ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка автоматизованого тестування з використанням фреймворків та середовища IntelliJ IDEA»

на здобуття освітнього ступеня бакалавра (магістра)
зі спеціальності _____123 Комп'ютерної інженерія_____
(код, найменування спеціальності)
освітньо-професійної програми «Комп'ютерна інженерія»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ _____ДМИТРО ЗУБЕЦЬ_____
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач(ка) вищої освіти гр. КІД-41_
_____ДМИТРО ЗУБЕЦЬ_____
Ім'я, ПРІЗВИЩЕ
Керівник: ____ЯРОСЛАВ ТОРОШАНКО___
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти – «Бакалавр»

Спеціальність – 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач кафедрою

Лащевська Наталія Олександрівна

«____» _____ 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Зубцю Дмитру Ігоровичу

1. Тема кваліфікаційної роботи: Розробка автоматизованого тестування з використанням фреймворків та середовища IntelliJ IDEA

Керівник кваліфікаційної роботи Ярослав Торошанко, доцент кафедри (Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання) затверджені наказом Державного університету інформаційно-комунікаційних технологій від «___» _____ 2025 р. № ___

2. Строк подання кваліфікаційної роботи «___» _____ 2025 р.

3. Вихідні дані до кваліфікаційної роботи: сучасні технології автоматизованого тестування (Selenium, JUnit, Maven); середовище розробки IntelliJ IDEA; методи організації процесу тестування; підходи до побудови тестових сценаріїв; вимоги до ефективного впровадження автоматизованих рішень; приклади тестування веб-застосунків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1. Теоретичні основи автоматизованого тестування

4.2. Підходи до автоматизованого тестування

4.3. Інструменти та середовища для автоматизованого тестування

4.4. Методи підвищення ефективності автоматизованого тестування

4.5. Сучасні тенденції розвитку автоматизованого тестування

4.6. Дослідження та реалізація автоматизованого тестування

4.7. Висновки

5. Перелік ілюстративного матеріалу

5.1. Мета, об’єкт, предмет

5.2. Актуальність

5.3. Порівняльна таблиця інструментів

5.4. Ключові поняття та роль автоматизації

5.5. Інструменти та середовище тестування

5.6. Порівняльна таблиця браузерів для автоматизованого тестування з Selenium

5.7. Реалізація прикладного тесту

5.8. Переваги обраного підходу

5.9. Складнощі під час реалізації

5.10. Отримані результати

5.11. Висновки

6. Дата видачі завдання «___» _____ 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір джерел інформації	01.03-06.03.2025	Виконано
2	Написання вступної частини	07.03-08.03.2025	Виконано
3	Написання теоретичної частини (розділи 1-3)	09.03-27.04.2025	Виконано
4	Виконання практичної частини	28.04-08.05.2025	Виконано
5	Написання висновків та оформлення списку використаної літератури	09.05-11.05.2025	Виконано
6	Підготовка демонстраційного матеріалу	12.05-14.05.2025	Виконано
7	Попередній захист роботи	29.05.2025	Виконано
8	Пред’явлення роботи в деканат		

Студент _____ Зубець Д.І.

Керівник роботи _____ Торошанко Я.І.

РЕФЕРАТ

Текстова частина бакалаврської роботи: 43 с., 5 табл., 6 рис., 23 джерела.

РОЗРОБКА АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ ФРЕЙМВОРКІВ ТА СЕРЕДОВИЩА INTELLIJ IDEA

Об'єкт дослідження – процес автоматизованого тестування програмного забезпечення з використанням фреймворків у середовищі розробки IntelliJ IDEA.

Предмет дослідження – методи, інструменти та технології автоматизованого тестування, зокрема використання фреймворків (JUnit, TestNG, Selenium тощо) у середовищі IntelliJ IDEA для забезпечення якості та стабільності програмного забезпечення.

Мета і завдання дослідження – розробити ефективну систему автоматизованого тестування програмного забезпечення з використанням фреймворків та середовища розробки IntelliJ IDEA, що дозволить підвищити якість коду, знизити кількість помилок та прискорити процес розробки.

Методи дослідження:

1. Аналіз літературних джерел та наукових публікацій – вивчення теоретичних основ автоматизованого тестування, порівняння підходів та інструментів.

2. Порівняльний аналіз – дослідження різних фреймворків для тестування (JUnit, TestNG, Selenium тощо) з метою виявлення їх переваг та недоліків у різних сценаріях тестування.

3. Експериментальний метод – розробка та запуск тестових сценаріїв у середовищі IntelliJ IDEA, аналіз результатів тестування.

4. Моделювання – створення тестових сценаріїв для різних рівнів тестування (юнит-тестування, інтеграційне тестування, енд-ту-енд тестування).

5. Статистичний аналіз – оцінка результатів тестування, визначення покриття коду тестами, аналіз ефективності та виявлення слабких місць у програмному забезпеченні.

6. Метод спостереження – моніторинг виконання тестів та аналіз поведінки системи під час автоматизованого тестування.

У даній бакалаврській роботі розглянуто теоретичні засади, методологічні підходи та сучасні інструменти для реалізації автоматизованого тестування програмного забезпечення. Особливу увагу приділено аналізу фреймворків і середовищ, зокрема використанню IntelliJ IDEA у поєднанні з Java, Selenium, JUnit та Maven.

У процесі дослідження було класифіковано основні види тестування, розглянуто принципи побудови тестової стратегії, методи підвищення ефективності тестування та сучасні технологічні тенденції, зокрема використання штучного інтелекту, хмарних рішень, а також підходів Continuous Integration / Continuous Delivery (CI/CD).

У практичній частині здійснено налаштування середовища розробки для тестування, а також реалізовано приклад автоматизованого тесту вебсайту з використанням описаних інструментів. Окремий розділ присвячено аспектам забезпечення якості та безпеки в процесі тестування.

Ключові слова: тестування програмного забезпечення, автоматизація тестування, IntelliJ IDEA, Selenium, JUnit, Maven, фреймворки, CI/CD, ефективність, безпека.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8

ВСТУП 13

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ, МЕТОДИ ТА ІНСТРУМЕНТИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ 14

1.1 Сутність і значення тестування програмного забезпечення 14

1.2 Автоматизація тестування: переваги, недоліки та огляд фреймворків 21

1.3 Підходи до організації автоматизованого тестування 26

1.4 Стратегія тестування і роль IntelliJ IDEA 28

РОЗДІЛ 2. ІНСТРУМЕНТИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ТА КРИТЕРІЇ ЇХ ВИБОРУ 33

2.1 Класифікація інструментів автоматизованого тестування 33

2.2 Фреймворки тестування та їх взаємодія з середовищем розробки 37

2.3 Інтеграція автоматизованого тестування у CI/CD процес 39

2.4 Метрики, звітність і візуалізація результатів тестування 41

РОЗДІЛ 3. СУЧАСНІ ТЕНДЕНЦІЇ ТА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ В АВТОМАТИЗОВАНОМУ ТЕСТУВАННІ 42

3.1 Актуальні тренди: штучний інтелект, no-code та хмарні рішення 42

3.2 Тестування в сучасних архітектурах і хмарних середовищах 43

3.3 Забезпечення якості ПЗ: вимоги, стандарти, надійність 46

3.4 Безпека тестування та інтеграція з QA-процесами 48

РОЗДІЛ 4. НАЛАШТУВАННЯ СЕРЕДОВИЩА ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ 51
4. Підготовка середовища розробки 51
4.1 Встановлення IntelliJ IDEA 51
4.2 Встановлення Java Development Kit (JDK) 52
4.3 Створення нового Maven-проєкту 52
4.4 Налаштування файлу pom.xml 53
4.5 Налаштування WebDriver для браузера Chrome 53
4.6. Опис технологій і бібліотек 54
4.6.1 JUnit 54
4.6.2 Selenium 55
4.6.3 Maven 56
4.6.4 Як ці технології взаємодіють 57
4.7 Написання простого тесту для сайту YouTube з використанням Selenium та JUnit 58
ВИСНОВКИ 58
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 60
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (ПРЕЗЕНТАЦІЯ) 63

ПЕРЕЛІК УМОВНИХ ПОЗНЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення.

Allure – це інструмент побудови звітів автотестів, що спрощує їх аналіз.

API (application programming interface), або інтерфейс програмування застосунків – це спосіб взаємодії комп’ютерних програм між собою.

Git – це розподілена система керування версіями файлів та спільної роботи.

IDE (integrated development environment), або інтегроване середовище розробки – це комплексне програмне рішення для розробки програмного забезпечення.

IntelliJ IDEA – це комерційне інтегроване середовище розробки для різних мов програмування (Java, Python, Scala, PHP та ін.) від компанії JetBrains.

JUnit – це фреймворк для написання та запуску тестів на мові програмування Java.

Maven – це інструмент для побудови та управління проєктами Java.

Mockito – це бібліотека для створення та управління об’єктами-замінниками (mocks) в Java.

REST Assured – це бібліотека для автоматизації тестування API.

Selenide – це бібліотека для автоматизації тестування веб-додатків у браузері, створена на основі Selenium WebDriver.

Selenium WebDriver – це гнучкий інструмент для автоматизованого тестування веб-проєктів на базі набору бібліотек для різних мов програмування, таких як Java, .Net(C#), Python, Ruby, PHP, Perl, JavaScript.

TestNG – це інструмент для автоматизованого тестування, заснований на Java, який надає більш гнучкі та потужні можливості порівняно з JUnit.

UI (user interface), або інтерфейс користувача – це засіб зручної взаємодії користувача з інформаційною системою.

ВСТУП

У сучасному світі якість програмного забезпечення є ключовим фактором його конкурентоспроможності та успішності. Через зростаючий обсяг і складність програмних продуктів традиційне ручне тестування вже не задовольняє потреб сучасної розробки. Воно є трудомістким, повільним і схильним до помилок, особливо в умовах динамічних змін у програмному коді. У зв’язку з цим усе більшого поширення набуває автоматизоване тестування — процес створення та виконання тестів із використанням спеціалізованих інструментів та фреймворків.

Автоматизоване тестування дозволяє підвищити ефективність перевірки якості програмного забезпечення, забезпечити швидкий зворотний зв’язок для розробників і мінімізувати ризики, пов’язані з людським фактором. Застосування сучасних фреймворків, таких як JUnit, Selenium, TestNG та інших, у поєднанні з середовищем IntelliJ IDEA значно спрощує реалізацію автоматизованих тестів і їх інтеграцію в процес розробки.

Актуальність теми дипломної роботи полягає у потребі забезпечення високої якості ПЗ шляхом впровадження ефективних рішень з автоматизованого тестування, що особливо важливо для компаній, які працюють за гнучкими методологіями розробки (Agile, DevOps). Використання потужного інтегрованого середовища розробки IntelliJ IDEA дозволяє створювати, налаштовувати та запускати тести у зручний спосіб, що робить його одним із найбільш популярних інструментів серед тестувальників та розробників.

Мета даної бакалаврської роботи полягає в аналізі методів, інструментів і технологій автоматизованого тестування та реалізації прикладного проєкту, що

демонструє процес налаштування середовища і написання тестів за допомогою IntelliJ IDEA, JUnit та Selenium.

Для досягнення мети в роботі було поставлено наступні завдання:

- дослідити теоретичні основи тестування програмного забезпечення;
- проаналізувати види, підходи та стратегії автоматизованого тестування;
- розглянути сучасні фреймворки та інструменти тестування;
- вивчити особливості налаштування середовища IntelliJ IDEA для автоматизації тестів;
- реалізувати приклад автоматизованого тестування з використанням актуальних технологій.

Об’єктом дослідження є процес автоматизованого тестування програмного забезпечення.

Предметом дослідження є інструменти, фреймворки та підходи до реалізації автоматизованого тестування у середовищі IntelliJ IDEA.

Практична цінність роботи полягає у створенні прикладного проєкту з автоматизованого тестування, що може бути використаний як основа для реальних розробницьких процесів або для навчання майбутніх спеціалістів у галузі інженерії програмного забезпечення.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ, МЕТОДИ ТА ІНСТРУМЕНТИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

1.1 Сутність і значення тестування програмного забезпечення

Розробка програмного забезпечення — це складний і багатоступеневий процес, у якому важливе місце займає контроль якості створюваних продуктів. Тестування програмного забезпечення — це одна з ключових фаз життєвого циклу програмного забезпечення, спрямована на виявлення помилок, дефектів або невідповідностей між фактичними і очікуваними результатами виконання програми. Метою тестування є забезпечення якості програмного забезпечення, а також зниження ризиків, пов'язаних з можливими помилками у продуктивному середовищі.

Тестування допомагає відповісти на кілька важливих питань: чи правильно працює система, чи відповідає вона специфікації, чи зручна вона для користувача, наскільки вона надійна та безпечна. Крім того, тестування дозволяє підвищити рівень довіри замовників і кінцевих користувачів до програмного продукту. Важливо розуміти, що тестування не гарантує повної відсутності дефектів у програмі, однак воно значно зменшує їх кількість і ризик впливу на бізнес-процеси.

У сучасному ІТ-середовищі вимоги до якості програмного забезпечення зростають, як і складність самих систем. Це зумовлює необхідність ефективних стратегій тестування, що охоплюють різні аспекти функціонування програмного продукту. У цьому контексті тестування виступає не як фінальна стадія розробки, а як невід’ємна складова кожного етапу створення ПЗ — від аналізу вимог до підтримки вже готової системи.

Слід також зазначити, що значення тестування особливо зростає у критичних системах, де помилки можуть призвести до серйозних наслідків — фінансових, юридичних або навіть загроз для життя. Саме тому у великих проєктах тестування виділяється в окрему фазу або навіть цілий процес, що інтегрується у загальну стратегію розробки.

У процесі тестування програмного забезпечення використовується ціла система класифікацій, яка допомагає краще зрозуміти його охоплення, цілі та підходи. Найбільш поширеною є класифікація за видами, методами та рівнями.

Щодо видів тестування, найпоширенішими є:

- Функціональне тестування — перевірка, чи виконує система необхідні функції відповідно до специфікації.
- Нефункціональне тестування — охоплює такі характеристики, як продуктивність, безпека, зручність використання, сумісність тощо.
- Регіональне тестування — перевірка коректності роботи після внесення змін у програму.
- Інтеграційне тестування — тестування взаємодії між окремими модулями або компонентами системи.
- Системне тестування — повна перевірка інтегрованої системи на відповідність вимогам.
- Тестування приймання — фінальна перевірка перед передачею продукту замовнику або в експлуатацію.

За методами тестування розрізняють:

- Статичне тестування — здійснюється без запуску програми (наприклад, аналіз коду, рецензії, перевірка документації).

- Динамічне тестування — виконується шляхом запуску програми та аналізу її поведінки.
- Також використовуються такі методи:
- Тестування "чорного ящика" — коли тестувальник не має доступу до коду і перевіряє лише зовнішню поведінку програми.
- Тестування "білого ящика" — тестувальник має доступ до внутрішньої логіки системи.
- Тестування "сірого ящика" — комбінований підхід, який враховує як внутрішню структуру, так і зовнішню поведінку.

Щодо рівнів тестування, зазвичай виокремлюють:

- Модульне тестування — тестування окремих функцій або методів.
- Інтеграційне тестування — тестування зв’язків між модулями.
- Системне тестування — тестування повністю зібраної системи.
- Тестування приймання (acceptance testing) — фінальна перевірка для замовника.

Ця багаторівнева система дозволяє будувати комплексний підхід до забезпечення якості, де кожен рівень і метод виконує свою функцію і доповнює інші.

Якість програмного забезпечення (ПЗ) — це сукупність властивостей, що визначають його здатність задовольняти встановлені та приховані потреби користувача. У процесі розробки програмного забезпечення тестування виконує ключову функцію — виявлення дефектів**,** підтвердження відповідності вимогам та оцінка працездатності системи в реальному чи симульованому середовищі.

Уявімо розробку банківської системи. Без повноцінного тестування існує ризик, що система некоректно округлятиме грошові операції або дозволятиме подвійне списання коштів. Тестування критично важливе для:

- моделювання сценаріїв перевантаження серверів;
- перевірки багаторівневої авторизації;
- оцінки реакції на несанкціоновані запити.

Упродовж десятиліть підходи до тестування змінювалися відповідно до трансформацій у розробці програмного забезпечення. Від ручного тестування окремих модулів до комплексної автоматизації в рамках DevOps — тестування стало ключовим елементом розробки, а не лише фінальним етапом перевірки.

Умовно історію тестування можна поділити на кілька етапів:

Таблиця 1.1 – Історичний розвиток тестування

Період	**Характеристика підходу**
1950–1970-ті	Тестування як відл

	агодження — ручне, без методологій
1980–1990-ті	З’являються структури як V-модель, принципи юніт-тестування

2000-ні	Початок автоматизації, поява TDD, BDD, інструментів типу JUnit
2010+	Інтеграція в CI/CD, масове вико

	ристання автоматизованих фреймворків
2020+	Тестування з використанням ШІ, автоматичне генерування тестів

Тестування — це не лише технічна, а й когнітивна діяльність, що вимагає аналітичного мислення, уваги до деталей, здатності до абстрагування та комунікації. Тестувальники часто працюють із неповною інформацією, шукаючи помилки в роботі інших і водночас забезпечуючи партнерську співпрацю з розробниками.

Професійне тестування потребує інверсного мислення: замість того, щоб підтверджувати правильність, тестувальник шукає, що може піти не так. Це вимагає:

- Скептицизму: не приймати код за правильний лише тому, що він працює;
- Допитливості: дослідження системи у «нестандартних» сценаріях;
- Уважності: виявлення неочевидних помилок;
- Креативності: створення граничних умов та негативних сценаріїв.

Тестувальник повинен не лише виявляти помилки, а й ефективно комунікувати про них. Конструктивний зворотний зв’язок розробнику знижує ймовірність конфлікту та прискорює виправлення. Наприклад, замість фрази «Кнопка не працює» — краще: «Після натискання кнопки “Підтвердити” не відбувається очікуваний перехід, як зазначено в специфікації п.3.2».

Психологічні бар’єри в тестуванні включають:

- Ефект упередження підтвердження — схильність шукати докази правильності, а не помилок;
- Втома від повторення — ігнорування помилок через монотонність;
- Ілюзія контролю — надмірна впевненість у якості коду, особливо після кількох «успішних» тестів.

Для боротьби з цим застосовують:

- чергування завдань;
- парне тестування;
- ротацію модулів між тестувальниками.

Тестувальник має відповідальність не лише за технічні аспекти, а й за етичну чесність: не приховувати помилки, не фальсифікувати результати, не спрощувати сценарії для «позитивного» звіту. Від цього залежить не лише якість продукту, а й репутація компанії.

1.2 Автоматизація тестування: переваги, недоліки та огляд фреймворків

Автоматизація тестування — це використання спеціальних інструментів та фреймворків для автоматичного запуску тестів, збору результатів та формування звітів. Такий підхід має як переваги, так і обмеження.

Серед основних переваг автоматизованого тестування можна виділити:

- Швидкість виконання: автоматизовані тести виконуються значно швидше, ніж ручні, особливо при багаторазовому повторенні.
- Повторюваність: можна багаторазово запускати один і той самий набір тестів без змін.
- Масштабованість: легко запускати тести на великій кількості конфігурацій, платформ, середовищ.
- Зниження людського фактору: автоматизація виключає вплив втоми або неуважності тестувальника.
- Інтеграція з CI/CD: автотести можна інтегрувати у процес безперервної інтеграції і доставки програмного забезпечення.

- Швидке виявлення дефектів: регулярне виконання тестів дозволяє оперативно виявити помилки після зміни коду.
- Відсутність людської помилки: Автоматизовані тести, виконувані програмою, виключають можливість людських помилок, які можуть виникнути при ручному тестуванні, таких як пропуск тестів чи помилкова інтерпретація результатів.
- Ефективність тестування на різних етапах розробки: Автоматизоване тестування особливо корисне при інтеграції нових функцій, що дозволяє одразу перевіряти сумісність із іншими частинами програмного продукту на ранніх етапах розробки.

Однак автоматизація має і певні недоліки:

- Висока початкова вартість: необхідно інвестувати час і ресурси в налаштування середовища, написання скриптів.
- Складність підтримки: автотести можуть ставати застарілими або нестабільними після змін у програмі.
- Обмежене охоплення: не всі типи тестів доцільно автоматизувати (наприклад, юзабіліті-тести).
- Витрати на навчання: потрібна спеціалізована підготовка персоналу.
- Фальшиве відчуття безпеки: якщо автотести не охоплюють усі сценарії, вони можуть дати ілюзію повного контролю над якістю.
- Необхідність постійної актуалізації: З часом тести можуть застарівати через зміни в програмному продукті, що вимагає регулярного оновлення тестових скриптів для підтримки їхньої актуальності.
- Ризик помилок у тестах: Автоматизовані тести можуть бути неефективними, якщо сценарії тестування не були належним чином сформульовані або якщо вони не відображають реальні користувацькі

сценарії. Це може призвести до неефективного виявлення дефектів або пропуску важливих багів.

Вибір між ручним та автоматизованим тестуванням часто базується на характеристиках проєкту, доступних ресурсах та цільових пріоритетах.

Автоматизоване тестування базується на використанні різноманітних фреймворків та інструментів, кожен з яких має свою специфіку, сферу застосування та особливості налаштування. Ось короткий огляд основних з них:

Selenium — один із найпопулярніших інструментів для автоматизованого тестування веб-додатків. Підтримує різні мови програмування, зокрема Java, Python, C#, JavaScript. Складається з кількох компонентів: WebDriver, IDE, Grid. Selenium дозволяє писати кросбраузерні автотести з мінімальними змінами у коді.

JUnit — класичний фреймворк для модульного тестування на мові Java. Підтримується у більшості середовищ розробки, зокрема IntelliJ IDEA. Дає змогу створювати тести з анотаціями, групувати їх, перевіряти винятки, валідувати результати.

TestNG — розширений фреймворк тестування, подібний до JUnit, але з більшою кількістю функцій, таких як параметризація тестів, залежності, паралельне виконання. Чудово підходить для великих і складних проєктів.

Cucumber — фреймворк для BDD (Behaviour-Driven Development), що дозволяє писати тести у вигляді зрозумілих людині сценаріїв (мовою Gherkin). Сприяє кращій співпраці між технічною та нетехнічною частинами команди.

Mockito — бібліотека для створення mock-об’єктів (заглушок), які замінюють залежності під час тестування. Використовується разом з JUnit або TestNG. Дозволяє тестувати окремі компоненти без реальних підключень до бази даних, мережі тощо.

Allure — інструмент для побудови красивих, детальних звітів після виконання автотестів. Підтримується багатьма фреймворками, включно з JUnit, TestNG, Selenide, Cucumber. Дає змогу візуалізувати результати, переглядати логи, скріншоти, діаграми покриття тестами.

Selenide — інструмент для автоматизованого тестування веб-додатків, який побудований на основі Selenium, але забезпечує простоту використання завдяки вбудованим методам для роботи з веб-елементами. Selenide активно використовується для тестування на основі Java, оскільки його API є дуже зручним для швидкого написання тестів.

Jest — фреймворк для тестування JavaScript-додатків, особливо популярний серед розробників фронтенду, оскільки він надає простий у використанні інтерфейс для тестування компонентів, функцій, а також інтеграційних тестів.

Katalon Studio — інструмент, який об'єднує функціональність для автоматизації тестів веб-додатків і мобільних додатків. Katalon Studio пропонує простий інтерфейс для створення та запуску тестів, а також має вбудовані звіти, що полегшує аналіз результатів.

Завдяки широкому вибору фреймворків і гнучким можливостям налаштування, автоматизоване тестування сьогодні стало невід'ємною частиною сучасної розробки програмного забезпечення.

1.3 Підходи до організації автоматизованого тестування

Тестування програмного забезпечення охоплює різноманітні техніки і підходи, які дозволяють забезпечити якість продукту на різних етапах його життєвого циклу. Одним із ключових етапів проектування автоматизованих тестів є правильна класифікація типів тестування. Вони поділяються за кількома критеріями: рівнем абстракції, метою, способом виконання, доступністю вихідного коду тощо.

За рівнем тестування найчастіше виділяють наступні види:

- Модульне (Unit testing): охоплює окремі функції чи класи;
- Інтеграційне (Integration testing): перевіряє взаємодію між модулями;
- Системне (System testing): перевіряє всю систему як єдине ціле;
- Приймальне (Acceptance testing): проводиться на основі вимог замовника.
- За доступом до коду тестування поділяється на:
- White-box (тестування з урахуванням внутрішньої структури коду);
- Black-box (тестування без знань про реалізацію);
- Grey-box (комбінація двох попередніх підходів).
- Також важливо враховувати класифікацію за способом виконання:
- Ручне тестування (Manual Testing);
- Автоматизоване тестування (Automated Testing).

Окремі типи тестування спрямовані на виявлення певних типів дефектів, як-от:

- Регресійне тестування (Regression Testing);
- Тестування навантаження (Load Testing);
- Стрес-тестування (Stress Testing);

– Тестування безпеки (Security Testing);
– Кросбраузерне тестування (Cross-browser Testing).

Таблиця 1.2 – Класифікація видів тестування

Критерій	**Основні види**
Рівень	Модульне, інтеграційне, системне
Доступ до коду	White-box, Black-box, Grey-box
Мета	Регресійне, навантаження, приймальне
Спосіб виконання	Ручне, автоматизоване

Розуміння класифікації видів тестування дає змогу більш точно обрати фреймворк або набір інструментів, відповідно до вимог конкретного проєкту.

Тестова стратегія — це високорівневий документ або план, який визначає загальний підхід до тестування програмного забезпечення. Вона є основою для створення тест-планів і описує мету, обсяг, критерії якості, типи тестування, інструменти, методології та відповідальність команди.

Ключові компоненти тестової стратегії включають:

– Мету тестування (що саме потрібно перевірити);
– Сфери застосування (які функції покриватимуться тестами);
– Підходи до тестування (ручне чи автоматизоване);
– Інструменти (наприклад, Selenium, JUnit, TestNG);
– Ризики (можливі проблеми при тестуванні).

Нижче представлено схему типового змісту тестової стратегії:

1. Вступ

2. Мета тестування
3. Обсяг і обмеження
4. Критерії початку/завершення тестування
5. Технічний стек
6. Вимоги до ресурсів
7. План дій у разі дефектів

Правильно сформована стратегія дозволяє команді тестування працювати злагоджено, скорочуючи витрати на виправлення помилок на пізніх етапах розробки.

1.4 Стратегія тестування і роль IntelliJ IDEA

Вибір правильного підходу до автоматизації залежить від специфіки продукту, цілей тестування, бюджету та наявних ресурсів. Найпоширенішими моделями є:

- Підхід із використанням фреймворків: побудова тестів на основі шаблонів, структур і бібліотек (наприклад, JUnit, TestNG, Selenium).
- Keyword-driven testing: створення тестів на основі ключових слів, що дозволяє навіть нефахівцям у програмуванні брати участь у створенні тестів.
- Data-driven testing: тести виконуються на різних наборах вхідних даних, що дозволяє забезпечити ширше покриття.
- BDD (Behavior Driven Development): фокус на поведінку системи, використання зрозумілої доменної мови (наприклад, Cucumber, Gherkin).

Підхід із використанням фреймворків

Це найбільш технічно орієнтований підхід, при якому автоматизовані тести будуються на основі готових бібліотек, інструментів і структур, таких як JUnit, TestNG, Selenium, Appium, RestAssured тощо.

Переваги:

- Гнучкість і контроль над логікою тестування;
- Широкі можливості інтеграції з іншими інструментами (логування, звіти, CI/CD);
- Підтримка багатьох мов програмування.

Недоліки:

- Потребує знань програмування;
- Вища вартість входу для новачків.

Застосування: рекомендований для команд із досвідом розробки, складних проєктів та інтеграції в CI/CD-процеси.

Keyword-driven testing

У цьому підході тестові сценарії створюються з використанням ключових слів, які описують дії (наприклад, "натисни", "введи", "перевір") у табличному вигляді (наприклад, Excel, CSV або спеціальні GUI-редактори). Використовуються такі інструменти, як Robot Framework, TestComplete, Ranorex.

Переваги:

- Можливість залучення не технічних спеціалістів до створення тестів;
- Покращення читабельності тестів;
- Швидка побудова сценаріїв.

Недоліки:

– Складність у масштабуванні та підтримці великої кількості ключових слів;
– Обмежена гнучкість порівняно з чистим кодом.

Застосування: підходить для команд з учасниками без глибоких технічних знань або коли важлива швидкість розробки тестів.

Data-driven testing

Цей підхід базується на розділенні тестової логіки та даних, де тестовий сценарій виконується багаторазово з різними наборами вхідних значень, зазвичай з таблиць або баз даних. Підтримується фреймворками типу JUnit (Parameterized Tests), TestNG (DataProvider), Selenium + Excel/CSV/XML.

Переваги:

– Ефективне покриття багатьох комбінацій даних без дублювання тест-коду;
– Полегшує зміну вхідних даних без редагування логіки тестів.

Недоліки:

– Ускладнення відладки при помилках у великих масивах даних;
– Може потребувати налаштування додаткових засобів для зчитування даних.

Застосування: ідеально підходить для валідації форм, калькуляторів, таблиць, баз даних, API.

BDD (Behavior-Driven Development)

Behavior-Driven Development — підхід, що орієнтований на опис поведінки системи мовою, зрозумілою бізнесу та замовнику. В основі лежить створення

сценаріїв у вигляді Given-When-Then, що пишуться мовою Gherkin. Інструменти: Cucumber, JBehave, SpecFlow, Behave (Python).

Переваги:

- Покращена комунікація між технічною та бізнес-командою;
- Прозорість і зрозумілість вимог;
- Можливість перевикористання сценаріїв як документації.

Недоліки:

- Вимагає більше часу на підтримку читабельності сценаріїв;
- Потребує тісної співпраці між усіма учасниками процесу.

Застосування: підходить для проєктів із сильною участю бізнес-аналітиків, де важлива деталізація очікуваної поведінки.

Таблиця 1.3 – Порівняльна таблиця підходів

Підхід	**Переваги**	**Недоліки**
Framework-based	Гнучкість, модульність	Потреба у програмуванні
Keyword-driven	Доступність для нетехнічних користувачів	Складність масштабування
Data-driven	Широке покриття тестами	Підтримка великого обсягу даних
BDD	Комунікація між командами, читабельність	Більше налаштувань, складність

Підхід обирається залежно від поставлених цілей та ресурсоємності проекту.

IntelliJ IDEA — одна з найпопулярніших IDE серед Java-розробників, яка забезпечує ефективну інтеграцію з численними фреймворками та інструментами для автоматизованого тестування. Серед основних можливостей IDE, які підтримують тестування, можна виділити:

- Підтримка JUnit і TestNG «з коробки»;
- Гнучке створення конфігурацій запуску для різних типів тестів;
- Вбудовані засоби перегляду результатів тестів і діагностики;
- Інтеграція з системами CI/CD (наприклад, Jenkins);
- Інтеграція з системами контролю версій Git, що дозволяє відстежувати зміни у тестах.

Завдяки багатому набору плагінів (наприклад, плагіни для Cucumber або Allure) IntelliJ IDEA дозволяє швидко розгорнути повноцінне середовище для тестування.

Схематично, робота в IntelliJ IDEA з тестування виглядає так:

1. Створення нового тестового класу через шаблон;
2. Додавання залежностей до build.gradle або pom.xml;
3. Виконання тестів за допомогою вбудованого тест-раннера;
4. Перегляд результатів та логів тестування в окремій панелі.

Завдяки потужним засобам навігації та рефакторингу IntelliJ IDEA зменшує час на налаштування та оновлення автоматизованих тестів, що позитивно впливає на швидкість зворотного зв'язку та якість продукту.

РОЗДІЛ 2. ІНСТРУМЕНТИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ТА КРИТЕРІЇ ЇХ ВИБОРУ

2.1 Класифікація інструментів автоматизованого тестування

Сучасне автоматизоване тестування програмного забезпечення ґрунтується на широкому спектрі інструментів, які підтримують різні типи тестів, мови програмування, інтеграцію з CI/CD системами та звітність. Кожен із таких інструментів має власні переваги та обмеження, що зумовлює потребу в їх ретельному доборі залежно від контексту проєкту.

Інструменти тестування класифікуються за кількома критеріями:

1. За рівнем тестування:
   - інструменти модульного тестування (наприклад, JUnit, TestNG);
   - інструменти інтеграційного тестування (наприклад, Spring Test, Arquillian);
   - інструменти системного тестування (наприклад, Selenium, Cypress);
   - інструменти навантажувального тестування (наприклад, JMeter, Gatling).
2. За типом інтерфейсу:
   - інструменти тестування GUI (Selenium, TestComplete);
   - інструменти тестування API (Postman, Rest Assured).
3. За способом взаємодії:
   - інструменти з графічним інтерфейсом (Katalon Studio, TestComplete);

- інструменти з консольним або скриптовим керуванням (JUnit, PyTest, Robot Framework).

4. За ліцензією:

- з відкритим кодом (JUnit, Selenium, Appium);
- комерційні (TestComplete, Ranorex, UFT).

Таблиця 2.1 – Коротке порівняння популярних інструментів

Інструмент	**Тип тестів**	**Мова**	**Ліцензія**	**Інтерфейс**
JUnit	Юніт-тести	Java	Open source	Консольний
Selenium	UI тестування	Java, Python	Open source	Скриптовий
TestNG	Юніт, інтеграційні	Java	Open source	Консольний
Postman	API тестування	REST	Free	Графічний
Katalon Studio	UI, API, мобільні	Скрипти Groovy	Free/Pro	Графічний
JMeter	Навантажувальні	Java	Open source	Графічний

Завдяки великому вибору інструментів, команди можуть формувати власний стек технологій відповідно до потреб і цілей тестування. Варто пам’ятати, що використання надмірної кількості інструментів може ускладнити процеси підтримки та навчання.

IntelliJ IDEA є популярним середовищем розробки, яке широко застосовується не лише для написання коду, а й для автоматизації тестування. Його гнучкість, інтеграція з різними фреймворками та системами керування проєктами забезпечує ефективну роботу QA-команд.

Переваги використання IntelliJ IDEA в тестуванні:

- автоматичне розпізнавання тестових класів та методів;
- інтеграція з JUnit, TestNG, Cucumber, Selenium, REST Assured тощо;
- зручний механізм запуску тестів і перегляду результатів;
- можливість створення шаблонів тестів;
- підтримка плагінів для перевірки покриття коду (наприклад, JaCoCo);
- інтеграція з Git, Maven, Gradle та CI/CD платформами.



Рисунок 2.1 – Загальна архітектура процесу тестування у середовищі IntelliJ IDEA

Тестовий клас у IntelliJ автоматично позначається спеціальним значком і має контекстне меню для запуску. Результати тестів відображаються у вікні Run з детальними логами, що спрощує аналіз помилок.

IntelliJ IDEA також підтримує конфігурації запуску, що дає змогу запускати лише певну групу тестів або створювати послідовності тестування. Це особливо корисно при роботі з великими наборами сценаріїв.

2.2 Фреймворки тестування та їх взаємодія з середовищем розробки

Фреймворки тестування є основою, на якій будуються інструменти тестування. Вони надають структуру, шаблони, готові функції та механізми обробки результатів. Основна функція фреймворків — забезпечити зручне середовище для написання та виконання тестів.

Розглянемо деякі відомі фреймворки:

- JUnit: класичний фреймворк для модульного тестування Java-додатків. Має анотації для визначення тестів, механізм перевірки результатів (assert) та підтримку інтеграції з CI.
- TestNG: розширена альтернатива JUnit, підтримує паралельне тестування, параметризацію, залежності між тестами.
- Cucumber: підтримує BDD (Behavior Driven Development), дозволяє писати тести мовою Gherkin у форматі Given-When-Then.
- Selenium WebDriver: фреймворк для UI-тестування, забезпечує взаємодію з браузером на рівні елементів DOM.
- REST Assured: фреймворк для тестування REST API, базується на Java, дозволяє будувати запити й перевіряти відповіді за допомогою зрозумілого DSL.

Вибір фреймворку залежить від типу проєкту, мови програмування, типу тестів та вимог до масштабованості.

Таблиця 2.2 – Порівняльна характеристика фреймворків

Фреймворк	**Призначення**	**Мова**	**Особливості**
JUnit	Юніт-тестування	Java	Простота, підтримка IDE
TestNG	Розширене юніт-тестування	Java	Паралельність, конфігурації
Cucumber	BDD тестування	Java	Формат Given-When-Then
Selenium WebDriver	UI тестування браузера	Java та інші	Кросбраузерність, WebDriver API
REST Assured	API тестування	Java	Fluent API для HTTP-запитів

Вибір інструментів автоматизації тестування є критично важливим етапом, який впливає на ефективність роботи всієї команди. Критерії вибору зазвичай включають:

- сумісність з мовою проєкту;
- простота інтеграції з існуючим середовищем;
- активність спільноти та підтримка;
- ліцензійні обмеження;
- можливість масштабування;
- вимоги до навантаження та типів тестування.

У практиці часто зустрічаються такі ситуації:

1. Якщо проєкт має REST API, але немає UI — доцільно використовувати REST Assured або Postman.
2. Якщо важлива поведінка користувача — застосовують Selenium або Cypress.
3. Для покриття бізнес-сценаріїв мовою, зрозумілою замовнику — Cucumber.
4. У багатокомпонентних системах — використовують комбінацію кількох фреймворків.

На практиці іноді створюються гібридні фреймворки, що поєднують переваги кількох інструментів. Наприклад, UI-тестування із Selenium + TestNG + Allure для звітності.

2.3 Інтеграція автоматизованого тестування у CI/CD процес

Ефективність автоматизованого тестування залежить від багатьох чинників: якості написаних тестів, покриття коду, правильності вибору фреймворку, організації процесу тощо. Проте існують базові принципи, дотримання яких дозволяє суттєво покращити результати автоматизованого тестування:

- Раннє тестування. Виявлення дефектів на ранніх етапах знижує вартість їх усунення та прискорює цикл розробки. Саме тому важливо впроваджувати практики тестування з перших етапів життєвого циклу ПЗ.
- Повторне використання коду. Написання універсальних бібліотек, повторне використання конфігурацій та сценаріїв економить час і знижує ймовірність помилок.
- Ізоляція тестів. Тест має перевіряти окрему одиницю функціоналу, бути незалежним від інших тестів і не залежати від стану зовнішніх систем.

- Регулярний аналіз результатів. Ефективне тестування передбачає не лише запуск тестів, а й аналіз результатів, логів та звітів для виявлення слабких місць у коді.
- Автоматизація рутинних задач. Автоматизоване тестування саме по собі зменшує кількість ручної роботи, однак додаткова автоматизація підготовки даних, очищення середовища або запуску звітів теж суттєво підвищує ефективність.

CI/CD — це практика безперервної інтеграції та доставки, яка дозволяє прискорити релізи, уникати помилок та автоматизувати більшість процесів розробки і тестування. Основні переваги впровадження CI/CD:

- автоматичний запуск тестів після кожного коміту в репозиторій;
- швидке виявлення та усунення помилок;
- стабільність середовища тестування та розгортання;
- зменшення людського фактору при складанні та розгортанні ПЗ.

Серед популярних інструментів CI/CD, що добре інтегруються з IntelliJ IDEA та Java-проєктами:

- Jenkins
- GitLab CI/CD
- GitHub Actions
- TeamCity
- CircleCI

Використання CI/CD дозволяє значно підвищити ефективність автоматизованого тестування шляхом швидкого зворотного зв'язку та

стабільного процесу збірки. Цей підхід також сприяє кращому управлінню якістю продукту.

2.4 Метрики, звітність і візуалізація результатів тестування

Інформативні звіти та графічна візуалізація результатів допомагають краще аналізувати ефективність тестування, знаходити "вузькі місця" у коді та приймати зважені рішення щодо якості релізу. Інструменти звітності, які зазвичай використовуються:

- Allure — створює зручні інтерактивні звіти з графіками, таблицями та деталізацією кожного тесту.
- ExtentReports — дозволяє створювати HTML-звіти з високим рівнем кастомізації.
- Surefire Reports — стандартне рішення для звітності в Maven-проєктах.

Перевагою інтеграції таких інструментів є можливість швидко знаходити неуспішні тести, причини їх падіння, відсоток покриття, а також бачити динаміку змін від коміту до коміту.

Для того щоб оцінити успішність автоматизованого тестування, необхідно використовувати метрики, що відображають якість процесу. Серед найбільш поширених:

- Test Coverage (покриття коду тестами) — показує, яку частину коду перевіряють тести.
- Defect Density — співвідношення кількості помилок до розміру коду.
- Pass Rate / Fail Rate — відсоток успішних / неуспішних тестів за певний період.

- Mean Time to Detect (MTTD) — середній час до виявлення дефекту.
- Mean Time to Repair (MTTR) — середній час на виправлення дефекту.

Ці метрики дозволяють об'єктивно оцінювати якість автоматизованого тестування, а також приймати рішення щодо доцільності подальшої автоматизації або рефакторингу існуючих тестів.

РОЗДІЛ 3. СУЧАСНІ ТЕНДЕНЦІЇ ТА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ В АВТОМАТИЗОВАНОМУ ТЕСТУВАННІ

3.1 Актуальні тренди: штучний інтелект, no-code та хмарні рішення

Інтеграція технологій штучного інтелекту (ШІ) та машинного навчання (ML) у сферу автоматизованого тестування є однією з найперспективніших тенденцій. Ці підходи дозволяють вирішити низку проблем, пов'язаних із рутинністю процесів, динамікою інтерфейсів та великою кількістю даних.

Основні переваги використання ШІ/ML у тестуванні:

- автоматичне генерування тест-кейсів на основі історії використання додатка;
- прогнозування найбільш ризикованих областей коду для цілеспрямованого тестування;
- самонавчальні системи, які з часом покращують якість тестування;
- зменшення кількості помилкових спрацювань у тестах завдяки адаптивним алгоритмам.

Приклади застосування:

- Testim та Mabl використовують ML для автоматичної генерації та оновлення тестів;
- Applitools застосовує комп'ютерний зір для візуального тестування інтерфейсів;
- Functionize пропонує інтелектуальне планування тестів на основі змін у коді.

У відповідь на зростаючу потребу у швидкому створенні тестів навіть без глибоких технічних знань, з'явилися no-code та low-code платформи. Вони дозволяють створювати автоматизовані тести за допомогою графічного інтерфейсу без написання коду або з мінімальним програмуванням.

Переваги таких платформ:

- швидке навчання для нетехнічних спеціалістів;
- скорочення часу на розробку та підтримку тестів;
- зниження навантаження на розробників та автоматизаторів.

Приклади платформ:

- Katalon Studio
- TestProject
- Leapwork
- Ranorex Studio

Хоча такі інструменти мають свої обмеження, вони ефективні у проектах із простими сценаріями або для створення прототипів автоматизації.

3.2 Тестування в сучасних архітектурах і хмарних середовищах

Сучасні додатки часто базуються на мікросервісній архітектурі, що вимагає специфічного підходу до автоматизованого тестування. Кожен сервіс працює незалежно, може масштабуватись окремо й оновлюватись без зупинки всієї системи.

Особливості тестування мікросервісів:

- необхідність тестування API кожного сервісу окремо;
- важливість контрактного тестування (наприклад, з використанням Pact);
- потреба в симуляції взаємодії між сервісами (інструменти типу WireMock);
- широке використання контейнеризації (Docker) для створення ізольованого середовища.

Мікросервіси підвищують гнучкість, але ускладнюють як ручне, так і автоматизоване тестування, що потребує нових стратегій та інструментів.

Тестування у хмарному середовищі стає дедалі популярнішим завдяки своїй масштабованості, доступності та можливості швидкого розгортання різноманітних конфігурацій. Хмарні сервіси забезпечують:

- віддалене виконання тестів на різних ОС і браузерах;
- динамічне масштабування ресурсів;
- централізоване зберігання звітів та логів;
- швидке розгортання середовищ для тестування.

Приклади хмарних платформ:

- BrowserStack
- Sauce Labs

- LambdaTest
- AWS Device Farm

Тестування мікросервісів:

- Тестування безперервної інтеграції та доставки (CI/CD): У мікросервісних архітектурах важливо забезпечити ефективне тестування в рамках безперервної інтеграції та доставки. Для цього використовуються інструменти, такі як Jenkins, GitLab CI/CD, які автоматизують процеси тестування при кожному коміті в репозиторій, забезпечуючи швидке виявлення помилок.
- Тестування без стану (Stateless Testing): Мікросервіси часто працюють без збереження стану між запитами, що вимагає від тестувальників особливих підходів при перевірці функціональності кожного окремого сервісу та його взаємодії з іншими.
- Тестування оркестрації контейнерів: Важливою частиною тестування мікросервісів є перевірка правильної роботи оркестрації контейнерів, що здійснюється за допомогою Kubernetes або Docker Compose. Тести мають перевіряти не лише функціональність сервісів, але й їхню взаємодію в середовищах, де сервіси розгортаються та масштабуються динамічно.

Тестування в хмарних середовищах:

- Інфраструктура як код (IaC): Тестування в хмарних середовищах активно інтегрується з підходом Інфраструктури як код (IaC), де конфігурації для тестових середовищ автоматизуються та зберігаються у вигляді коду (наприклад, за допомогою Terraform, AWS CloudFormation). Це дозволяє швидко створювати та знищувати інфраструктуру, що значно спрощує тестування на різних середовищах.

- Хмарні сервіси для мобільного тестування: Для мобільних додатків хмарні платформи пропонують можливості для тестування на фізичних пристроях, що забезпечує вищу точність результатів порівняно з емуляторами або симуляторами. Такі сервіси, як AWS Device Farm, дозволяють тестувати додатки на реальних пристроях з різними версіями операційних систем і налаштуваннями.
- Тестування serverless архітектур: З розвитком serverless архітектур (наприклад, AWS Lambda) необхідно розробити специфічні стратегії тестування, оскільки обчислювальні ресурси створюються та знищуються за потребою. Це вимагає тестування функціональності на рівні кожної окремої функції, а не на рівні цілих серверів або середовищ.

Застосування хмарних сервісів особливо ефективне для тестування веб- і мобільних застосунків, де важливо перевірити поведінку на багатьох пристроях та конфігураціях.

3.3 Забезпечення якості ПЗ: вимоги, стандарти, надійність

Якість програмного забезпечення (ПЗ) — це сукупність характеристик, що визначають здатність ПЗ задовольняти встановлені та передбачувані потреби користувачів. Поняття якості охоплює не лише коректну роботу продукту, але й зручність використання, масштабованість, надійність, безпеку тощо.

Основним міжнародним стандартом, що визначає характеристики якості ПЗ, є ISO/IEC 25010:2011. У ньому виділено вісім основних атрибутів якості:

- Функціональна придатність (Functional suitability) — наскільки програмне забезпечення виконує необхідні функції;

- Продуктивність (Performance efficiency) — ефективність використання ресурсів;
- Сумісність (Compatibility) — здатність ПЗ взаємодіяти з іншими системами;
- Юзабіліті (Usability) — зручність у використанні;
- Надійність (Reliability) — здатність функціонувати стабільно;
- Безпека (Security) — захист від несанкціонованого доступу;
- Супровідність (Maintainability) — легкість модифікації;
- Портованість (Portability) — можливість переносу на інші платформи.

Автоматизоване тестування значно полегшує досягнення цих характеристик. Завдяки регулярному запуску тестів, програмне забезпечення швидше проходить перевірку на відповідність вимогам якості. Наприклад, регресійне тестування дозволяє виявити порушення функціональності після внесення змін, а навантажувальне тестування — оцінити продуктивність системи.

Автоматизоване тестування повинно не лише виявляти помилки, але й забезпечувати безпеку тестованої системи та середовища. Безпека тестування стосується як самої тестової інфраструктури, так і збереження конфіденційних даних у процесі тестування.

Ключові аспекти забезпечення безпеки:

- Ізоляція тестового середовища — тестування повинно проводитися у відокремленому середовищі, щоб уникнути ризику пошкодження продуктивної системи;

- Знеособлення тестових даних — конфіденційна інформація має бути замінена або знеособлена під час використання у тестах;
- Використання захищених каналів — при взаємодії з API або іншими сервісами мають використовуватися захищені протоколи (наприклад, HTTPS);
- Контроль доступу — лише уповноважені особи повинні мати доступ до тестових сценаріїв, логів і результатів;
- Автоматичне очищення після тестування — очищення тимчасових даних, cookie, сесій тощо.

Тестувальники також повинні враховувати загрози безпеці, пов’язані з тестуванням вебзастосунків, наприклад:

- SQL-ін’єкції;
- міжсайтовий скриптинг (XSS);
- підробка міжсайтових запитів (CSRF);
- відкриті порти та неконфігуровані сервіси.

Усі ці загрози можна частково відслідковувати за допомогою спеціалізованих автоматизованих інструментів безпеки (наприклад, OWASP ZAP, Burp Suite тощо).

3.4 Безпека тестування та інтеграція з QA-процесами

Забезпечення якості (Quality Assurance, QA) — це системний процес, який включає планування, моніторинг і поліпшення всіх аспектів розробки ПЗ. Автоматизоване тестування є одним із ключових інструментів у цьому процесі.

Основні підходи до інтеграції автоматизованого тестування в QA-процеси:

- Shift-left testing — раннє включення тестування в життєвий цикл розробки;
- Test-Driven Development (TDD) — написання тестів перед створенням коду;
- Behavior-Driven Development (BDD) — створення тестів на основі очікуваної поведінки системи;
- CI/CD — автоматичне виконання тестів при кожному коміті або перед релізом;
- Feedback loop — оперативне отримання результатів тестування для швидкого реагування.

Така інтеграція дозволяє команді швидко виявляти проблеми, знижувати вартість помилок і забезпечувати стабільність продукту. Тестувальники та розробники повинні тісно співпрацювати, що дозволяє зробити тестування частиною процесу розробки, а не окремим етапом.

Надійність автоматизованого тестування означає стабільне та передбачуване виконання тестів з однаковими результатами за однакових умов. Основні принципи забезпечення надійності включають:

- Уникнення "фальшивих" результатів — тести не повинні давати хибнопозитивні (успішний тест при наявності помилки) або хибнонегативні результати. Це досягається коректним описом очікуваної поведінки та контролем зовнішніх залежностей (наприклад, мережі, часу відповіді серверів);
- Повторюваність тестів — тести мають бути детермінованими, тобто завжди давати однаковий результат при однаковому наборі вхідних даних;

- Контроль версій тестів — усі тести мають зберігатися у системах контролю версій (Git), що дозволяє відслідковувати зміни та повертатися до попередніх версій у разі помилок;
- Регулярне оновлення тестів — у разі зміни вимог до системи тести мають оперативно адаптуватися до нової логіки;
- Аналіз результатів тестування — системи автоматичної візуалізації (наприклад, Allure Reports) допомагають швидко виявити нестабільні тести та часті помилки.

Також важливо регулярно проводити ревізії тестового набору, видаляти застарілі тести та впроваджувати best practices тестування, наприклад, Page Object Model, розділення тестів за категоріями та мінімізацію залежностей.

Автоматизоване тестування повинно враховувати аспекти безпеки на всіх етапах тестування, починаючи від розробки тестових сценаріїв до їх виконання в тестовому середовищі. Під час інтеграції автоматизованого тестування в процеси забезпечення якості важливо мінімізувати ризики, пов'язані з безпекою, зокрема з конфіденційністю даних і доступом до тестового середовища.

Основні принципи забезпечення безпеки тестування:

- Ізоляція тестових середовищ: Тестування не повинно виконуватися на продуктивних середовищах, оскільки це може призвести до витоку або пошкодження конфіденційних даних. Тестові середовища повинні бути максимально відокремлені від основної продуктивної системи.
- Захист тестових даних: Тестування часто передбачає використання даних, які можуть містити чутливу інформацію (наприклад, особисті дані користувачів). Важливо використовувати техніки анонімізації або маскування даних для забезпечення їх захисту.

- Контроль доступу: Доступ до тестових середовищ, тестових даних та результатів тестування має бути обмежений лише для уповноважених осіб. Використання багатофакторної автентифікації та принципу найменших привілеїв допомагає захистити інфраструктуру тестування від потенційних загроз.
- Використання зашифрованих каналів: Тестування API та взаємодія з іншими сервісами повинні здійснюватися через захищені протоколи (наприклад, HTTPS) для запобігання несанкціонованому доступу або перехопленню даних.
- Регулярні аудити тестових середовищ: Для виявлення потенційних вразливостей необхідно регулярно проводити аудити тестового середовища та перевіряти його на відповідність вимогам безпеки.

РОЗДІЛ 4. НАЛАШТУВАННЯ СЕРЕДОВИЩА ДЛЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

4. Підготовка середовища розробки

Щоб реалізувати автоматизоване тестування з використанням фреймворків JUnit і Selenium, спочатку потрібно налаштувати робоче середовище. У цьому підпункті буде детально описано кожен крок — від встановлення необхідного ПЗ до створення базового проєкту в IntelliJ IDEA.

4.1 Встановлення IntelliJ IDEA

IntelliJ IDEA — це інтегроване середовище розробки (IDE), яке підтримує роботу з Java і багатьма іншими мовами програмування. Для нашого проєкту достатньо безкоштовної версії IntelliJ IDEA Community Edition.

Етапи встановлення:

1. Перейдіть на офіційний сайт JetBrains: https://www.jetbrains.com/idea/
2. Завантажте інсталятор для вашої операційної системи.
3. Встановіть програму, дотримуючись інструкцій інсталятора.
4. Після запуску оберіть тему оформлення, бажано залишити стандартну.
5. Увійдіть у JetBrains Account або пропустіть, якщо цього не вимагає встановлення.

4.2 Встановлення Java Development Kit (JDK)

IntelliJ IDEA не містить вбудованого JDK, тому потрібно завантажити й встановити його окремо.

1. Завантажте JDK з офіційного сайту Oracle: https://www.oracle.com/java/technologies/javase-downloads.html
2. Встановіть його, запам’ятавши шлях до JDK (наприклад, C:\Program Files\Java\jdk-17).
3. У IntelliJ IDEA перейдіть до: File → Project Structure → SDKs → Add SDK → Java SDK → оберіть шлях до встановленого JDK.

4.3 Створення нового Maven-проєкту

Maven — це система керування проєктами, яка допомагає автоматично підключати бібліотеки (залежності), компілювати код, запускати тести тощо.

1. У IntelliJ IDEA натисніть File → New → Project.
2. Виберіть Maven → натисніть Next.
3. У полі GroupId введіть, наприклад: com.example
4. У полі ArtifactId введіть: selenium-test
5. Оберіть Java версії 17 (або ту, що встановлено).
6. Натисніть Finish — проєкт буде створено.

4.4 Налаштування файлу pom.xml

Файл pom.xml є конфігураційним — у ньому задаються всі залежності, необхідні для проєкту. Щоб працювати із Selenium і JUnit, необхідно додати до pom.xml такі залежності:

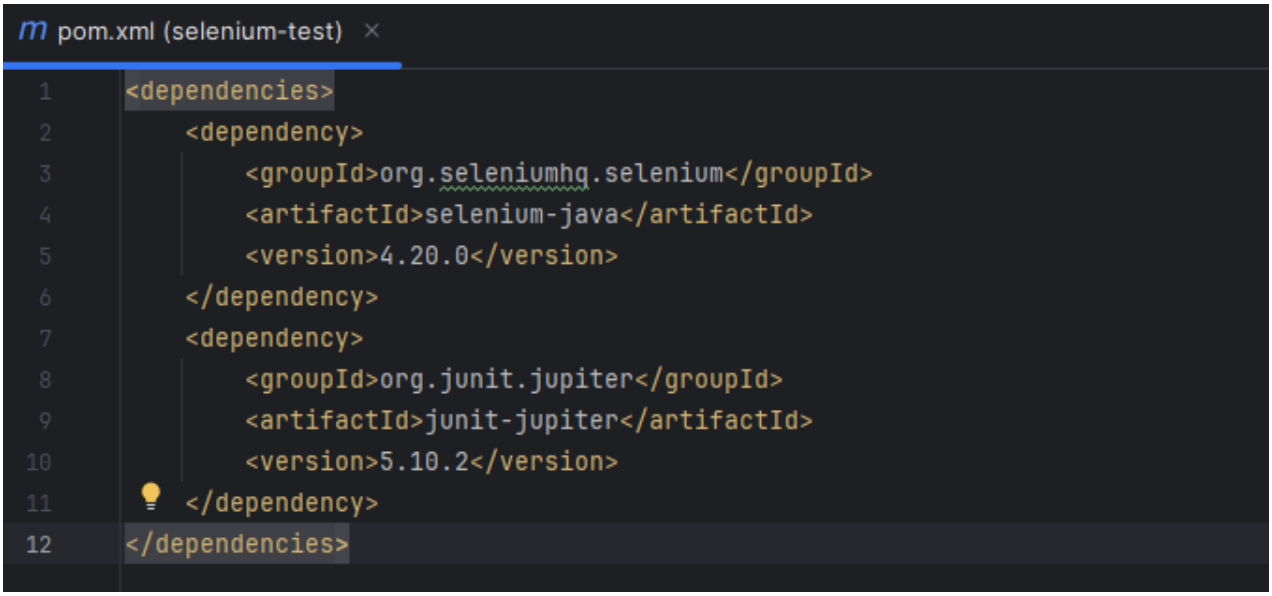


Рисунок 4.1 – Файл pom.xml

Пояснення:

- selenium-java — бібліотека для автоматизації браузера.
- junit-jupiter — сучасна версія фреймворку для написання тестів у Java.

4.5 Налаштування WebDriver для браузера Chrome

Selenium працює з браузером за допомогою WebDriver. Для Google Chrome потрібен chromedriver.

1. Дізнайтесь свою версію браузера Chrome (в адресному рядку введіть chrome://settings/help).

2. Перейдіть на сайт https://chromedriver.chromium.org/downloads і завантажте відповідну версію chromedriver.
3. Розпакуйте архів у будь-яку теку, наприклад: C:\webdriver\chromedriver.exe
4. У системних змінних (System Environment Variables) додайте шлях до цієї теки в змінну PATH.

Альтернатива: можна прописати шлях у коді вручну (покажу пізніше).

4.6. Опис технологій і бібліотек

У цьому підпункті ми розглянемо основні технології, які використовуються для автоматизованого тестування, а саме: JUnit, Selenium і Maven. Кожна з цих технологій має свою роль у процесі тестування та взаємодіє з іншими компонентами для досягнення ефективного результату.

4.6.1 JUnit

JUnit — це фреймворк для написання автоматизованих тестів в Java. Він дозволяє створювати тести для окремих частин коду та автоматично їх запускати, перевіряючи правильність роботи кожної з частин. JUnit є стандартом для юніт-тестування в Java.

Основні особливості JUnit:

- Анотації: JUnit використовує анотації для визначення методів тестування, таких як @Test, @Before, @After.
- Тестові методи: Кожен метод, який має анотацію @Test, є тестовим методом. У ньому виконуються перевірки.

– Асерт: Відповідно до умов тесту, ми можемо використовувати різні асерти, наприклад, assertEquals, assertTrue для перевірки результатів.



Рисунок 4.2 – Приклад тесту в JUnit

У цьому прикладі ми перевіряємо, чи сума 1 + 1 дорівнює 2.

4.6.2 Selenium

Selenium — це набір інструментів для автоматизації веб-застосунків через браузери. Selenium дозволяє писати автоматизовані тести, які взаємодіють з елементами вебсторінки, такими як кнопки, поля введення та інші компоненти.

Основні компоненти Selenium:

– WebDriver: Основний інструмент для взаємодії з браузером. WebDriver керує браузером, відкриває сторінки, заповнює форми та виконує інші дії.
– Selenium Grid: Дозволяє запускати тести на різних браузерах та операційних системах одночасно.

– Selenium IDE: Інтерфейс для запису тестів без необхідності програмування.

pom.xml (selenium-test) SimpleSeleniumTest.java

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class SimpleSeleniumTest {

    public static void main(String[] args) {
        // Вказуємо шлях до драйвера Chrome
        System.setProperty("webdriver.chrome.driver", "C:\\Tools\\chromedriver.exe");

        // Створюємо екземпляр WebDriver для Chrome
        WebDriver driver = new ChromeDriver();

        // Відкриваємо сайт
        driver.get("https://example.com");

        // Перевірка заголовка сторінки
        String title = driver.getTitle();
        System.out.println("Title of the page is: " + title);

        // Закриваємо браузер
        driver.quit();
    }
}
```

Рисунок 4.3 – приклад тесту з Selenium

У цьому прикладі Selenium відкриває сайт https://example.com, перевіряє заголовок сторінки та виводить його.

4.6.3 Maven

Maven — це система автоматизації збірки для проєктів на Java. Maven дозволяє спростити процес керування залежностями (бібліотеками, фреймворками) і забезпечує стандартизовану структуру проєкту.

Основні функції Maven:

– Управління залежностями: Maven дозволяє автоматично підключати бібліотеки, необхідні для проєкту, через файл pom.xml.
– Збірка проєкту: Maven автоматизує процес компіляції, запуску тестів, створення звітів і публікації артефактів.
– Плагіни: Maven підтримує плагіни для виконання різних завдань, таких як тестування, збірка документації, тощо.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>selenium-test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>4.20.0</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.10.2</version>
        </dependency>
    </dependencies>
</project>
```

Рисунок 4.4 – Приклад файлу pom.xml для Maven

Цей файл містить залежності для Selenium і JUnit, щоб у проєкті можна було їх використовувати.

4.6.4 Як ці технології взаємодіють

1. Maven керує залежностями й автоматизує процес зборки проєкту.
2. JUnit дозволяє писати тести, що перевіряють правильність роботи окремих компонентів програми.
3. Selenium використовується для автоматизації браузера, перевірки вебсайтів та інтерактивної роботи з елементами на сторінці.

Ці три інструменти працюють разом, щоб автоматизувати тестування, зробити процес тестування більш ефективним і зручним для розробників.

4.7 Написання простого тесту для сайту YouTube з використанням Selenium та JUnit

У цьому підпункті буде розглянуто приклад реалізації простого автоматизованого тесту, який відкриває сайт YouTube та перевіряє, чи відповідає заголовок сторінки очікуваному значенню. Для реалізації тесту використовуються такі інструменти: Selenium WebDriver для автоматизації взаємодії з веббраузером та JUnit як фреймворк для організації тестування.

Метою тесту є перевірка, що при відкритті головної сторінки сайту https://www.youtube.com її заголовок містить слово "YouTube", що дозволяє зробити висновок про успішне завантаження потрібного ресурсу.

Передумови:

- Середовище розробки: IntelliJ IDEA;
- Установлений браузер Google Chrome;
- Завантажений та налаштований ChromeDriver, сумісний із версією браузера;
- Додані залежності до selenium-java та junit у pom.xml.

pom.xml (Unknown) | YouTubeTest.java

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

import static org.junit.Assert.assertTrue;

public class YouTubeTest {
    5 usages
    private WebDriver driver;

    @Before
    public void setUp() {
        // Вказуємо шлях до ChromeDriver
        System.setProperty("webdriver.chrome.driver", "C:\\path\\to\\chromedriver.exe");

        // Ініціалізуємо екземпляр ChromeDriver
        driver = new ChromeDriver();
    }

    @Test
    public void testYouTubeTitle() {
        // Відкриваємо головну сторінку YouTube
        driver.get("https://www.youtube.com");

        // Отримуємо заголовок відкритої сторінки
        String title = driver.getTitle();

        // Перевіряємо, що заголовок містить слово "YouTube"
        assertTrue( message: "Заголовок не містить 'YouTube'", title.contains("YouTube"));
    }

    @After
    public void tearDown() {
        // Закриваємо браузер після виконання тесту
        if (driver != null) {
            driver.quit();
        }
    }
}
```

Рисунок 4.5 – Приклад коду автоматизованого тесту

Пояснення до коду:

Клас YouTubeTest містить усі компоненти тесту.

Анотація @Before позначає метод setUp(), який виконується перед кожним тестом. У цьому методі виконується ініціалізація драйвера браузера.

Метод testYouTubeTitle() містить сам тест:

driver.get("https://www.youtube.com") відкриває сторінку.

driver.getTitle() зчитує заголовок сторінки.

assertTrue(...) перевіряє, що заголовок містить потрібний фрагмент тексту.

Анотація @After вказує, що метод tearDown() виконується після завершення тесту, і відповідає за коректне закриття браузера.

Такий тест демонструє базові принципи побудови автоматизованих тестів: налаштування, виконання дій, перевірка результату та завершення.

Цей приклад можна розглядати як основу для розширення функціональності тестів, наприклад, шляхом перевірки інших елементів на сторінці або взаємодії з формами.

ВИСНОВКИ

У процесі виконання дипломної роботи було розглянуто теоретичні та практичні аспекти автоматизованого тестування програмного забезпечення із застосуванням сучасних фреймворків і середовища розробки IntelliJ IDEA. Робота охопила широкий спектр питань, що стосуються організації, реалізації та вдосконалення процесу автоматизованого тестування, а також відображає сучасні тенденції та виклики в цій галузі.

У теоретичній частині дипломної роботи було проаналізовано основні поняття тестування ПЗ, класифікацію видів тестування, переваги й недоліки автоматизації. Було визначено, які фреймворки та інструменти найбільш ефективні в сучасних умовах, а також розглянуто методологічні підходи до побудови стратегії тестування, роль середовища IntelliJ IDEA, CI/CD, хмарних рішень і технологій машинного навчання у тестуванні.

Особливу увагу приділено забезпеченню якості та безпеки при автоматизованому тестуванні. Розглянуто методи контролю якості, стандарти, метрики, а також питання кібербезпеки, що є критично важливими в умовах сучасної розробки програмного забезпечення.

У практичній частині дипломної роботи реалізовано приклад налаштування середовища автоматизованого тестування із використанням IntelliJ IDEA, Java, Maven, Selenium та JUnit. Створено приклад простого автотесту для вебсайту, що демонструє основні принципи реалізації тестування у реальному проєкті.

У результаті виконання дипломної роботи були досягнуті поставлені цілі:

– сформовано цілісне уявлення про сучасні підходи до автоматизованого тестування;

- обґрунтовано вибір інструментів та фреймворків;
- реалізовано практичний приклад тестування із повною конфігурацією середовища;
- висвітлено актуальні тенденції та проблеми галузі.

Отримані результати можуть бути використані для організації ефективного процесу тестування в IT-компаніях, при впровадженні автоматизації на підприємствах, а також як навчальний матеріал для студентів і початківців у сфері QA.

Таким чином, дипломна робота підтвердила важливість автоматизованого тестування як ключового етапу забезпечення якості програмного забезпечення, а також практичну цінність сучасних фреймворків і інструментів у процесі його реалізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ammann, P., & Offutt, J. Introduction to Software Testing. Cambridge University Press, 2016.

2. Myers, G. J., Sandler, C., Badgett, T. The Art of Software Testing. Wiley, 2011.

3. Fewster, M., Graham, D. Software Test Automation. Addison-Wesley, 1999.

4. Selenium Official Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.selenium.dev/documentation/

5. JUnit 5 User Guide [Електронний ресурс] – Режим доступу до ресурсу: https://junit.org/junit5/docs/current/user-guide/

6. TestNG Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://testng.org/doc/

7. Cucumber Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://cucumber.io/docs/

8. Mockito Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://site.mockito.org/

9. Allure Framework Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.qameta.io/allure/

10. IntelliJ IDEA Official Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/idea/documentation/

11. Meszaros G. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007.

12. Arnon Axelrod. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. Apress, 2018.

13. Rex Black. Foundations of Software Testing ISTQB Certification. Cengage Learning, 2012.

14. Khorikov V. Unit Testing: Principles, Practices, and Patterns. Manning Publications, 2020.

15. REST Assured Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://rest-assured.io/

16. Atlassian. What is CI/CD? [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/continuous-delivery

17. D. Graham, E. Veenendaal, I. Evans. Foundations of Software Testing. ISTQB Certification. Cengage Learning, 2008.

18. Jason Arbon. "How AI Is Changing Software Testing". TechBeacon [Електронний ресурс] – Режим доступу до ресурсу: https://techbeacon.com/enterprise-it/how-ai-changing-software-testing

19. Katalon. No-code Test Automation Guide [Електронний ресурс] – Режим доступу до ресурсу: https://katalon.com

20. Pact.io Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.pact.io

21. BrowserStack Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/docs

22. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models/

23. OWASP Foundation. OWASP Top Ten Web Application Security Risks [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-project-top-ten

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (ПРЕЗЕНТАЦІЯ)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Комп'ютерної інженерії

РОЗРОБКА АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ З ВИКОРИСТАННЯМ ФРЕЙМВОРКІВ ТА СЕРЕДОВИЩА INTELLIJ IDEA

Виконав:
Зубець Дмитро Ігорович
Науковий керівник:
Торошанко Ярослав Іванович

Київ 2025

Мета, об’єкт, предмет

- Метою роботи є розробка ефективного підходу до автоматизованого тестування програмного забезпечення з використанням сучасних фреймворків у середовищі IntelliJ IDEA.
- Об’єктом дослідження є процес тестування програмного забезпечення.
- Предметом дослідження є засоби та методи автоматизації тестування, а також інструменти та фреймворки, що використовуються у середовищі розробки IntelliJ IDEA.

Актуальність

Актуальність теми зумовлена постійним зростанням вимог до якості програмного забезпечення та необхідністю швидкого і ефективного тестування. Автоматизація значно скорочує час перевірки програмного коду та знижує ризик людських помилок.

Порівняльна таблиця інструментів

Характеристика	Maven	Selenium	JUnit
Тип інструменту	Система управління проєктами та залежностями	Інструмент для автоматизованого тестування вебінтерфейсів	Фреймворк для модульного тестування
Основна функція	Автоматичне керування бібліотеками та збіркою	Емуляція дій користувача у веббраузері	Написання та запуск юніт-тестів
Мова програмування	Java	Підримує Java, Python, C#, JavaScript	Java
Інтеграція з IntelliJ IDEA	Повна підтримка (автоматичне налаштування проєкту)	Підтримується через драйвери та залежності	Вбудована підтримка
Головні переваги	Зручне керування залежностями, плагіни, автоматизація	Підтримка багатьох браузерів та платформ	Простота написання тестів, анотації, звіти
Недоліки	Не тестує напряму – потрібен у зв’язці з іншими	Обмежене до UI, складна підтримка стабільності	Підходить лише для модульних тестів
Сценарій використання	Побудова та керування структурою тестового середовища	Тестування користувацького інтерфейсу вебдодатків	Тестування логіки додатку (методів, класу)

Ключові поняття та роль автоматизації

- Автоматизоване тестування забезпечує повторюваність, швидкість і ефективність перевірки ПЗ.
- Основні підходи: модульне, інтеграційне, системне тестування.
- Автоматизація дозволяє зменшити людський фактор і покращити стабільність випусків.

Інструменти та середовище тестування

- Використано IntelliJ IDEA як основне середовище.
- Налаштовано JDK, Maven, WebDriver.
- Обрано Selenium – для автоматизації взаємодії з браузером, JUnit – для структуризації тестів.









Порівняльна таблиця браузерів для автоматизованого тестування з Selenium

Характеристика	Chrome	Firefox	Edge	Safari
Підтримка Selenium	Повна	Повна	Повна	Часткова (тільки на macOS)
WebDriver	ChromeDriver	GeckoDriver	EdgeDriver	SafariDriver
Швидкість запуску	Висока	Висока	Висока	Середня
Платформи	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS	Лише macOS
Частота оновлень драйвера	Регулярні	Регулярні	Регулярні	Менш часті
Документація	Багато ресурсів	Помірна кількість	Достатньо	Обмежено
Проблеми з сумісністю	Мінімальні	Мінімальні	Іноді після оновлень	Часті для Windows

Реалізація прикладного тесту

Реалізація автоматизованого тесту на прикладі сайту YouTube з використанням Selenium та JUnit. У результаті тесту перевіряється правильність відкриття сайту, робота пошуку та загальна відповідь інтерфейсу

pom.xml (selenium-test) | SimpleSeleniumTest.java

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>selenium-test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>4.20.0</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.10.2</version>
        </dependency>
    </dependencies>
</project>
```

Рисунок 4.4 – Приклад файлу pom.xml для Maven

pom.xml (Unknown) | YouTubeTest.java

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

import static org.junit.Assert.assertTrue;

public class YouTubeTest {
    5 usages
    private WebDriver driver;

    @Before
    public void setUp() {
        // Вказуємо шлях до ChromeDriver
        System.setProperty("webdriver.chrome.driver", "C:\\path\\to\\chromedriver.exe");

        // Ініціалізуємо екземпляр ChromeDriver
        driver = new ChromeDriver();
    }

    @Test
    public void testYouTubeTitle() {
        // Відкриваємо головну сторінку YouTube
        driver.get("https://www.youtube.com");

        // Отримуємо заголовок відкритої сторінки
        String title = driver.getTitle();

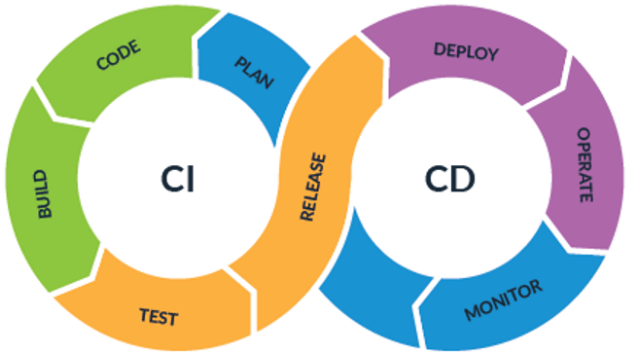
        // Перевіряємо, що заголовок містить слово "YouTube"
        assertTrue( message: "Заголовок не містить 'YouTube'", title.contains("YouTube"));
    }

    @After
    public void tearDown() {
        // Закриваємо браузер після виконання тесту
        if (driver != null) {
            driver.quit();
        }
    }
}
```

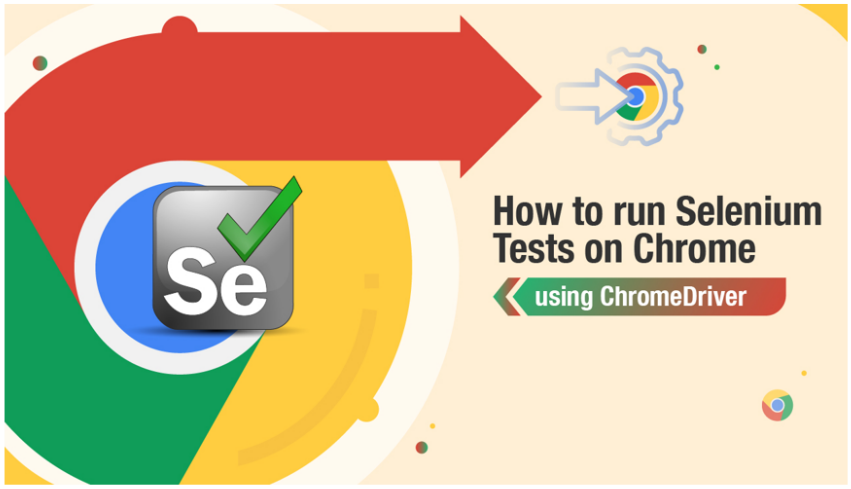
Рисунок 4.5 – Приклад коду автоматизованого тестування

Переваги обраного підходу

- Простота підтримки тестів.
- Чітка структура і можливість масштабування.
- Використання поширених технологій, сумісних з CI/CD.



Складнощі під час реалізації



- Налаштування середовища потребувало врахування сумісності версій бібліотек.
- Пошук стабільного WebDriver для актуальної версії Chrome.
- Вирішення проблеми синхронізації завдяки WebDriverWait.

Отримані результати



- Створено повноцінне середовище для автотестів.
- Реалізовано базовий приклад автоматизованого функціонального тесту.
- Показано практичну взаємодію популярних бібліотек у реальному кейсі.

Висновки

У процесі виконання бакалаврської роботи було досліджено теоретичні основи та практичні аспекти автоматизованого тестування програмного забезпечення. Розглянуто сучасні інструменти, методи та підходи, що дозволяють підвищити ефективність тестування.

Проведено налаштування середовища розробки з використанням IntelliJ IDEA, Maven, Selenium та JUnit, а також реалізовано приклад функціонального автотесту.

Отримані результати підтверджують доцільність застосування автоматизації для забезпечення якості програмного продукту та зменшення витрат на тестування.