Кафедра - Комп'ютерна інженерія та програмування

Спеціальність - 123 Комп'ютерна інженерія

Освітня програма - Сучасне програмування, мобільні пристрої та комп'ютерні ігри

Рівень освіти - бакалавр

ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
курс лекцій

Лектор: Кандидат технічних наук, доцент
Сергій БУЛЬБА

Протокол засідання кафедри No. 5 від 20 січня, 2025 року

ЗМІСТ

Модель розробки програмного забезпечення

Модель розробки програмного забезпечення (Soft ware Development Model, SDM) — структура, яка систематизує різні види проектної діяльності, їхню взаємодію та послідовність у процесі розробки програмного забезпечення. Вибір тієї чи іншої моделі залежить від масштабу та складності проекту, предметної галузі, доступних ресурсів та множини інших факторів.

1. Каскадна (водоспадна) модель

Тестування з'являється лише з середини розвитку проекту, досягаючи свого максимуму наприкінці.

2. V-модель

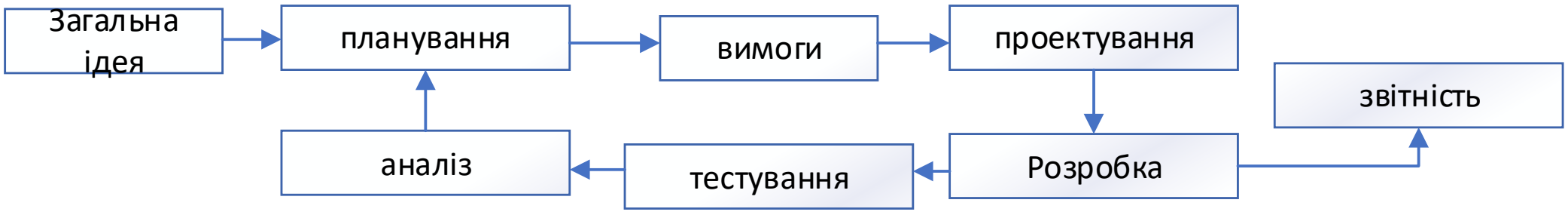
При використанні v-подібної моделі на кожній стадії на спуску потрібно думати про те, що і як відбуватиметься на відповідній стадії на підйомі. Тестування тут з'являється вже на ранніх стадіях розвитку проекту, що дозволяє мінімізувати ризики, а також виявити та усунути безліч потенційних проблем до того, як вони стануть проблемами реальними.

3. Ітеративна та інкрементальна модель

Ітераційна інкрементальна модель є фундаментальною основою сучасного підходу розробки ПЗ. Як випливає з назви моделі, їй властива певна двоїстість

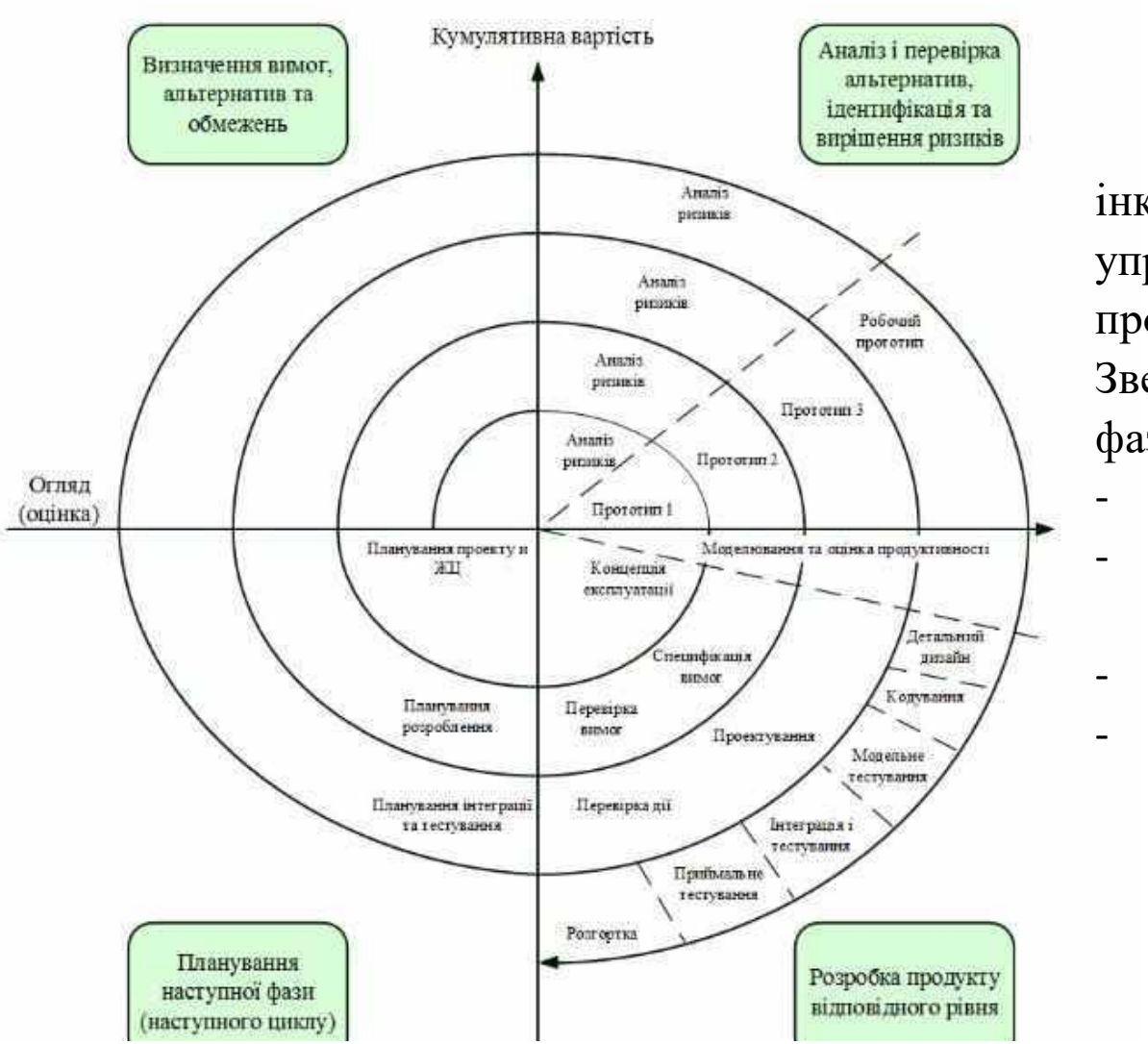
- з погляду життєвого циклу модель є ітераційною, тобто передбачає багаторазове повторення тих самих стадій;
- з погляду розвитку продукту (прирощення його корисних функцій) модель є інкрементальною.

Ключовою особливістю даної моделі є розбиття проекту на відносно невеликі проміжки (ітерації), кожен з яких у загальному випадку може включати всі класичні стадії, властиві водоспадній і v-подібної моделям



Ітераційна інкрементальна модель дуже добре зарекомендувала себе на об'ємних та складних проектах, які виконують великі команди протягом тривалих термінів. Проте до основних недоліків цієї моделі часто відносять високі накладні витрати, спричинені високою «бюрократизованістю» та загальною громіздкістю моделі.

4. Спіральна модель

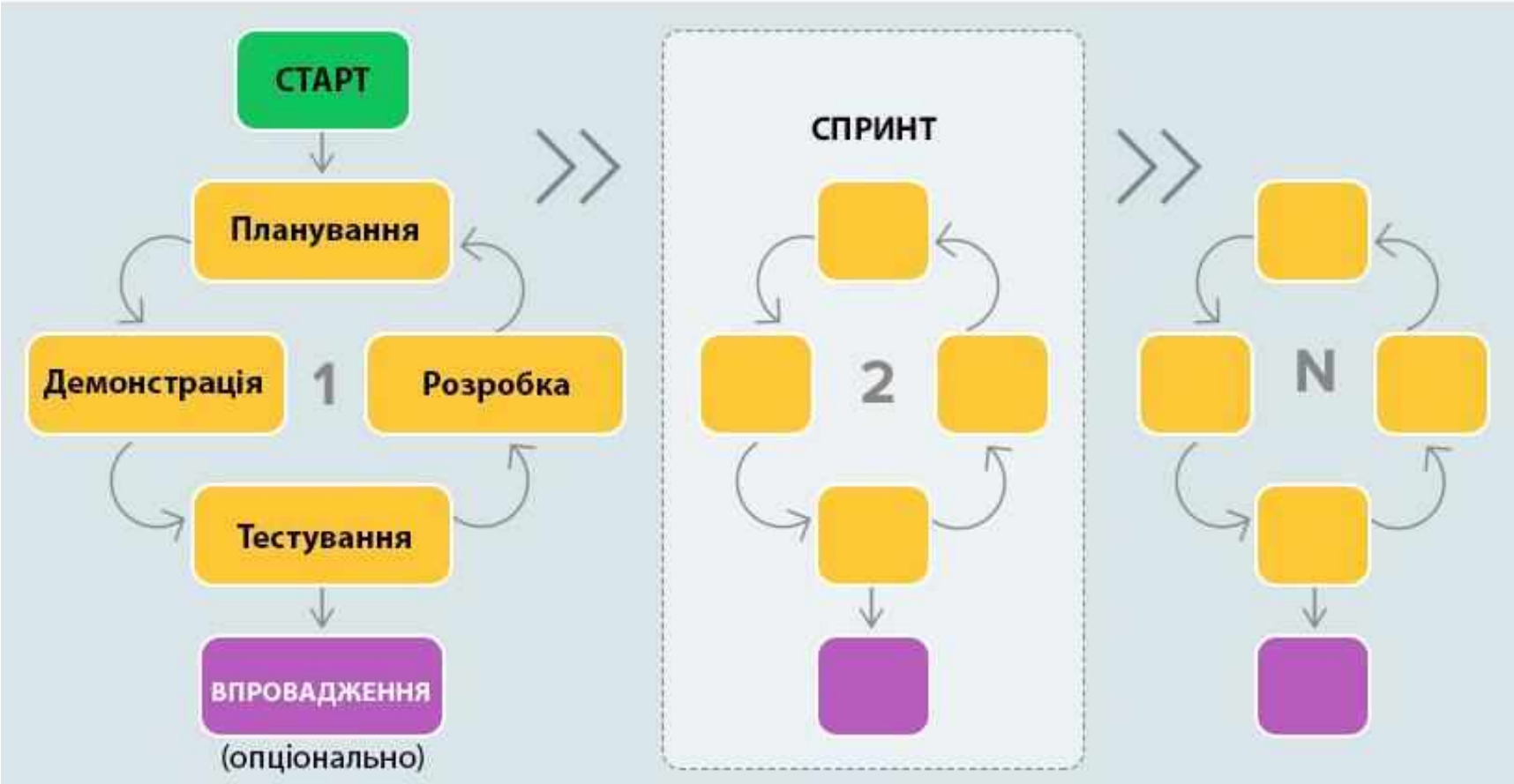


Спіральна модель є окремим випадком ітераційної інкрементальної моделі, в якій особлива увага приділяється управлінню ризиками, що особливо впливають на організацію процесу розробки проекту та контрольні точки.

Зверніть увагу на те, що тут явно виділено чотири ключові фази:

- Визначення вимог, альтернатив та обмежень;
- Аналіз та перевірка альтернатив, ідентифікація та вирішення ризиків;
- Розробка продукту відповідного рівня;
- Планування наступної фази (наступного циклу)

5. Гнучка модель (скрам)



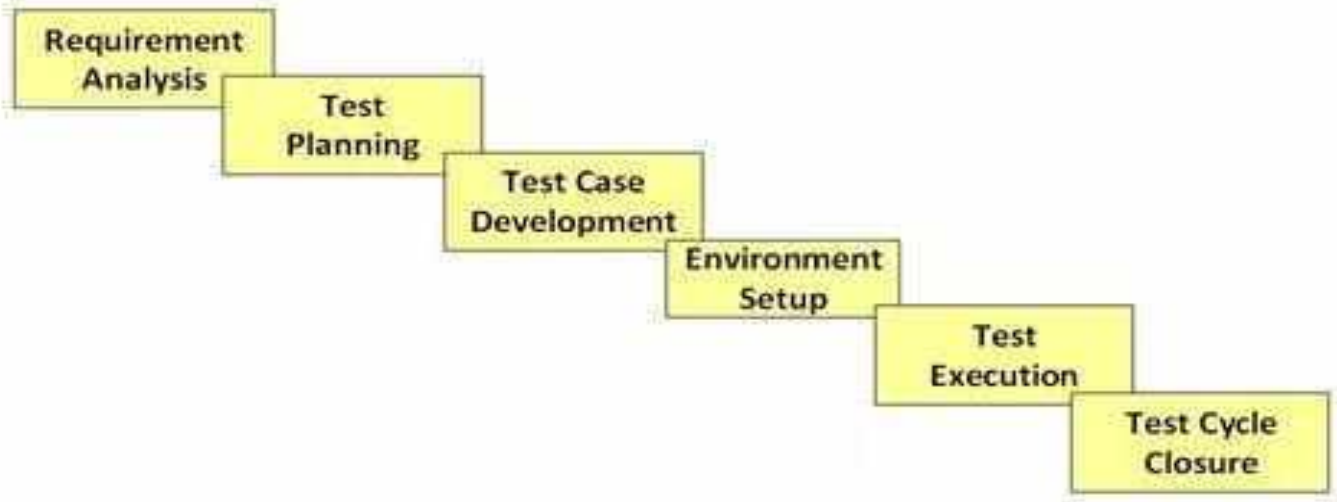
Модель	Переваги	Недоліки	Тестування
Каскад	• У кожній стадії є точний результат, що перевіряється. • Кожного часу команда виконує один вид роботи. • • Добре працює для невеликих завдань.	• Повна нездатність адаптувати проект до змін у вимогах. • Надзвичайно пізніше створення працюючого продукту..	З середини проекту
V-подібна	• Кожен етап має чіткий результат, який можна перевірити. • Увага приділяється тестуванню з самого першого етапу. • Добре працює для проектів зі стабільними вимогами.	• Відсутність гнучкості • і адаптивність. • Раннього прототипування немає. • Складність усунення проблем упущена на ранніх стадіях розробки проекту.	При переходах між етапами.
Ітеративно інкрементна	• Досить раннє створення прототипів. • Простота управління ітераціями. • Розкладіть проект на керовані ітерації.	• Відсутність гнучкості в межах ітерацій. • Складність усунення проблем упущена на ранніх стадіях розробки проекту.	• У певні моменти ітерацій. • Повторне тестування (після доопрацювання) того, що вже було протестовано раніше.
Спіральна	• Поглиблений аналіз ризиків. • Підходить для великих проектів. • Досить раннє створення прототипів.	• Високі накладні витрати. • Складність застосування для невеликих проектів. • Висока залежність успіху від якості аналізу ризиків.	• У певні моменти ітерацій. • Повторне тестування (після доопрацювання) того, що вже було протестовано раніше.
Гнучка	• Максимальне залучення клієнтів. • Багато роботи з вимогами. • Тісна інтеграція тестування та розробки. • Мінімізація документації.	• Складність реалізації для великих проектів. • Складність побудови стабільних процесів.	У певні моменти ітерацій і в будь-який необхідний момент.

Життєвий цикл тестування програмного забезпечення

Життєвий цикл тестування програмного забезпечення – це послідовність конкретних заходів, що проводяться в процесі тестування для забезпечення досягнення цілей щодо якості програмного забезпечення. Цей цикл – процес побудови і розвитку програмного забезпечення.

У кожній моделі життєвого циклу тестування програмного забезпечення може бути шість основних етапів:

1. Аналіз вимог
2. Планування тестів
3. Розробка кейсів
4. Налаштування тестового середовища
5. Виконання тесту
6. Закриття тестового циклу



ЦІЛІ ТЕСТУВАННЯ

Підвищити ймовірність того, що додаток, призначений для тестування, працюватиме правильно за будь-яких обставин.

Підвищити ймовірність того, що програма, призначена для тестування, буде відповідати всім описаним вимогам.

Надання актуальної інформації про стан продукту зараз.

Етапи тестування:

1. Аналіз продукту
2. Робота з вимогами
3. Розробка стратегії тестування
та планування процедур контролю якості
4. Створення тестової документації
5. Тестування прототипу
6. Основне тестування
7. Стабілізація
8. Експлуатація

Специфікація вимог

Специфікація вимог – це документ, в якому міститься набір вимог до програмного продукту. Вимоги структуруються та описують логіку роботи продукту (функціональні вимоги), його зовнішній вигляд (користувацький інтерфейс), обмеження в розробці, а також нефункціональні вимоги. Для опису функціональних вимог часто використовуються користувацькі сценарії (use cases). У користувацьких сценаріях представлені варіанти того, як користувач може взаємодіяти з ПЗ. Нефункціональні вимоги описують обмеження пов'язані з дизайном продукту та його реалізацією (продуктивність продукту, безпека, надійність, сумісність, проєктні обмеження, стандарти якості).

Рівні та типи вимог

- **Бізнес-вимоги** виражають мету, заради якої розробляється товар.
- **Вимоги користувача** описують завдання, які користувач може виконувати за допомогою системи, що розробляється.
- **Бізнес-правила** описують особливості прийнятих у предметній галузі процесів, обмежень та інших правил.
- **Атрибути якості** розширюють собою нефункціональні вимоги і на рівні вимог користувача можуть бути представлені у вигляді опису ключових для проекту показників якості.
- **Функціональні вимоги** описують поведінку системи, тобто. її дії (обчислення, перетворення, перевірки, обробку тощо.). У контексті проектування функціональні вимоги переважно впливають на дизайн системи.
- **Нефункціональні вимоги** описують властивості системи (зручність використання, безпека, надійність, розширюваність тощо), якими вона повинна мати при реалізації своєї поведінки.
- **Обмеження** є факторами, що обмежують вибір способів і засобів реалізації продукту.
- **Вимоги до інтерфейсів** описують особливості взаємодії системи, що розробляється, з іншими системами та операційним середовищем.
- **вимоги до даних** що описують структури даних (і самі дані), які є невід'ємною частиною системи, що розробляється. Часто сюди відносять опис бази даних та особливостей її використання.

Властивості вимог

Атомарність

Відповідно до цього критерію, вимоги повинні бути описані так, щоб їх не можна було уточнити ще детальніше або розбити одну вимогу на декілька. Приклад поганої вимоги – користувач може зареєструватися та додати у профіль особисту інформацію. Реєстрація та додавання інформації це дві різні функції, які повинні бути описані і уточнені в різних пунктах специфікації. Приклад хорошої вимоги – користувач може підписатися на оновлення інших користувачів.

Завершеність

Вимоги до одного функціоналу повинні бути описані в одному пункті специфікації. Не можна допускати ситуацію, коли один і той самий функціонал описаний в різних частинах документа.

Послідовність

Вимога не повинна суперечити іншим вимогам та обмеженням системи.

Наприклад: із соціальних мереж повинен запитуватися телефонний номер користувача. Очевидно, що дана вимога суперечить обмеженням соціальних мереж, так як номер телефону є прихованою інформацією, яку соціальні мережі не надають. Користувач повинен сам ввести номер телефону, щоб система його отримала. Подібних протиріч не повинно бути і між пунктами специфікації. Наприклад, одна і та ж кнопка повинна називатися однаково у всьому документі.

Відстеження (Трасування)

Критерій, який дозволяє зрозуміти, чому було прописано саме таку вимогу. Наприклад, для сайту з продажу алкогольних напоїв вимогою може бути реєстрація тільки повнолітніх користувачів у зв'язку із законодавчими обмеженнями.

Актуальність

Критерій, який перевіряє відповідність вимог до сучасних реалій (наприклад, із законодавчого/технічного боку або з боку зручності використання продукту, актуальності його користувацького інтерфейсу). Не варто прописувати у вимогах застарілі браузери, наприклад IE, або застарілі мобільні девайси та їх ОС, тому що вони практично не використовуються.

Здійсненність

Перевірка на те, що вимогу можливо реалізувати за допомогою актуальних існуючих на даний момент технологій. Наприклад, вимога того, що система повинна давати швидкий відгук (тут потрібна конкретизація – не довше, ніж 3 секунди) при помірному навантаженні (тут також потрібна конкретизація за кількістю користувачів та виконуваних ними дій) може існувати. Але вимога відгуку менше секунди при дуже великому навантаженні – нездійсненна в сучасних реаліях вимога.

Зрозумілість (доступність)

Вимога має бути сформульована достатньо чітко, конкретно та однозначно, для того, щоб вона однаково розумілася всією командою розробки.

Верифікованість

Критерій, за яким визначається, чи можна порівняти готовий продукт з вимогою і перевірити, чи виконується вона. Якщо вимога описана дуже розмито, перевірити її не вийде. Наприклад: фото повинно завантажуватися швидко/налаштування профілю повинні бути інтуїтивно-зрозумілими/оформлення замовлення повинно відбуватись легко. Для того, щоб перевірити такі вимоги, явно необхідні уточнення, як це швидко/інтуїтивно-зрозуміло/легко.

Обов'язковість

Визначення того, наскільки важливе та пріоритетне виконання даної вимоги. Всім вимогам в специфікації повинні бути призначені важливість, стабільність та пріоритет. За допомогою цього вибираються першочергові значення для виконання командою розробки завдання.

Ідентифікація

У кожної вимоги повинен бути унікальний ідентифікатор, за допомогою якого вимога прив'язується до інших артефактів проєкту. Наприклад, в описі тест-кейса вказується ідентифікатор вимоги, яка перевіряється цим тест-кейсом.

Модифікація

Вимоги повинні бути легко модифіковані (внесення змін) за необхідності.

Повнота

Найскладніший критерій, згідно з яким вимоги повинні вичерпно описувати весь функціонал системи. Все, що система повинна виконувати, повинно бути зафіксовано у вимогах, інакше можна зробити серйозну помилку під час проєктування архітектури. Складність у тому, що на початкових етапах проєктування системи дуже важко точно вказати та описати всі функції. Вимоги спочатку описуються більш загальними твердженнями, а далі уточнюються.

Техніки тестування вимог

Тестування документації та вимог стосується розряду нефункціонального тестування. Основні техніки такого тестування у контексті вимог такі.

Взаємний перегляд. Взаємний перегляд («рецензування») є однією з найбільш активно використовуваних технік тестування вимог і може бути представлений в одній з трьох наступних форм (у міру наростання його складності та ціни):

• **Побіжний перегляд** може виражатися як у показ автором своєї роботи колегам з метою створення загального розуміння та отримання зворотного зв'язку, так і в простому обміні результатами роботи між двома і більше авторами для того, щоб колега висловив свої питання та зауваження. Це найшвидший, найдешевший і найпоширеніший вид перегляду. Для запам'ятовування: аналог побіжного перегляду — це ситуація, коли ви в школі з однокласниками перевіряли перед складанням один одного, щоб знайти описки та помилки.

• **Технічний перегляд** виконується групою спеціалістів. В ідеальній ситуації кожен фахівець повинен представляти свою галузь знань. Тестований продукт не може вважатися досить якісним, поки що хоча б у одного проглядаючого залишаються зауваження. Для запам'ятовування: аналог технічного перегляду — це ситуація, коли певний договір визує юридичний відділ, бухгалтерія тощо.

• **Формальна інспекція** є структурованим, систематизованим та документованим підходом до аналізу документації. Для його виконання залучається велика кількість спеціалістів, саме виконання займає досить багато часу, і тому цей варіант перегляду використовується досить рідко (як правило, при отриманні на супровід та доопрацювання проекту, створенням якого раніше займалася інша компанія).Для запам'ятовування: аналог формальної інспекції — це ситуація генерального прибирання квартири (включаючи вміст усіх шаф, холодильника, комори тощо).

Запитання. Наступною очевидною технікою тестування та підвищення якості вимог є (повторне) використання технік виявлення вимог, а також (як окремий вид діяльності) – питання. Якщо хоч щось у вимогах викликає у вас нерозуміння чи підозру — запитуйте. Можна спитати представників замовника, можна звернутися до довідкової інформації. З багатьох питань можна звернутися до більш досвідчених колег за умови, що вони мають відповідну інформацію, раніше отриману від замовника. Головне, щоб ваше питання було сформульоване таким чином, щоб отримана відповідь дозволила покращити вимоги.

Тест-кейси та чек-листи. Ми пам'ятаємо, що хороша вимога є перевіреною, а отже, мають існувати об'єктивні способи визначення того, чи правильно реалізовано вимогу. Продумування чек-аркушів або навіть повноцінних тест-кейсів у процесі аналізу вимог дозволяє нам визначити, наскільки вимогу перевіряємо. Якщо ви можете швидко вигадати кілька пунктів чек-листа, це ще не ознака того, що з вимогою все добре (наприклад, воно може суперечити якимось іншим вимогам). Але якщо жодних ідей щодо тестування вимоги на думку не спадає — це тривожний знак.

Дослідження поведінки системи. Ця техніка логічно випливає з попередньої (продумування тест-кейсів та чек-листів), але відрізняється тим, що тут тестуванню піддається, як правило, не одна вимога, а цілий набір. Тестувальник подумки моделює процес роботи користувача із системою, створеною за тестованими вимогами, і шукає неоднозначні або зовсім неописані варіанти поведінки системи.

Графічне уявлення. Щоб побачити загальну картину вимог цілком, дуже зручно використовувати малюнки, схеми, діаграми, інтелект-карти тощо. Графічне уявлення зручне одночасно своєю наочністю та стислістю.

Прототипування. Можна сказати, що прототипування часто є наслідком створення графічного представлення та аналізу поведінки системи.

Приклад документів

Нижче наведено структуру SRS, рекомендовану цим стандартом **IEEE 830**:

1. Введення
- Мета
- Умовні позначення термінів
- Передбачувана аудиторія і послідовність сприйняття
- Масштаби проекту
- Посилання на джерела

2. Загальний опис
- Бачення продукту
- функціональність продукту
- Класи та характеристики користувачів
- Операційне середовище продукту (операційне середовище)
- Рамки, обмеження, правила та стандарти
- Документація для користувачів
- Припущення і залежності

3. функціонал системи
- Функціональний блок X (таких блоків може бути більше одного)
  - •Опис і пріоритет
  - •Причинно-наслідкові зв'язки, алгоритми (рух процесів, робочі процеси)
  - •Функціональні вимоги

4. Вимоги до зовнішнього інтерфейсу
- Інтерфейси користувача (UX)
- Інтерфейси прикладного програмування
- Апаратні інтерфейси
- Комунікаційні та комунікаційні інтерфейси

5. Нефункціональні вимоги
- Вимоги до продуктивності
- Вимоги до безпеки (даних)
- Критерії якості програмного забезпечення
- Вимоги безпеки системи

6. Інші вимоги
- Додаток A: Глосарій
- Додаток B: Моделі процесів і доменів та інші діаграми
- Додаток B: Перелік ключових завдань

КЛАСИФІКАЦІЯ ТЕСТУВАННЯ

По запуску коду виконання:

- Статичне тестування — без запуску.
- Динамічне тестування - із запуском. •

За доступом до коду та архітектури програми:

- Метод білої скриньки — доступ до коду є.
- Метод чорної скриньки – доступу до коду немає.
- Метод сірої скриньки – до частини коду доступ є, до частини – ні.

За ступенем автоматизації:

- Ручне тестування – тест-кейси виконує людина.
- Автоматизоване тестування — тест-кейси частково або повністю виконує спеціальний інструментальний засіб.

За рівнем деталізації програми (за рівнем тестування):

- Модульне (компонентне) тестування — перевіряються окремі невеликі частини програми.
- Інтеграційне тестування — перевіряється взаємодія між кількома частинами програми.

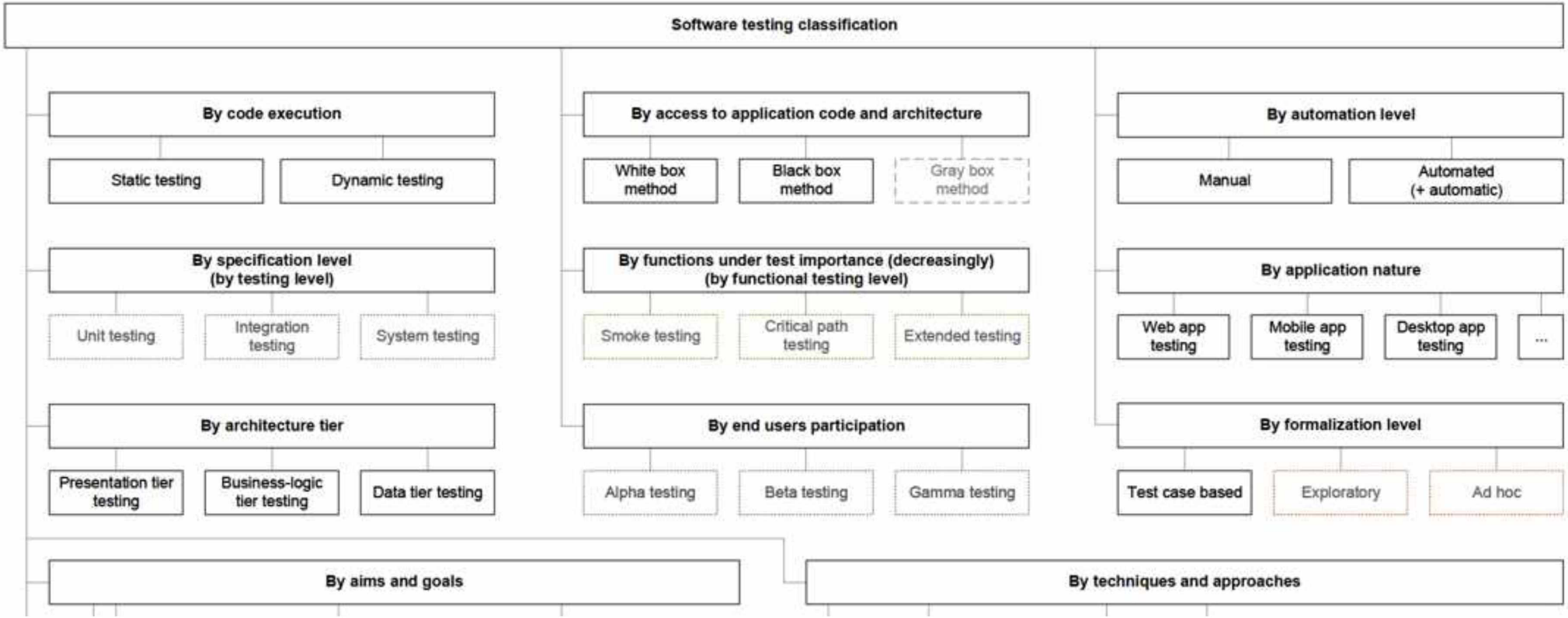
Системне тестування — програма перевіряється як єдине ціле.

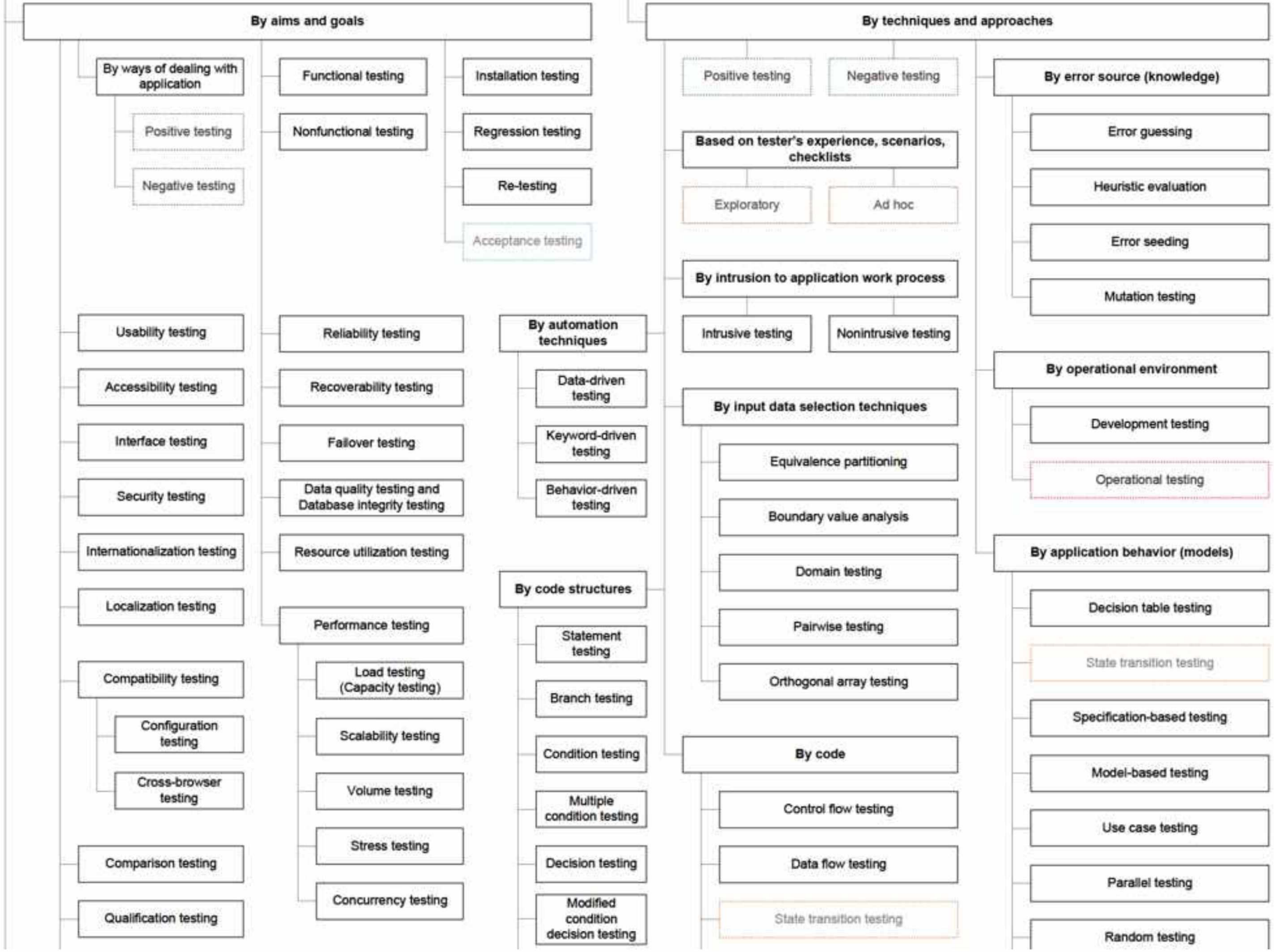
За (зменшенням) ступенем важливості тестованих функцій (за рівнем функціонального тестування):

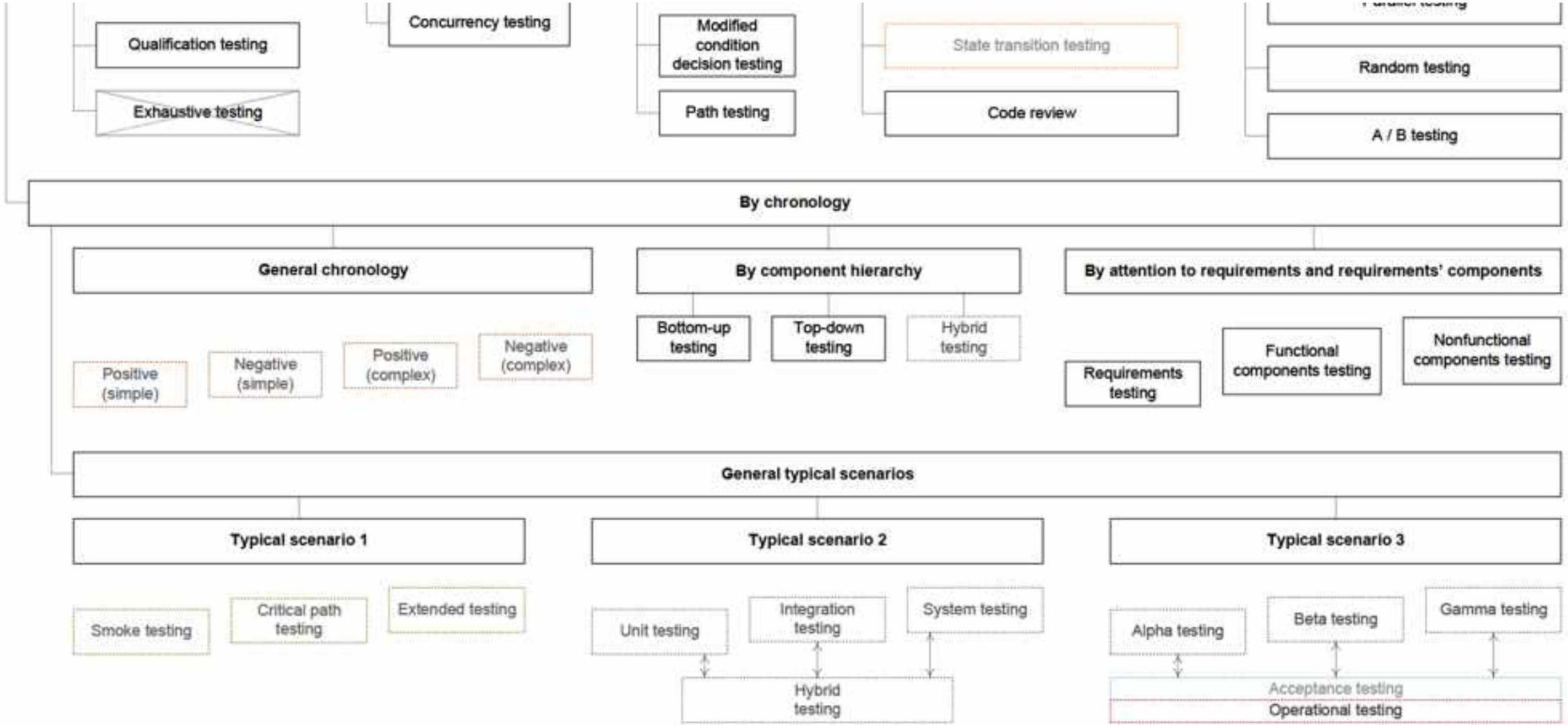
- Димове тестування — перевірка найважливішої, найголовнішої функціональності, непрацездатність якої робить безглуздою саму ідею використання програми.
- Тестування критичного шляху — перевірка функціональності, яка використовується типовими користувачами у типовій повсякденній діяльності.
- Розширене тестування — перевірка всієї (іншої) функціональності, заявленої у вимогах.

За принципами роботи з додатком:

- Позитивне тестування — всі дії з додатком виконуються суворо за інструкцією без жодних неприпустимих дій, некоректних даних тощо. Можна сказати, що додаток досліджується в «тепличних умовах».
- Негативне тестування — у роботі з додатком виконуються (некоректні) операції та використовуються дані, які потенційно призводять до помилок (класика жанру — розподіл на нуль).


Software testing classification
By code execution
Static testing
Dynamic testing
By specification level (by testing level)
Unit testing
Integration testing
System testing
By architecture tier
Presentation tier testing
Business-logic tier testing
Data tier testing
By access to application code and architecture
White box method
Black box method
Gray box method
By functions under test importance (decreasingly) (by functional testing level)
Smoke testing
Critical path testing
Extended testing
By end users participation
Alpha testing
Beta testing
Gamma testing
By automation level
Manual
Automated (+ automatic)
By application nature
Web app testing
Mobile app testing
Desktop app testing
...
By formalization level
Test case based
Exploratory
Ad hoc
By aims and goals
By techniques and approaches


By aims and goals
By ways of dealing with application
Positive testing
Negative testing
Functional testing
Nonfunctional testing
Installation testing
Regression testing
Re-testing
Acceptance testing
Usability testing
Accessibility testing
Interface testing
Security testing
Internationalization testing
Localization testing
Compatibility testing
Configuration testing
Cross-browser testing
Comparison testing
Qualification testing
Reliability testing
Recoverability testing
Failover testing
Data quality testing and Database integrity testing
Resource utilization testing
Performance testing
Load testing (Capacity testing)
Scalability testing
Volume testing
Stress testing
Concurrency testing
By automation techniques
Data-driven testing
Keyword-driven testing
Behavior-driven testing
By code structures
Statement testing
Branch testing
Condition testing
Multiple condition testing
Decision testing
Modified condition decision testing
By techniques and approaches
Positive testing
Negative testing
Based on tester's experience, scenarios, checklists
Exploratory
Ad hoc
By intrusion to application work process
Intrusive testing
Nonintrusive testing
By input data selection techniques
Equivalence partitioning
Boundary value analysis
Domain testing
Pairwise testing
Orthogonal array testing
By code
Control flow testing
Data flow testing
State transition testing
By error source (knowledge)
Error guessing
Heuristic evaluation
Error seeding
Mutation testing
By operational environment
Development testing
Operational testing
By application behavior (models)
Decision table testing
State transition testing
Specification-based testing
Model-based testing
Use case testing
Parallel testing
Random testing


Qualification testing
Exhaustive testing
Concurrency testing
Modified condition decision testing
Path testing
State transition testing
Code review
Random testing
A / B testing
By chronology
General chronology
Positive (simple)
Negative (simple)
Positive (complex)
Negative (complex)
By component hierarchy
Bottom-up testing
Top-down testing
Hybrid testing
By attention to requirements and requirements' components
Requirements testing
Functional components testing
Nonfunctional components testing
General typical scenarios
Typical scenario 1
Smoke testing
Critical path testing
Extended testing
Typical scenario 2
Unit testing
Integration testing
System testing
Hybrid testing
Typical scenario 3
Alpha testing
Beta testing
Gamma testing
Acceptance testing
Operational testing

КЛАСИФІКАЦІЯ ЗА ЗАПУСКУ КОДУ НА ВИКОНАННЯ

Статичне тестування (static testing) — тестування без запуску коду на виконання. В рамках цього підходу тестуванню можуть бути піддані:

- Документи (вимоги, тест-кейси, описи архітектури додатка, схеми баз даних тощо).
- Графічні прототипи (наприклад, ескізи інтерфейсу користувача).
- Код програми (що часто виконується самими програмістами в рамках аудиту коду (code review), що є специфічною варіацією взаємного перегляду у застосуванні до вихідного коду). Код програми також можна перевіряти з використанням технік тестування на основі структур коду.
- Параметри (налаштування) середовища виконання програми.
- Підготовлені тестові дані.

Динамічне тестування (dynamic testing) — тестування із запуском коду на виконання. Запускатися на виконання може як код всієї програми повністю (системне тестування), так і код кількох взаємопов'язаних частин (інтеграційне тестування), окремих частин (модульне або компонентне тестування) і навіть окремі ділянки коду. Основна ідея цього виду тестування полягає в тому, що перевіряється реальна поведінка (частини) програми.

КЛАСИФІКАЦІЯ ЗА ДОСТУПОМ ДО КОДУ І АРХІТЕКТУРИ ДОДАТКА

Метод білої скриньки — тестувальник має доступ до внутрішньої структури та коду програми, а також достатньо знань для розуміння побаченого. Виділяють навіть супутню тестування методом білого ящика глобальну техніку — тестування з урахуванням дизайну.

Метод чорної скриньки — у тестувальника або немає доступу до внутрішньої структури та коду додатка, або недостатньо знань для їх розуміння, або він свідомо не звертається до них у процесі тестування.

В рамках тестування за методом чорної скриньки основною інформацією для створення тест-кейсів виступає документація (особливо вимоги) та загальний здоровий глузд (для випадків, коли поведінка додатку в деякій ситуації не регламентована явно; іноді; це називають «тестуванням на основі неявних вимог», але канонічного визначення цього підходу немає).

Метод сірої скриньки — комбінація методів білої скриньки та чорної скриньки, яка полягає в тому, що до частини коду та архітектури у тестувальника доступ є, а до частини — ні.

	Переваги	Недоліки
Метод «білої скриньки»	• Показує приховані проблеми і спрощує їх діагностику. • Це дозволяє досить просто автоматизувати тестові кейси та їх реалізацію на самих ранніх стадіях розробки проекту. • Він має розвинену систему метрик, збір та аналіз яких легко автоматизується. • Заохочує розробників писати якісний код. • Багато методик цього методу є перевіреними, усталеними рішеннями, заснованими на строгому технічному підході.	• Її не можуть виконати тестувальники, які не володіють достатніми знаннями в області програмування. • Тестування орієнтоване на реальну функціональність, що збільшує ймовірність відсутності нереалізованих - вимог. • Поведінка програми досліджується у відриві від реального середовища виконання і не враховує його вплив. • Поведінка програми досліджується ізольовано від реальних сценаріїв користувача{148}.
Метод чорної скриньки	• Тестувальник не обов'язково повинен володіти (глибокими) знаннями в області програмування. • Поведінка програми досліджується в контексті реального часу виконання та враховує його вплив. • Поведінка програми досліджується в контексті реальних сценаріїв користувача{148}. • Тест-кейси можна створювати, як тільки з'являються стабільні вимоги. • Процес створення тест-кейсів дозволяє виявити дефекти вимог. • Дозволяє створювати тестові приклади, які можна повторно використовувати в різних проектах.	• Є можливість повторити деякі тестові ключі, вже виконані розробниками. • Існує велика ймовірність того, що деякі з можливих поведінки додатків залишаться неперевіреними. • Для розробки високопродуктивних тест-кейсів необхідна якісна документація. • Діагностика виявлених дефектів складніше в порівнянні з методиками методу білого ящика. • Через широкий спектр методик і підходів складно планувати і оцінювати трудовитрати. • У разі автоматизації можуть знадобитися складні, дорогі інструменти.
Метод сірої скриньки	Поєднує в собі переваги і недоліки методів білого і чорного ящика.	

КЛАСИФІКАЦІЯ ЗА СТУПЕНЬЮ АВТОМАТИЗАЦІЇ

• **Ручне тестування** (manual testing) — тестування, в якому тест-кейси виконуються людиною вручну без використання засобів автоматизації. Незважаючи на те, що це звучить дуже просто, від тестувальника в ті чи інші моменти часу потрібні такі якості, як терплячість, спостережливість, креативність, уміння ставити нестандартні експерименти, а також уміння бачити і розуміти, що відбувається «всередині системи»

• **Автоматизоване тестування** (automated testing, test automation) – набір технік, підходів та інструментальних засобів, що дозволяє виключити людину з виконання деяких завдань у процесі тестування. Тест-кейси частково або повністю виконує спеціальний інструментальний засіб, проте розробка тест-кейсів, підготовка даних, оцінка результатів виконання, написання звітів про виявлені дефекти — все це і багато іншого робить людина.

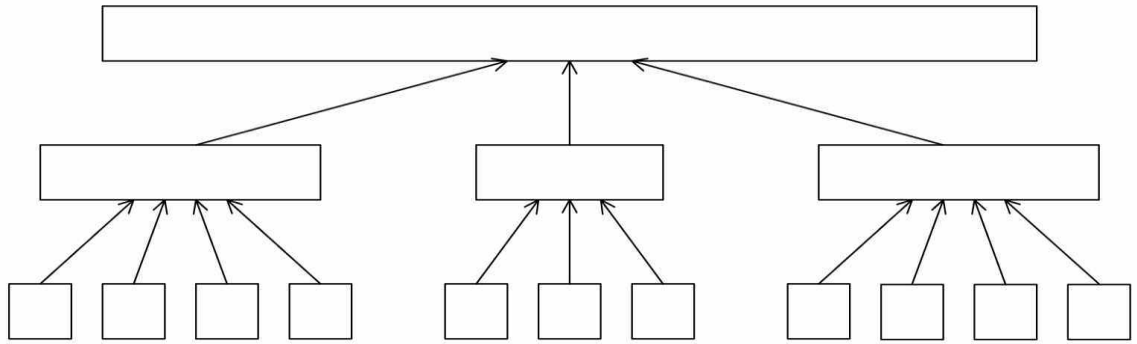
Переваги	Недоліки
• Швидкість виконання тестових випадків може бути в кілька разів і на порядки більше, ніж можливості людини. • Відсутність впливу людського фактора в процесі виконання тест-кейсів (втома, неуважність • Мінімізація витрат при повторному виконанні тест-кейсів (участь людини тут потрібно лише епізодично). • Здатність засобів автоматизації виконувати тест-кейси, в принципі, нестерпна для людини в силу своєї складності, швидкості або інших факторів. • Здатність засобів автоматизації збирати, зберігати, аналізувати, агрегувати і представляти величезні обсяги даних в зручній для людини формі. • Можливість засобів автоматизації виконувати низькорівневі дії з додатком, операційною системою, каналами передачі даних	• Висококваліфікований персонал необхідний у зв'язку з тим, що автоматизація - це «проект в рамках проекту» (зі своїми вимогами, планами, кодом) • Високі витрати на складні засоби автоматизації, розробку та підтримку коду тест-кейсу. • Автоматизація вимагає більш ретельного планування та управління ризиками, оскільки в противному випадку проект може бути серйозно пошкоджений. • Інструментів автоматизації дуже багато, що ускладнює проблему вибору того чи іншого інструменту і може спричинити за собою фінансові витрати (і ризики), необхідність навчання персоналу (або пошук фахівців). • У разі відчутної зміни вимог, зміни технологічної області, обробки інтерфейсів (як призначених для користувача, так і програмних) багато тест-кейси безнадійно застарівають і вимагають повторного створення.

КЛАСИФІКАЦІЯ З РІВНЯ ДЕТАЛІЗАЦІЇ ДОДАТКА (ЗА РІВНЕМ ТЕСТУВАННЯ)

Модульне (компонентне) тестування - спрямоване на перевірку окремих невеликих частин програми, які (як правило) можна досліджувати ізольовано від інших подібних частин. Під час виконання цього тестування можуть перевірятись окремі функції або методи класів, самі класи, взаємодія класів, невеликі бібліотеки, окремі частини додатка. Часто даний вид тестування реалізується з використанням спеціальних технологій та інструментальних засобів автоматизації тестування, які значно спрощують та прискорюють розробку відповідних тест-кейсів.

Інтеграційне тестування спрямоване на перевірку взаємодії між декількома частинами додатка, кожна з яких, у свою чергу, перевірено окремо на стадії модульного тестування). На жаль, навіть якщо ми працюємо з дуже якісними окремими компонентами, «на стику» їхньої взаємодії часто виникають проблеми. Саме ці проблеми виявляє інтеграційне тестування.

Системне тестування - спрямовано перевірку всього додатка як єдиного цілого, зібраного з частин, перевірених двох попередніх стадіях. Тут не тільки виявляються дефекти «на стиках» компонентів, а й з'являється можливість повноцінно взаємодіяти з додатком з точки зору кінцевого користувача



КЛАССИФИКАЦИЯ ПО (УБЫВАНИЮ) СТЕПЕНИ ВАЖНОСТИ ТЕСТИРУЕМЫХ ФУНКЦИЙ (ПО УРОВНЮ ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ)

Димове тестування - спрямоване на перевірку, найважливішої, найголовнішої функціональності, непрацездатність якої робить безглуздою саму ідею використання програми (або іншого об'єкта, що піддається димовому тестуванню).

Димове тестування проводиться після виходу нового білда, щоб визначити загальний рівень якості програми та прийняти рішення про (не)доцільність виконання тестування критичного шляху та розширеного тестування. Оскільки тест-кейсів на рівні димового тестування відносно небагато, а самі вони досить прості, але дуже часто повторюються, вони є хорошими кандидатами на автоматизацію.

Тестування критичного шляху - спрямоване на дослідження функціональності, що використовується типовими користувачами у типовій повсякденній діяльності. Саме ці функції і потрібно перевірити, як ми переконалися, що додаток «в принципі працює» (димовий тест пройшов успішно). Якщо з якихось причин програма не виконує ці функції або виконує їх некоректно, багато користувачів не зможуть досягти безлічі своїх цілей. Порогове значення метрики успішного проходження «тесту критичного шляху» вже трохи нижче, ніж у димовому тестуванні, але все одно досить високо (як правило, близько 70–80–90 % — залежно від суті проекту)

Розширене тестування - спрямоване на дослідження всієї заявленої у вимогах функціональності — навіть тієї, яка низько проранжована за ступенем важливості. При цьому тут також враховується, яка функціональність є більш важливою, а яка менш важливою. Але за наявності достатньої кількості часу та інших ресурсів тест-кейси цього рівня можуть торкнутися навіть найнижчих пріоритетів.

КЛАСИФІКАЦІЯ З ПРИНЦИПІВ РОБОТИ З ДОДАТКОМ

Позитивне тестування (positive testing) спрямоване на дослідження програми у ситуації, коли всі дії виконуються суворо за інструкцією без будь-яких помилок, відхилень, введення невірних даних тощо. Якщо позитивні тест-кейси завершуються помилками, це тривожна ознака — програма працює неправильно навіть в ідеальних умовах (і можна припустити, що в неідеальних умовах вона працює ще гірше).

Для прискорення тестування кілька позитивних тест-кейсів можна поєднувати (наприклад, перед відправкою заповнити всі поля форми правильними значеннями) — іноді це може ускладнити діагностику помилки, але суттєва економія часу компенсує цей ризик.

Негативне тестування — спрямоване на дослідження роботи програми у ситуаціях, коли з ним виконуються (некоректні) операції та/або використовуються дані, які потенційно призводять до помилок (класика жанру — поділ на нуль). Оскільки в реальному житті таких ситуацій значно більше (користувачі припускаються помилок, зловмисники усвідомлено «ламають» додаток, у середовищі роботи програми виникають проблеми тощо), негативних тест-кейсів виявляється значно більше, ніж позитивних. На відміну від позитивних, негативні тест-кейси не варто об'єднувати, бо подібне рішення може призвести до неправильного трактування поведінки програми та пропуску (невиявлення) дефектів.

КЛАСИФІКАЦІЯ З ПРИРОДИ ДОДАТКУ

Тестування веб-додатків пов'язане з інтенсивною діяльністю в галузі тестування сумісності (особливо - крос-браузерного тестування), тестування продуктивності, автоматизації тестування з використанням широкого спектру інструментальних засобів.

Тестування мобільних додатків також потребує підвищеної уваги до тестування сумісності, оптимізації продуктивності (у тому числі клієнтської частини з точки зору зниження енергоспоживання), автоматизації тестування із застосуванням емуляторів мобільних пристроїв.

Тестування настільних додатків є найкласичнішим серед усіх перелічених у даній класифікації, і його особливості залежать від предметної галузі застосування, нюансів архітектури, ключових показників якості.

КЛАСИФІКАЦІЯ З ФОКУСУВАННЯ НА РІВНІ АРХІТЕКТУРИ ДОДАТКА

• **Тестування рівня представлення** (presentation tier testing) сконцентроване на тій частині додатка, яка відповідає за взаємодію із «зовнішнім світом» (користувачами та іншими додатками). Тут досліджуються питання зручності використання, швидкості відгуку інтерфейсу, сумісності з браузерами, коректності роботи інтерфейсів.

• **Тестування рівня бізнес-логіки** (business logic tier testing) відповідає за перевірку основного набору функцій програми та будується на базі ключових вимог до додатку, бізнес-правил та загальної перевірки функціональності.

• **Тестування рівня даних** (data tier testing) сконцентровано на тій частині додатку, яка відповідає за зберігання та деяку обробку даних (найчастіше — у базі даних чи іншому сховищі). Тут особливий інтерес є тестування даних, перевірка дотримання бізнес-правил, тестування продуктивності.

КЛАСИФІКАЦІЯ З ЗАЛУЧЕННЯ КІНЦЕВИХ КОРИСТУВАЧІВ

• **Альфа-тестування** (alpha testing) виконується всередині організації-розробника з можливим частковим залученням кінцевих користувачів. Може бути формою внутрішнього приймального тестування. Суть цього виду коротко: продукт вже можна періодично показувати зовнішнім користувачам, але він ще досить «сирий», тому основне тестування виконується організацією-розробником.

• **Бета-тестування** (beta testing) виконується поза організацією-розробником з активним залученням кінцевих користувачів/замовників. Може бути формою зовнішнього приймального тестування. Суть цього виду коротко: продукт вже можна відкрито показувати зовнішнім користувачам, він вже досить стабільний, але проблеми все ще можуть бути, і для їх виявлення потрібний зворотний зв'язок від реальних користувачів.

• **Гамма-тестування** (gamma testing) — фінальна стадія тестування перед випуском продукту, спрямована на виправлення незначних дефектів, виявлених у бета-тестуванні. Як правило, також виконується з максимальним залученням кінцевих користувачів/замовників. Може бути формою зовнішнього приймального тестування. Суть цього виду коротко: продукт вже майже готовий, і зараз зворотний зв'язок від реальних користувачів використовується для усунення останніх недоробок.

КЛАСИФІКАЦІЯ ЗА СТУПЕНЬЮ ФОРМАЛІЗАЦІЇ

Тестування на основі тест-кейсів (scripted testing, test case based testing) — формалізований підхід, в якому тестування проводиться на основі підготовлених тест-кейсів, наборів тест-кейсів та іншої документації. Це найпоширеніший спосіб тестування, який також дозволяє досягти максимальної повноти дослідження програми за рахунок суворої систематизації процесу, зручності застосування метрик та широкого набору вироблених за десятиліття та перевірених на практиці рекомендацій.

Дослідницьке тестування (exploratory testing) — частково формалізований підхід, у межах якого тестувальник виконує роботу з додатком по обраному сценарію, який, своєю чергою, допрацьовується у виконання з метою повного дослідження докладання. Ключовим фактором успіху при виконанні дослідницького тестування є робота за сценарієм, а не виконання розрізнених бездумних операцій. Існує навіть спеціальний сценарний підхід, який називають сесійним тестуванням (session-based testing). В якості альтернативи сценаріям при виборі дій з додатком іноді можуть використовуватися чек-аркуші, і тоді цей вид тестування називають тестуванням на основі чек-аркушів (checklist-based testing).

Вільне (інтуїтивне) тестування (ad hoc testing) — повністю неформалізований підхід, у якому не передбачається використання ні тест-кейсів, ні чек-листів, ні сценаріїв — тестувальник повністю спирається на свій професіоналізм та інтуїцію (experience-based testing) спонтанного виконання з додатком дій, які, як він вважає, можуть виявити помилку. Цей вид тестування використовується рідко і виключно як доповнення до повністю або частково формалізованого тестування у випадках, коли для дослідження певного аспекту поведінки програми (поки що?) немає тест-кейсів.

КЛАСИФІКАЦІЯ ЗА МЕТОЮ ТА ЗАВДАННЯМИ

• **Функціональне тестування** (functional testing) — вид тестування, спрямований на перевірку коректності роботи функціональності програми (коректність реалізації функціональних вимог). Часто функціональне тестування асоціюють з тестуванням за методом чорного ящика, проте і методом білого ящика цілком можна перевіряти коректність реалізації функціональності.

- функціональне тестування (як антонім нефункціонального) спрямоване на перевірку того, які функції програми реалізовані, і що вони працюють правильно;

- тестування функціональності спрямоване на ті ж завдання, але акцент зміщений у бік дослідження додатка в реальному робочому середовищі, після локалізації та в подібних ситуаціях.

• **Нефункціональне тестування** (non-functional testing) — вид тестування, спрямований на перевірку нефункціональних особливостей програми (коректність реалізації нефункціональних вимог), таких як зручність використання, сумісність, продуктивність, безпека тощо.

• **Тестування безпеки** (security testing) — тестування, спрямоване на перевірку здатності програми протистояти зловмисним спробам отримання доступу до даних або функцій, права на доступ до яких зловмисник не має.

• **Тестування інтернаціоналізації** — тестування, спрямоване на перевірку готовності продукту до роботи з використанням різних мов та з урахуванням різних національних та культурних особливостей. Цей вид тестування не має на увазі перевірки якості відповідної адаптації (цим займається тестування локалізації), воно сфокусоване саме на перевірці можливості такої адаптації (наприклад: що буде, якщо відкрити файл з ієрогліфом в імені; як буде працювати інтерфейс, якщо все перекласти на японський, чи може програма шукати дані в тексті корейською і т.д.).

• **Інсталяційне тестування** (installation testing, installability testing) — тестування, спрямоване на виявлення дефектів, що впливають на перебіг стадії інсталяції (установки) програми. У загальному випадку таке тестування перевіряє безліч сценаріїв та аспектів роботи інсталятора в таких ситуаціях, як:

- нове середовище виконання, в якому програма раніше не була інстальована;
- оновлення існуючої версії («апгрейд»);
- зміна поточної версії більш стару («даунгрейд»);
- повторне встановлення програми з метою усунення проблем, що виникли («переінсталяція»);
- повторний запуск інсталяції після помилки, що призвела до неможливості продовження інсталяції;
- видалення програми;
- встановлення нової програми із сімейства додатків;
- автоматична інсталяція без участі користувача.

• **Регресійне тестування** (regression testing) — тестування, спрямоване на перевірку того факту, що у раніше працездатній функціональності не з'явилися помилки, спричинені змінами у додатку чи середовищі його функціонування. Тому регресійне тестування є невід'ємним інструментом забезпечення якості та активно використовується практично у будь-якому проекті.

• **Повторне тестування** (re-testing, confi rmation testing) — виконання тест-кейсів, які раніше виявили дефекти, з метою підтвердження усунення дефектів. Фактично цей вид тестування зводиться до дій на фінальній стадії життєвого циклу звіту про дефект, спрямований на те, щоб перевести дефект у стан «перевірений» та «закритий».

• **Тестування локалізації** — тестування, спрямоване на перевірку коректності та якості адаптації продукту до використання тією чи іншою мовою з урахуванням національних та культурних особливостей. Це тестування слідує за тестуванням інтернаціоналізації (див. попередній пункт) і перевіряє коректність перекладу та адаптації продукту, а не готовність продукту до таких дій.

• **Тестування сумісності** (compatibility testing, interoperability testing) — тестування, спрямоване на перевірку здатності програми працювати у вказаному оточенні.

Тестування даних та баз даних — два близькі за змістом види тестування, спрямовані на дослідження таких характеристик даних, як повнота, несуперечність, цілісність, структурованість тощо.

У контексті баз даних дослідженню може піддаватися адекватність моделі предметної області, здатність моделі забезпечувати цілісність і консистентність даних, коректність роботи тригерів, процедур, що зберігаються і т.д.

• **Тестування використання ресурсів**) — сукупність видів тестування, що перевіряють ефективність використання додатком доступних йому ресурсів та залежність результатів роботи програми від кількості доступних йому ресурсів. Часто ці види тестування безпосередньо чи опосередковано примикають до технік тестування продуктивності.

• **Порівняльне тестування** (comparison testing) — тестування, спрямоване на порівняльний аналіз переваг і недоліків продукту, що розробляється по відношенню до його основних конкурентів.

• **Демонстраційне тестування** (qualifi cation testing) — формальний процес демонстрації замовнику продукту з метою підтвердження, що продукт відповідає всім заявленим вимогам. На відміну від приймального тестування{89} цей процес більш суворий і всеосяжний, але може проводитися і на проміжних стадіях розробки продукту.

• **Вичерпне тестування** (exhaustive testing) – тестування програми з усіма можливими комбінаціями всіх можливих вхідних даних у всіх можливих умовах виконання. Для будь-якої складної системи нереалізовано, але може застосовуватися для перевірки окремих вкрай простих компонентів.

• **Тестування надійності** (reliability testing) — тестування здатності програми виконувати свої функції в заданих умовах протягом заданого часу або заданої кількості операцій.

• **Тестування відновлюваності** (recoverability testing) — тестування здатності програми відновлювати свої функції та заданий рівень продуктивності, а також відновлювати дані у разі виникнення критичної ситуації, що призводить до тимчасової (часткової) втрати працездатності програми.

• **Тестування відмовостійкості** (failover testing) — тестування, що полягає в емуляції або реальному створенні критичних ситуацій з метою перевірки здатності застосування задіяти відповідні механізми, що запобігають порушенню працездатності, продуктивності та пошкодження даних.

• **Тестування продуктивності** (performance testing) — дослідження показників швидкості реакції програми на зовнішні впливи за різного характеру та інтенсивності навантаження. У рамках тестування продуктивності виділяють такі підвиди:

– **Навантажувальне тестування** (load testing, capacity testing) — дослідження здатності додатка зберігати задані показники якості при навантаженні в допустимих межах та деякому перевищенні цих меж (визначення «запасу міцності»).
– **Тестування масштабованості** (scalability testing) — дослідження здатності програми збільшувати показники продуктивності відповідно до збільшення кількості доступних додатку ресурсів.
– **Об'ємне тестування** (volume testing) — дослідження продуктивності програми при обробці різних (зазвичай великих) обсягів даних.
– **Стресове тестування** (stress testing) — дослідження поведінки програми при позаштатних змінах навантаження, які значно перевищують розрахунковий рівень, або в ситуаціях недоступності значної частини необхідних додатку ресурсів. Стресове тестування може виконуватися і поза контекстом тестування навантаження: тоді воно, як правило, називається «тестуванням на руйнування» (destructive testing) і є крайньою формою негативного тестування.

– **Конкурентне тестування** (concurrency testing) — дослідження поведінки додатка в ситуації, коли йому доводиться обробляти велику кількість запитів, що одночасно надходять, що викликає конкуренцію між запитами за ресурси (базу даних, пам'ять, канал передачі даних, дискову підсистему та і т.д.). Іноді під конкурентним тестуванням розуміють дослідження роботи багатопоточних додатків і коректність синхронізації дій, що проводяться в різних потоках.

КЛАСИФІКАЦІЯ ПО ТЕХНІКАМ І ПІДХОДАМ

• Тестування на основі досвіду тестувальника, сценаріїв, чек-листів:

– **Дослідницьке тестування** (розглянуто раніше).
– **Вільне** (інтуїтивне) тестування (розглянуто раніше).

КЛАСИФІКАЦІЯ ЗА СТУПЕНЕМ ВТРУЧАННЯ В РОБОТУ ПРОГРАМИ:

- **Інвазивне тестування** (intrusive testing) — тестування, виконання якого може вплинути на функціонування програми через роботу інструментів тестування (наприклад, будуть спотворені показники продуктивності) або через втручання в сам код програми (наприклад, для аналізу роботи програми було додано додаткове протоколювання, включено виведення налагоджувальної інформації тощо). Деякі джерела розглядають інвазивне тестування як форму негативного або навіть стресового тестування.
- **Неінвазивне тестування** (nonintrusive testing) — тестування, виконання якого непомітно для застосування і не впливає на процес його звичайної роботи.

•КЛАСИФІКАЦІЯ З ТЕХНІК АВТОМАТИЗАЦІЇ:

- **Тестування під управлінням даними** — спосіб розробки автоматизованих тест-кейсів, в якому вхідні дані та очікувані результати виносяться за межі тест-кейсу і зберігаються поза ним — у файлі, базі даних тощо .
- **Тестування під керуванням ключовими словами** (keyword-driven testing) — спосіб розробки автоматизованих тест-кейсів, в якому за межі тест-кейсу виноситься не тільки набір вхідних даних та очікуваних результатів, але й логіка поведінки тест-кейсу, яка описується ключовими словами (командами).
- **Тестування під управлінням поведінкою** — спосіб розробки автоматизованих тест-кейсів, у якому основна увага приділяється коректності роботи бізнес-сценаріїв, а не окремим деталям функціонування програми.

КЛАСИФІКАЦІЯ НА ОСНОВІ (ЗНАННЯ) ДЖЕРЕЛ ПОМИЛОК:

- **Тестування передбаченням помилок** (error guessing) — техніка тестування, в якій тести розробляються на основі досвіду тестувальника та його знань про те, які дефекти типові для тих чи інших компонентів або областей функціональності додатку . Може комбінуватися з технікою т.зв. «помилково-орієнтоване» тестування (failure-directed testing), в якому нові тести будуються на основі інформації про раніше виявлені в додатку проблеми.

- **Евристична оцінк**а (heuristic evaluation) — техніка тестування зручності використання, спрямована на пошук проблем в інтерфейсі користувача, що є відхиленням від загальноприйнятих норм.

- **Мутаційне тестування** (mutation testing) — техніка тестування, в якій порівнюється поведінка кількох версій одного і того ж компонента, причому частина таких версій може бути спеціально розроблена з додаванням помилок (що дозволяє оцінити ефективність тест-кейсів — якісні тести виявлять ці фахівці - ально додані помилки). Може комбінуватися з наступним у списку видом тестування (тестуванням додаванням помилок).

- **Тестування додаванням помилок** (error seeding) — техніка тестування, в яку додаток спеціально додаються заздалегідь відомі, спеціально продумані помилки з метою моніторингу їх виявлення та усунення і, таким чином, формування більш точної оцінки показників процесу тестування. Може комбінуватись із попереднім у цьому списку видом тестування (мутаційним тестуванням).

КЛАСИФІКАЦІЯ НА ОСНОВІ ВИБОРУ ВХІДНИХ ДАНИХ:

- **Тестування на основі класів еквівалентності** — техніка тестування, спрямована на скорочення кількості тест-кейсів, що розробляються та виконуються при збереженні достатнього тестового покриття. Суть техніки полягає у виявленні наборів еквівалентних тест-кейсів (кожен з яких перевіряє одну й ту саму поведінку додатка) та виборі з таких наборів невеликого підмножини тест-кейсів, які з найбільшою ймовірністю виявляють проблему.
- **Тестування на основі граничних умов** — інструментальна техніка тестування на основі класів еквівалентності, що дозволяє виявити специфічні значення досліджуваних параметрів щодо меж класів еквівалентності. Ця техніка значно спрощує виявлення наборів еквівалентних тест-кейсів та вибір таких тест-кейсів, які виявлять проблему з найбільшою ймовірністю.
- **Доменне тестування** — техніка тестування на основі класів еквівалентності та граничних умов, що дозволяє ефективно створювати тест-кейси, що стосуються кількох параметрів (змінних) одночасно. Ця техніка також описує підходи до вибору мінімальної множини показових тест-кейсів з усього набору можливих тест-кейсів.
- **Попарне тестування** — техніка тестування, в якій тест-кейси будуються за принципом перевірки пар значень параметрів (змінних) замість того, щоб намагатися перевірити всі можливі комбінації всіх значень усіх параметрів. Ця техніка є окремим випадком N-комбінаційного тестування і дозволяє істотно скоротити трудовитрати на тестування (а іноді й зовсім уможливити тестування у разі, коли кількість «всіх комбінацій усіх значень усіх параметрів» вимірюється мільярдами).
- **Тестування на основі ортогональних масивів** — інструментальна техніка попарного та N-комбінаційного тестування, заснована на використанні «ортогональних масивів» (двовимірних масивів, що мають наступну властивість: якщо взяти дві будь-які колонки такого масиву, то «підмасив», що вийшов, міститиме всі можливі попарні комбінації значень, представлених у вихідному масиві).

КЛАСИФІКАЦІЯ НА ОСНОВІ СЕРЕДОВИЩА ВИКОНАННЯ:

• **Тестування в процесі розробки** (development testing) — тестування, яке виконується безпосередньо в процесі розробки програми та/або в середовищі виконання, відмінного від середовища реального використання програми. Як правило, виконується самими розробниками.

• **Тестування на основі коду** (code based testing). У різних джерелах цю техніку називають по-різному (найчастіше тестуванням на основі структур, причому деякі автори змішують в один набір тестування по потоку управління і потоку даних, а деякі суворо поділяють ці стратегії). Підвиди цієї техніки також організують у різні комбінації, але найбільш універсально їх можна класифікувати так:

– **Тестування по потоку управління** — сімейство технік тестування, в яких тест-кейси розробляються з метою активації та перевірки виконання різних послідовностей подій , які визначаються за допомогою аналізу вихідного коду програми. Додаткове докладне пояснення див. в цьому розділі (див. тестування на основі структур коду).

– **Тестування по потоку даних** — сімейство технік тестування, заснованих на виборі окремих шляхів з потоку управління з метою дослідження подій, пов'язаних із зміною стану змінних. Додаткове докладне пояснення див. далі в цьому розділі (у частині, де тестування потоку даних пояснено з точки зору стандарту ISO/IEC/IEEE 29119-4).

– **Тестування за діаграмою або таблицею станів** — техніка тестування, в якій тест-кейси розробляються для перевірки переходів програми з одного стану до іншого. Стан може бути описаний діаграмою станів або таблицею станів.

– **інспекція (аудит) коду** — сімейство технік підвищення якості коду за рахунок того, що в процесі створення або вдосконалення коду беруть участь кілька осіб. Ступінь формалізації аудиту коду може варіюватися від досить швидкого перегляду до ретельної формальної інспекції. На відміну від технік статичного аналізу коду (по потоку управління та потоку даних) аудит коду також покращує такі його характеристики, як зрозумілість, підтримуваність, відповідність угод про оформлення. Аудит коду виконується переважно самими програмістами.

• **Тестування на основі структур коду** (structure-based techniques) передбачає можливість дослідження логіки виконання коду в залежності від різних ситуацій і включає:

– **Тестування на основі виразів** (statement testing) — техніка тестування (за методом білої ящика), в якій перевіряється коректність (і сам факт) виконання окремих виразів у коді.
– **Тестування на основі гілок** (branch testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання окремих гілок коду (під гілкою розуміється атомарна частина коду, виконання якої відбувається або не відбувається залежно від істинності чи хибності певної умови).
– **Тестування на основі умов** (condition testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання окремих умов (умовою вважається вираз, який може бути обчислений до значення «істина» або «брехня»).
– **Тестування на основі комбінацій умов** (multiple condition testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання складних умов.
– **Тестування на основі окремих умо**в, що породжують розгалуження («вирішальних умов») (modifi ed condition decision coverage testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання таких окремих умов у складі складних умов, які поодинці визначають результат обчислення всього складного умови.
– **Тестування на основі рішень** (decision testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання складних розгалужень (з двома та більш можливими варіантами). Незважаючи на те, що «два варіанти» сюди також підходять, формально таку ситуацію варто віднести до тестування на основі умов.

• **Тестування на основі шляхів** (path testing) — техніка тестування (за методом білої скриньки), в якій перевіряється виконання всіх або деяких спеціально вибраних шляхів у коді програми.

• **Тестування на основі (моделей) поведінки програми** (application behavior/model-based testing):

– **Тестування за таблицею прийняття рішень** (decision table testing) — техніка тестування (за методом чорної скриньки), в якій тест-кейси розробляються на основі т.д. н. таблиці прийняття рішень, в якій відображені вхідні дані (та їх комбінації) та впливу на додаток, а також відповідні вихідні дані та реакції додатка.

– **Тестування за діаграмою або таблицею станів** (розглянуто раніше).

– **Тестування за специфікаціями** (specifi cation-based testing, black box testing) (розглянуто раніше).

– **Тестування за моделями поведінки програми** (model-based testing) — техніка тестування, в якій дослідження програми (і розробка тест-кейсів) будується на якійсь моделі: таблиці прийняття рішень {101}, таблиці або діаграмі станів {98} , сценаріїв користувача, моделі навантаження і т.д.

– **Тестування на основі варіантів використання** (use case testing) — техніка тестування (за методом чорної скриньки), в якій тест-кейси розробляються на основі варіантів використання. Варіанти використання виступають переважно джерелом інформації для кроків тест-кейсу, тоді як набори вхідних даних зручно розробляти з допомогою технік вибору вхідних даних. У загальному випадку джерелом інформації для розробки тест-кейсів у цій техніці можуть виступати не тільки варіанти використання, але й інші вимоги користувача в будь-якому їх вигляді. Якщо методологія розробки проекту передбачає використання історій користувача, цей вид тестування може бути замінений тестуванням на основі історій користувача (user story testing).

– **Паралельне тестування** (parallel testing) - техніка тестування, в якій поведінка нового (або модифікованого) додатка порівнюється з поведінкою еталонного додатка (імовірно працює правильно). Термін «паралельне тестування» також може використовуватися для позначення способу проведення тестування, коли кілька тестувальників або систем автоматизації виконують роботу одночасно, тобто. паралельно. Дуже рідко (і не зовсім правильно) під паралельним тестуванням розуміють мутаційне тестування.

– **Тестування на основі випадкових даних** (random testing) — техніка тестування (за методом чорної скриньки), в якій вхідні дані, дії або навіть самі тест-кейси вибираються на основі (псевдо)випадкових значень так, щоб відповідати операційному профілю (operational profile) — підмножині дій, що відповідають певній ситуації або сценарію роботи з додатком.

• **A/B-тестування** (A/B testing, split testing) — техніка тестування, в якій використовується вплив на результат виконання операції зміни одного з вхідних параметрів. Однак набагато частіше можна зустріти трактування A/B-тестування як техніку тестування зручності використання, в якому користувачам випадково пропонують різні варіанти елементів інтерфейсу, після чого оцінюється різниця в реакції користувачів.

КЛАСИФІКАЦІЯ ЗА МОМЕНТОМ ВИКОНАННЯ (ХРОНОЛОГІЇ)

• **Висхідне тестування** (bottom-up testing) — інкрементальний підхід до інтеграційного тестування, в якому в першу чергу тестуються низькорівневі компоненти, після чого процес переходить на все більш високорівневі компоненти.

• **Східне тестування** (top-down testing) — інкрементальний підхід до інтеграційного тестування, в якому в першу чергу тестуються високорівневі компоненти, після чого процес переходить на дедалі більш низькорівневі компоненти.

• **Гібридне тестування** (hybrid testing) — комбінація висхідного та низхідного тестування, що дозволяє спростити та прискорити отримання результатів оцінки програми.

ТЕСТОВІ АРТЕФАКТИ

Чек-лист (checklist) – це список, який містить ряд необхідних перевірок під час тестування програмного продукту. Чек-лист – це просто набір ідей: ідей із тестування, ідей із розробки, ідей із планування та управління – **будь-яких** ідей.

Чек-лист найчастіше є звичайним і звичний нам список:

- у якому послідовність пунктів немає значення (наприклад, список значень якогось поля);
- у якому послідовність пунктів важлива (наприклад, кроки у стислій інструкції);
- структурований (багаторівневий) список, що дозволяє відобразити ієрархію ідей.

Для того щоб чек-лист був дійсно корисним інструментом, він повинен мати ряд важливих властивостей:

- логічність;
- послідовність та структурованість;
- повнота та надмірність.

Типовими варіантами такої логіки є створення окремих чек-листів для:

- типових користувальницьких сценаріїв;
- різних рівнів функціонального тестування;
- окремих частин (модулів та підмодулів) програми;
- окремих вимог, груп вимог, рівнів та типів вимог;
- частин або функцій програми, найбільш схильних до ризиків.

Щоб проілюструвати принципи побудови чек-листів, ми скористаємося логікою розбиття функцій програми за ступенем їх важливості на три категорії:

• Базові функції, без яких існування програми втрачає сенс (тобто найважливіші — те, заради чого додаток взагалі створювався), або порушення роботи яких створює об'єктивні серйозні проблеми для середовища виконання.

• Функції, затребувані більшістю користувачів у повсякденній роботі.

• Інші функції (різноманітні «дрібниці», проблеми з якими не сильно вплинуть на цінність програми кінцевого користувача).

Можливі такі варіанти статусів:

- «Passed» – перевірка пройдена успішно, багів не знайдено;
- «Failed» – знайдений один або більше багів;
- «Blocked» – неможливо перевірити, тому що один з багів блокує поточну перевірку
- «In Progress» – поточний пункт, над яким працює тестувальник;
- «Not run» – ще не перевірено;
- «Skipped» – пункт перевірятися не буде з певної причини. Наприклад, поточний функціонал ще не реалізований.

Сайт "example.edu"
Реєстрація і Особистий профіль
Реєстрація на сайті
Редагування профілю
Форма зворотного зв'язку
Валідація полів
Відправка листа/повідомлення
Доставка листа/повідомлення
Пошук
Пошук по назві
Перехід за посиланням
Робота пошуку по різноманітним параметрам
Коментарі
Додавання коментаря
Відображення коментаря

	Іванов О.О.	Петров Б.Б.	Федоров В.В.	Васєчкін Г.Г.
Сайт "example.edu"	Google Chrome 91.0.4472.80	Mozilla Firefox 89.0.1	Opera 77.0.4054.172	Safari 14.1
Реєстрація і Особистий профіль				
Реєстрація на сайті	Passed			Passed
Редагування профілю	Failed			Failed
Форма зворотного зв'язку				
Валідація полів	Failed	Failed	Passed	Passed
Відправка листа/повідомлення	Blocked	Blocked	Passed	Passed
Доставка листа/повідомлення	Blocked	Blocked	Skipped	Skipped

https://bt-w.qatestlab.com/view.php?id=

- Після завершення проходження чекліста, у ньому не повинно залишитися комірок зі статусом «Not run».
- Всі комірки зі статусом «Failed» та «Blocked» обов'язково повинні мати примітки з посиланнями на баг-репорти.
- Статус «Passed» встановлюється тільки для пунктів, які перевірені та не містять помилок.

Переваги використання чеклістів

- Використання чеклістів сприяє структуруванню інформації у співробітника.
- При правильному записі необхідних дій у співробітника з'являється однозначне розуміння завдань. Це сприяє підвищенню швидкості навчання нових співробітників.
- Чеклісти допомагають уникнути невизначеності та помилок, що пов'язані з людським фактором. Збільшується покриття тестами програмного продукту.
- Підвищується ступінь взаємозамінності співробітників.
- Економія робочого часу.

Чекліст потрібен щоб:

Не забути необхідні тести.

Ділити завдання за рівнем кваліфікації.

Зберігати звітність та результати тестування.

Чекліст містить:

Список перевірок (з необхідним ступенем деталізації).

Оточення перевірки:

- збірка, на якій проводилося тестування;
- тестове оточення (якщо є);
- інформація про тестувальників.

Результат перевірки.

ТЕСТ КЕЙС

Тест – Набір з одного або декількох тест-кейсів

Тест-кейс – набір вхідних даних, умов виконання та очікуваних результатів, розроблений з метою перевірки тієї чи іншої властивості або поведінки програмного засобу.

Під тест-кейсом також може розумітися відповідний документ, який представляє формальний запис тест-кейсу.

Критично важливо зрозуміти і запам'ятати: якщо у тест-кейсу не вказані вхідні дані, умови виконання та очікувані результати, та/або не зрозуміла мета тест-кейсу – це поганий тест-кейс (Іноді він не має сенсу, іноді його зовсім неможливо виконати).

Поділяються на:

- високоврівневий тест-кейс;
- низькорівневий тест-кейс;.
- специфікація тест-кейсу;
- специфікація тесту.

МЕТА НАПИСАННЯ ТЕСТ-КЕЙСІВ

Тестування можна проводити і без тест-кейсів, але наявність тест-кейсів дозволяє:

• Структурувати та систематизувати підхід до тестування (без чого великий проект майже гарантовано приречений на провал).

• Обчислювати метрики тестового покриття та вживати заходів щодо його збільшення (тест-кейси тут є головним джерелом інформації, без якого існування подібних метрик втрачає сенс).

• Відслідковувати відповідність поточної ситуації плану (скільки приблизно знадобиться тест-кейсів, скільки вже є, скільки виконано із запланованої на даному етапі кількості тощо).

• Уточнити взаєморозуміння між замовником, розробниками та тестувальниками (тест-кейси часто набагато наочно показують поведінку програми, ніж це відображено у вимогах).

• Зберігати інформацію для тривалого використання та обміну досвідом між співробітниками та командами (або, як мінімум, не намагайтеся втримати в голові сотні сторінок тексту).

• Проводити регресійне тестування та повторне тестування (які без тест-кейсів було б взагалі неможливо виконати).

• Підвищувати якість вимог (ми це вже розглядали: написання чек-листів та тест-кейсів – хороша техніка тестування вимог).

• Швидко вводити в курс справи нового співробітника, який нещодавно підключився до проекту.

ЖИТТЯНИЙ ЦИКЛ ТЕСТ-КЕЙСУ

- Створено (new);
- Запланований (planned, ready for testing));
- **Не виконаний (**not tested**);**
- Виконується (work in progress);
- **Пропущений (**skipped**);**
- **Провален (**failed**);**
- **Пройдено успішно (**passed**);**
- **Заблокований (**blocked**);**
- Закритий (closed);
- Вимагає доопрацювання (not ready).

АТРИБУТИ ПОЛІВ ТЕСТ-КЕЙСУ

Як було зазначено вище, термін «тест-кейс» може ставитися до формальної запису тест-кейса як технічного документа. Цей запис має загальноприйняту структуру, компоненти якої називаються атрибутами (полями) тест-кейсу.

Залежно від інструменту керування тест-кейсами зовнішній вигляд їх запису може трохи відрізнятися, можуть бути додані або прибрані окремі поля, але концепція залишається незмінною.

Ідентифікатор (identifier);

Пріоритет (priority)

Пріоритет тест-кейсу може корелювати з:

- важливістю вимоги, сценарію користувача або функції, з якими пов'язаний тест-кейс;
- потенційною важливістю дефекту, на пошук якого спрямований тест-кейс;
- ступенем ризику, пов'язаного з тест-кейсом, що перевіряється вимогою, сценарієм або функцією.

Пов'язана з тест-кейсом вимога (requirement);

Модуль і підмодуль програми (module and submodule);

Погано	Добре
Тест 1	Запуск, одиничне копіювання, правильні налаштування
Тест 2	Запуск однієї копії з недійсними контурами
Тест 78 (покращений)	Виконуйте, багато копій, без конфліктів
Зупинка	Зупинка по Ctrl+C
Закриття	Зупиниться, закривши консоль

Назва (суть) тест-кейсу (title);

- Інформативність.
- Хоча б відносна унікальність (щоб не плутати різні тест-кейси).

Вихідні дані, необхідні для виконання тест-кейсу (precondition, preparation, initial data, setup);

Кроки тест-кейсу (steps);

Очікувані результати (expected results).

Тест кейс

Ім'я:	Тест на подання коментарів	
Функція:	Контактні запитання	
Дія	**Очікуваний результат**	**Результат тесту:** • Пройшов • Не вдалося • заблоковано
Передумовою:		
Перейдіть на сайт Pro Testing: http://www.протестуючі.ru	Веб-сайт Pro Testing відкритий і доступний	
Натисніть на посилання "Задати питання" внизу сторінки	Сторінка «Питання, Запити та Запити» відкрита та доступна	
Кроки тїсть:		
Заповніть форму коментарів: "Тип справи": коментар «Контактна особа»: Ольга "Електронна пошта": test@test.com "Повідомлення": *Доброго дня, шановна команда "ProTesting"! Підкажіть, будь ласка, де я можу з ними познайомитися?*	Дані успішно введені	
Натисніть кнопку «Надіслати»	Сторінка «Ваш запит успішно надіслано!» відкрито	
Постумова:		
Натисніть на посилання "Повернутися до форми заявки"	Відкрита сторінка «Питання, прохання та запити»	

ВЛАСТИВОСТІ ЯКІСНИХ ТЕСТ-КЕЙСІВ

Правильна технічна мова, точність та однаковість формулювань. Ця властивість однаково стосується і вимог, і тест-кейсів, і звітів про дефекти – будь-якої документації.

Баланс між специфічністю та спільністю. Тест-кейс вважається тим паче специфічним, що детальніше у ньому розписані конкретні дії, конкретні значення тощо., тобто. Ніж у ньому більше конкретики. Відповідно, тест-кейс вважається тим більш загальним, чим у ньому менше конкретики.

Баланс між простотою та складністю. Тут немає академічних визначень, але прийнято вважати, що простий тест-кейс оперує одним об'єктом (чи у ньому явно видно головний об'єкт), і навіть містить небагато тривіальних дій; складний тест-кейс оперує кількома рівноправними об'єктами та містить багато нетривіальних дій.

"Показовість" (висока ймовірність виявлення помилки). Починаючи з рівня тестування критичного шляху можна стверджувати, що тест-кейс є тим добрішим, чим він більш показовий (з більшою ймовірністю виявляє помилку). Саме тому ми вважаємо непридатними надто прості тест-кейси – вони не показові.

Послідовність у досягненні мети. Суть цієї властивості виявляється у тому, що це дії в тест-кейсі спрямовані слідування єдиної логіці і досягнення єдиної мети і містять ніяких відхилень.

Відсутність зайвих дій. Найчастіше ця властивість має на увазі, що не потрібно в кроках тест-кейсу довго і по пунктах розписувати те, що можна замінити однією фразо

Ненадмірність по відношенню до інших тест-кейсів. У процесі створення великої кількості тест-кейсів дуже легко опинитися в ситуації, коли два і більше тест-кейс фактично виконують одні і ті ж перевірки, переслідують одні і ті ж цілі, спрямовані на пошук одних і тих же проблем.

Демонстративність (здатність демонструвати виявлену помилку очевидним чином). Очікувані результати мають бути підібрані і сформульовані таким чином, щоб будь-яке відхилення від них відразу ж впадало в око і ставало очевидним, що сталася помилка. Порівняйте витримки із двох тест-кейсів.

Простежуваність. З інформації, що міститься в якісному тест-кейсі, має бути зрозуміло, яку частину програми, які функції і які вимоги він перевіряє. Частково це властивість досягається через заповнення відповідних полів тест-кейсу («Посилання на вимогу», «Модуль», «Підмодуль»), але й сама логіка тест-кейсу грає не останню роль, т.я. у разі серйозних порушень цієї властивості можна довго з подивом дивитися, наприклад, на яку вимогу посилається тест-кейс, і намагатися зрозуміти, як вони пов'язані один з одним.

Можливість повторного використання. Ця властивість рідко виконується для низькорівневих тест-кейсів, але при створенні високорівневих тест-кейсів можна досягти таких формулювань, при яких:

• тест-кейс буде придатним до використання з різними налаштуваннями програми, що тестується, і в різних тестових оточеннях;

• тест-кейс практично без змін можна буде використовувати для тестування аналогічної функціональності в інших проектах або в інших областях програми.

Повторюваність. Тест-кейс має бути сформульований таким чином, щоб при багаторазовому повторенні він показував однакові результати.

Відповідність прийнятим шаблонам оформлення та традиціям. З шаблонами оформлення, як правило, проблем не виникає: вони суворо визначені зразком або взагалі екранною формою інструментального засобу управління тест-кейсами. Що ж до традицій, то вони відрізняються навіть у різних командах у рамках однієї компанії, і тут неможливо дати іншої поради, окрім як «шануйте вже готові тест-кейси перед тим як писати свої».

ЛОГІКА СТВОРЕННЯ ЕФЕКТИВНИХ ПЕРЕВІРОК

Існує досить простий алгоритм, що дозволяє нам створювати ефективні перевірки навіть за таких умов. Приступаючи до продумування чек-листа, тест-кейс або набору тест-кейсів, задайте собі такі запитання та отримайте чіткі відповіді:

- Що перед вами?
- Кому і навіщо воно потрібне (і наскільки це важливо)?
- Як воно зазвичай використовується?
- Як може зламатися, тобто. почати працювати неправильно?

До цього алгоритму можна додати ще невеликий перелік універсальних рекомендацій, які дозволять вам проводити тестування краще:

- Починайте якомога раніше – вже з моменту появи перших вимог можна займатися їх тестуванням та покращенням, можна писати чек-листи та тест-кейси, можна уточнювати план тестування, готувати тестове оточення тощо.
- Якщо ви маєте тестувати щось велике і складне, розбивайте його на модулі та підмодулі, функціональність піддавайте функціональній декомпозиції. Тобто, досягайте такого рівня деталізації, при якому ви можете легко пам'ятати всю інформацію про об'єкт тестування.
- Обов'язково пишіть чек-листи. Якщо вам здається, що ви зможете запам'ятати всі ідеї і потім легко відтворити їх, ви помиляєтеся. Винятків не буває.
- У міру створення чек-листів, тест-кейсів тощо. прямо в текст вписуйте питання, що виникають. Коли питань накопичиться достатньо, зберіть їх окремо, уточніть формулювання та зверніться до того, хто може дати відповіді.

• Якщо інструментальний засіб, який ви використовуєте, дозволяє використовувати косметичне оформлення тексту – використовуйте (так текст краще читатиметься), але намагайтеся дотримуватися загальноприйнятих традицій і не розфарбовувати кожне друге слово у свій колір, шрифт, розмір тощо.

• Використовуйте техніку швидкого перегляду для отримання відгуку від колег та покращення створеного вами документа.

• Плануйте час на покращення тест-кейсів (виправлення помилок, доопрацювання за фактом зміни вимог тощо).

• Починайте опрацювання (і виконання) тест-кейсів із простих позитивних перевірок найважливішої функціональності. Потім поступово підвищуйте складність перевірок, пам'ятаючи не тільки про позитивні , але й про негативні перевірки.

• Пам'ятайте, що основою тестування є мета. Якщо ви не можете швидко і просто сформулювати ціль створеного вами тест-кейсу, ви створили поганий тест-кейс.

• Уникайте надлишкових тест-кейсів, що дублюють один одного. Мінімізувати їх кількість вам допоможуть техніки класів еквівалентності, граничних умов, доменного тестування.

• Якщо показник тестування можна збільшити, при цьому не сильно змінивши його складність і не відхилившись від вихідної мети, зробіть це.

• Пам'ятайте, що багато тест-кейсів потребують окремої підготовки, яку потрібно описати у відповідному полі тест-кейсу.

• Кілька позитивних тест-кейсів можна безбоязно поєднувати, але об'єднання негативних тест-кейсів майже завжди заборонено.

• Подумайте, як можна оптимізувати створений вами тест-кейс (набір тест-кейсів тощо) так, щоб знизити витрати на його виконання.

• Перед тим як надсилати фінальну версію створеного вами документа, ще раз перечитайте написане (у добрій половині випадків знайдете друкарську помилку або іншу недоробку).

ТИПОВІ ПОМИЛКИ

- відсутність назви тест-кейсу або погано написана назва;
- відсутність нумерації кроків та/або очікуваних результатів;
- посилання на множину інших вимог;
- використання особистої форми дієслів;
- використання минулого чи майбутнього часу в очікуваних результатах;
- постійне використання слів "перевірити" (і йому подібних) у чек-листах;
- опис стандартних елементів інтерфейсу замість використання їх усталених назв;
- пунктуаційні, орфографічні, синтаксичні та подібні до них помилки.

ЛОГІЧНІ ПОМИЛКИ

- посилання на інші тест-кейс або кроки інших тест-кейсів.
- деталізація, що не відповідає рівню функціонального тестування
- розпливчасті двозначні описи дій та очікуваних результатів.
- опис дій як найменування модуля/підмодуля
- опис подій або процесів як кроки або очікувані результати
- вигадування» особливостей поведінки програми
- відсутність опису приготування для виконання тест-кейсу.
- некоректне найменування елементів інтерфейсу чи його властивостей.
- нерозуміння принципів роботи програми та викликана цим некоректність тест-кейсів
- перевірка типової "системної" функціональності.
- неправильна поведінка програми як очікуваний результат
- тест-кейси, що не належать до тестованого додатка
- формальні та/або суб'єктивні перевірки.

Use Case (сценарій користування) – це перелік дій, сценарій, за яким користувач взаємодіє з додатком або програмою для виконання будь-якої дії та досягнення конкретної мети.

Тестування за юзкейсами проводиться для того, щоб виявити додаткові логічні прогалини та баги у web-додатку, які складно знайти під час тестування окремих індивідуальних модулів або частин цього web-додатку.

У більшості випадків Use Case описує, що робить система, а не як. Власне, цього правила і варто дотримуватися, створюючи такі сценарії.

За допомогою юзкейсів можна описувати взаємодію двох або більшої кількості учасників, що мають конкретну мету, наприклад:

- покупка товару у магазині (Покупець – Продавець);
- відправка листа електронною поштою (Адресант – Поштовий клієнт);
- запит сторінки браузером (Браузер – Web-сервер).

УК02. Знайдіть свій пристрій

Передумови:

- УС01.

Крок No.	Дія користувача	Відповідь додатка
1	В поле «Пошук» введіть дані, за якими потрібно знайти пристрій. (Код/Ім'я/Місцезнаходження/Коментар)	Пошук у всьому списку пристроїв.
2		Відображає список (що складається з коду, назви) пристроїв, що задовольняє запит користувача.
3	Вибирає потрібний пристрій.	Відображає дані (ім'я, код, місцезнаходження, коментарі) про вибраний пристрій.

Пост-умови:

- Виводиться список пристроїв, який відповідає пошуковому запиту користувача.

УЦ02а. Пристрій не знайдено

Крок No.	Дія користувача	Відповідь додатка
1а	В поле «Пошук» введіть дані, за якими потрібно знайти пристрій. (Код/Ім'я/Місцезнаходження/Коментар)	Пошук у всьому списку пристроїв.
2А		Пристроїв за запитом користувача не знайдено.
3А		Відображає порожній список і повідомлення «Не знайдено».

Пост-умови:

- Виконайте UC02 з кроку 1.

ЗВІТИ О ДЕФЕКТАХ

Дефект - це невідповідність між очікуваним результатом і фактичним.

Очікуваним результатом є поведінка системи так, як описано у вимогах.

Фактичним результатом є поведінка системи, яка спостерігалася в процесі тестування.

Помилка – це людська дія, яка призводить до неправильних результатів.

Цей термін дуже часто використовується як найбільш універсальний, що описує будь-які проблеми («людська помилка», «помилка в коді», «помилка в документації», «помилка при виконанні операції», «помилка при передачі даних», «помилковий результат» і т.д.)

Дефект, (defect, bug, problem, fault) — недолік компонента або системи, який може призвести до відмови або відмови.

Цей термін також розуміється досить широко, коли йдеться про дефекти документації, налаштувань, вхідних даних тощо.

Збій (interruptio) або відмова (failur) – вілхилення поведінки системи від очікуваного

Збій – самовиправляєма відмова, або одноразова відмова, який може бути усунутий незначним втручанням оператора.

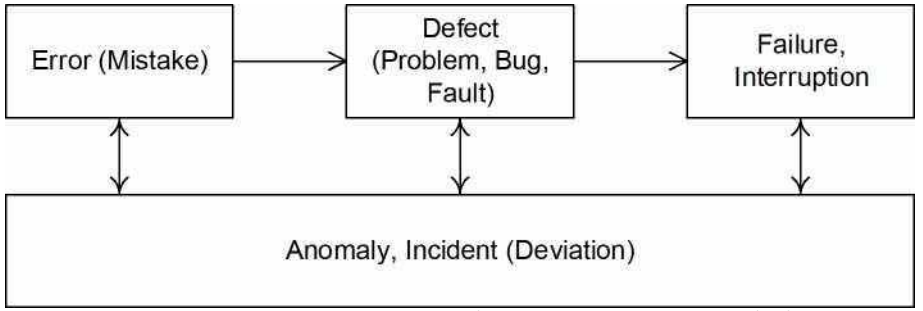
Відмова - це подія, яка пов'язана з порушенням працездатного стану об'єкта.

Ці терміни більше пов'язані з теорією надійності і не часто зустрічаються в повсякденній роботі тестувальника, але саме збої та відмови є тим що помічає тестувальник в процесі тестування (і на підставі цього проводить дослідження з метою виявлення дефекту і його причин).

Аномалія (anomaly) або інцидент (incident, deviation) - це будь-яке відхилення спостережуваного (фактичного) стану, поведінки, значення, результату, властивості від очікувань спостерігача, сформоване на основі вимог, специфікацій тощо. документація або досвіду і здорового глузду.

Дефект - відхилення (deviatio) фактичного результату від очікувань спостерігача сформованих з урахуванням вимог, специфікацій, іншої документації чи досвіду і здорового глузду.

Звідси логічно випливає, що дефекти можуть зустрічатися не тільки в коді додатка, а й у будь-якій документації, в архітектурі та дизайні, в налаштуваннях тестованого додатка або тестового оточення де завгодно.



Взаємозв'язок проблем при розробці програмного забезпечення

ЗВІТ О ДЕФЕКТАХ ТА ЙОГО ЖИТТЕВИЙ ЦИКЛ

Звіт о дефектах (defect report) - це документ, який описує і пріоритезує виявлений дефект, а також полегшує його усунення

Як випливає з визначення, звіт про дефекти пишеться з наступними основними цілями:

- **Надати інформацію про проблему** — повідомити команду проєкту та інші зацікавлені сторони про проблему, описати суть проблеми;
- **Пріоритезувати проблему** – визначити ступінь небезпеки проблеми для проєкту та бажані часові рамки для її усунення;
- **Сприяти усуненню проблеми** – якісний звіт про дефект не тільки містить усі необхідні деталі, щоб зрозуміти, що сталося, але також може містити аналіз причини проблеми та рекомендації щодо виправлення ситуації.

Звіт про дефект (і сам дефект разом з ним) проходить певні стадії життєвого циклу:

• **Виявлено** (submitted) — початковий стан звіту (іноді називається «Новий» (new)), якому він знаходиться одразу після створення. Деякі засоби також дозволяють спочатку створювати чернетку (draft) і лише потім публікувати звіт.

• **Призначений** (assigned) — цей стан звіт переходить з моменту, коли хтось із проектної команди призначається відповідальним за виправлення дефекту. Призначення відповідального проводиться або рішенням лідера команди розробки, або колегіально, або за добровільним принципом, або іншим способом, прийнятим у команді, або виконується автоматично на основі певних правил.

• **Виправлений** (fixed) — у цей стан звіт перекладає відповідальний за виправлення дефекту член команди після виконання відповідних дій з виправлення.

• **Перевірено** (verified) — у цей стан звіт перекладає тестувальник, який переконався, що дефект насправді було усунено. Як правило, таку перевірку виконує тестувальник, який спочатку написав звіт про дефект.

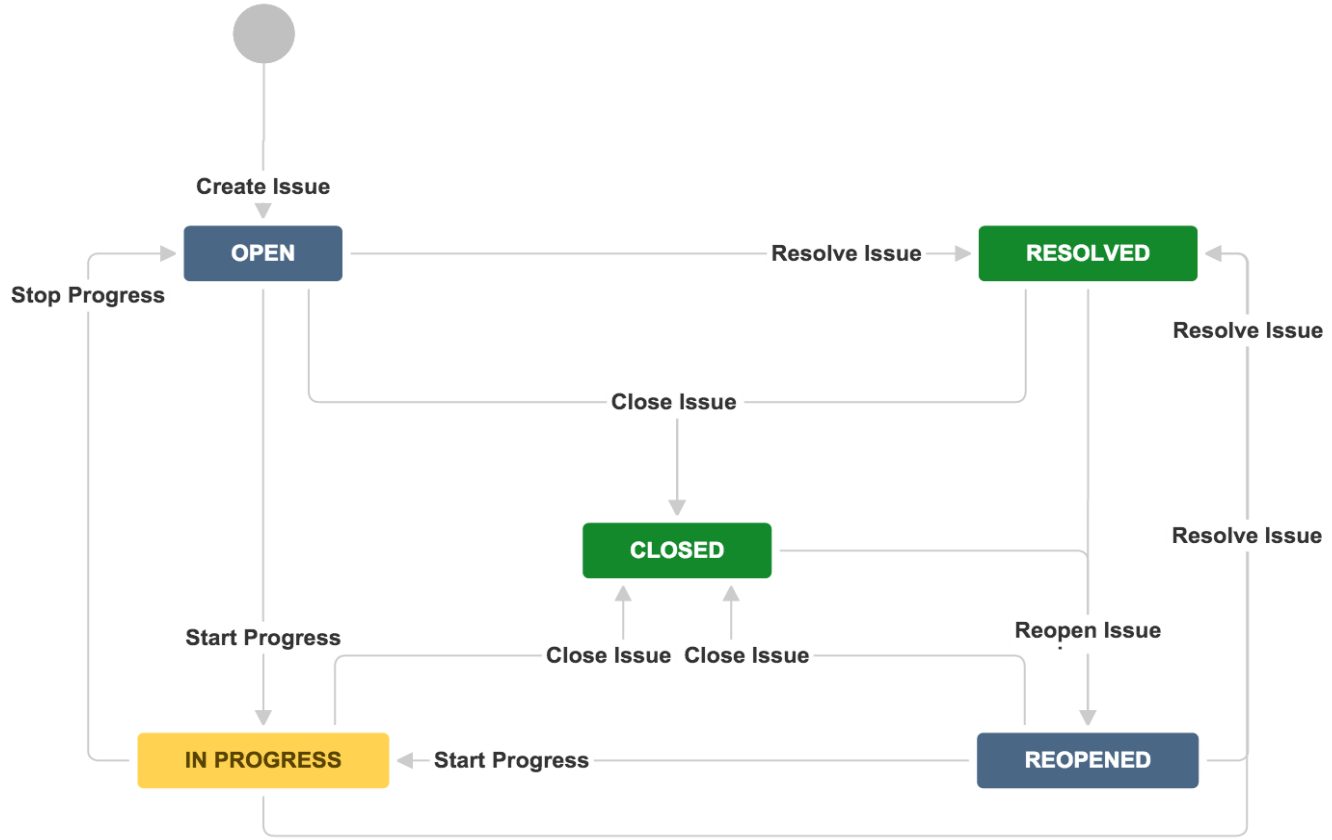
• **Закритий** (closed) — стан звіту, що означає, що з даного дефекту не планується подальших дій.

• **Відкритий заново** (reopened) — у цей стан (як правило, зі стану «Виправлено») звіт перекладає тестувальник, який переконався, що дефект, як і раніше, відтворюється на білді, в якому він уже має бути виправлений.

• **Рекомендований до відхилення** (to be declined) — у цей стан звіт про дефект може бути переведений з багатьох інших станів з метою винести на розгляд питання про відхилення звіту з тієї чи іншої причини. Якщо рекомендація є обґрунтованою, звіт переводиться у стан «Відхилений» (див. наступний пункт).

• **Відхилений** (declined) — цей стан звіт перекладається у випадках, докладно описаних у пункті «Закритий», якщо засіб керування звітами про дефекти передбачає використання цього стану замість стану «Закритий» для тих чи інших резолюцій щодо звіту.

• **Відкладено** (deferred) — у цей стан звіт перекладається у випадку, якщо виправлення дефекту найближчим часом є нераціональним або неможливим, однак є підстави вважати, що в найближчому майбутньому ситуація виправиться (вийде нова версія бібліотеки, повернеться з відпустки фахівець за якоюсь технологією, зміняться вимоги замовника і т.д.).



Ідентифікатор	Короткий опис	Повний опис	Кроки по відновленню	Відтворюваність	Важливість	Терміновість	Симптом	Можливість обійти	Коментар	Програми
19	Нескінченний цикл обробки вхідного файлу з атрибутом **«тільки для читання»**	Якщо у вхідного файлу виставлено атрибут «тільки для читання», після обробки додатку не вдається перемістити його до каталогу- приймача: створюється копія файлу, але оригінал не видаляється, і додаток знову і знову обробляє цей файл безуспішно намагається перемістити його в каталог- приймач. **Очікуваний результат:** після обробки файл переміщено з каталогу-джерела в каталог-приймач **Фактичний результат:** оброблений файл копіюється в каталог-приймач. але його оригінал залишається в каталозі-джерелі. Вимога: ДС-2.1.	1. Помістити в каталог-джерело файл допустимого типу та розміру. 2. Встановити файлу атрибут «тільки для читання». 3. Запустити програму. Дефект: оброблений файл з’являється в каталозі-приймачі. але не видаляється з каталогу-джерела, файл у каталозі-приймачі безперервно оновлюється (видно за значенням часу останньої зміни).	Завжди	Середня	Звичайна	Некоректна операція	ні	Якщо замовник не планує використовувати установку атрибуту «тільки для читання» файлів у каталозі-джерелі для досягнення якихось цілей, можна просто знімати цей атрибут і спокійно переміщати файл.	-

Шапка	
Короткий опис (Summary)	Короткий опис проблеми, вказівка на причину та тип помилкової ситуації.
Проект (Project)	Назва тестового проекту
Компонент програми (Component)	Назва частини або функції тестованого продукту
Номер версії (Version)	Версія на якій було знайдено помилку
Серйозність (Severity)	Найбільш поширена п'ятирівнева система градації серйозності дефекту: 1. Блокуючий (Blocked) 2. Критичний (Critical) 3. Значний (Major) 4. Незначний (Minor) Тривіальний (Trivial)
Пріоритет (Priority)	Пріоритет дефекту: 1. Високий (High) 2. Середній (Medium) Низький (Low)
Статус (Statud)	Статус бага. Залежність від процедури та життєвого циклу бага (bug workflow and life cycle)
Автор (Author)	Творець баг репорту
Назначен на (Assigned To)	Ім'я співробітника призначеного для вирішення проблем
Оточення	

ОС / Сервіс пак і т.д. / Браузера + версія	Інформація про оточення, на якому було знайдено баг: операційна система, сервіс пак, для тестування ім'я і версія браузера і т.д.
…	
Опис	
Кроки відтворення (Steps to Reproduce)	Кроки, якими можна легко відтворити ситуацію, що призвела до помилки.
Фактичний результат (Result)	Результат, отриманий після проходження кроків до відтворення
Очікуваний результат (Expected Result)	Очікуваний правильний результат
Доповнення	
Прикріплений файл (Attachment)	Файл із логами, Скріншот або будь-який інший документ, який може допомогти прояснити причину помилки або вказати на спосіб вирішення проблеми

Ідентифікатор (identifier) є унікальним значенням, що дозволяє однозначно відрізнити один звіт про дефект від іншого і використовуваний у різних посиланнях.

Короткий опис (summary) має у гранично лаконічній формі давати вичерпну відповідь на запитання «Що сталося?» "Де це сталося"? «За яких умов це сталося?». Наприклад: «Відсутній логотип на сторінці привітання, якщо користувач є адміністратором»:

• Що сталося? Немає логотипу.

• Де це сталося? На сторінці вітання.

• За яких умов це сталося? Якщо користувач є адміністратором

Детальний опис (description) представляє у розгорнутому вигляді необхідну інформацію про дефект, а також (обов'язково!) опис фактичного результату, очікуваного результату та посилання на вимогу (якщо це можливо).

Якщо адміністратор входить до системи, на сторінці привітання відсутній логотип.
Фактичний результат: логотип відсутній у верхньому лівому куті сторінки.
Очікуваний результат: логотип відображається у верхньому лівому куті сторінки.
Вимога: R245.3.23b.

Кроки відтворення (steps to reproduce, STR) описують дії, які необхідно виконати для відтворення дефекту. Це поле схоже на кроки тест-кейсу, за винятком однієї важливої відмінності: тут дії прописуються максимально докладно, із зазначенням конкретних значень, що вводяться, і найдрібніших деталей, т.к. відсутність цієї інформації у складних випадках може призвести до неможливості відтворення дефекту

1. Відкрити http://testapplication/admin/login/.
2. Авторизуватися з ім'ям "defaultadmin" та паролем "dapassword".
Дефект: у лівому верхньому куті сторінки немає логотипу (замість нього відображається порожній простір з написом «logo»).

Відтворюваність (reproducibility) показує, чи при кожному проходженні кроків відтворення дефекту вдається викликати його прояв. Це поле набирає всього два значення: завжди (**always**) або іноді (**sometimes**).

Важливість (severity) показує ступінь шкоди, яка завдається проекту існуванням дефекту.

У випадку виділяють такі градації важливості:

• **Критична** (critical) — існування дефекту призводить до масштабних наслідків катастрофічного характеру, наприклад: втрата даних, розкриття конфіденційної інформації, порушення ключової функціональності програми тощо.

• **Висока** (major) — існування дефекту завдає відчутних незручностей багатьом користувачам у межах їх типової діяльності, наприклад: недоступність вставки з буфера обміну, непрацездатність загальноприйнятих клавіатурних комбінацій, необхідність перезапуску програми під час виконання типових сценаріїв роботи.

• **Середня** (medium) — існування дефекту слабко впливає на типові сценарії роботи користувачів, та/або існує обхідний шлях досягнення мети, наприклад: діалогове вікно не закривається автоматично після натискання кнопок «OK»/«Cancel», при роздруковуванні кількох документів поспіль не зберігається значення поля «Двосторонній друк», переплутані напрями сортувань за полем таблиці.

• **Низька** (minor) — існування дефекту рідко виявляється незначним відсотком користувачів і (майже) не впливає на їхню роботу, наприклад: друкарська помилка в глибоко вкладеному пункті меню налаштувань, якесь вікно відразу при відображенні розташоване незручно (потрібно перетягнути його) у зручне місце), неточно відображається час до завершення операції копіювання файлів.

Терміновість (priority) показує, як швидко дефект має бути усунений.

У випадку виділяють такі градації терміновості:

• **Найвища** (ASAP, as soon as possible) терміновість вказує на необхідність усунути дефект настільки швидко, наскільки це можливо. Залежно від контексту «настільки швидко, наскільки можливо» може змінюватись від «у найближчому білді» до одиниць хвилин.

• **Висока** (high) терміновість означає, що дефект слід виправити позачергово, т.к. його існування або вже об'єктивно заважає роботі, або почне створювати такі перешкоди у найближчому майбутньому.

• **Звичайна** (normal) терміновість означає, що дефект слід виправити у порядку загальної черговості. Таке значення терміновості набуває більшість дефектів.

• **Низька** (low) терміновість означає, що в найближчому майбутньому виправлення даного дефекту не вплине на підвищення якості продукту.

Симптом (symptom) - дозволяє класифікувати дефекти щодо їх типового прояву. Не існує загальноприйнятого списку симптомів. Більше того, далеко не в кожному інструментальному засобі управління звітами про дефекти є таке поле, а там, де воно є, його можна налаштувати. Як приклад розглянемо такі значення симптомів дефекту.

• **Косметичний дефект** (cosmetic fl aw) — візуально помітний недолік інтерфейсу, що не впливає на функціональність програми (наприклад, напис на кнопці виконаний шрифтом не тієї гарнітури).

• **Пошкодження/втрата даних** (data corruption/loss) — внаслідок дефекту спотворюються, знищуються (або не зберігаються) деякі дані (наприклад, при копіюванні файлів копії виявляються пошкодженими).

• **Проблема в документації** (documentation issue) — дефект не стосується програми, а документації (наприклад, немає розділу посібника з експлуатації).

• **Неправильна операція** (incorrect operation) — деяка операція виконується неправильно (наприклад, калькулятор показує відповідь 17 при множенні 2 на 3).

• **Проблема інсталяції** (installation problem) — дефект проявляється на стадії встановлення та/або конфігурування програми.

• **Помилка локалізації** (localization issue) — щось у додатку не перекладено або переведено неправильно вибраною мовою інтерфейсу.

• **Нереалізована функціональність** (missing feature) — якась функція програми не виконується або не може бути викликана (наприклад, у списку форматів для експорту документа немає кількох пунктів, які там мають бути).

• **Проблема масштабованості** (scalability) — при збільшенні кількості доступних додатку ресурсів не відбувається очікуваного приросту продуктивності програми.

• **Низька продуктивність** (low performance) - виконання деяких операцій займає неприпустимо великий час

• **Крах системи** (system crash) — програма припиняє роботу або втрачає здатність виконувати свої ключові функції (також може супроводжуватися крахом операційної системи, веб-сервера тощо).

• **Несподівана поведінка** (unexpected behavior) — у процесі виконання певної типової операції програма поводиться незвичайним (на відміну від загальноприйнятого) чином (наприклад, після додавання до списку нового запису активним стає не новий запис, а перший у списку).

• **Недружня поведінка** (unfriendly behavior) — поведінка програми створює користувачеві незручності в роботі (наприклад, на різних діалогових вікнах у різному порядку розташовані кнопки «OK» та «Cancel»).

• **Розбіжність з вимогами** (variance from specs) — цей симптом вказують, якщо дефект складно співвіднести з іншими симптомами, проте додаток поводиться не так, як описано в вимогах.

• **Пропозиція щодо поліпшення** (enhancement) — у багатьох інструментальних засобах управління звітами про дефекти цього випадку є окремий вид звіту, так як пропозицію щодо поліпшення формально не можна вважати дефектом: додаток поводиться згідно з вимогами, але у тестувальника є обґрунтована думка про те, як ту чи іншу функціональність можна покращити.

Можливість обійти (workaround) — показує, чи існує альтернативна послідовність дій, виконання якої дозволило б користувачеві досягти поставленої мети (наприклад, клавіатурна комбінація Ctrl+P не працює, але роздрукувати документ можна, вибравши відповідні пункти в меню).

Коментар (comments, additional info) – може містити будь-які корисні для розуміння та виправлення дефекту дані. Іншими словами, сюди можна писати все те, що не можна писати до інших полів.

Вкладення (attachments) — це не так поле, як список прикріплених до звіту про дефект додатків (копій екрана, що викликають збій файлів тощо).

ІНСТРУМЕНТАЛЬНІ ЗАСОБИ

Загальний набір функцій, як правило, що реалізуються баг-трекінговими системами:

• Створення звітів про дефекти, керування їх життєвим циклом з урахуванням контролю версій, прав доступу та дозволених переходів зі стану.

• Збір, аналіз та надання статистики у зручній для сприйняття людиною формі.

• Розсилка повідомлень, нагадувань та інших артефактів відповідним співробітникам.

• Організація взаємозв'язків між звітами про дефекти, тест-кейсами, вимогами та аналіз таких зв'язків з можливістю формування рекомендацій.

• Підготовка інформації для включення до звіту про результати тестування.

• Інтеграція із системами управління проектами.

Jira

1. **Project** (проект) дозволяє вказати, якого проекту належить дефект.

2. **Issue type** (тип запису/артефакту) дозволяє вказати, що саме являє собою артефакт, що створюється. JIRA дозволяє створювати не тільки звіти про дефекти, а й безліч інших артефактів, типи яких можна налаштовувати. За замовчуванням наведено:

• **Improvement** (пропозиція щодо покращення) — детально описано в розділі, присвяченому полям звіту про дефект.

• **New feature** (нова особливість) — опис нової функціональності, нової властивості, нової особливості продукту.

• **Task** (завдання) — завдання для виконання тим чи іншим учасником проектної команди.

• **Custom issue** (довільний артефакт) — зазвичай це значення при налаштуванні JIRA видаляють, замінюючи своїми варіантами, або перейменовують на Issue.

3. **Summary** (короткий опис) дозволяє вказати короткий опис дефекту.

4. **Priority** (Терміновість) дозволяє вказати терміновість виправлення дефекту. За замовчуванням JIRA пропонує такі варіанти:

• **Highest** (найвища терміновість).

• **High** (висока терміновість).

• **Medium** (звичайна терміновість).

• **Low** (низька терміновість).

• **Lowest** (найнижча терміновість).

5. **Components** (компоненти) містить перелік компонентів програми, порушених дефектом (хоча іноді тут перераховують симптоми дефектів).

6. **Aff ected versions** (зачеплені версії) містить перелік версій продукту, в яких є дефект.

7. **Environment** (оточення) містить опис апаратної та програмної конфігурації, в якій проявляється дефект.

8. **Description** (Докладний опис) дозволяє вказати докладний опис дефекту.

9. **Original estimate** (початкова оцінка часу виправлення) дозволяє вказати початкову оцінку того, скільки часу займе усунення дефекту.

10. **Remaining estimate** (розрахунковий залишковий час виправлення) показує, скільки часу залишилося від початкової оцінки.

11. **Story points** (оціночні одиниці) дозволяє вказати складність дефекту (або іншого артефакту) у спеціальних оцінних одиницях, прийнятих у гнучких методологіях управління проектами.

12. **Labels** (мітки) містить мітки (теги, ключові слова), за якими можна групувати та класифікувати дефекти та інші артефакти.

13. **Epic/Theme** (історія/область) містить перелік високорівневих міток, що описують відношення до дефекту великі області вимог, великі модулі додатка, великі частини предметної області, об'ємні історії користувача і т.д.

14. **External issue id** (ідентифікатор зовнішнього артефакту) дозволяє пов'язати звіт про дефект або інший артефакт із зовнішнім документом.

5. **Epic link** (посилання на історію/область) містить посилання на історію/область (див. пункт 13), найближче до дефекту.

16. **Has a storys** (історії) містить посилання та/або опис історій, пов'язаних з дефектом (як правило, тут наводяться посилання на зовнішні документи).

17. **Tester** (тестувальник) містить ім'я автора опису дефекту.

18. **Additional information** (додаткова інформація) містить додаткову інформацію про дефект.

19. **Sprint** (спринт) містить номер спринту (2–4-тижнева ітерація розробки проекту в термінології гнучких методологій управління проектами), під час якого було виявлено дефект.

Create Issue

Configure Fields

Project*

Issue Type* Bug

Summary*

Priority Major

Component/s

Start typing to get a list of possible matches or press down to select.

Affects Version/s

Start typing to get a list of possible matches or press down to select.

Environment

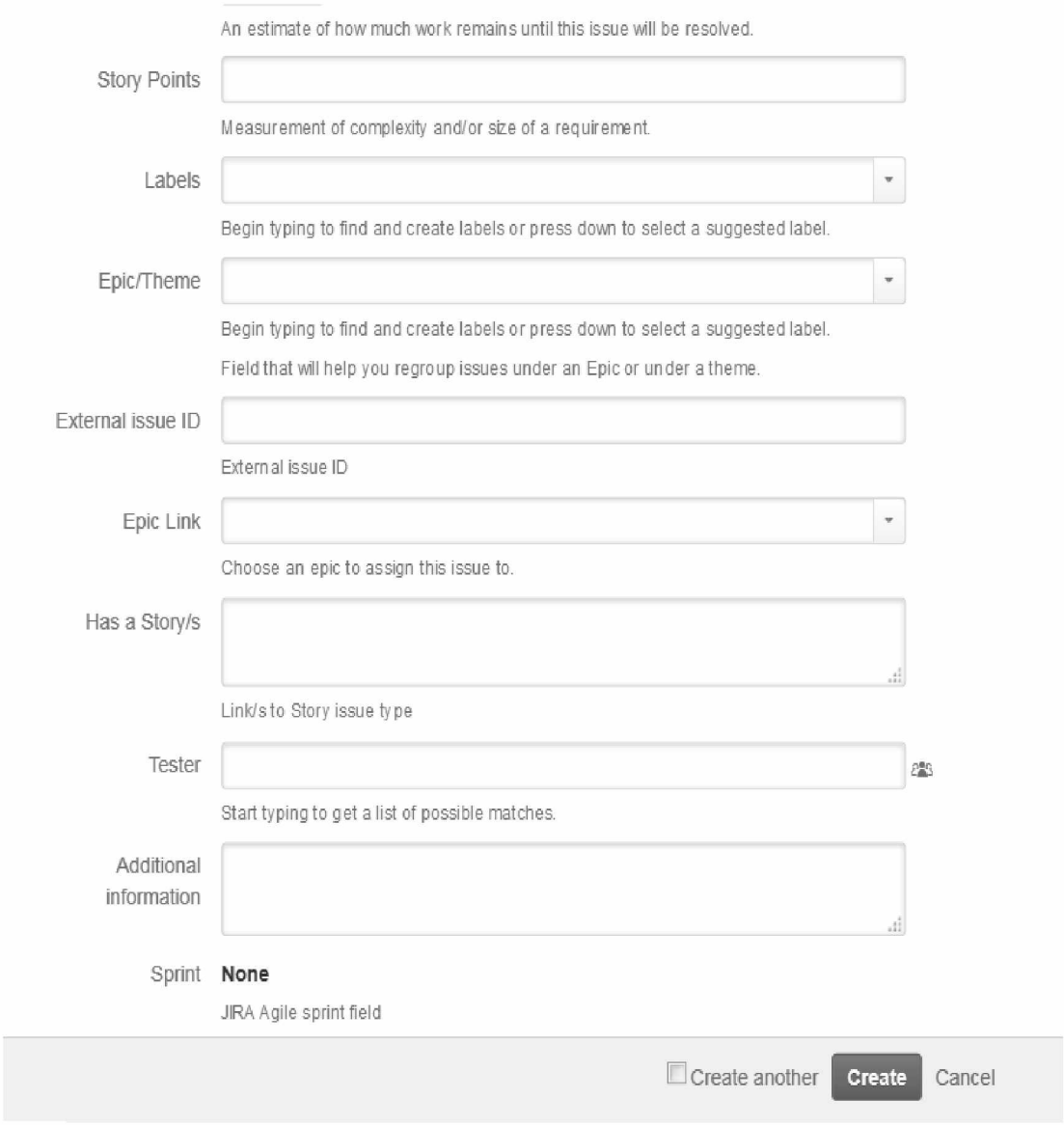
For example operating system, software platform and/or hardware specifications (include as appropriate for the issue).

Description

Original Estimate (eg. 3w 4d 12h)

The original estimate of how much work is involved in resolving this issue.

Remaining Estimate (eg. 3w 4d 12h)


An estimate of how much work remains until this issue will be resolved.
Story Points
Measurement of complexity and/or size of a requirement.
Labels
Begin typing to find and create labels or press down to select a suggested label.
Epic/Theme
Begin typing to find and create labels or press down to select a suggested label.
Field that will help you regroup issues under an Epic or under a theme.
External issue ID
External issue ID
Epic Link
Choose an epic to assign this issue to.
Has a Story/s
Link/s to Story issue type
Tester
Start typing to get a list of possible matches.
Additional information
Sprint
None
JIRA Agile sprint field
Create another
Create
Cancel

Bugzilla

1. **Product** (продукт) дозволяє вказати, якого продукту (проекту) належить дефект.
2. **Reporter** (автор звіту) містить e-mail автора опису дефекту.
3. **Component** (компонент) містить вказівку компонента додатка, до якого належить описуваний дефект.
4. **Component description** (опис компонента) містить опис компонента додатку, до якого відноситься дефект, що описується. Ця інформація завантажується автоматично під час вибору компонента.
5. **Version** (версія) містить вказівку версії продукту, де було виявлено дефект.
6. **Severity** (важливість) містить вказівку на важливість дефекту. За замовчуванням запропоновано такі варіанти:
   - **Blocker** (Блокуючий дефект) — дефект не дозволяє вирішити за допомогою програми певне завдання.
   - **Critical** (критична важливість).
   - **Major** (висока важливість).
   - **Normal** (звичайна важливість).
   - **Minor** (низька важливість).
   - **Trivial** (найнижча важливість).
   - **Enhancement** (пропозиція щодо покращення) — детально описано в розділі, присвяченому полям звіту про дефект (див. опис поля «симптом», значення «пропозиція щодо покращення»{182}).
7. **Hardware** (апаратне забезпечення) дозволяє вибрати профіль апаратного оточення, в якому проявляється дефект.
8. **OS** (операційна система) дозволяє вказати операційну систему, під якою виявляється дефект.
9. **Priority** (Терміновість) дозволяє вказати терміновість виправлення дефекту. За замовчуванням Bugzilla пропонує такі варіанти:
   - **Highest** (найвища терміновість).
   - **High** (висока терміновість).
   - **Normal** (звичайна терміновість).
   - **Low** (низька терміновість).
   - **Lowest** (найнижча терміновість).

10. **Status** (Статус) дозволяє встановити статус звіту про дефект. За замовчуванням Bugzilla пропонує такі варіанти статусів:

• **Unconfi** rmed (не підтверджено) — дефект поки що не вивчений, і немає гарантії того, що він справді коректно описаний.

• **Confi rmed** (підтверджено) – дефект вивчений, коректність опису підтверджена.

• **In progress** (в роботі) — ведеться робота з вивчення та усунення дефекту.

В офіційній документації рекомендується відразу після встановлення Bugzilla сконфігурувати набір статусів та правила життєвого циклу звіту про дефекти відповідно до прийнятих у вашій компанії правил.

11. **Assignee** (відповідальний) вказує e-mail учасника проектної команди, відповідального за вивчення та виправлення дефекту.

12. **CC** (повідомляти) містить список e-mail адрес учасників проектної команди, які будуть отримувати повідомлення про те, що відбувається з цим дефектом.

13. **Default CC** (повідомляти за замовчуванням) містить e-mail адресу(и) учасників проектної команди, які за умовчанням отримуватимуть повідомлення про те, що відбувається з будь-якими дефектами (найчастіше тут зазначаються e-mail адреси розсилок).

14. **Original estimation** (початкова оцінка) дозволяє вказати початкову оцінку того, скільки часу займе усунення дефекту.

15. **Deadline** (крайній термін) дозволяє вказати дату, до якої дефект обов'язково потрібно виправити.

16. **Alias** (псевдонім) дозволяє вказати коротку назву дефекту (можливо, у вигляді якоїсь абревіатури) для зручності згадування дефекту в різноманітних документах.

17. **URL** (URL) дозволяє вказати URL, де проявляється дефект (особливо актуально для веб-додатків).

18. **Summary** (короткий опис) дозволяє вказати короткий опис дефекту.

19. **Description** (Докладний опис) дозволяє вказати докладний опис дефекту.

20. **Attachment** (вкладення) дозволяє додати до звіту про дефект вкладення у вигляді прикріплених файлів.

21. **Depends on** (залежить від) дозволяє вказати перелік дефектів, які мають бути усунені до початку роботи з цим дефектом.

22. **Blocks** (блокує) дозволяє вказати перелік дефектів, до роботи з якими можна буде розпочати лише після усунення цього дефекту.


* Product: TestProduct
Reporter: user@user.com
* Component: TestComponent
Component Description
This is a test component.
* Version: unspecified
Severity: enhancement
Hardware: PC
OS: Windows
Priority: ---
Status: CONFIRMED
Assignee: adm@adm.com
CC:
Default CC:
Orig. Est.:
Deadline:
Alias:
URL: http://
* Summary:
Description:
Attachment: Add an attachment
Depends on:
Blocks:
Submit Bug
Remember values as bookmarkable template

ВЛАСТИВОСТІ ЯКІСНИХ ЗВІТІВ О ДЕФЕКТАХ

Звіт про дефект може виявитися неякісним (а отже, ймовірність виправлення дефекту знизиться), якщо в ньому порушено одну з таких властивостей.

Ретельне заповнення всіх полів точною та коректною інформацією. Порушення цієї властивості відбувається з багатьох причин: недостатній досвід тестувальника, неуважність, лінь і т.д. Найяскравішими проявами такої проблеми вважатимуться такі:

• Частина важливих розуміння проблеми полів не заповнена. В результаті звіт перетворюється на набір уривчастих відомостей, використовувати які для виправлення дефекту неможливо.

• Наданої інформації недостатньо для розуміння проблеми. Наприклад, з такого поганого детального опису взагалі не ясно, про що йдеться: «Додаток іноді неправильно конвертує деякі файли».

• Надана інформація є некоректною (наприклад, вказані неправильні повідомлення програми, неправильні технічні терміни тощо). Найчастіше таке відбувається через неуважність (наслідки помилкового copy-paste та відсутності фінальної вичитки звіту перед публікацією).

• «Дефект» (саме так, у лапках) знайдений у функціональності, яка ще не була оголошена як готова до тестування. Тобто тестувальник констатує, що неправильно працює те, що й не мало (поки що!) правильно працювати.

• У звіті є жаргонова лексика: як у прямому сенсі — нелітературні висловлювання, так і деякі технічні жаргонізми, зрозумілі вкрай обмеженому колу людей. Наприклад: "Фігово підчепилися чартники". (Мали на увазі: «Не всі таблиці кодувань завантажені успішно».)

• Звіт замість опису проблеми із додатком критикує роботу когось із учасників проектної команди. Наприклад: "Ну яким дурнем треба бути, щоб таке зробити?!"

• У звіті втрачено якусь незначну на перший погляд, але за фактом критичну для відтворення дефекту проблему. Найчастіше це проявляється у вигляді пропуску якогось кроку з відтворення, відсутності або недостатньої подробиці опису оточення, надмірно узагальненої вказівки значень, що вводяться і т.п.

• Звіту виставлено неправильні (зазвичай занижені) важливість чи терміновість. Щоб уникнути цієї проблеми, варто ретельно дослідити дефект, визначати його найнебезпечніші наслідки та аргументовано відстоювати свою точку зору, якщо колеги вважають інакше.

• До звіту не додані копії екрана (особливо важливі для косметичних дефектів) або інші файли. Класика такої помилки: звіт описує неправильну роботу програми з деяким файлом, але сам файл не доданий.

• Звіт написаний безграмотно з погляду людської мови. Іноді на це можна заплющити очі, але іноді це стає реальною проблемою, наприклад: «Not keyboard in parameters accepting values» (це реальна цитата; і сам автор так і не зміг пояснити, що було на увазі).

Правильна технічна мова. Ця властивість однаково стосується і вимог, і тест-кейсів, і звітів про дефекти — будь-якої документації

Специфічність опису кроків. Говорячи про тест-кейси, ми підкреслювали, що в їхніх кроках варто дотримуватися золотої середини між специфічністю та спільністю. У звітах про дефекти перевага, як правило, надається специфічності з дуже простої причини: брак якоїсь дрібної деталі може призвести до неможливості відтворення дефекту. Тому, якщо у вас є хоч найменший сумнів, чи важлива якась деталь, вважайте, що вона важлива.

Відсутність зайвих дій та/або їх довгих описів. Найчастіше ця властивість має на увазі, що не потрібно в кроках по відтворенню дефекту довго і по пунктах розписувати те, що можна замінити однією фразою.

Друга за частотою помилка — початок кожного звіту про дефект із запуску програми та докладного опису щодо приведення його в той чи інший стан. Цілком допустимою практикою є написання у звіті про дефект приготувань (за аналогією з тест-кейсами) або опис потрібного стану програми в одному (першому) кроці.

Відсутність дублікатів. Коли в проектній команді працює велика кількість тестувальників, може виникнути ситуація, коли один і той же дефект буде описаний кілька разів різними людьми. А іноді буває так, що навіть той самий тестувальник уже забув, що колись давно вже виявляв якусь проблему, і тепер описує її наново. Уникнути подібної ситуації дозволяє наступний набір рекомендацій:

Очевидність та зрозумілість. Описуйте дефект так, щоб у вашого звіту не виникло жодного сумніву в тому, що це дійсно дефект. Найкраще ця властивість досягається за рахунок ретельного пояснення фактичного та очікуваного результату, а також вказівки посилання на вимогу у полі «Детальний опис»

Простежуваність. З інформації, що міститься в якісному звіті про дефект інформації, має бути зрозуміло, яку частину програми, які функції і які вимоги зачіпає дефект. Найкраще ця властивість досягається правильним використанням можливостей інструментального засобу управління звітами про дефекти: вказуйте у звіті про дефект компоненти додатка, посилання на вимоги, тест-кейси, суміжні звіти про дефекти (схожі дефекти; залежні та залежать від даного дефекти), розставляючи теги і т.д.

Окремі звіти для кожного нового дефекту. Існує два непорушні правила**:**

• У кожному звіті описується рівно один дефект (якщо той самий дефект проявляється в декількох місцях, ці прояви перераховуються в докладному описі).

• У разі виявлення нового дефекту створюється новий звіт. Не можна для опису нового дефекту правити старі звіти, переводячи їх у стан «відновлено».

Відповідність прийнятим шаблонам оформлення та традиціям. Як і у випадку з тест-кейсами, з шаблонами оформлення звітів про дефекти проблем не виникає: вони визначені зразком або екранною формою інструментального засобу управління життєвим циклом звітів про дефекти.

ЛОГІКА СТВОРЕННЯ ЄФЕКТИВНИХ ЗВІТІВ

При створенні звіту про дефект рекомендується слідувати наступному алгоритму:

0. Виявити дефект J.
1. Зрозуміти суть проблеми.
2. Відтворити дефект.
3. Перевірити наявність опису знайденого вами дефекту у системі управління дефектами.
4. Сформулювати суть проблеми у вигляді «що зробили, що отримали, що очікували отримати».
5. Заповнити поля звіту, починаючи з детального опису.
6. Після заповнення всіх полів уважно перечитати звіт, виправивши неточності та додавши подробиці.
7. Ще раз перечитати звіт, т.к. у пункті 6 ви точно щось упустили J.

ПОМИЛКИ ОФОРМЛЕННЯ ТА ФОРМУЛЮВАННЯ

Погані короткі описи (summary). Формально ця проблема відноситься до оформлення, але фактично вона набагато небезпечніша: адже читання звіту про дефект і усвідомлення суті проблеми починається саме з короткого опису.

Ідентичні короткий та докладний опис (summary та description). Так, зрідка бувають настільки прості дефекти, що для них достатньо одного короткого опису (наприклад, «Опечатка в імені пункту головного меню “File” (зараз “Fille”)»), але якщо дефект пов'язаний з більш-менш складною поведінкою програми, варто продумати як мінімум три способи опису проблеми:

• короткий для поля «короткий опис» (його краще формулювати наприкінці роздумів);

• докладний для поля «докладний опис» (що пояснює та розширює інформацію з «короткого опису»);

ще один короткий для останнього кроку в кроках по відтворенню дефекту.

Відсутність у докладному описі явної вказівки фактичного результату, очікуваного результату та посилання на вимогу, якщо вони важливі, і їх можна вказати.

Ігнорування лапок, що призводить до спотворення сенсу. Як ви зрозумієте такий короткий опис, як запис зникає при наведенні миші? Якийсь запис зникає при наведенні миші? А ось і ні, виявляється, «поле “Запис” зникає при наведенні миші». Навіть якщо недописати слово «поле», лапки підкажуть, що мають на увазі власне ім'я, тобто. назва якогось елемента.

Загальні проблеми із формулюваннями фраз українською та англійською мовами. нелегко одразу навчитися формулювати думку одночасно дуже коротко та інформативно, але не менш складно читати подібні твори (цитати дослівні):

Зайві пункти на кроках відтворення. Не варто починати «зі створення світу», більшість проектної команди досить добре знає додаток, щоб «пізнати» його ключові частини.

Копії екрана як «копій всього екрана цілком». Найчастіше потрібно зробити копію якогось конкретного вікна програми, а не всього екрана, тоді допоможе Alt+PrintScreen. Навіть якщо важливо захопити більше одного вікна, практично будь-який графічний редактор дозволяє відрізати непотрібну частину картинки.

Копії екрана, на яких не позначено проблему. Якщо обвести проблемну область червоною лінією, це в рази підвищить швидкість і простоту розуміння суті проблеми здебільшого.

Відкладання написання звіту «на потім». Прагнення спочатку знайти більше дефектів, а вже потім їх описувати призводить до того, що якісь важливі деталі (а іноді й самі дефекти!) забуваються. Якщо «на потім» вимірюється не хвилинами, а годинами чи навіть днями, проектна команда не отримує вчасної інформації.

ЛОГІЧНІ ПОМИЛКИ

Вигадані дефекти. Однією з найбільш образливих для тестувальника причин відхилення звіту про дефект є так звана «описана поведінка не є дефектом» («not a bug»), коли з якоїсь причини коректна поведінка програми описується як помилкова.

Віднесення розширених можливостей застосування до дефектів. Найяскравішим прикладом цього випадку є опис як дефект того факту, що програма запускається під операційними системами, які не зазначені явно у списку підтримуваних.

Невірно вказані симптоми. Це не смертельно, завжди можна підправити, але якщо спочатку звіти будуть згруповані за симптомами, їхня неправильна вказівка створює безліч незручностей, що дратують.

Надмірно занижені (або завищені) важливість та терміновість. З цією бідою досить ефективно борються проведенням загальних зборів та переглядом звітів про дефекти силами всієї команди (або хоча б кількох осіб), але якщо ці показники занижені надмірно, є висока ймовірність, що пройде дуже багато часу, перш ніж до такого звіту просто дійде черга наступних зборах з перегляду.

Концентрація на дрібницях на шкоду головному. Тут варто згадати хрестоматійний приклад, коли тестувальник знайшов проблему, що призводить до краху додатка зі втратою даних, але записав її як косметичний дефект (у повідомленні про помилку, яке «перед смертю» показував додаток, була помилка). Завжди думайте про те, як неприємність, що сталася з додатком, вплине на користувачів, які складності вони можуть через це випробувати, наскільки це для них важливо, тоді шанс побачити реальну проблему різко підвищується.

Вказання на кроках відтворення інформації, неважливу для відтворення помилки. Прагнення прописати все максимально докладно іноді набуває нездорової форми, коли у звіт про дефект починає потрапляти чи не інформація про погоду за вікном та курс національної валюти.

Ігнорування "Послідовних дефектів".

ПЛАНУВАННЯ ТА ЗВІТНІСТЬ

Життєвий цикл тестування: кожна ітерація починається з планування і закінчується звітністю, яка стає основою для планування наступної ітерації — і так далі. Таким чином, планування та звітність перебувають у тісному взаємозв'язку, і проблеми з одним із цих видів діяльності неминуче призводять до проблем з іншим видом, а зрештою і до проблем із проектом загалом.

- Без якісного планування не ясно, кому і що потрібно робити.
- Коли не зрозуміло, кому і що робити, робота виконується погано.
- Коли робота виконана погано і не зрозумілі точні причини, неможливо зробити правильні висновки про те, як виправити ситуацію.
- Без правильних висновків неможливо створити якісний звіт про результати роботи.
- Без якісного звіту про результати роботи неможливо створити якісний план подальшої роботи.
- Всі. Порочне коло замкнулося. Проект вмирає.

Планування (planning) — безперервний процес прийняття управлінських рішень та методичної організації зусиль щодо їх реалізації з метою забезпечення якості деякого процесу протягом тривалого періоду часу.

До високорівневих завдань планування належать:

• зниження невизначеності;

• підвищення ефективності;

• покращення розуміння цілей;

• створення основи управління процесами.

Звітність (reporting) — збирання та розповсюдження інформації про результати роботи (включаючи поточний статус, оцінку прогресу та прогноз розвитку ситуації).

До високорівневих завдань звітності належать:

• збір, агрегація та надання у зручній для сприйняття формі об'єктивної інформації про результати роботи;

• формування оцінки поточного статусу та прогресу (порівняно з планом);

• позначення існуючих та можливих проблем (якщо такі є);

• формування прогнозу розвитку ситуації та фіксація рекомендацій щодо усунення проблем та підвищення ефективності роботи.

ТЕСТ-ПАЛН ТА ЗВІТ О РЕЗУЛЬТАТАХ ТЕСТУВАННЯ

Тест-план (test plan) — документ, що описує та регламентує перелік робіт з тестування, а також відповідні техніки та підходи, стратегію, сферу відповідальності, ресурси, розклад та ключові дати.

До низькорівневих завдань планування у тестуванні відносяться:

• оцінка обсягу та складності робіт;

• визначення необхідних ресурсів та джерел їх отримання;

• визначення розкладу, термінів та ключових точок;

• оцінка ризиків та підготовка превентивних контрзаходів;

• розподіл обов'язків та відповідальності;

• узгодження робіт із тестування з діяльністю учасників проектної команди, які займаються іншими завданнями.

Якісний тест-план має більшість властивостей якісних вимог, а також розширює їх набір наступними пунктами:

• Реалістичність (запланований підхід реально виконаємо).

• Гнучкість (якісний тест-план не тільки модифікується з точки зору роботи з документом, але й побудований таким чином, щоб при виникненні непередбачуваних обставин допускати швидку зміну будь-якої зі своїх частин без порушення взаємозв'язку з іншими частинами).

• Узгодженість із загальним проектним планом та іншими окремими планами (наприклад, планом розробки).

У ЗАГАЛЬНОМУ ВИПАДКУ ТЕСТ-ПЛАН ВКЛЮЧАЄ НАСТУПНІ РОЗДІЛИ

• **Ціль.** Гранично короткий опис мети розробки програми (частково це нагадує бізнес-вимоги, але тут інформація подається в ще більш стислому вигляді і в контексті того, на що слід звертати першорядну увагу при організації тестування та підвищення якості).

• **Області, що піддаються тестуванню**. Перелік функцій та/або нефункціональних особливостей програми, які будуть тестовані.

У деяких випадках тут також наводиться пріоритет відповідної галузі.

• **Області, які не піддаються тестуванню**. Перелік функцій та/або нефункціональних особливостей програми, які не будуть піддані тестуванню. Причини виключення тієї чи іншої області зі списку тестованих можуть бути різними — від гранично низької їх важливості для замовника до нестачі часу чи інших ресурсів. Цей перелік складається, щоб у проектної команди та інших заінтересованих осіб було чітке єдине розуміння, що тестування таких особливостей програми не заплановане — такий підхід дозволяє виключити появу хибних очікувань та неприємних сюрпризів.

• **Тестова стратегія та підходи**. Опис процесу тестування з точки зору застосовуваних методів, підходів, видів тестування, технологій, інструментальних засобів тощо.

• **Критерії**. Цей розділ включає наступні підрозділи:

- **Приймальні критерії, критерії якості** — будь-які об'єктивні показники якості, яким продукт, що розробляється, повинен відповідати з точки зору замовника або користувача, щоб вважатися готовим до експлуатації.
- **Критерії початку тестування** — перелік умов, при виконанні яких команда розпочинає тестування. Наявність цього критерію страхує команду від безглуздої витрати зусиль за умов, коли тестування не принесе очікуваної користі.
- **Критерії припинення тестування** — перелік умов, під час яких тестування припиняється. Наявність цього критерію також страхує команду від безглуздої витрати зусиль в умовах, коли тестування не принесе очікуваної користі.
- **Критерії відновлення тестування** — перелік умов, при виконанні яких тестування відновлюється (як правило, після зупинки).
- **Критерії завершення тестування** — перелік умов, під час яких тестування завершується. Наявність цього критерію страхує команду як від передчасного припинення тестування, так і від продовження тестування в умовах, коли воно вже припиняє приносити відчутний ефект.

• **Ресурси** (Resources). У цьому розділі тест-плану перераховуються всі необхідні для успішної реалізації стратегії тестування ресурси, які можна розділити на:

º програмні ресурси (яке ПЗ необхідне команді тестувальників, скільки копій та з якими ліцензіями (якщо йдеться про комерційне ПЗ));

º апаратні ресурси (яке апаратне забезпечення, в якій кількості та до якого моменту необхідно команді тестувальників);

º людські ресурси (скільки фахівців якого рівня та зі знаннями в яких галузях має підключитися до команди тестувальників у той чи інший момент часу);

º часові ресурси (скільки за часом займе виконання тих чи інших робіт);

º фінансові ресурси (у яку суму обійдеться використання наявних або одержання ресурсів, перелічених у попередніх пунктах цього списку); у багатьох компаніях фінансові ресурси можуть бути подані окремим документом, т.к. є конфіденційною інформацією

• **Розклад.** Фактично це календар, у якому зазначено, що й до якого моменту має бути зроблено. Особлива увага приділяється т.зв. «ключовим точкам» (milestones), на момент настання яких має бути отриманий якийсь значний відчутний результат.

• **Ролі та відповідальність.** Перелік необхідних ролей (наприклад, «провідний тестувальник», «експерт з оптимізації продуктивності») та сфера відповідальності фахівців, які виконують ці ролі.

• **Оцінка ризиків.** Перелік ризиків, які з ймовірністю можуть виникнути в процесі роботи над проектом. По кожному ризику дається оцінка загрози, що їм представляються, і наводяться варіанти виходу з ситуації.

• **Документація**. Перелік тестової документації, що використовується, із зазначенням, хто і коли повинен її готувати і кому передавати.

Метрика (metric) – числова характеристика показника якості. Може включати опис способів оцінки та аналізу результату.

Обчислені значення можна використовувати для:

• прийняття рішень про початок, призупинення, відновлення або припинення тестування (див. вище розділ «Критерії» тест-плану);

• визначення ступеня відповідності продукту заявленим критеріям якості;

• визначення ступеня відхилення фактичного розвитку проекту від плану;

• виявлення «вузьких місць», потенційних ризиків та інших проблем;

• оцінки результативності прийнятих управлінських рішень;

• підготовки об'єктивної інформативної звітності.

Метрики може бути як прямими (не вимагають обчислень), і розрахунковими (обчислюються за формулою). Типові приклади прямих метрик — кількість розроблених тест-кейсів, кількість знайдених дефектів тощо. У розрахункових метриках можуть використовуватися як абсолютно тривіальні, так і досить складні формули.

Метрики може бути як прямими (не вимагають обчислень), і розрахунковими (обчислюються за формулою). Типові приклади прямих метрик — кількість розроблених тест-кейсів, кількість знайдених дефектів тощо. У розрахункових метриках можуть використовуватися як абсолютно тривіальні, так і досить складні формули.

$$T^{SP} = \frac{T^{Success}}{T^{Total}} \cdot 100\%,$$

Приклад простої метрики

$$T^{SC} = \sum_{Level}^{MaxLevel} \frac{(T_{Level} \cdot I)^{R_{Level}}}{B_{Level}},$$

$\mathbf{T^{SP}}$ — відсотковий показник успішного проходження тест-кейсів, $\mathbf{T^{Suuccess}}$ — кількість успішно виконаних тест-кейсів, $\mathbf{T^{Total}}$ — загальна кількість виконаних тест-кейсів.

Мінімальні межі значень:

- Початкова фаза проекту: 10%.
- Основна фаза проекту: 40%.
- Фінальна фаза проекту: 85%.

$\mathbf{T^{SC}}$ — інтегральна метрика проходження тест-кейсів у взаємозв'язку з вимогами та дефектами, $\mathbf{T_{Level}}$ — ступінь важливості тест-кейс, **I** — кількість виконань тест-кейс, $\mathbf{R_{Level}}$ — ступінь важливості вимоги , що перевіряється тест-кейсом, $\mathbf{T}_{\text{Level}}$ — кількість дефектів, виявлених тест-кейсом.

Спосіб аналізу:

• Ідеальним станом є безперервне зростання значення $\mathbf{T^{SC}}$.

• У разі негативної динаміки зменшення значення $\mathbf{T^{SC}}$ на 15 % і більше за останні три спринти може трактуватися як неприпустиме і бути достатнім приводом для припинення тестування.

У тестуванні існує велика кількість загальноприйнятих метрик, багато з яких можна зібрати автоматично з використанням інструментальних засобів управління проектами.

Наприклад:

• процентне відношення (не) виконаних тест-кейсів до всіх наявних;

• процентний показник успішного проходження тест-кейсів;

• відсотковий показник заблокованих тест-кейсів;

• густина розподілу дефектів;

• ефективність усунення дефектів;

• розподіл дефектів за важливістю та терміновістю;

Деякі метрики можуть обчислюватися на основі даних про розклад, наприклад метрика зсуву розкладу:

$$ScheduleSlippage = \frac{DaysToDeadline}{NeededDays} - 1$$

ScheduleSlippage — значення зсуву розкладу, **DaysDeadline** — кількість днів до запланованого завершення роботи, **NeededDays** — кількість днів, необхідне завершення роботи.

Значення **ScheduleSlippage** не повинно ставати негативним.

ПОКРИТТЯ

Покриття - відсотковий вираз ступеня, в якому досліджуваний елемент (coverage item) торкнуться відповідного набору тест-кейсів.

Найпростішими представниками метрик покриття можна вважати:

• **Метрику покриття вимог** (вимога вважається «покритою», якщо на неї посилається хоча б один тест-кейс):

$$R^{SimpleCoverage} = \frac{R^{Covered}}{R^{Total}} \cdot 100\%,$$

$\mathbf{R^{SimpleCoverage}}$ — метрика покриття вимог,

$\mathbf{R^{Covered}}$ — кількість вимог, покритих хоча б одним тест-кейсом,

$\mathbf{R^{Total}}$ — загальна кількість вимог .

• **Метрику щільності покриття вимог** (враховується, скільки тест-кейсів посилається на кілька вимог):

$$R^{DensityCoverage} = \frac{\sum T_i}{T^{Total}.R^{Total}} \cdot 100\%,$$

$\mathbf{R^{DensityCoverage}}$ — щільність покриття вимог,

$\mathbf{T_i}$ — кількість тест-кейсів, що покривають **i**-у вимогу,

$\mathbf{T^{Total}}$ — загальна кількість тест-кейсів,

$\mathbf{R^{Total}}$ — загальна кількість вимог.

• **Метрику покриття класів еквівалентності** (аналізується, скільки класів еквівалентності торкнулося тест-кейсів).

$$E^{Coverage} = \frac{E^{Covered}}{E^{Total}} \cdot 100\%$$

E^{Coverege} — метрика покриття класів еквівалентності,

E^{Covered} — кількість класів еквівалентності, покритих хоча б одним тест-кейсом,

E^{Total} — загальна кількість класів еквівалентності.

• **Метрику покриття граничних умов** (аналізується, скільки значень із групи граничних умов зачеплено тест-кейсами).

$$B^{Coverage} = \frac{B^{Covered}}{B^{Total}} \cdot 100\%,$$

B^{Coverege} — метрика покриття граничних умов,

B^{Covered} — кількість граничних умов, покритих хоча б одним тест-кейсом,

B^{Total} — загальна кількість граничних умов.

• **Метрики покриття коду модульними тест-кейсами**. Таких метрик дуже багато, але вся їх суть зводиться до виявлення певної характеристики коду (кількість рядків, гілок, шляхів, умов і т.д.) та визначення, який відсоток представників цієї характеристики покритий тест-кейсами.

ЗВІТ О РЕЗУЛЬТАТАХ ТЕСТУВАННЯ

звіт про результати тестування — документ, що узагальнює результати робіт із тестування та містить інформацію, достатню для співвіднесення поточної ситуації з тест-планом та прийняття необхідних управлінських рішень.

До низькорівневих завдань звітності у тестуванні відносяться:

• оцінка обсягу та якості виконаних робіт;

• порівняння поточного прогресу із тест-планом (у тому числі за допомогою аналізу значень метрик);

• опис наявних складнощів та формування рекомендацій щодо їх усунення;

• надання особам, зацікавленим у проекті, повної та об'єктивної інформації про поточний стан якості проекту, виражену у конкретних фактах та числах.

Як і будь-який інший документ, звіт про результати тестування може бути якісним або мати недоліки. Якісний звіт про результати тестування має багато властивостей якісних вимог, а також розширює їх набір наступними пунктами:

• Інформативність (в ідеалі після прочитання звіту не повинно залишатися жодних відкритих питань про те, що відбувається з проектом у контексті якості).

• Точність та об'єктивність (за жодних умов у звіті не допускається спотворення фактів, а особисті думки мають бути підкріплені твердими обґрунтуваннями).

Звіт про результати тестування насамперед потрібен наступним особам:

• менеджеру проекту — як джерело інформації про поточну ситуацію та основу для ухвалення управлінських рішень;

• керівнику команди розробників («дев-ліду») — як додатковий об'єктивний погляд на те, що відбувається на проекті;

• керівнику команди тестувальників («тест-лід») — як спосіб структурувати власні думки та зібрати необхідний матеріал для звернення до менеджера проекту з нагальних питань, якщо це потребує;

• замовнику — як найбільше об'єктивне джерело інформації про те, що відбувається на проекті, за який він платить свої гроші.

У загальному випадку звіт про результати тестування включає такі розділи:

• **Короткий опис**. У гранично стислій формі відображає основні досягнення, проблеми, висновки та рекомендації. В ідеальному випадку прочитання короткого опису може бути достатньо для формування повноцінного уявлення про те, що позбавить необхідності читати весь звіт.

• **Команда випробувачів**. Список учасників проектної команди, задіяних у забезпеченні якості, із зазначенням їх посад та ролей у підзвітний період.

• **Опис процесу тестування**. Послідовний опис того, які роботи було виконано за підзвітний період.

• **Розклад**. Детальний розклад роботи команди тестувальників та/або особисті розклади учасників команди.

• **Статистика щодо нових дефектів**. Таблиця, в якій представлені дані щодо виявлених за підзвітний період дефектів (з класифікацією по стадії життєвого циклу та важливості).

• **Список нових дефектів**. Список виявлених за підзвітний період дефектів зі своїми короткими описами та важливістю.

• **Статистика з усіх дефектів** Таблиця, в якій представлені дані щодо виявлених за весь час існування проекту дефектів (з класифікацією по стадії життєвого циклу та важливості). Як правило, до цього ж розділу додається графік, що відображає такі статистичні дані.

• **Рекомендації** . Обґрунтовані висновки та рекомендації щодо прийняття тих чи інших управлінських рішень (зміни тест-плану, запиту чи звільнення ресурсів тощо). Тут цій інформації можна відвести більше місця, ніж у короткому описі (summary), наголосивши саме на тому, що і чому рекомендується зробити в існуючій ситуації.

• **Програми** Фактичні дані (як правило, значення метрик та графічне подання їх зміни у часі).

ЛОГІКА ПОБУДОВИ ЗВІТУ О РЕЗУЛЬТАТАХ ТЕСТУВАННЯ

• Висновки будуються на основі цілей (які були відображені у плані).
• Висновки доповнюються рекомендаціями.
• Як висновки, і рекомендації суворо обгрунтовуються.
• Обґрунтування ґрунтується на об'єктивних фактах.

Висновки мають бути:
• Короткі.
• Інформативними.
• Корисні для читача звіт.

Рекомендації мають бути:
• Короткі.
• Реально здійсненними.
• Ті, хто дає як розуміння того, що треба зробити, так і деякий простір для прийняття власних рішень.

Обґрунтування висновків та рекомендацій — проміжна ланка між гранично стислими результатами аналізу та величезною кількістю фактичних даних. Воно дає відповіді на запитання на кшталт:

• «Чому ми так вважаємо?»
• «Невже це так?!»
• Де взяти додаткові дані?

Фактичний матеріал містить найрізноманітніші дані, отримані в процесі тестування. Сюди можуть відноситися звіти про дефекти, журнали засобів автоматизації, створені різними додатками набори файлів і т.д. Як правило, до звіту про результати тестування додаються лише скорочені агреговані вибірки подібних даних (якщо це можливо), а також наводяться посилання на відповідні документи, розділи системи управління проектом, шляхи до сховища даних тощо.

ОЦІНКА ВИТРАТ

Трудовитрати — кількість робочого часу, необхідного для виконання роботи (виражається в людино-годинах).

Будь-яка оцінка краща за її відсутність.

Оптимізм згубний.

Оцінка має бути аргументована.

Простий спосіб навчитися оцінювати – оцінювати.

Алгоритм навчання для формування оцінки:

• **Сформуйте оцінку.** Раніше вже було зазначено, що немає нічого страшного в тому, що набуте значення може виявитися дуже далеким від реальності. Для початку воно просто має бути.

• **Запишіть отриману оцінку.** Це застрахує вас як мінімум від двох ризиків: забути отримане значення (особливо якщо робота зайняла багато часу), збрехати собі в стилі «ну, я якось приблизно так і думав».

• **Виконайте роботу.** В окремих випадках люди схильні підлаштовуватися під заздалегідь сформовану оцінку, прискорюючи або сповільнюючи виконання роботи, — це також корисна навичка, але зараз така поведінка заважатиме. Однак якщо ви тренуватиметеся на десятках і сотнях різних завдань, ви фізично не зможете «підлаштуватися» під кожну з них і почнете отримувати реальні результати.

• **Звірте реальні результати з раніше сформованою оцінкою**.

• **Зважте на помилки при формуванні нових оцінок.** На цьому етапі дуже корисно не просто відзначити відхилення, а подумати, що спричинило його появу.

• **Повторюйте цей алгоритм якомога частіше для різних областей життя.** Зараз ціна ваших помилок вкрай мала, а напрацьований досвід від цього стає не менш цінним.

Корисні ідеї щодо формування оцінки трудовитрат:

• Додайте невеликий «буфер» (за часом, бюджетом чи іншими критичними ресурсами) на непередбачені обставини. Чим дальніший прогноз ви будуєте, тим більшим може бути цей «буфер» — від 5–10 до 30–40%. Але в жодному разі не варто свідомо завищувати оцінку в рази.

• З'ясуйте свій «коефіцієнт спотворення»: більшість людей через особливості свого мислення схильні постійно або занижувати, або завищувати оцінку. Багаторазово формуючи оцінку трудовитрат і порівнюючи її згодом із реальністю, ви можете помітити певну закономірність, яку можна виразити числом. Наприклад, може бути, що ви схильні занижувати оцінку в 1.3 раза. Спробуйте наступного разу внести відповідне виправлення.

• Зважайте на обставини, що не залежать від вас. Наприклад, ви точно впевнені, що виконаєте тестування чергового білда за N людино-годин, ви врахували всі фактори, що відволікають, і т.д. і вирішили, що точно закінчите до такої дати. А потім насправді випуск білда затримується на два дні, і ваш прогноз по моменту завершення роботи виявляється нереалістичним.

• Заздалегідь замислюйтеся над необхідні ресурси. Приміром, необхідну інфраструктуру можна підготувати (або замовити) заздалегідь, т.я. на подібні допоміжні завдання може бути витрачено багато часу, до того ж основна робота часто не може бути розпочата, доки не будуть завершені всі приготування.

• Шукайте способи організувати паралельне виконання завдань. Навіть якщо ви працюєте один, все одно якісь завдання можна і потрібно виконувати паралельно (наприклад, уточнення тест-плану, поки відбувається розгортання віртуальних машин). Якщо робота виконується кількома людьми, розпаралелювання роботи можна вважати життєвою необхідністю.

• Періодично звіряйтесь з планом, вносите коригування в оцінку та повідомляйте зацікавлених осіб про внесені зміни заздалегідь. Наприклад, ви зрозуміли (як у згаданому вище прикладі із затримкою білда), що завершите роботу як мінімум на два дні пізніше. Якщо ви повідомите проектну команду негайно, у ваших колег з'являється шанс скоригувати свої власні плани. Якщо ж ви в «годину ікс» подасте сюрприз про зсуви терміну на два дні, ви створите колегам об'єктивну проблему.

• Використовуйте інструментальні засоби - від електронних календарів до можливостей вашої системи управління проектом: це дозволить вам як мінімум не тримати в пам'яті купу дрібниць, а як максимум - підвищить точність оцінки, що формується.

Структурна декомпозиція — ієрархічна декомпозиція об'ємних завдань на дедалі більші підзадачі з метою спрощення оцінки, планування та моніторингу виконання роботи.

У процесі виконання структурної декомпозиції великі завдання поділяються на дедалі дрібніші підзавдання, що дозволяє нам:

• описати весь обсяг робіт з точністю, достатньої для чіткого розуміння суті завдань, формування досить точної оцінки трудовитрат і вироблення показників досягнення результатів;

• визначити весь обсяг трудовитрат як суму трудовитрат з окремих завдань (з урахуванням необхідних поправок);

• від інтуїтивного уявлення перейти до конкретного переліку окремих дій, що полегшує побудову плану, прийняття рішень про розпаралелювання робіт і т.д.

Якщо абстрагуватися від наукового підходу та формул, то суть такої оцінки зводиться до наступних кроків:

• декомпозиції вимог до рівня, на якому з'являється можливість створення якісних чек-листів;

• декомпозиції завдань із тестування кожного пункту чек-листа до рівня «тестування дій» (створення тест-кейсів, виконання тест-кейсів, створення звітів про дефекти і т.д.);

• виконання оцінки з урахуванням власної продуктивності.

Позитивні та негативні сторони автоматизованого тестування

До переваг можна віднести

- Швидкість виконання тест-кейс може в рази і на порядки перевищувати можливості людини.
- Відсутній вплив людського фактора у процесі виконання тест-кейсів.
- Засоби автоматизації здатні виконати тест-кейси, в принципі непосильні для людини через свою складність, швидкість або інші фактори.
- Засоби автоматизації здатні збирати, зберігати, аналізувати, агрегувати та представляти у зручній для сприйняття людиною формі колосальні обсяги даних.
- Засоби автоматизації здатні виконувати низькорівневі дії з програмою, операційною системою, каналами передачі даних тощо.

Використанням автоматизації ми отримуємо можливість збільшити тестове покриття за рахунок:

- виконання тест-кейсів, про які раніше не варто і думати;
- багаторазового повторення тест-кейсів із різними вхідними даними;
- вивільнення часу створення нових тест-кейсів.

До недоліків та ризиків можна віднести:

- Необхідність наявності висококваліфікованого персоналу через те, що автоматизація — це «проект усередині проекту».
- Розробка та супровід як самих автоматизованих тест-кейсів, так і всієї необхідної інфраструктури займає дуже багато часу.
- Автоматизація вимагає ретельнішого планування та управління ризиками, так, як в іншому випадку проекту може бути завдано серйозної шкоди.
- Комерційні засоби автоматизації коштують відчутно дорого, а наявні безкоштовні аналоги не завжди дозволяють ефективно вирішувати поставлені завдання.
- Засобів автоматизації дуже багато, що ускладнює проблему вибору того чи іншого засобу, ускладнює планування та визначення стратегії тестування, може спричинити додаткові тимчасові та фінансові витрати, а також необхідність навчання персоналу або найму відповідних фахівців.

Автоматизація тестування вимагає відчутних інвестицій та сильно підвищує проектні ризики, а тому існують спеціальні підходи щодо оцінки застосовності та ефективності автоматизованого тестування. В першу чергу слід врахувати:

• **Витрати часу на ручне виконання тест-кейсів і виконання цих тест-кейсів, але вже автоматизованих.** Чим відчутніша різниця, тим вигіднішою є автоматизація.

• **Кількість повторень виконання тих самих тест-кейсів**. Чим вона більша, тим більше часу ми зможемо заощадити за рахунок автоматизації.

• **Витрати часу на налагодження, оновлення та підтримку автоматизованих тест-кейсів**. Цей параметр найскладніше оцінити, і саме він становить найбільшу загрозу успіху автоматизації, тому тут для проведення оцінки слід залучати найдосвідченіших фахівців.

• **Наявність у команді відповідних фахівців та їхнє робоче завантаження**. Автоматизацією займаються найкваліфікованіші співробітники, які в цей час не можуть вирішувати інших завдань.

Список завдань, вирішити які допомагає автоматизація:

• Виконання тест-кейсів, непосильних людині.

• Розв'язання рутинних завдань.

• Прискорення тестування.

• Вивільнення людських ресурсів для інтелектуальної роботи.

• Збільшення тестового покриття.

• Поліпшення коду за рахунок збільшення тестового покриття та застосування спеціальних технік автоматизації.

ВИПАДКИ НАЙБІЛЬШОЇ ЗАСТОСОВНОСТІ АВТОМАТИЗАЦІЇ

Кейс / Виклик	Яку проблему вирішує автоматизація?
Регресійне тестування.	Необхідність виконання ручних тестів, кількість яких неухильно збільшується з кожною збіркою, але весь сенс яких полягає в тому, щоб перевірити той факт, що раніше відпрацьований функціонал продовжує коректно працювати.
Тестування інсталяції та налаштування тестового середовища.	Безліч повторюваних рутинних операцій для перевірки роботи інсталятора, розміщення файлів у файловій системі, вмісту конфігураційних файлів, реєстру тощо. Підготовка програми в заданому середовищі та із заданими налаштуваннями для базового тестування.
Тестування конфігурації та тестування сумісності	Запуск одних і тих же тест-кейсів на великому наборі вхідних даних, на різних платформах і в різних умовах. Класичний приклад: є конфігураційний файл зі 100 параметрами, кожен з яких може приймати 100 значень: є 100^{100} варіантів конфігураційного файлу, всі з яких потрібно перевірити.
Використання методів комбінаторного тестування	Генеруйте комбінації значень і запускайте тести кілька разів, використовуючи ці згенеровані комбінації як вхідні дані.
Модульне тестування	Перевірка коректності роботи фрагментів атомарного коду і елементарних взаємодій таких фрагментів коду - практично нездійсненне завдання для людини, за умови, що потрібно виконати тисячі таких перевірок і ніде не помилитися.
Інтеграційне тестування	Поглиблене тестування взаємодії компонентів в ситуації, коли людині практично нема чого спостерігати, так як всі процеси, що цікавлять і підлягають тестуванню, відбуваються на рівнях, більш глибоких, ніж призначений для користувача інтерфейс.

Кейс / Виклик	Яку проблему вирішує автоматизація?
Тестування безпеки	Необхідність перевірки прав доступу, паролів за замовчуванням, відкритих портів, вразливостей актуальних версій програмного забезпечення і т.д., тобто швидкого виконання дуже великої кількості перевірок, під час яких неможливо щось пропустити, забути або «неправильно зрозуміти».
Тестування продуктивності	Створення навантаження з інтенсивністю і точністю, недоступними людині. Швидкісний збір великого набору параметрів програми. Аналіз великих обсягів даних з журналів автоматизації.
Димовий тест для великих систем.	Виконання великої кількості тест-кейсів при отриманні кожної збірки досить просте для автоматизації тест-кейсів.
Програми (або їх частини) без графічного інтерфейсу.	Сканування консольних програм за великими наборами значень параметрів командного рядка (і їх комбінацій). Скануйте додатки та їх компоненти, які взагалі не призначені для взаємодії з людиною (веб-сервіси, сервери, бібліотеки тощо).
Тривалі, рутинні, виснажливі та/або потребують уваги операції.	Перевірки, що вимагають порівняння великих обсягів даних, високої точності обчислень, обробки великої кількості файлів, розміщених по всьому дереву каталогів, значно більшого часу виконання і т.д.
Перевірка «внутрішнього функціоналу» веб-додатків (посилання, доступність сторінок і т.д.)	Автоматизація вкрай рутинних дій (наприклад, перевірка всіх 30 000+ посилань, щоб переконатися, що всі вони ведуть на реальні сторінки). Автоматизація тут спрощена за рахунок стандартизації завдання — є багато готових рішень.
Стандартний, аналогічний функціонал для багатьох проектів.	У цьому випадку навіть висока складність первинної автоматизації окупиться за рахунок простоти багаторазового використання одержуваних рішень в різних проектах.
Технічні виклики.	Перевірка коректності ведення протоколу, роботи з базами даних, коректності пошуку, операцій з файлами, коректності форматів і вмісту документів, що формуються і т.д.

ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

	Підхід	Суть	Переваги	Недоліки
1	Приватні рішення.	Для вирішення кожного окремого завдання пишеться окрема програма.	Швидко, просто.	Немає послідовності, багато часу витрачається на підтримку. Практично неможливо використовувати повторно.
2	Тестування на основі даних (DDT).	Вхідні дані та очікувані результати виносяться з тест-кейса.	Один і той же тест може повторюватися багато разів з різними даними.	Логіка тест-кейсу все ще строго визначена всередині, і тому тест-кейс повинен бути переписаний, щоб змінити її.
3	Управління ключовими словами тестування (KDT).	З тест-кейса береться опис його поведінки.	Зосередьтеся на діях високого рівня. Дані та особливості поведінки зберігаються ззовні і можуть бути змінені без зміни коду тест-кейса.	Складність виконання низькорівневих операцій.
4	Використання фреймворків.	Конструктор, який дозволяє використовувати інші підходи.	Потужність і гнучкість.	Відносна складність (особливо в створенні каркаса).
5	Запис і відтворення.	Інструмент автоматизації фіксує дії тестувальника і може відтворювати їх, керуючи програмою, що тестується.	Простота, висока швидкість створення тест-кейсів.	Вкрай низька якість, лінійність, непідтримувані тест-кейси. Потрібне серйозне доопрацювання отриманого коду.
6	Тестування під керуванням поведінкою (BDT).	Розвиток ідей тестування під керуванням даними та ключовими словами. Відмінність — у концентрації на бізнес-сценаріях без виконання дрібних перевірок.	Висока зручність перевірки високорівневих сценаріїв користувача.	Такі тест-кейси пропускають велику кількість функціональних та нефункціональних дефектів, а тому мають бути доповнені класичними більш низькорівневими тест-кейсами.

ПРИВАТНІ РІШЕННЯ

Зручність приватних рішень полягає в тому, що їх можна реалізувати швидко, просто, «ось зараз». Але у них є й величезний недолік — це «кустарні рішення», якими може скористатися вся пара людей. І при появі нового завдання, навіть дуже схожого на раніше вирішене, швидше за все, доведеться все автоматизувати заново.

Переваги	Недоліки
• Швидко і легко реалізується. • Можливість використовувати будь-які доступні інструменти, якими вміє користуватися тестувальник. • Ефект від застосування миттєвий. • Можливість знаходити дуже ефективні рішення в тому випадку, якщо основні інструменти, які використовуються в проекті для автоматизації тестування, виявляться малопридатними для цієї конкретної задачі. • Здатність швидко створювати та оцінювати прототипи перед застосуванням більш важких рішень.	• Відсутність універсальності і, як наслідок, неможливість або надзвичайна складність повторного використання (адаптації для вирішення інших завдань). • Фрагментарність і неузгодженість рішень один з одним (різні підходи, технології, інструменти, принципи вирішення). • Вкрай висока складність розробки, підтримки та супроводу таких рішень (найчастіше, крім самого автора, ніхто не розуміє, що і навіщо було зроблено, і як це працює). • Це ознака «кустарного виробництва» і не вітається в промисловому розвитку програм.

ТЕСТУВАННЯ ПІД УПРАВЛІННЯМ ДАНИМИ (DDT)

До типових прикладів використання тестування під керуванням даними належать:

• Перевірка авторизації та прав доступу на великому наборі імен користувачів та паролів.

• Багаторазове заповнення полів форм різними даними та перевірка реакції програми.

• Виконання тест-кейсу на основі даних, отриманих за допомогою комбінаторних технік

Дані для аналізованого підходу до організації тест-кейсів можуть надходити з файлів, баз даних та інших зовнішніх джерел або навіть генеруватися в процесі виконання тест-кейс

Переваги А	**Недоліки**
• Усунення надмірності коду в тест-кейсах. • Зручне зберігання та зручний для читання формат даних. • Можливість довірити генерацію даних співробітнику, який не має навичок програмування. • Можливість використовувати один і той же набір даних для запуску різних тест-кейсів. • Можливість повторного використання набору даних для вирішення нових завдань. • Можливість використовувати один і той же набір даних в одному тест-кейсі, але реалізований під різні платформи	• Якщо логіка поведінки тест-кейса зміниться, його код все одно доведеться переписати. • При невдалому виборі формату представлення даних різко знижується їх зрозумілість для непідготовленого фахівця. • Необхідність використання технологій генерації даних. • Висока складність коду тест-кейсу у випадку складних різнорідних даних. • Ризик некоректної роботи тест-кейсів у випадку, коли кілька тест-кейсів працюють з одним і тим же набором даних, і він був змінений таким чином, що деякі тест-кейси не були розраховані. • Слабка здатність збирати дані в разі дефектів. • Якість тест-кейса залежить від професіоналізму співробітника, який реалізує код тест-кейса.

ТЕСТУВАННЯ ПІД УПРАВЛІННЯМ КЛЮЧОВИМИ СЛОВАМИ

Найяскравішим прикладом інструментального засобу автоматизації тестування, що ідеально наступного ідеології тестування під керуванням ключовими словами, є Selenium IDE.

Тестування під керуванням ключовими словами стало тим переломним моментом, починаючи з якого стало можливим залучення до автоматизації тестування нетехнічних спеціалістів.

Другою природною перевагою тестування під управлінням ключовими словами стала можливість використання різних інструментів одними й тими самими наборами команд та даних. Так, наприклад, ніщо не заважає нам взяти показані CSV-файли і написати нову логіку їхньої обробки не на PHP, а на C#, Java, Python або навіть із використанням спеціалізованих засобів автоматизації тестування

Переваги А	**Недоліки**
• Максимально виключення надмірності коду в тест-кейсах. • Можливість побудови міні-фреймворків, які вирішують широкий спектр завдань. • Підвищення рівня абстракції тест-кейсів та можливості адаптації їх для роботи з різними технічними рішеннями. • Зручне зберігання та зручний для читання формат даних та команд тест-кейсів. • Можливість делегувати опис логіки тест-кейсу співробітнику, який не має навичок програмування. • Багаторазове використання для вирішення нових завдань. • Розширюваність (можливість додавати нову поведінку тест-кейсу на основі вже реалізованої поведінки).	• Висока складність (а можливо, і тривалість) розробки. • Велика ймовірність помилок у коді тест-кейса. • Висока складність (або неможливість) виконання низькорівневих операцій, якщо фреймворк не підтримує відповідні команди. • Ефект від використання такого підходу не миттєвий (спочатку йде тривалий період розробки та налагодження самих тест-кейсів та допоміжного функціоналу). • Реалізація такого підходу вимагає наявності висококваліфікованих кадрів. • Необхідно навчити персонал мові ключових слів, які використовуються в тест-кейсах.

ВИКОРИСТАННЯ ФРЕЙМВОРКІВ

Фреймворків автоматизації тестування дуже багато, вони дуже різні, але їх поєднує кілька спільних рис:

• висока абстракція коду (немає необхідності описувати кожну елементарну дію) із збереженням можливості виконання низькорівневих дій;

• універсальність та переносимість використовуваних підходів;

• досить висока якість реалізації (для популярних фреймворків).

Як правило, кожен фреймворк спеціалізується на своєму вигляді тестування, рівні тестування, наборі технологій. Існують фреймворки для модульного тестування (наприклад, сімейство xUnit), тестування веб-додатків (наприклад, сімейство Selenium), тестування мобільних додатків, тестування продуктивності тощо.

Переваги	Недоліки
• Поширена. • Універсальність у своєму технологічному комплексі. • Хороша документація та велика спільнота спеціалістів для консультації. • Високий рівень абстракції. • Наявність великого набору готових рішень та описів релевантних найкращих практик використання того чи іншого фреймворку для вирішення тих чи інших завдань.	• Щоб вивчити фреймворк, потрібен час. • У разі написання власного фреймворку ви де-факто отримуєте новий проект з розробки програмного забезпечення. • Висока складність переходу на інший фреймворк. • Якщо фреймворк більше не підтримується, тест-кейси рано чи пізно доведеться переписувати за допомогою нового фреймворку. • Існує великий ризик вибору неправильного фреймворку.

ЗАПИС І ВІДТВОРЕННЯ (RECORD & PLAYBACK)

Технологія запису та відтворення стала актуальною з появою досить складних засобів автоматизації, що забезпечують глибоку взаємодію з додатком, що тестується, та операційною системою. Використання цієї технології, як правило, зводиться до наступних основних кроків:

1. Тестувальник вручну виконує тест-кейс, а засіб автоматизації записує всі його дії.
2. Результати запису подаються у вигляді коду високорівневою мовою програмування (у деяких засобах спеціально розробленою).
3. Тестувальник редагує отриманий код.
4. Готовий код автоматизованого тест-кейсу виконується для тестування в автоматизованому режимі.

Сама технологія при досить високій складності внутрішньої реалізації дуже проста у використанні і по самій своїй суті, тому часто застосовується для навчання фахівців-початківців з автоматизації тестування.

Переваги	Недоліки
• Надзвичайно простий в освоєнні (достатньо всього декількох хвилин, щоб почати користуватися цією технологією). • Швидко створюйте «скелетний тест-кейс», записуючи ключові дії з програмою, що тестується. • Автоматичний збір технічних даних про додаток, що тестується (ідентифікатори та локатори елементів, мітки, назви тощо). • Автоматизація рутинних дій (заповнення полів, перехід за посиланнями, кнопками тощо). • У деяких випадках його можуть використовувати тестувальники без навичок програмування.	• Лінійність тест-кейсів: не буде циклів, умов, викликів функцій та інших явищ, характерних для програмування та автоматизації. • Фіксація непотрібних дій (як помилкових випадкових дій тестувальника з тестованим додатком, так і (у багатьох випадках) перемикання на інші додатки і роботи з ними). • Так зване «хардкодування», тобто запис конкретних значень всередині коду тест-кейсу, що вимагатиме ручного доопрацювання для передачі тест-кейсу на технологію тестування на основі даних. • Незручні назви змінних, незручне оформлення коду тест-кейсу, відсутність коментарів та інші недоліки, що ускладнюють підтримку та супровід тест-кейсу в подальшому. • Низька надійність самих тест-кейсів через відсутність обробки винятків, перевірки стану тощо.

ТЕСТУВАННЯ ПІД УПРАВЛІННЯМ ПОВОДЖЕННЯМ

Розглянуті вище технології автоматизації максимально сфокусовані на технічних аспектах поведінки програми і мають загальний недолік: з їх допомогою складно перевіряти високорівневі сценарії користувача. Цей недолік покликане виправити тестування під керуванням поведінкою, в якому акцент робиться не на окремих технічних деталях, а на загальній працездатності програми при вирішенні типових завдань користувача.

Такий підхід не тільки спрощує виконання цілого класу перевірок, а й полегшує взаємодію між розробниками, тестувальниками, бізнес-аналітиками та замовником, т.д. в основі підходу лежить дуже проста формула «given-when-then»:

• Given («маючи, припускаючи, за умови») описує початкову ситуацію, в якій знаходиться користувач у контексті роботи з додатком.

• When («коли») описує набір дій користувача у цій ситуації.

• Th en («тоді») описує очікувану поведінку програми.

Такий принцип опису перевірок дозволяє навіть учасникам проекту, які не мають глибокої технічної підготовки, брати участь у розробці та аналізі тест-кейсів, а для фахівців з автоматизації спрощується створення коду автоматизованих тест-кейсів, т.я. така форма є стандартною, єдиною і при цьому надає достатньо інформації для написання високорівневих тест-кейсів. Існують спеціальні технічні рішення (наприклад, Behat, JBehave, NBehave, Cucumber), які спрощують реалізацію тестування під керуванням поведінкою.

Переваги	Недоліки
• Орієнтація на потреби кінцевих користувачів. • Спрощує співпрацю між різними фахівцями. • Простота та швидкість створення та аналізу тест-кейсів (що, в свою чергу, збільшує корисний ефект від автоматизації та знижує накладні витрати).	• Поведінкові тести високого рівня упускають багато деталей, а тому можуть не виявити деякі проблеми в додатку або не надати інформацію, необхідну для розуміння виявленої проблеми. • У деяких випадках інформації, наданої в поведінковому тест-кейсі, недостатньо для його безпосередньої автоматизації.

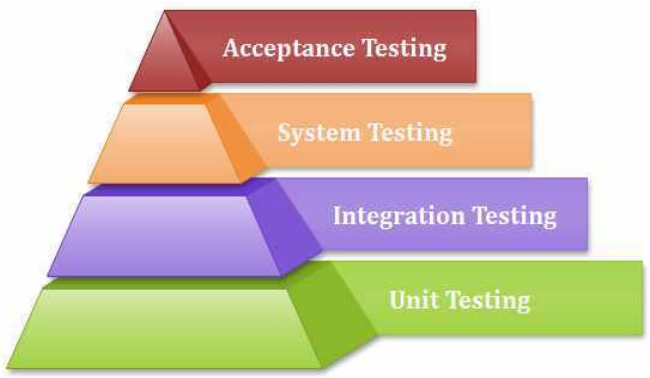
МОДУЛЬНЕ ТЕСТУВАННЯ

Модульне тестування визначається як метод забезпечення якості, при якому код програми розбивається на складові будівельні блоки – разом з відповідними даними кожного блоку або блоку, процесами використання і функціями – щоб гарантувати, що кожен блок працює належним чином.

Перш ніж будь-яке програмне забезпечення буде розроблено і випущено воно повинно пройти ряд тестів, щоб переконатися в його точності і функціональності. Тестування програмного забезпечення починається ще до завершення роботи програми. Таким чином, помилки і баги виявляються на ранній стадії, перш ніж вони загубляться в різних кодах.

Як правило, кожне програмне забезпечення проходить чотири етапи тестування. Перший - модульне тестування, за яким слід інтеграційне тестування, системне тестування і, нарешті, приймальне тестування. Модульне тестування формує фундамент, на якому будуються всі інші тести. Таким чином, точність і ретельність модульного тестування є важливими факторами, що впливають на те, наскільки добре можуть бути виконані інші тести, і на продуктивність програмного забезпечення в цілому.

В ієрархії рівнів тестування модульне тестування є першим рівнем тестування, яке виконується перед інтеграцією та іншими рівнями тестування. Він використовує модулі для процесу тестування, що зменшує залежність від очікування модульного тестування. Фреймворки, заглушки, драйвери та макетні об’єкти використовуються для допомоги в модульному тестуванні.



Як правило, **програмне** забезпечення проходить чотири рівні тестування: модульне тестування, інтеграційне тестування, системне тестування та приймальне тестування, але іноді через витрату часу тестувальники програмного забезпечення проводять мінімальне модульне тестування, але пропуск модульного тестування може призвести до більшої кількості дефектів під час інтеграційного тестування, системи Тестування та приймальне тестування або навіть під час бета-тестування, яке відбувається після завершення програмного забезпечення.

Мета модульного тестування

Метою модульного тестування є:

- Ізолювати розділ коду.
- Перевірити правильність коду.
- Протестувати кожну функцію і процедуру.
- Виправляти помилки на ранніх стадіях циклу розробки і економити витрати.
- Допомогти розробникам зрозуміти кодову базу і дати їм можливість швидко вносити зміни.
- Допомогти з повторним використанням коду.

Як працюють модульні тести

Модульне тестування зазвичай складається з трьох етапів: план, кейси та сценарії та сам модульний тест. На першому етапі модульний тест готується та переглядається. Наступним кроком є створення тестів і сценаріїв, а потім тестування коду.

Розробка, орієнтована на тестування, вимагає, щоб розробники спочатку написали невдалі модульні тести. Потім вони пишуть код і рефакторинг програми, поки тест не пройде. TDD зазвичай призводить до чіткої та передбачуваної кодової бази.

Кожен тестовий приклад перевіряється незалежно в ізольованому середовищі, щоб гарантувати відсутність залежностей у коді. Розробник програмного забезпечення повинен кодувати критерії для перевірки кожного тестового прикладу, а для звітування про будь-які невдалі тести можна використовувати структуру тестування. Розробники не повинні робити тест для кожного рядка коду, оскільки це може зайняти надто багато часу. Потім розробники повинні створити тести, зосереджені на коді, який може вплинути на поведінку програмного забезпечення, що розробляється.

Модульне тестування включає лише ті характеристики, які є життєво важливими для продуктивності тестованого блоку. Це заохочує розробників змінювати вихідний код, не хвилюючись про те, як такі зміни можуть вплинути на функціонування інших модулів або програми в цілому. Після того, як буде встановлено, що всі блоки програми працюють максимально ефективно та без помилок, більші компоненти програми можна оцінити за допомогою інтеграційного тестування . Модульні тести слід виконувати часто, їх можна робити вручну або автоматизовано.

Переваги та недоліки модульного тестування

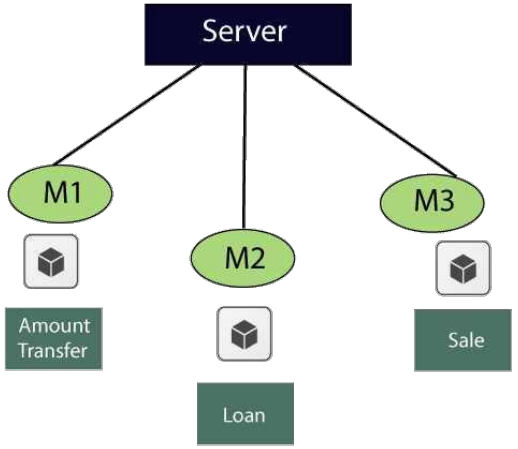
Переваги модульного тестування включають:

- Чим раніше виявлено проблему, тим менше складних помилок виникає.
- Витрати на раннє усунення проблеми можуть швидко перевищити витрати на її вирішення пізніше.
- Процеси налагодження спрощуються.
- Розробники можуть швидко вносити зміни в кодову базу.
- Розробники також можуть повторно використовувати код, переносячи його в нові проекти.

До недоліків можна віднести:

- Тести не виявлять кожну помилку.
- Модульні тести перевіряють лише набори даних і їх функціональність — вони не виявлять помилок під час інтеграції.
- Щоб перевірити один рядок коду, може знадобитися написати більше рядків тестового коду, що потенційно займе час.
- Модульне тестування може мати круту криву навчання , наприклад, необхідно навчитися використовувати певні автоматизовані програмні засоби.

КОНЦЕПЦІЇ МОДУЛЬНОГО ТЕСТУВАННЯ



Для **переказу суми** вимоги наступні:

1.	**Переказ суми**
1.1	З номера рахунку (FAN)→ текстове поле
1.1.1	FAN→ приймає лише 4 цифри
1.2	До номера рахунку (TAN)→ текстове поле
1.2.1	TAN→ Приймати лише 4 цифри
1.3	Сума→ Текстове поле
1.3.1	Сума → Приймайте максимум 4 цифри
1.4	Передача→ Кнопка
1.4.1	Передача → Увімкнено
1.5	Скасувати→ Кнопка
1.5.1	Скасувати→ Увімкнено

Нижче наведено деталі доступу до програми, які надає клієнт

- URL→ сторінка входу
- Ім'я користувача/пароль/ОК → домашня сторінка
- Щоб отримати модуль переказу суми, виконайте наведені нижче дії

Кредити → продаж → Переказ суми

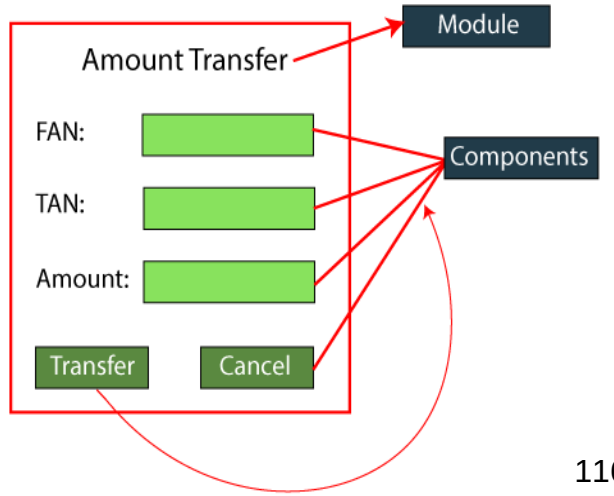
Виконуючи модульне тестування, ми повинні дотримуватися деяких правил, а саме:

- Щоб почати модульне тестування, у нас повинен бути принаймні один модуль.
- Випробуйте позитивні значення
- Перевірка на від'ємні значення
- Без надмірного тестування
- Припущення не потрібні

Коли ми відчуємо, що досягнуто максимального тестового покриття , ми припинимо тестування.

Тепер ми почнемо виконувати модульне тестування на різних компонентах, таких як

- **З номера рахунку (FAN)**
- **На номер рахунку (TAN)**
- **Сума**
- **Трансфер**
- **Скасувати**



Техніки модульного тестування:

Існує 3 типи методів модульного тестування. Вони є

Тестування чорного ящика: ця методика тестування використовується для модульних тестів для вхідних даних, інтерфейсу користувача та вихідних частин.

Тестування білого ящика: цей метод використовується для тестування функціональної поведінки системи шляхом надання вхідних даних і перевірки виходу функціональності, включаючи внутрішню структуру дизайну та код модулів.

Тестування сірого ящика: ця техніка використовується для виконання відповідних тестових випадків, методів тестування, тестових функцій і аналізу продуктивності коду для модулів.

Найкращі методи модульного тестування

- Випадки модульного тестування повинні бути незалежними. У разі будь-яких удосконалень або змін у вимогах це не повинно вплинути на випадки модульного тестування.
- Тестуйте лише один код за раз.
- Дотримуйтеся чітких і узгоджених умов імен для своїх модульних тестів
- У разі зміни коду в будь-якому модулі переконайтеся, що для модуля є відповідний тестовий приклад , і модуль пройшов тести перед зміною реалізації
- Помилки, виявлені під час модульного тестування, необхідно виправити перед переходом до наступного етапу в SDLC
- Прийміть підхід «тест як ваш код». Чим більше коду ви пишете без тестування, тим більше шляхів вам потрібно перевірити на наявність помилок.

ЯКІ НАЙКРАЩІ ІНСТРУМЕНТИ ДЛЯ МОДУЛЬНОГО ТЕСТУВАННЯ?

1. JUnit

Це загальнодоступна платформа модульного тестування, використовує програмування на Java. JUnit був вперше створений в 1997 році двома програмістами. З тих пір він отримував часті оновлення версій.

JUnit перевіряє дані перед їх додаванням в код. З допомогою JUnit ви можете підвищити ефективність роботи розробників, забезпечити узгодженість програмного коду і усунути втрати часу, що витрачається на налагодження. Крім надання тверджень для тестування модуля, JUnit також використовується для швидкого створення кодів, які можуть підвищити якість програмного коду. Junit здатний запускати тестові приклади протягом короткого періоду часу.

2. **NUnit**

NUnit, дуже схожий на JUnit, також є інструментом модульного тестування з відкритим вихідним кодом. Істотна відмінність між ним і JUnit полягає в тому , що NUnit був запрограмований для .NET Framework. Це поширений інструмент, спочатку написаний з використанням мови програмування C#. NUnit рішуче підтримує тести, засновані на даних. Він також підтримує ідею паралельного тестування з використанням консольного бігуна для завантаження і виконання численних тестів. NUnit використовує твердження як ефективний метод класу активів. Деякі з додатків, підтримуваних NUnit, включають silver light, .NET core, Xamarin mobile і ін.

3. DBUnit

DBUnit - це захоплююче додаток до процесу модульного тестування. Він був створений як розширення JUnit, але також сумісний з Ant. DBUnit особливо хороший для проектів з базою даних, оскільки однією з його основних функцій є приведення бази даних у відоме стан між періодами тестування. Виконання цього може мати велике значення для запобігання різних проблем, які можуть виникнути із-за збоїв в роботі бази даних після тестування. Без відновлення бази даних наступні результати тестування можуть бути невірними, в той час як фактичної помилки в коді немає.

Оновлені DBUnits в потоковому режимі можуть ефективно працювати з великими наборами даних. DBUnit також може виконувати імпорт і експорт даних в XML-набір даних з нього. Нарешті, розробники можуть використовувати DBUnits для перевірки того, що дані в базі даних відповідають необхідному значенням, встановленим для даної функції.

4. SimpleTest

SimpleTest - це інструмент модульного тестування, створений на мові програмування PHP і для нього. Це ресурс з відкритим вихідним кодом, який можна легко використовувати для тестування стандартних, але складних завдань PHP, таких як вхід на сайт. SimpleTest framework підтримує SSL, форми, аутентифікацію і проксі. Вбудований в саму просту структуру - це autorun.php файл, який використовується для отримання виконуваних тестових сценаріїв з тестових прикладів.

5. **ABAP Unit**

ABAP Unit - це інструмент тестування, що використовується для автоматичних і ручних процесів тестування. У нього є як безкоштовна, так і комерційна версія, як і у Embunit. Весь процес тестування від програмування до реалізації може бути виконаний в ABAP, що дозволяє користувачеві ретельно перевіряти одиницю коду на наявність помилок.

Модульне тестування за допомогою JUnit

JUnit Test - це звичайний Java-клас, в якому у формі методів визначено різні тестові приклади (тестовіприклади) для тестування програмного компонента. В рамках цих тестових прикладів кожна конкретна функція (зазвичай у формі методу або цілого класу) відповідного програмного компонента тестується шляхом її виконання (при необхідності з використанням фіктивних даних). Потім він перевірить, чи призводить до бажаного результату.

Тестовий приклад позначається анотацією @Test як такої і "затверджує" (образно кажучи), що, скажімо, функція A дає результат X при певних попередніх умовах, встановлених в тестовому прикладі. Якщо це твердження виявиться вірним, тестовий приклад пройдено, в іншому випадку він завершився невдачею. Це працює з так званим. Затвердження (просто "затвердження"), які JUnit надає в самих різних формах у вигляді статичних методів. Прикладом може служити метод assertEquals(...), який перевіряє, чи приймає вираз очікуване значення:

```
@Test
void testSumMethod(){
    Calculator calc = new Calculator();
    assertEquals(5, calc.sumOf(3, 2));
}
```

У цьому прикладі перевіряється метод sumOf(...) класу Calculator, який повинен складати два числа (3 і 2). Очікуваний результат такий 5.

Крім цих тестових прикладів, тести JUnit можуть також містити інші методи, функції (та інші властивості), яких також відзначені за допомогою анотацій для тіста. Основні анотації стисло перераховані тут (не повністю!):

@Test Позначає метод як тестовий приклад.

@BeforeAll Виділяє метод, який запускається один раз перед всіма тестовими прикладами (повинен виконуватися тільки один раз для кожного тестового класу). Це робиться, наприклад, для підготовки фіктивних даних або т. п.

@AfterAll Наприклад ***@BeforeAll***, тільки один раз після всіх тестовихприкладів. Служить, наприклад, для закриття ресурсів, що використовуються в тесті, або тому подібного.

@BeforeEach Виконується по одному разу перед кожним тестовим прикладом.

@AfterEach Виконується один раз за раз після кожного окремого тестового прикладу.

Припустимо, ми хочемо LetterCounter запрограмувати клас, який пропонує методи для підрахунку певних буквених класів в довільних рядках. Ці методи називаються countAllLetters(String text), coundVowels(String text) та countConsonants(String text), кожен з яких повертає значення з кількістю відповідних букв.

Чому б нам не спробувати Test-driven development і не написати наш тест перед програмуванням самого класу:

```java
class LetterCounterTest {

    private LetterCounter lc;
    private String text;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
            System.out.println("Starting LetterCounter Test
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
            System.out.println("Finished LetterCounter Test
    }

    @BeforeEach
    void setUp() throws Exception {
            lc = new LetterCounter();
            text = "Hello World! The End Is Near!";
    }

    @Test
    @DisplayName("Count all letters")
    void testCountAllLetters() {
            assertEquals(22, lc.countAllLetters(text));
    }

    @Test
    @DisplayName("Count vowels")
    void testCountVowels() {
            assertEquals(8, lc.countVowels(text));
    }

    @Test
    @DisplayName("Count consonants")
    void testCountConsonants() {
            assertEquals(14, lc.countConsonants(text));
    }
}
```

Цей тест містить кілька зайвих з (@BeforeAll, @AfterAlli @BeforeEach) зазначені методи, а також три тестових прикладу, кожен з (@Test) яких відзначений, і перевіряє три описаних методів.

Тепер ми створюємо наш клас LetterCounteri його методи, але для демонстраційних цілей ми ще не реалізуємо ніяких функцій, але повертаємо скрізь 0:

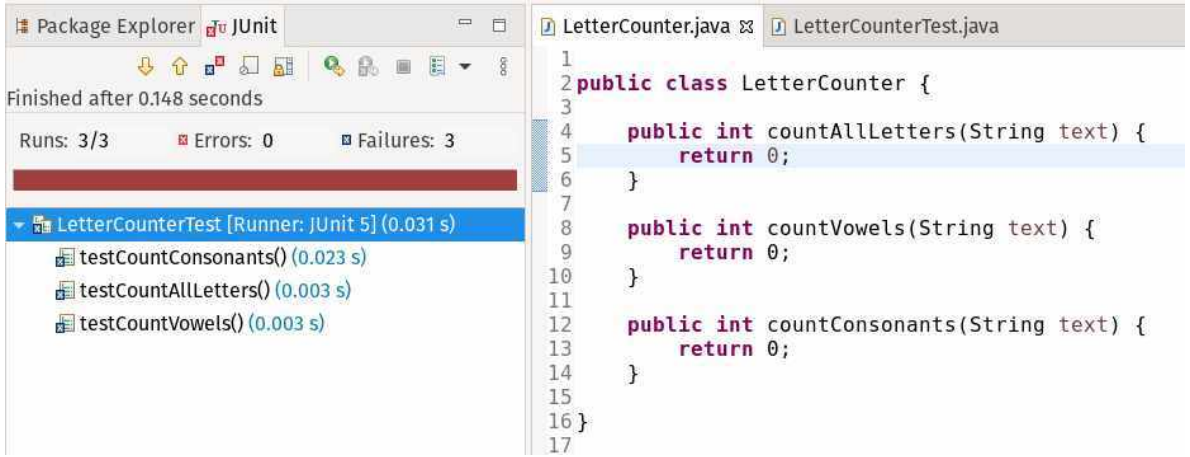
```java
public class LetterCounter {

        public int countAllLetters(String text) {
                return 0;
        }

        public int countVowels(String text) {
                return 0;
        }

        public int countConsonants(String text) {
                return 0;
        }
}
```

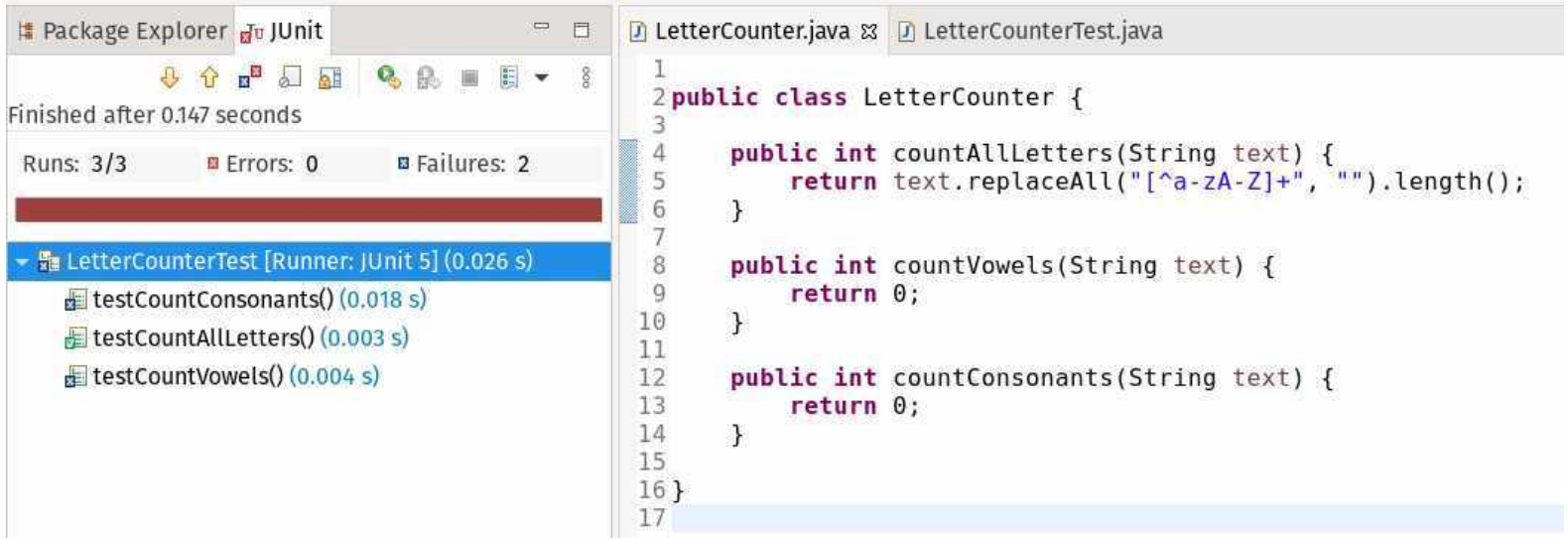
Тепер, якщо ми запустимо тест в нашому середовищі IDE Eclipse клацніть правою кнопкою миші проект → *Виконати* → *JUnit Test*), то, звичайно, всі тестові приклади завершаться помилкою:



Тепер давайте додамо робочий код в один з наших методів:

```
public int countAllLetters(String text) {
    return text.replaceAll("[^a-zA-Z]+", "").length();
}
```

Тепер ми бачимо, що один з тестових прикладів пройшов успішно. Але поки не *всі* з них успішні, тест в цілому вважається *невдалим* (*failed*).



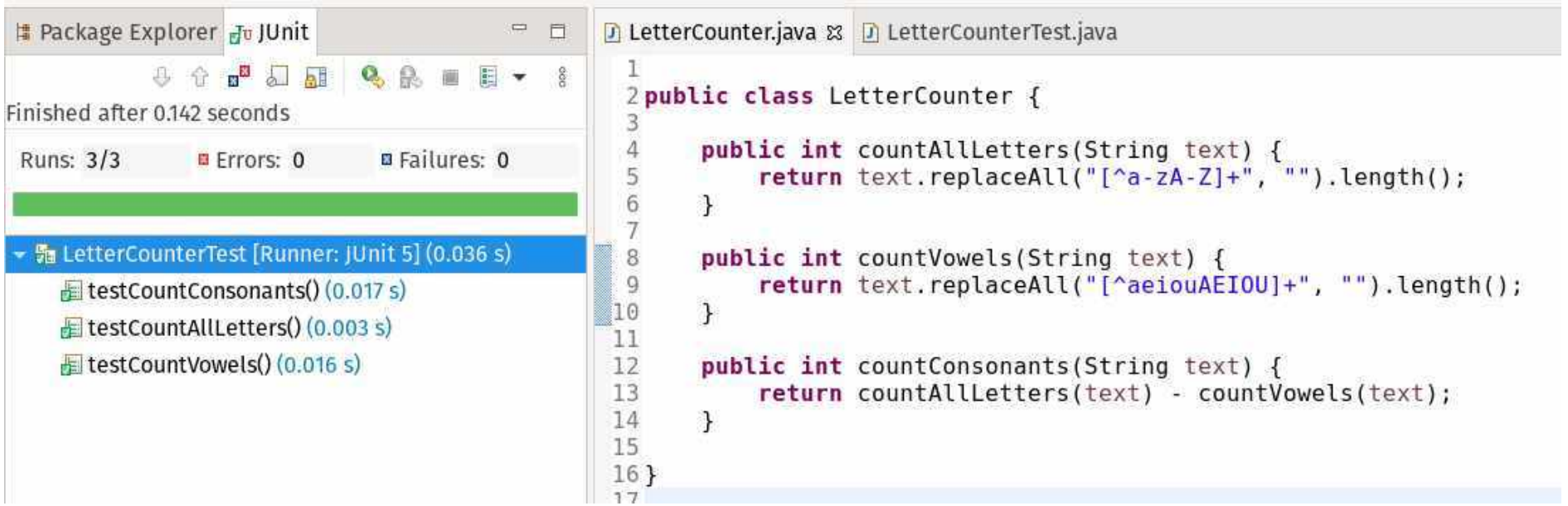
Тепер давайте завершимо наш клас:

```
public class LetterCounter {

	public int countAllLetters(String text) {
		return text.replaceAll("[^a-zA-Z]+", "").length();
	}

	public int countVowels(String text) {
		return text.replaceAll("[^aeiouAEIOU]+", "").length();
	}

	public int countConsonants(String text) {
		return countAllLetters(text) - countVowels(text);
	}
}
```



Selenium
Webdriver + java

Введення

- Webdriver - популярний інструмент для керування реальним браузером, який можна використовувати як для автоматизації тестування веб-застосунків, так і для виконання інших рутинних завдань, пов'язаних з роботою в Інтернеті.
- Крім того, Webdriver – проект з відкритим вихідним кодом, підтримує безліч мов програмування та має велику спільноту користувачів.
- Офіційний сайт Selenium проекту: https://www.selenium.dev

Переваги використання Java з Selenium:

- Багаті бібліотеки: Java пропонує багатий набір бібліотек та інструментів, що спрощує створення підтримуваного та масштабованого коду для автоматизованих тестів.
- Великі фреймворки: Існують численні фреймворки для тестування, засновані на Java (наприклад, TestNG, JUnit), які легко інтегруються з Selenium WebDriver.
- Зручність для новачків: Java має синтаксис, що читається, що робить його більш доброзичливим для новачків у програмуванні та автоматизації тестування.

Сумісність із інструментами розробки:

- Інтеграція із сучасними IDE: Java широко підтримується сучасними інтегрованими середовищами розробки (IDE), такими як IntelliJ IDEA, Eclipse та NetBeans. Це забезпечує зручність розробки та налагодження тестових сценаріїв.
- Підтримка Maven та Gradle: Використання інструментів автоматизації складання проектів, таких як Maven та Gradle, спрощує управління залежностями та складанням проекту.

Налаштування оточення для Selenium WebDriver та Java

- Установка Java Development Kit (JDK):
- Забезпечуємо правильне встановлення JDK для вашої операційної системи (Windows, Linux, macOS).
- Встановлення змінних середовища JAVA_HOME та додавання шляху до виконуваних файлів JDK до змінної PATH.
- Перевірка успішної установки за допомогою команди java-version у терміналі.

Налаштування оточення для Selenium WebDriver та Java(продовження)

- Встановлення та налаштування IntelliJ IDEA:
- Скачування та встановлення IntelliJ IDEA, інтегрованого середовища розробки для Java.
- Створення нового проекту в IntelliJ IDEA та вибір JDK для проекту.
- Додавання залежності для Selenium WebDriver у файлі проекту (зазвичай через Maven).

Налаштування оточення для Selenium WebDriver та Java(продовження)

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.141.59</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-chrome-driver -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-chrome-driver</artifactId>
        <version>3.141.59</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>RELEASE</version>
        <scope>test</scope>
    </dependency>


</dependencies>
```

Додані залежності для Selenium Webdriver, Chrome driver та junit
У файлі pom.xml.
Це спеціальний XML-файл, який завжди зберігається в базовій директорії проекту та називається pom. xml. Файл POM містить інформацію про проект та різні деталі конфігурації, які використовуються Maven для створення проекту.

Основні програмні компоненти

1. Браузер, роботу якого користувач хоче автоматизувати. Це реальний браузер певної версії, встановлений на певній ОС і має налаштування
2. Для керування браузером необхідний driver браузера. Driver насправді є веб-сервером, який запускає браузер та надсилає йому команди, а також закриває його. У кожного браузера свій driver. Пов'язано це з тим, що у кожного браузера свої чудові команди управління та реалізовані вони по-своєму.
3. Скрипт/тест, який містить набір команд певною мовою програмування для драйвера браузера. Такі скрипти використовують Selenium Webdriver bindings (готові бібліотеки), які доступні користувачам різними мовами.

Основні поняття

- Webdriver – найважливіша сутність, відповідальна за керування браузером. Основний хід скрипту/тесту будується саме навколо екземпляра цієї сутності.
- Webelement - друга важлива сутність, що є абстракцією над веб-елементом (кнопки, посилання, поля введення та ін.). Webelement інкапсулює методи взаємодії користувача з елементами та отримання їх поточного статусу.
- By – абстракція над локатором веб-елемента. Цей клас інкапсулює інформацію про селектор (наприклад, CSS), а також сам локатор елемента, тобто всю інформацію, необхідну для знаходження потрібного елемента на сторінці.

Основи Selenium WebDriver з Java

- Ініціалізація WebDriver: Створення об'єкта WebDriver для керування браузером.
- Відкриття веб-сторінки: Як завантажити сторінку та взаємодіяти з її елементами.
- Основи локаторів: Пошук елементів на веб-сторінці за допомогою різних стратегій.

Ініціалізація Webdriver

- Як використовувати метод get() для відкриття веб-сторінки.
- Обговорення відносних та абсолютних URL.

```
driver.get("https://www.yahoo.com");
```

Приклад відкриття веб-сторінки

```
WebDriver driver = new ChromeDriver();
```

Приклад ініціалізації для Chrome

Взаємодія з елементами

- Розбір методів взаємодії з елементами: кліки, введення тексту, очищення полів тощо.
- Обробка списків, що випадають, чекбоксів, радіокнопок.

```
elementById.click();    // Клкк по элементу
driver.manage().window().maximize();// відкрити тест у повноекранному режимі
elementByName.sendKeys("Текст для введення");   // Введення  тексту в полі вводу
elementByCss.clear();   // Очистка поля вводу
sleep( millis: 7000);//
driver.quit();// закриття браузера через 7 секунд
```

Типи локаторів

- Оскільки Webdriver - це інструмент для автоматизації веб-додатків, то більшість роботи з ним це робота з веб-елементами (WebElements). WebElements - ні що інше, як DOM об'єкти, що знаходяться на веб-сторінці. А для того, щоб здійснювати якісь дії над DOM об'єктами / веб-елементами необхідно їх точно визначити (знайти).
- Таким чином в Webdriver определяется нужный элемент. By - класс, содержащий статические методы для идентификации элементов:
- Локатори визначають, яким чином Selenium повинен знаходити елемент на веб-сторінці. Вибір правильного локатора залежить від структури HTML і конкретного завдання. Краще використовувати ID або інші унікальні атрибути, але у разі відсутності таких можливостей можна використовувати інші стратегії, такі як CSS Selector або XPath. Заходьте в світ локаторів і робіть вибір з розумінням та ефективно.

Локатори By.id та By.name

By.id

```
<div id="element_id">
    <p>some content</p>
</div>
```

- Пошук елемента

```
WebElement element = driver.findElement(By.id("element_id"));
```

By.name

```
<div name="element_name">
    <p>some content</p>
</div>
```

Пошук елемента:

```
WebElement element =driver.findElement(By.name("element_name"));
```

Локатори By.className та By.TagName

<img class="element_class">

- Пошук елемента:

WebElement element = driver.findElement(By.className("element_class"));

By.TagName

    ...

    ...

- Пошук елемента:

List<WebElement> elements = driver.findElements(By.tagName("a"));

Локатор By.XPath

```
<div class='main'>
    <p>text</p>
    <p>Another text</p>
</div>
```

Пошук елемента:

```
WebElement element =
driver.findElement(By.xpath("//div[@class='main']"));
```

Приклад автоматизованої реєстрації на сайті

```xml
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.141.59</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-chrome-driver -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-chrome-driver</artifactId>
        <version>3.141.59</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>RELEASE</version>
        <scope>test</scope>
    </dependency>


</dependencies>
```

Вміст файлу pom.xml, з усіма потрібними залежностями. Залежност скопійовані з https://mvnrepository.cor

Реєстрація

```
4 usages
public class Data {

2 usages
public String getUrl = "https://unsplash.com/login";
    2 usages
    public String firstName = "Иван ";
    2 usages
    public String surName = "Ивановичъ";
    2 usages
    public String emailAdress = "ivan.ivanovic4h20023@gmail.com";
    1 usage
    public String userName = "Caramelka228322";
    2 usages
    public String password = "Zasedf342fdsj2123Ad";


}
```

- Вміст класу Data, з якого ми переноситимемо дані, до тесту

```java
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
```

Під’єднані пакети, після додавання залежностей у pom.xml

```java
14 usages
WebDriver driver = new ChromeDriver(); // импорт chromeDriver
6 usages
Data getDate = new Data();
```

Підключення ChromeDriver та конструктора, за допомогою якого, підтягуватимуться дані з класу Data

```
private By crtanaccount = By.xpath( xpathExpression: "/html/body/div[2]/div/div/div/div/div[4]/a");//Код кнопки реєстрації
private By inputFirstName = By.xpath( xpathExpression: "//*[@id=\"user_first_name\"]");//Код поля введення імені
private By inputSurtName = By.xpath( xpathExpression: "//*[@id=\"user_last_name\"]");//Код поля введення прізвища
private By inputEmailAdress = By.xpath( xpathExpression: "//*[@id=\"user_email\"]");//Код поля введення пошти
private By inputuserName = By.xpath( xpathExpression: "//*[@id=\"user_username\"]");//Код поля введення нікнейму
private By inputPassword = By.xpath( xpathExpression: "//*[@id=\"user_password\"]");//Код поля введення паролю
private By btnContinue = By.xpath( xpathExpression: "//*[@id=\"new_user\"]/div[5]/input[2]");//Код поля введення кнопка зареєструватись
```

Заведення елементів сторінки через змінні, за допомогою шляху By.xpath
та коду елементу на сайті

```
driver.get(getDate.getUrl);
driver.manage().window().maximize();// відкрити тест у повноекранному режимі
driver.findElement(crtanaccount).click(); //клік на кнопку створення акаунту
driver.findElement(inputFirstName).sendKeys(getDate.firstName); // обробник подій sandKeys, надає можливість вивести  строкове значення
driver.findElement(inputSurtName).sendKeys(getDate.surName); // вставити прізвище
driver.findElement(inputEmailAdress).sendKeys(getDate.emailAdress); // вставити ел. адресу
driver.findElement(inputuserName).sendKeys(getDate.userName); //  вставити нікнейм
driver.findElement(inputPassword).sendKeys(getDate.password); // вставити пароль
driver.findElement(btnContinue).click(); // клік на кнопку продовжити
sleep( millis: 10000);//
driver.quit();// закриття браузера через 10 секунд
```

Написання тесту, знаходження елеменетів, та присвоєння їм значень змінних з класу Data.(обробник sendKeys)

Висновок

Сьогодні ми дізналися, як використовувати Selenium WebDriver та Java для автоматизації тестування веб-додатків. Ця потужна комбінація дозволяє ефективно створювати стабільні та повторювані тести.

Java виявляється ідеальним вибором завдяки своїй крос-платформенності, багатими бібліотеками та підтримкою спільноти. IntelliJ IDEA та Maven полегшують налаштування та управління проектами.