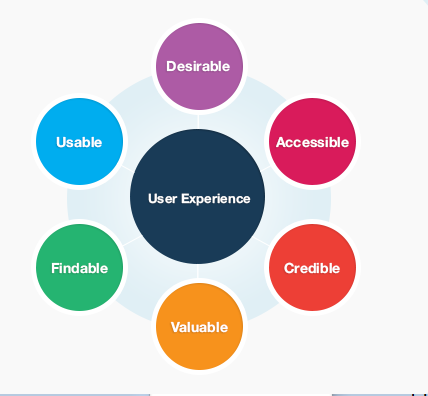


Рис.9 - Аспекти взаємодії людини з комп'ютером включені до UX.

UX включає в себе поведінку людини, ставлення і емоції при використанні конкретного продукту, системи або послуги.

Аспекти взаємодії людини з комп'ютером включені до UX:

* практичність;
* емпіричність;
* емоційність;
* багатозначність.

Аспекти сприйняття персоною:

* корисність;
* простота використання;
* вправність.

Спільні характеристики:

* Обидва можуть бути застосовані до (майже) будь-якого продукту.
* Обидва вирішують завдання, пов'язані з дизайнерським аспектом продукту.
* Обидва орієнтовані на комфорт користувача.

В чому ж їхні обов’язки?

* UI дизайнер турбується про розташування елементів та гаму кольорів (розробляє розташування елементів, вибирає кольори та шрифти, що доповнюють одне одного), а також підбирає зображення (визначаючи, чи тренди дизайну підходять інтерфейсу, створює ілюстрації для полегшення користування сайтом).
* UX дизайнер відповідальний за організацію системи, тобто її покращення для користувачів, щоб їм було зручно знайти потрібну інформацію, а також, що є досить очевидним, він повинен розуміти користувача, проводити інтерв’ю та оцінювання. Роль UX дизайнера включає багато викликів i є багатогранною.
* Хоча між UI і UX дизайнерами існують докорінні відмінності, вони доповнюють один одного, тим самим допомагаючи полегшити користування сайтом.

**Usability.**

ISO визначає зручність як «Ступінь з якою продукт може бути використано визначеними користувачами для досягнення визначених цілей з ефективністю, продуктивністю та вдоволеністю для специфічного контексту використання».

**Usability testing:** *(перевірка ергономічності)* — дослідження, що виконується з метою визначення зручності деякого штучного об'єкту (веб-сторінка, користувальницький інтерфейс) для його подальшого застосування:

* легко розуміти,
* легко вчитися,
* легко працювати,
* є привабливим для користувачів з урахуванням конкретних умови.

**Юзабіліті-тестування** відноситься до оцінки продукту або сервісу при тестуванні його з представниками користувачів.

* Як правило, під час випробування, учасники намагатимуться завершити типові задачі в той час як спостерігачі дивляться, слухають і роблять нотатки.
* Мета полягає в тому, щоб виявити будь-які проблеми з юзабіліті, збирати кількісну і якісну інформацію і визначити задоволення учасника продуктом.

Переваги usability-тестування до кінцевого споживача або клієнта:

* Краща якість програмного забезпечення
* Програмне забезпечення простіше у використанні
* Програмне забезпечення більш легко прийнятне для користувачів
* Скорочує час навчання для нових користувачів

Отже,

**Usability testing:** протестувати, наскільки легко користувач може використовувати систему.

**UI testing:** протестувати зовнішній вигляд екрану кольору, шрифти, макети і т.д.

**Питання для самоконтролю:**

1. Які основні типи інтерфейсу користувача.
2. Що таке **Usability testing?**
3. **Що таке UI testing?**
4. В чому ж обов’язки UI та UX дизайнерів?

**Лекція 5. Типи тестування**

**План лекції:**

1. **Рівні тестування.**
2. **Види тестування ПО. Функціональне тестування (Functional Testing). Тестування безпеки (Security and Access Control Testing). Тестування взаємодії (Interoperability Testing).**
3. **Нефункціональне тестування ПЗ.**
4. **Види тестування, пов’язані зі змінами. Кросбраузерність.**

**Рівні тестування**

Тестування на різних рівнях проводиться протягом усього життєвого циклу розробки і супроводу ПЗ. Рівень тестування визначає те, над чим виробляються тести: над окремим модулем, групою модулів або системою в цілому. Проведення тестування на всіх рівнях системи – це запорука успішної реалізації та здачі проекту.

**Рівні тестування (Testing levels):**

* Компонентне або Модульне тестування (Component testing or Unit testing)
* Інтеграційне тестування (Integration testing)
* Системне тестування (System testing)
* Приймальне тестування (Acceptance testing)

**Компонентне тестування** перевіряє функціональність і шукає дефекти в частинах програми, які доступні і можуть бути протестовані окремо (модулі програми, об’єкти, функції і т.д.).

Зазвичай компонентне (модульне) тестування проводиться викликаючи код, який необхідно перевірити чи за підтримки середовищ розробки, таких як фреймовки (каркаси) для модульного тестування або інструменти для дебагу. Всі знайдені дефекти, як правило виправляються в коді без формального їх опису в системі багів (Bug Tracking System).

Один з найбільш ефективних підходів до компонентного (модульного) тестування – це підготовка автоматизованих тестів до початку основного кодування ПЗ. Це називається розробка від тестування (test-driven development) або підхід тестування спочатку (test first approach). При цьому підході створюються і інтегруються невеликі шматки коду, навпроти яких запускаються тести, написані до початку кодування. Розробка ведеться до тих пір, поки всі тести не будуть успішними.

**Інтеграційне тестування** призначено для перевірки зв’язку між компонентами, а також взаємодії з різними частинами системи.  
Рівні інтеграційного тестування:

* Компонентний інтеграційний рівень (Component Integration testing) перевіряє взаємодія між різними системами після проведення компонентного тестування.
* Системний інтеграційний рівень (System Integration testing) перевіряє взаємодію між різними системами після проведення системного тестування.

**Системне тестування** – основним завданням є перевірка як функціональних так і не функціональних вимог у системі 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. Для мінімізації ризиків, пов’язаних з особливостями поведінки системи в тому чи іншому середовищі, під час тестування рекомендується використовувати оточення максимально наближене до того, на яке буде встановлений продукт після релізу. Існує два підходи до системного тестування:

* На базі вимог (requirements based) – Для кожної вимоги пишуться тестові випадки (test cases), перевіряючі виконання даної вимоги.
* На базі випадків використання (use case based) – На основі уявлення про способи використання продукту створюються випадки використання системи (Use Cases). По конкретному випадку використання можна визначити один або більше сценаріїв. На перевірку кожного сценарію пишуться тест кейси (test cases), які повинні бути протестовані.

**Приймальне тестування** проводиться з метою: визначення чи задовольняє система приймальні критерії, винесення рішення замовником або іншою уповноваженою особою приймається програма чи ні.  
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 (Product Acceptance Plan) або іншим документом, де описаний набір дій, пов’язаних з проведенням приймального тестування, дата проведення, відповідальні і т.д.

**Методи приймального тестування:**

* 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.
* Тестування (Аудит) третьою стороною. Наймається спеціалізована компанія на тестуванні або підписується договір з конкурентом постачальника на надання послуг аудиту. Оптимально.
* 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 неї 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.

# **Види тестування ПО.**

Всі види тестування програмного забезпечення, залежно від переслідуваних цілей, можна умовно розділити на наступні групи:

* Функціональні (Functional testing)
* Нефункціональні (Non-functional testing)
* Пов’язані зі змінами (Regression testing)

**Функціональні види тестування**

Функціональні тести базуються на функціях і особливостях, а також 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. Далі перераховані одні з найпоширеніших видів функціональних тестів:

Функціональне тестування (Functional testing)

Тестування безпеки (Security and Access Control Testing)

Тестування взаємодії (Interoperability Testing)

**Нефункціональні види тестування**

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. В цілому, це тестування того, “Як” система працює. Далі перераховані основні види нефункціональних тестів:

Всі види тестування продуктивності:

тестування навантаження (Performance and Load Testing)

стресове тестування (Stress Testing)

тестування стабільності або надійності (Stability / Reliability Testing)

об’ємне тестування (Volume Testing)

Тестування установки (Installation testing)

Тестування зручності користування (Usability Testing)

Тестування на відмову і відновлення (Failover and Recovery Testing)

Конфігураційне тестування (Configuration Testing)

**Тестування, пов’язане зі змінами**

Після проведення необхідних змін, таких як виправлення бага / дефекту, програмне забезпечення повинне бути перетестоване для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

Димове тестування (Smoke Testing)

Регресійне тестування (Regression Testing)

Тестування збірки (Build Verification Test)

Санітарне тестування або перевірка узгодженості / справності (Sanity Testing)

**Функціональне тестування** (Functional testing)

Тестування функціональності може проводиться у двох аспектах:

* вимоги
* бізнес-процеси

Тестування в аспекті «вимоги» використовує специфікацію функціональних вимог до системи як основу для дизайну тестових випадків (Test Cases). У цьому випадку необхідно зробити список того, що буде тестуватися, а що ні, пріоритезувати вимоги на основі ризиків (якщо це не зроблено в документі з вимогами), а на основі цього пріоритезувати тестові сценарії (test cases). Це дозволить сфокусуватися і не упустити при тестуванні найбільш важливий функціонал.

Тестування в сенсі «бізнес-процеси» використовує знання цих самих бізнес-процесів, які описують сценарії щоденного використання системи. У цьому випадку тестові сценарії (test scripts), як правило, ґрунтуються на випадках використання системи (use cases).

Переваги функціонального тестування:

імітує фактичне використання системи.

Недоліки функціонального тестування:

можливість упущення логічних помилок у програмному забезпеченні;

ймовірність надмірного тестування.

Досить поширеною є автоматизація функціонального тестування.

**Тестування безпеки (Security and Access Control Testing)** – це стратегія тестування, що використовується для перевірки безпеки системи, а також для аналізу ризиків, пов’язаних із забезпеченням цілісного підходу до захисту додатків, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних.

Загальна стратегія безпеки ґрунтується на трьох основних принципах: конфіденційність, цілісність, доступність.

**Конфіденційність** – це приховування певних ресурсів або інформації. Під конфіденційністю можна розуміти обмеження доступу до ресурсу деякої категорії користувачів, або іншими словами, за яких умов користувач авторизований отримати доступ до даного ресурсу.  
**Цілісність** – Існує два основних критерії при визначенні поняття цілісності:

**Довіра** – Очікується, що ресурс буде змінений тільки відповідним способом певною групою користувачів.

**Пошкодження і відновлення** – у разі коли дані пошкоджуються або неправильно змінюються авторизованим або авторизованим користувачем, потрібно визначити наскільки важливою є процедура відновлення даних.

**Доступність** являє собою вимоги про те, що ресурси повинні бути доступні авторизованому користувачеві, внутрішньому об’єкту або пристрою. Як правило, чим більш критичний ресурс, тим вище рівень доступності.

**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) і інтеграційне тестування (integration testing).

# **Нефункціональне тестування ПЗ**

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами. В цілому, це тестування того, “Як” система працює. Далі перераховані основні види нефункціональних тестів:

Всі види тестування продуктивності (Performance testing):

тестування навантаження (Load Testing)

стресове тестування (Stress Testing)

тестування стабільності або надійності (Stability / Reliability Testing)

об’ємне тестування (Volume Testing)

Тестування установки (Installation testing)

Тестування зручності користування (Usability Testing)

Тестування на відмову і відновлення (Failover and Recovery Testing)

Конфігураційне тестування (Configuration Testing)

**Тестування навантаження або тестування продуктивності** – це автоматизоване тестування, що імітує роботу певної кількості бізнес користувачів на ресурсі.

Основні види тестування продуктивності.

У тестування навантаження входять наступні види тестування продуктивності:

**Тестування продуктивності**

Завданням тестування продуктивності є визначення масштабованості програми під навантаженням, при цьому відбувається:

* вимір часу виконання обраних операцій при певних інтенсивностях 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 на високих, граничних, стресових навантаженнях.

**Стресове тестування** (Stress Testing)

Стресове тестування дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу і також оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення впливу стресу. Стресом в даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійна зміна конфігурації сервера.

**Тестування стабільності або надійності** (Stability / Reliability Testing)  
Завданням тестування стабільності (надійності) є перевірка працездатності програми при тривалому (багатогодинному) тестуванні з середнім рівнем навантаження. Часи виконання операцій можуть грати в даному вигляді тестування другорядну роль. При цьому на перше місце виходить відсутність втрат пам’яті, перезапусків серверів під навантаженням та інших аспектів, що впливають саме на стабільність роботи.

**Об’ємне тестування** (Volume Testing)

Завданням об’ємного тестування є отримання оцінки продуктивності при збільшенні обсягів даних в базі даних програми, при цьому відбувається:

* вимір часу виконання обраних операцій при певних інтенсивностях виконання цих операцій;
* може проводитися визначення кількості користувачів, що одночасно працюють з додатком.

**Тестування Установки** (Installation Testing)

Тестування установки направлено на перевірку успішної інсталяції і настройки, а також оновлення або видалення програмного забезпечення.

**Тестування зручності користування** (Usability Testing)

Для того щоб додаток був популярним, йому мало бути функціональним – воно має бути ще й зручним (попередня лекція).

Тестування зручності користування – це метод тестування, спрямований на встановлення ступеня зручності використання, зрозумілості та привабливості для користувачів розроблюваного продукту в контексті заданих умов. [ISO 9126].

Тестування зручності користування дає оцінку рівня зручності використання програми за наступними пунктами:

* продуктивність, ефективність (efficiency) – скільки часу і кроків знадобиться користувачеві для завершення основних завдань програми, наприклад, розміщення новини, реєстрації, покупка і т.д. (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 призупинення роботи з ним на тривалий період часу (повторне виконання операцій після перерви повинно проходити швидше ніж у нового користувача);
* емоційна реакція (emotional response) – як користувач себе почуває після завершення завдання – розгублений, випробував стрес? 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 і успішно відновлюватися після можливих збоїв, що виникли у зв’язку з помилками програмного забезпечення, відмовами обладнання або проблемами зв’язку (наприклад, відмова мережі). Метою даного виду тестування є перевірка систем відновлення (або дублюючих основний функціонал систем), які, у разі виникнення збоїв, забезпечать збереження і цілісність даних тестованого продукту. Методика подібного тестування полягає в симулюванні різних умов збою і наступному вивченні та оцінці реакції захисних систем. У процесі подібних перевірок з’ясовується, чи була досягнута необхідна ступінь відновлення системи після виникнення збою.

**Конфігураційне тестування** (Configuration Testing) – називається тестування сумісності продукту, що випускається (програмне забезпечення) з різним апаратним і програмним засобами.

Основні цілі – визначення оптимальної конфігурації і перевірка сумісності програми з необхідним оточенням (обладнанням, ОС і т.д.)

# **Види тестування, пов’язані зі змінами. Кросбраузерність.**

Після проведення необхідних змін, таких як виправлення бага / дефекту, програмне забезпечення повинне бути перетестоване для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

* Регресійне тестування (Regression Testing)
* Димове тестування (Smoke Testing)
* Санітарне тестування або перевірка узгодженості / справності (Sanity Testing)
* Тестування збірки (Build Verification Test)

**Регресійне тестування**(Regression Testing) – це вид тестування спрямований на перевірку змін, зроблених в додатку або середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб сервер або сервер додатки), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше.  
Регресійними можуть бути як функціональні, так і нефункціональні тести.  
Як правило, для регресійного тестування використовуються тест кейси, написані на ранніх стадіях розробки і тестування. Це дає гарантію того, що зміни в новій версії програми не пошкодили вже існуючу функціональність.

Рекомендується робити автоматизацію регресійних тестів, для прискорення подальшого процесу тестування і виявлення дефектів на ранніх стадіях розробки програмного забезпечення.

Сам по собі термін “Регресійне тестування”, залежно від контексту використання може мати різний зміст. Сем Канер, наприклад, описав 3 основних типи регресійного тестування:

* Регресія багів (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена.
* Регресія старих багів (Old bugs regression) – спроба довести, що недавня зміна коду чи даних зламала виправлення старих помилок, тобто старі баги стали знову відтворюватися.
* Регресія побічного ефекту (Side effect regression) – спроба довести, що недавня зміна коду чи даних зламала інші частини продукту.

**Димове тестування** (Smoke Testing)

Поняття пішло з інженерної 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 застосовується для поверхневої перевірки всіх модулів програми на предмет працездатності і наявності швидкого знаходження критичних і блокуючих дефектів.

**Санітарне тестування** (Sanity Testing)

Це вузьконаправлене тестування, достатнє для доказу того, що конкретна функція працює згідно заявленим в специфікації вимогам. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін вироблених в ній або навколишньому середовищі. Зазвичай виконується вручну.

**Тестування збірки** (Build Verification Test)

Тестування спрямоване на визначення відповідності, випущеної версії, критеріям якості для початку тестування. За своїми цілями є аналогом димового тестування, спрямованого на приймання нової версії в подальше тестування або експлуатацію. Вглиб воно може проникати далі, залежно від вимог до якості випущеної версії.

**Відмінність санітарного тестування від димового** (Sanity vs Smoke testing)

У деяких джерелах помилково вважають, що санітарне та димове тестування – це одне і теж. Ми ж вважаємо, що ці види тестування мають “вектори руху”, що 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).

**Питання для самоконтролю:**

1. В чому полягає різниця між функціональним і не функціональним тестуванням?
2. Дайте характеристику тестуванню навантаження?
3. Що входить до тестування, що пов'язане зі змінами?
4. В чому відмінність димового та санітарного тестування?
5. В чому особливості тестування установки?
6. В чому особливості регресійного тестування?
7. Назвіть рівні тестування і їх основні характеристики.
8. В чому різниця між компонентним і системним тестуванням?
9. Які підходи до системного тестування, ви знаєте?
10. Які підходи до інтеграційного тестування, ви знаєте?

**Лекція 6. Аналіз вимог (Requirement Analysis)**

**План лекції:**

1. **Визначення і класифікація вимог**
2. **Характеристики хороших вимог**
3. **Типи документів, через які вимоги можуть бути повідомлені**
4. **Аналіз вимог**
5. **Відстеження і залежні матриці.**

**Вимоги (requirement) - умови або можливості, необхідні користувачеві для вирішення певних завдань або досягнення певних цілей, які повинні бути досягнуті для виконання контракту, стандартів, специфікації, або інших формальних документів [ISTQB].**

**IEEE Standart Glossary of Software Engineering Terminology визначає вимоги, як:**

1. Умови чи можливості, необхідні користувачу для вирішення проблем або досягнення цілей.
2. Умови або можливості, якими повинна володіти система, щоб виконати контракт або задовольняти стандарти, специфікації або інші формальні документи.
3. Документування представлення умов або можливостей.

**Вимоги** визначаються в процесі аналізу вимог та фіксуються в специфікації вимог, діаграмі прецедентів та інших артефактах процесу аналізу та розробки вимог.

**Розробка вимог** до програмної системи може бути розділена на декілька етапів:

* 1. **Знаходження вимог (збір, визначення потреб заінтересованих осіб та систем).**
  2. **Аналіз вимог (перевірка цілісності та закінченості).**
  3. **Специфікація (документування вимог).**
  4. **Тестування вимог.**

Помилки у вимогах є найбільш поширеним типом помилки розробки систем і найдорожчим для виправлення.

**Рівні та типи вимог:**

* Функціональні:
  + Бізнес вимоги
  + Вимоги користувача
  + Функціональні вимоги
* Нефункціональні:
  + фізичне середовище
  + продуктивність
  + документація
  + атрибути якості

Характеристики хороших вимог:

* + Necessary - Необхідність вимоги.
  + Complete - Завершеність вимоги.
  + Consistent - Узгодженість (несуперечливість) вимоги.
  + Unambiguous – Однозначність.
  + Verifiable – Здатність бути перевіреню.
  + Traceable – Простеження.

Зазвичай програмні вимоги представлені в проекті в наступних уявленнях:

* SRS (Специфікація вимог до програмного забезпечення) - документ, в якому перераховані всі вимоги до ПЗ, згруповані за типом на логічні розділи;
* Product Backlog- список вимог (так звані юзер-сторі користувачів), який зберігається протягом продукту, розробленого з використанням методології Agile Scrum (SDLC);
* Юз-кейси - описують взаємодію користувачі з майбутнім програмним забезпеченням.

**Мета тестування** - забезпечення того, що програмне забезпечення працює відповідно до взаємно узгоджених вимог.

Для цього інженеру QC необхідно виконати наступне:

* Провести аналіз вимог та довести їх до ідеалу.
* Написати Acceptance Criteria при участі замовника.
* Спроектувати тест-кейси так, щоб повністю покрити всі вимоги.
* Виконати тести.
* Репортувати про дефекти (якщо такі є).

Аналіз вимог включає в себе:

* Ретельний огляд вимог (reviews).
* Документування висновків огляду.
* Зустрічі з зацікавленими сторонами з врегулювання конфліктів у вимогах.
* Оновлення вимог.

**Відстеження та матриці відстеження.**

**Requirements Traceability це** здатність стежити за життям вимоги, і в зворотному напрямку і в прямому.

ISTQB визначення: здатність ідентифікувати відповідні елементи в документації та програмного забезпечення, таких як вимоги з відповідними тестами.

**Traceability** може бути вертикальне і горизонтальне.

A **traceability matrix** – це документ, який візуалізує відстеження.

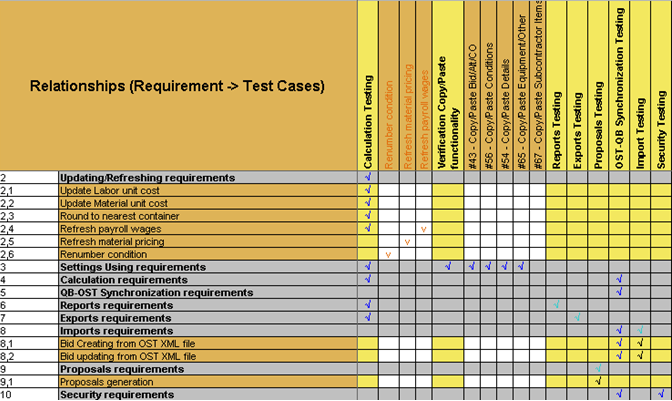


Рис. 10 - **Тraceability matrix.**

**Залежність (Dependency)** це залежності між умовами, подіями або завданнями, такого плану, що наступні можуть бути зроблені тоді, коли попередні зроблені повністю.

**Dependency Structure Matrix (DSM)-** це представлення залежностей проекту. Це компактний спосіб представлення складної системи.

Важливість матриці залежностей:

1. Структурування інформації
2. Впорядкування тестування (планування)
3. Аналіз впливу завдань одне на інше до реалізації або до тестування

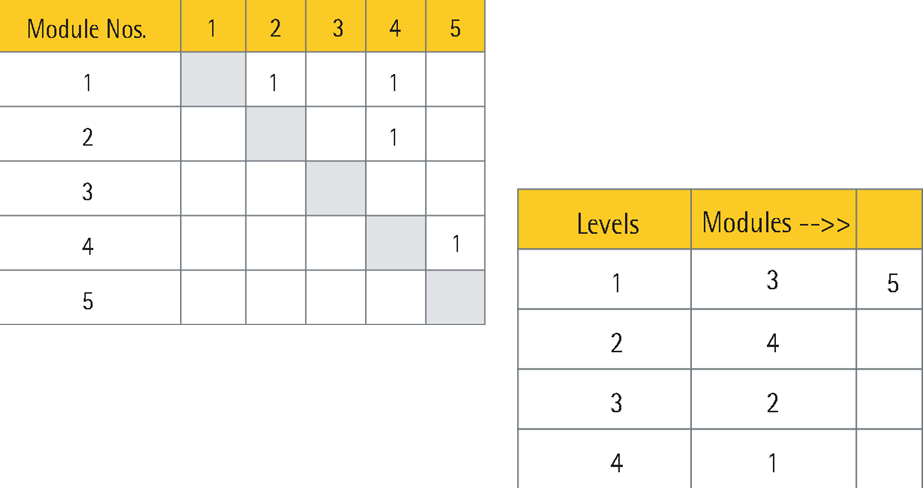


Рис.11- **Dependency Structure Matrix.**

**Питання для самоконтролю:**

1. Що таке вимоги?
2. Дайте характеристику хорошим вимогам.
3. Які типи документів, через які вимоги можуть бути повідомлені?
4. Що входить в аналіз вимог?
5. Що показують матриці відстеження і залежності?

**Лекція 7. Техніки тест дизайну (Test Design Techniques)**

**План лекції:**

1. **Процес тест дизайну (Test Design Process)**
2. **Типи технік дизайну (Design Techniques Types)**
3. **Статичні техніки (Static Techniques)**
4. **Динамічні техніки (Dynamic Techniques)**
   * + **Specification – Based;**
     + **Structure – Based;**
     + **Experience – Based.**
5. **Вибір техніки тест дизайну (Choosing A Test Design Technique)**

**Тест дизайн** – це етап процесу тестування ПЗ, на якому проектуються і створюються тестові випадки (тест кейси), відповідно до визначених раніше критеріїв якості і цілей тестування.

**Тестовий випадок (Test Case)** — це документ, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації функції або частини функції, яку потрібно протестувати.

**Тест аналіз** - це процес огляду всього, що може бути використано для отримання інформації для тестування. Це основа для тестів. Тест аналіз має такі основні завдання, в приблизно такому порядку:

* + Огляд тестової бази (Review **Test Basis** )
  + Визначити пункти тестів (умови тестів) (**Define Test Items )**
  + Оцінка testability вимог і системи
  + Визначити тестове середовище

**Тестова база** - всі документи, з яких можна витягнути вимоги до компонента або системи (документація, на якій базуються тестові випадки).

**Test Items** (Тестові умови) - елемент або подія компонента або системи, які можуть бути перевірені за допомогою одного або більше тестів, наприклад, функція, транзакція або елемент конструкції.

**Методи генерування тестів:**

* 1. Чернетка-чистовик (dirty list-white list).
* 2. Матрична розкладка (matrices).
* 3. Блок-схеми (flowchart).

Чудові результати дає комбінування методів.

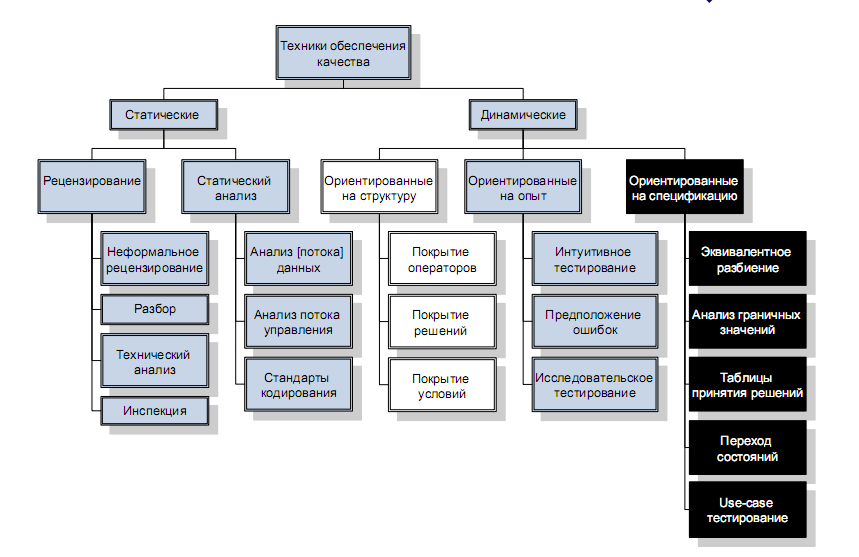


Рис.12 – Техніки забезпечення якості

Статичні техніки:

1. Рецензування (review):

- Неформальне рецензування (informal review).

- Розбір (walkthrough).

- Технічний аналіз (technical review).

- Інспекція (inspection).

1. Статичний аналіз (static analysis):

- Аналіз [потоку] даних (data [flow] analysis).

- Аналіз потоку управління (control flow analysis).

- Стандарти кодування / метрики коду.

Динамічні техніки:

1. **Орієнтовані на структуру** (structure-based):

- Покриття операторів (statement coverage)

- Покриття рішень (decision coverage)

- Покриття умов (condition coverage)

1. **Орієнтовані на досвід** (experience-based):

- Інтуїтивне тестування (ad hoc testing)

- Припущення помилок (error guessing)

- Дослідне тестування (exploratory testing)

1. **Орієнтовані на специфікацію** (specification-based):

- Еквівалентне розбиття (equivalence partitioning)

- Аналіз граничних значень (boundary value

analysis)

- Таблиці прийняття рішень (decision tables)

- Перехід станів (state transition)

- Тестування на основі сценаріїв використання (use-case testing)

**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 кожній області як мінімум один раз.

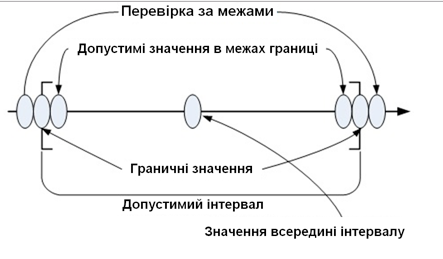


Рис.13 – Еквівалентне розбиття

Приклад:

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.

**Метод граничних значень**: метод проектування чорним ящиком, в якому тести-кейси розробляються на основі граничних значень.

Крайовими чи граничними є значення вхідного значення або вихідного, який знаходиться на краю перегородки еквівалентності або в найменшій додатковій відстані по обидві сторони від краю, наприклад, мінімальне або максимальне значення з діапазону.

Не методи повинні керувати вашою підготовкою, а ви повинні управляти методами так, щоб з їх допомогою створити саме ті тест-кейси, які з високою ймовірністю могли б знайти баги.

**Таблиці рішень -** хороший метод для збору системних вимог, містять логічні умови та документування внутрішнього дизайну системи. Вони можуть використовуватися для запису складних бізнес-правил, які повинна реалізовувати система. Аналізуються специфікації і визначаються умови і дії системи. Вхідні умови і дії найчастіше формулюються таким чином, щоб вони могли приймати логічні значення «Істина» або «брехня».



Рис.14 – Таблиці рішень

Кроки побудови таблиці:

1) Визначити/записати всі умови;

2) Порахувати кількість можливих комбінацій умов;

3) Заповнити комбінації;

4) Забрати лишні комбінації;

5) Записати дії.

**Тестування переходами станів** - метод проектування чорним ящиком, в якому тести призначені для виконання дійсних та недійсних переходів між станами.

**Перехідний стан** - перехід між двома станами компонента або системи.

**Ідея:** Дизайн-схема, яка показує події, які викликають зміну одного стану іншим. **Тести повинні охоплювати кожен шлях, починаючи з найдовшої комбінації станів.**

**Сценарій використання** - сценарій, який описує використання системи користувачем для досягнення певної мети.

 В якості користувачів (actors) можуть бути:

* люди;
* інші системи.

**Метод білого ящика або структурні методи.**

**Покриття операторів (Statement Testing).**

Кожен оператор програми повинен бути виконаний (покритий) хоча б один раз.

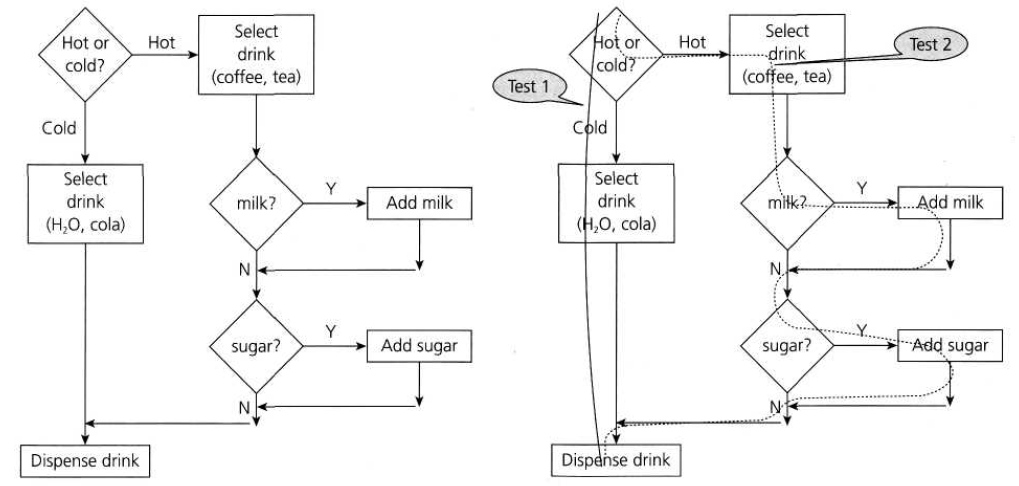


Рис.15 – Покриття операторів

Цей критерій є найбільш слабким з використовуваних у структурному тестуванні, тому що проходження всіх операторів не гарантує перевірку правильності послідовності попарних переходів між ними.

**Покриття рішень (Decision Testing).**

**Кожна гілка алгоритму (кожний перехід між вершинами) має бути пройдена (виконана) хоча б один раз.**

Виконання даного критерію, у загальному випадку, забезпечує й покриття операторів, проте цей критерій не є ідеальним. Так, наприклад, він не забезпечує перевірку правильності обробки операторів логічних переходів та циклів.

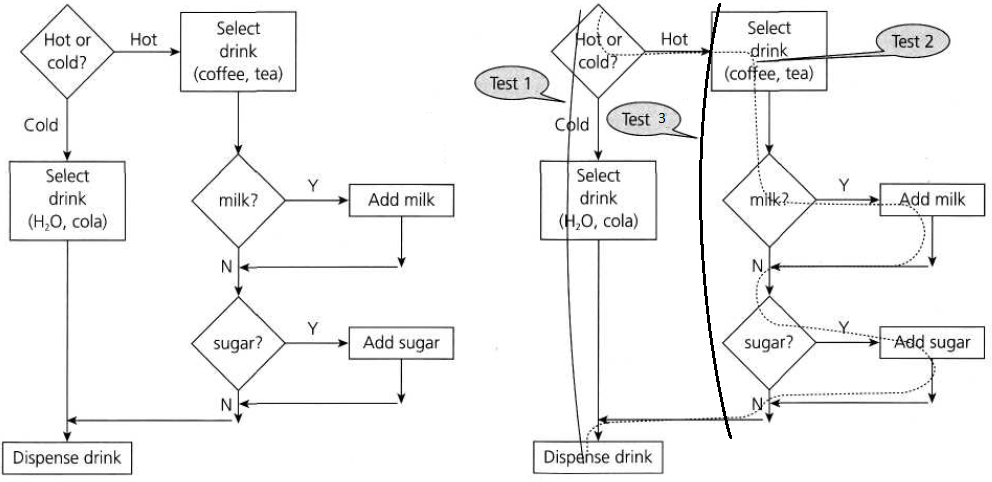


Рис.16 – Покриття рішень

**Критерій покриття шляхів.**

Це найбільш повний критерій, проте його реалізація ускладнена через величезну кількість необхідних тестів. Так, наприклад, проблемою є тестування циклічних структур, в силу необхідності їх багаторазового повторення.

* **Граничне тестування циклу:** відповідно до даного критерію має бути виконаний вхід у кожний цикл (проте виконання ітерацій не вимагається).
* **Внутрішнє тестування циклу:** відповідно до даного критерію має бути виконаний вхід у кожний цикл і як мінімум одна ітерація.
* **Критерій покриття умов:** використовуються при тестуванні логічних конструкцій розгалуження.

Коли тести створюються на основі майстерності тестувальника, його інтуїції і досвіді роботи з подібними додатками або технологіями, це називається тестуванням, **заснованим на досвіді.** Даний вид тестування корисний як додаток більш систематичних методів для розробки спеціальних тестів, не завжди очевидних при використанні більш формальних методик, особливо коли використовується після таких методик. Однак корисність цього методу може сильно варіюватися залежно від досвіду тестувальника.

Який метод кращий? Це неправильне питання!

Кожен метод хороший для певних речей, і не так добрий для інших речей. Деякі методи більш застосовні до певних ситуацій і рівнів випробувань, інші застосовні до всіх рівнів випробувань.

**Внутрішні чинники**, що впливають на рішення про те, які методи використовувати:

* тестера знання і досвід;
* очікувані дефекти;
* мета випробувань;
* документація;
* модель життєвого циклу.

**Зовнішні фактори,** що впливають на рішення про те, які метод використовувати, є:

* ризики;
* замовник і договірні вимоги;
* тип системи;
* нормативні вимоги;
* час і бюджет.

**Питання для самоконтролю:**

1. Що таке процес тест дизайну?
2. Типи технік дизайну?
3. Які Ви знаєте статичні техніки?
4. Які Ви знаєте динамічні техніки?
5. Як вибрати потрібну техніку тест дизайну.

**Лекція 8. Тест-кейс (Test Case)**

**План лекції:**

* 1. **Що таке тест-кейс?**
  2. **Структура тест-кейса.**
  3. **Специфікація тест-кейса.**
  4. **Інструменти управління тест-кейсом.**

**Тестовий випадок / Тест кейс** (Test Case) – це артефакт, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації функції, що тестуємо, або її частини.  
Основні **атрибути** Test Case:

1. ID (номер).
2. Name (ім’я).
3. Preconditions (умови і параметри).
4. Steps (кроки до відтворення).
5. Expected result (очікуваний результат).
6. Actual result (фактичний результат).
7. Postconditions (постумови).

**Проблемні тестові випадки:**

* Які залежні один від одного (наприклад, очікується, що частина кроків виконана в попередньому test case, є посилання на інші test cases).
* З нечітким формулюванням кроків.
* З нечітким формулюванням ідеї або очікуваного результату.

Test cases можуть бути:

– Позитивні – використовуються тільки коректні дані і перевіряють, чи правильно додаток виконує функцію.

– Негативні – використовуються як коректні, так і некоректні дані (мінімум 1 некоректний параметр) і ставить за мету перевірку виняткових ситуацій (спрацьовування валідаторів), а також перевіряє, що функція не виконується при спрацьовуванні валідатора.

Основні **стани** тест кейсу:

* Created (створений).
* Modified (змінений).
* Retired (більше не дійсний).

Основні **стани проходження** тест кейсу:

* No run (не запущено).
* Passed (пройдено).
* Failed (помилка).
* Blocked (заблоковано).
* Not Completed (не завершено).

**Система відстеження помилок** (багтрекінгова система, bugtracking system) – програма, розроблена з метою допомоги розробникам ПЗ враховувати і контролювати помилки (баги), знайдені в програмах, побажаннях користувачів, а також стежити за процесом усунення цих помилок і виконання або невиконання побажань.

Головний компонент такої системи – база даних, що містить відомості про виявлені дефекти:

1. Номер (ідентифікатор об’єкта).
2. Дата і час, коли був виявлений дефект.
3. Хто повідомив про дефект.
4. Хто відповідальний за усунення дефекту.
5. Короткий опис проблеми (обов’язкове поле).
6. Критичність дефекту (обов’язкове поле) і пріоритет рішення.
7. Опис кроків для виявлення дефекту (відтворення неправильної поведінки програми) (обов’язкове поле).
8. Очікуваний результат (обов’язкове поле).
9. Фактичний результат (обов’язкове поле).
10. Обговорення можливих рішень і їх наслідків.
11. Статус дефекту.
12. Версія продукту, в якій дефект був знайдений.
13. Версія продукту, в якій дефект виправлений.
14. Скріншот.

**Серйозність** (Severity) – це атрибут, що характеризує вплив дефекту на працездатність програми.

**Блокуюча** (Blocker) – розробка або використання продукту неможливо. Необхідно негайне виправлення проблеми.

**Критична** (Critical) – серйозні проблеми (не працює критичний блок функціоналу, спостерігаються серйозні помилки, пов’язані з втратою даних тощо).

**Значна** (Major) – проблема, пов’язана з важливим функціоналом продукту.

**Незначна** (Minor) – проблема пов’язана з другорядним функціоналом продукту або є легкий обхідний шлях для цієї проблеми.

**Тривіальна** (Trivial) – проблема косметичного рівня («недопрацьований» інтерфейс: друкарські помилки, різнобій з кольорами).

**Пріоритет** (Priority) – це атрибут, який вказує на черговість виконання завдання або усунення дефекту. Чим вище пріоритет, тим швидше потрібно виправити дефект.

**Високий** (High) – помилка повинна бути виправлена якомога швидше, тому що її наявність є критичною для проекту.

**Середній** (Medium) – помилка повинна бути виправлена, її наявність не є критичною, але вимагає обов’язкового рішення.

**Низький** (Low) – помилка повинна бути виправлена. Її наявність не є критичною, і не вимагає термінового вирішення.

Порядок виправлення помилок з їх пріоритетами:  
High -> Medium -> Low.

**Життєвий цикл дефекту:**

Система відслідковування помилок використовує один з варіантів «життєвого циклу» помилки, стадія якого визначається поточним станом, або статусом, в якому знаходиться помилка.

**Типовий цикл дефекту (Defect cycle)**:

**Новий** (New) – дефект зареєстрований тестувальником.

**Призначено** (Open) – призначений відповідальний за виправлення дефекту.

**Вирішений** (Fixed) – дефект переходить назад в сферу відповідальності тестувальника. Як правило супроводжується резолюцією, наприклад:

**Виправлено** (виправлення включені у версію таку-то),

**Дубль** (повторює дефект, вже знаходиться в роботі),

**НЕ виправлено** (працює відповідно до специфікації, має занадто низький пріоритет, виправлення відкладено до наступної версії і т.п.).

**Відхилений** (Rejected) – запит додаткової інформації про умови, в яких дефект проявляється, або не згоду з тим, що виявлена поведінка системи дійсно являєтся дефектом.

**Відкрито повторно** (Reopen) – дефект знову знайдений в іншій версії.

**Закритий** (Closed) – використовується системою для закриття вирішених завдань.

Принцип: «Де? Що? Коли?»

Де?: У якому місці інтерфейсу користувача або архітектури програмного продукту знаходиться проблема.

Що?: Що відбувається або не відбувається згідно специфікації або вашому уявленню про нормальну роботу програмного продукту. При цьому вказуйте на наявність або відсутність об’єкта проблеми, а не на його утримання (його вказують в описі).

Коли?: У який момент роботи програмного продукту, по настанню якої події або за яких умов проблема проявляється.

**Питання для самоконтролю:**

* 1. Що таке тест-кейс?
  2. Яка структура тест-кейса?
  3. Що містить специфікація тест-кейса?
  4. Які інструменти управління тест-кейсом.

**Лекція 9. Fundamental Test Process**

**План лекції:**

1. **Що таке тест-кейс?**
2. **Структура тест-кейса.**
3. **Специфікація тест-кейса.**
4. **Інструменти управління тест-кейсом.**

Виконання тестів необхідно, але не менш важливі і супроводжуючі дії - планування і документування процесу. В обов'язки тестувальників входить розробка тестових сценаріїв, а також підготовка тестування і оцінка його результатів. Становлення ідеї фундаментального тестового процесу на всіх рівнях тестування зайняло роки. У рамках цього процесу можна виділити ключові кроки:

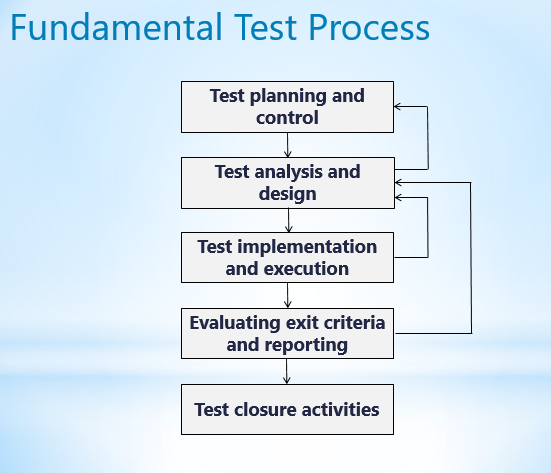


Рис.17 – Фундаментальний процес тестування

Тут дії описані в логічній послідовності, але в умовах реального проекту вони можуть накладатися, відбуватися одночасно або навіть повторюватися. Зазвичай, відбувається адаптація цих кроків під потреби конкретної системи або проекту.

У фундаментальному процесі тестування виділяють наступні основні етапи:

* планування і управління (planning and control);
* аналіз і проектування (analysis and designing);
* реалізація і виконання (implementation and execution);
* оцінка вихідного критерію і звітність (evaluating exit criteria and reporting);
* завершення процесу тестування (test closure activities).

**Планування і управління тестування**

На даному етапі визначається завдання і цілі тестування, характеристика етапів тестування для досягнення цілей і завдання.

Управління це безперервний процес порівнювання поточного стану з планом тестування і звіт про стан справ, включаючи відхилення від плану. Цей етап також включає прийняття дій, необхідних для досягнення цілей і завдань проекту. Щоб керувати тестуванням, потрібно перевіряти його стан протягом усього проекту. Планування враховує результати моніторингу та управління.

**Аналіз і проектування тестів**

Під час аналізу і проектування тестів відбувається трансформація загальних цілей тестування в конкретні тест кейси та умови.

**Аналіз і проектування мають такі завдання:**

* перегляд бази тестування (вимоги до проекту, архітектура, інтерфейси, проектування);
* оцінювання тестованості об'єктів;
* визначення і пріоритетність тест умов, заснованих на аналізі елементів тестування, специфікації, поведінці і структурі;
* проектування і пріоритетність тест кейсів;
* визначення необхідних даних для підтримки тест умов і кейсів;
* настройка середовища тестування і визначення необхідної інфраструктури та інструментів.

**Реалізація та виконання тестування**

Реалізація та виконання тестування це діяльність, при якій тест процедури і сценарії (test procedures and scripts) визначаються за допомогою розташування тест кейсів в певному порядку і включають будь-яку іншу інформацію, необхідну для виконання тестування, середовище тестування налаштоване і тести запускаються.

Реалізація та виконання тестування мають такі головні завдання:

* розробка, реалізація і пріоритетність тест кейсів;
* розробка, пріоритетність процедур тестування, формування тестових даних, підготовка test harnesses (необов'язково) і написання автоматизованих сценаріїв;
* створення набору тестів з тест процедур для ефективного виконання тестування;
* перевірка налаштувань середовища тестування;
* виконання тест процедур вручну або використовуючи спеціальні програми у відповідність планової послідовності;
* фіксування результатів виконання тестування і особливостей і версії тестованого програмного забезпечення, тестових інструментів та програм (testware);
* порівняння отриманих і передбачуваних результатів тестування.

**Оцінка вихідного критерію і звітність**

При оцінці вихідного критерію виконання тестування оцінюється щодо того, наскільки досягнуті поставлені цілі. Це повинно відбуватися на кожному етапі тестування.

Оцінка вихідного критерію має такі головні завдання:

* порівняння даних реєстраційного журналу з вихідним критерієм, визначеним при плануванні тестування;
* визначення потрібно більше тестів або можливо необхідно змінити вихідний критерій;
* написання короткого звіту по тестуванню для замовників.

**Завершення процесу тестування**

У процесі завершення тестування збираються дані попередніх пройдених етапів для накопичення досвіду, тестового по, фактів і чисел. Наприклад, коли виходить програма, проект тестування завершується (або скасовується), завершується черговий етап або реліз продукту без внесення змін або функцій в нього.

Процес завершення тестування має наступні завдання:

* перевірка того скільки запланованих цілей досягнуто, закриття інцидентів або підняття записів про зміни для тих, які залишилися відкритими, а також перевірка приймального тестування;
* закінчення роботи і архівування результатів програм тестування (testware), середовища тестування та інфраструктури для подальшого використання;
* передача програм тестування організації техобслуговування;
* аналіз отриманого досвіду, поліпшення результатів тестів на зрілість.

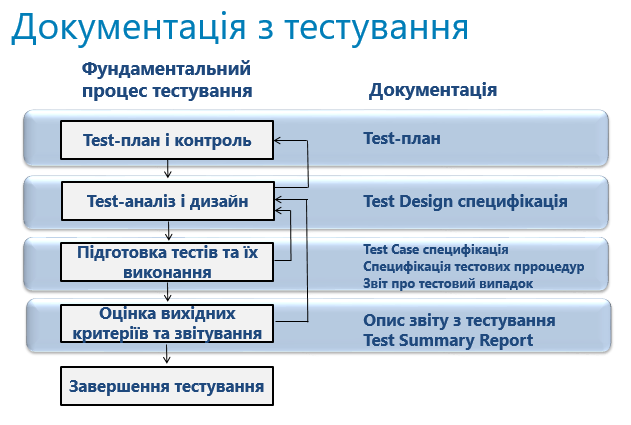


Рис. 18 – Документація з тестування

**Питання для самоконтролю:**

* 1. Назвіть основні етапи фундаментального процесу тестування.
  2. Які основні документи розробляються на цих етапах?

**Список використаних джерел:**

* 1. Дейкстра Дисциплина программирования [Текст] / Э. Дейкстра ; пер. с англ. И. Х. Зусман ; ред. Э. З. Любимский. — М. : Мир, 1978. — 275 с. — (Математическое обеспечение).].
  2. Липаев В.В. Тестирование программ [Текст] / В.В. Липаев. — М.: Радио и связь, 1986. — 296 с.
  3. ДСТУ 2844-94. Програмні засоби ЕОМ. Забезпечення якості. Терміни та визначення [Текст]. — Введ. 1.08.1995. — К.: Держстандарт України, 1995. — 57 с.
  4. ДСТУ 2850-94. Програмні засоби ЕОМ. Показники і методи оцінювання якості [Текст]. — К.: Держстандарт України, 1994. – 20 с.
  5. ISO/IEC 9126. 2001. Software engineering – Software product quality – Part 1: Quality model. Part 2: External metrics. Part 3: Internal metric. Part 4: Quality in use metrics [Text] — Geneva, Switzerland: International Organization for Standardization.].
  6. Laprie J.C. Dependable Computing and Fault Tolerance: Concepts and Terminology [Text] / J.C. Laprie // Proc. 15th IEEE Int'l Symp. Fault-Tolerant Computing (FTCS-15) — Ann Arbor, MI, June 1985. — pp. 2-11.
  7. Калбертсон Р. Быстрое тестирование / Р. Калбертсон, К. Браун, Г. Кобб. — М. : Издательский дом Вильямс, 2002. — 384 с.
  8. Канер С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес приложений / С. Канер. — К.: ДиаСофт, 2001. — 544 с.
  9. Кулямин В. В. Место тестирования среди методов оценки качества ПО / В. В. Кулямин, О. Л. Петренко. 2003 — 9 c.
  10. Patton R. Software Testing [Text] / R. Patton. — 2nd Edn. — Indianapolis: Sams, 2005. — 408p. — ISBN 0672327988.
  11. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем [Текст] / Б. Бейзер. — СбП: Питер, 2004. — 320 с. — ISBN 5-94723-698-2.
  12. Майерс Г.Дж. Искусство тестирования программ. (1979). Перевод с английского под редакцией Б.А. Позина. Переводчики: С.А. Блау, С.Г. Орлов, Б.А. Позин, Л.С. Черняк.