Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

факультет прикладних інформаційних технологій та електроінженерії
(повна назва факультету)
кафедра автоматизації технологічних процесів і виробництв
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: «Розробка і аналіз системи автоматизованого тестування програмних продуктів з використанням javascript (комплексна тема).

Виконав(ла): студент(ка) VI курсу, групи КАм-61
спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології»
(шифр і назва спеціальності)

(підпис) Ковтко А.М. (прізвище та ініціали)
(підпис) Лещук Н.В. (прізвище та ініціали)
Керівник (підпис) Коноваленко І.В. (прізвище та ініціали)
Нормоконтроль (підпис) Козбур В.Р. (прізвище та ініціали)
Завідувач кафедри (підпис) Савків В.Б. (прізвище та ініціали)
Рецензент (підпис) Чихіра І.В. (прізвище та ініціали)

Тернопіль
2023

АНОТАЦІЯ

Дана магістерська кваліфікаційна стосується розробки засобів автоматизованого тестування програмного забезпечення для підвищення його ефективності, економії часу, ресурсів і витрат.

В даній роботі проаналізовано аналоги платформ тестування програмного забезпечення, визначено їх функціональні можливості. Розглянуто різні сучасні інструменти автоматизованого тестування програмного забезпечення.

Розроблено програмне забезпечення для автоматизованого тестування веб-сайтів за допомогою двох типів тестування, – UI & API та API тестування. Окремо розроблено архітектуру автоматизованого фреймворку.

Для тестування UI частини нашого веб-сайту, було використано інструмент Playwright, для API тестування вибрано інструмент SuperTest. Для запуску тестів використовуємо Jest приєднаними відповідними бібліотеками, котрі додаємо до package.json.

Розроблено систему збору аналітичних даних по результатам проведених тестувань. Для цього використані тестові репорти двох типи тестів UI і API і для цього використовуються різні інструменти. Для API це окрема бібліотека яка на основі Jest_runner, котра агрегує дані і генерує репорт. Для UI тестування інструмент Playwright має хороший вбудований репорт. Для зручності додано ще один репорт – Allure, котрий агрерує всі дані і формує один загальний репорт.

Досліджено ефективність інструментів автоматичного тестування з використанням автоматичної генерації тестових даних, що має важливе значення для підтримки модульного тестування. У кваліфікаційній роботі проведено порівняння добре відомих загальнодоступні інструменти генерації даних модульного тестування, – TestGen4j , JCrasher та JUB. Їх застосували до класів Java та оцінили ефективність на основі шкали значень їхніх мутації.

ABSTRACT

This master's qualification concerns the development of automated software testing tools to increase its efficiency, save time, resources and costs.

In this paper, analogues of software testing platforms are analyzed, and their functionality is determined. Various modern automated software testing tools are considered.

Developed software for automated website testing using two types of testing, UI & API and API testing. The architecture of the automated framework was developed separately.

For testing the UI part of our website, the Playwright tool was used, and for the API testing, the SuperTest tool was chosen. To run the tests, we use Jest with the appropriate libraries attached, which we add to package.json.

A system for collecting analytical data based on the results of the tests has been developed. For this, test reports of two types of UI and API tests are used and various tools are used for this. For the API, this is a separate library based on Jest_runner, which aggregates data and generates a report. For UI testing, the Playwright tool has a nice built-in report. For convenience, one more report has been added - Allure, which aggregates all data and forms one general report.

The effectiveness of automatic testing tools using automatic generation of test data, which is important to support unit testing, is investigated. In the qualification work, a comparison of well-known publicly available data generation tools for unit testing, TestGen4j, JCrasher and JUB, was made. They were applied to Java classes and evaluated for performance based on a scale of their mutation values.

ЗМІСТ

АНОТАЦІЯ 3
ABSTRACT 4
ЗМІСТ 5
ВСТУП 11
1 АНАЛІТИЧНА ЧАСТИНА 14
1.1 Інструменти автоматизованого тестування програмного забезпечення 14
1.3 Огляд автоматизованих інструментів тестування (з основними функціями) 16
1.3.1 Платформа Katalon 16
1.3.2 Фреймворк Selenium 18
1.3.3 Інструмент автоматизації тестування програмного забезпечення Appium 19
1.3.4 Функціональне тестування інтерфейсу TestComplete 19
1.3.5 Система тестування програмного забезпечення Cypress. 20
1.3.6 Автоматизоване тестування GUI для веб-додатків Ranorex Studio 21
1.3.7 Хмарна платформа автоматизованого тестування Perfecto 22
1.3.8 Хмарна платформа автоматизованого тестування LambdaTest 23
1.3.9 Інструмент автоматизованого тестування API Postman 23
1.3.10 Інструмент тестування API SoapUI 24
1.3.11 Інструмент автоматизації тестування GUI Eggplant, Eggplant Functional 25
1.3.12 Комплексний інструмент автоматизації для веб-тестування Tricentis Tosca 25
1.3.13 Інструмент автоматизованого тестування продуктивності 26

1.3.14 Платформа автоматизації тестування програмного забезпечення Robot Framework 27
1.3.15 Автоматизований інструмент візуального тестування Applitools 28
2 НАУКОВО-ДОСЛІДНА ЧАСТИНА 30
Порівняння інструментів автоматизованої генерації тестів на рівні модуля 30
2.1 Вибір інструментів автоматизованої генерації тестів 31
2.2 Розробка інструментів експериментального дослідження 32
2.3 Суб'єкти та інструменти модульного тестування 33
2.4 Додаткові набори тестів 34
2.5 Перевірка Java класів. 34
2.6 Інструмент тестування мутацій MuJava 35
2.7 Порівняльне експериментальне дослідження інструментів JCrasher, TestGen4J, JUB. 36
2.8 Загрози достовірності результатів 37
3 ТЕХНОЛОГІЧНА ЧАСТИНА 50
3.1 Тестування програмного забезпечення – загальний огляд 50
3.2 Підтримка та покращення якості програмної продукції 51
3.3 Види тестування ПЗ 52
3.4 Підходи до тестування ПЗ 53
3.5 Життєвий цикл тестування програмного забезпечення 54
3.5.1. Аналіз вимог. 54
3.5.2. Планування тестування 55
3.5.3. Розробка тестового випадку 56
3.5.5. Виконання тесту 58
3.5.6. Закриття випробувального циклу 58
3.6 Популярні моделі тестування програмного забезпечення 58
3.6.1. V-модель 59

3.6.2. Тестова модель піраміди..... 59
3.6.3. Стільникова модель..... 60
3.8 Порівняння автоматизованого тестування проти ручного 62
4 ПРОЕКТНА ЧАСТИНА 65
4.1 Автоматизоване тестування веб-сайту..... 65
4.1.1. Стратегія тестування:..... 65
4.1.2. Підхід до написання автоматизованих тестів: 66
4.1.3. API тестування:..... 66
4.1.4. Ручне API тестування: 67
4.1.5. Автоматизоване API тестування:..... 68
4.1.6. Переваги автоматизованого API тестування:..... 69
4.1.7. Процес тестування API:..... 70
4.1.8. Реалізація API тестування: 70
4.1.9. Реалізація UI тестування: 74
4.2 Архітектура автоматизованого фреймворку 76
4.2.1. Написання тестів 77
4.2.2. Процес написання тестів: 78
4.2.3. Тестові Layers (шари):..... 78
4.3 CI/CD + репортинг..... 79
4.3.1.Тестові репорти: 79
4.3.2. CI/CD: 79
5 СПЕЦІАЛЬНА ЧАСТИНА..... 82
5.1 Фреймворк для тестування JavaScript Jest..... 82
5.1.1. Синтаксис та низький поріг вхідного бар'єру. 82
5.1.2. Автоматичне визначення тестових файлів. 82
5.1.3. Можливості мокування та підробки (Mocking). 82
5.1.4. Розширена підтримка асинхронного коду. 83
5.1.5. Зручний інтерфейс виведення результатів. 83

5.1.6. Сприяння тестам з великим обсягом даних..... 84
5.1.7. Підтримка Snapshot-Тестування. 84
5.1.8. Підтримка таймерів. 84
5.1.9. Розширюваність та плагіни. 84
5.1.10. Підтримка серверного та клієнтського коду. 85
5.1.11. Параметризовані тести:..... 85
5.1.12. Тайм-аути та Retry. 85
5.1.13. Глобальні функції beforeAll, afterAll, beforeEach, afterEach. 85
5.1.14. Тестування Async/await. 86
5.1.15. Динамічні Тести. 86
5.1.16. Виведення покриття коду. 86
5.1.17. Вбудований інтерфейс тестування React. 87
5.2 Детальний розбір бібліотеки SuperTest. 88
5.2.1. Синтаксис та інтеграція. 88
5.2.2. Підтримка різних HTTP-методів. 88
5.2.3. Перевірка заголовків та тіла відповіді. 88
5.2.4. Можливість використання Cookies та Сесій. 89
5.2.5. Асинхронна підтримка..... 89
5.2.6. Підтримка JSON та форм-даних. 89
5.2.7. Виведення тестових запитів. 90
5.2.8. Легка інтеграція з іншими фреймворками тестування. 90
5.2.9. Підтримка таймаутів та повторів. 90
5.2.10. Організація тестових середовищ. 91
5.2.11. Підтримка організації тестових сценаріїв. 91
5.2.12. Виведення деталей запиту та відповіді. 91
5.2.13. Використання Promise та Async/await. 92
5.2.14. Підтримка тестового збільшення навантаження..... 92
5.2.15. Розширюваність та плагіни. 92

5.3 Детальний розбір Playwright. 93
5.3.1. Багатофункціональність. 93
5.3.2. Підтримка різних браузерів..... 93
5.3.3. Асинхронна підтримка та Promise API. 94
5.3.4. Робота з багатьма сторінками, мультиплікація..... 94
5.3.5. Засоби для тестування продуктивності..... 94
5.3.6. Обробка подій та введення користувача..... 95
5.3.7. Взаємодія з фреймами та IFrames..... 95
5.3.8. Тестування мобільних веб-додатків..... 95
5.3.9. Синхронізація та очікування умов. 95
5.3.10. Скріншоти та відеозаписи тестів:..... 96
5.3.11. Інтеграція з іншими інструментами. 96
5.3.12. Обмеження та підтримка конфігурацій. 96
5.3.13. Відлагодження та інспектор браузера..... 97
5.3.14. Сценарії тестування інтерактивності. 97
5.3.15. Мови та підтримка тестування на віддалених серверах. 97
5.3.16. Взаємодія з локальним сервером та мокованими даними. 98
5.3.17. Підтримка регулярних виразів для селекторів..... 98
5.3.18. Підтримка темниць та робота з Cookies..... 98
5.4 Установка Jest. 99
5.5 Запуск тестування Jest..... 100
5.6 Установка Playwright..... 107
6 ОХОРОНА ПРАЦІ 109
6.1 Визначення оптимальних умов праці інженера-оператора системи мікроклімату 109
6.2 Розрахунок освітленості робочого місця 113
7 БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ..... 117

7.1 Основні вражаючі фактори ядерних вибухів, їхні параметри і наслідки впливу на людей 117
7.2 Методи захисту та безпека підприємств промисловості, відновлення інженерно-технічного комплексу цеху (заводу) 120
7.3 Висновки розділу 121
ВИСНОВОК 124
ПЕРЕЛІК ПОСИЛАНЬ 127

ВСТУП

Основний сплеск інтересу до теми тестування програмних продуктів припав на 90-ті роки і розпочався США. Бурхливий розвиток систем автоматизованої розробки програмного забезпечення (CASE-засобів) та мережевих технологій призвів до зростання ринку виробництва ПЗ та до перегляду питань забезпечення відповідної надійності та якості програмних продуктів, що розробляються. Різко посилилася конкуренція між виробниками програмного забезпечення, що відповідно привело до зростання особливої уваги щодо якості створюваних продуктів, так як тепер у споживача був вибір. Багато фірм пропонували свої продукти та послуги за досить прийнятними цінами, а тому можна було звернутися до тих, хто розробить програму не лише швидко та дешево, а й якісно. Ситуація ускладнилася тим фактом, що в даний час комп'ютеризації та цифровізації підлягають практично всі галузі людського життя. Якість та надійність програмного забезпечення починає набувати особливої важливості. На теперішній час це вже не тільки комфорт від роботи в тій чи іншій програмі, сьогодні відповідне програмне забезпечення керує обладнанням у лікарнях, диспетчерськими системами в аеропортах, атомними реакторами, космічними кораблями тощо.

Усвідомивши той факт, що забезпечення високої якості програмних продуктів – це реальний шлях «обійти» конкурентів, багато компаній у всьому світі вкладають все більше коштів у забезпечення якості своїх продуктів, створюючи власні групи та відділи, що займаються тестуванням, або передаючи тестування своїх продуктів стороннім організаціям .

Найбільші компанії, що дбають про свою репутацію і бажають пройти сертифікацію на високому рівні. CMMI (Capability Maturity Model Integration), створюють власні системи управління якістю (Quality

Management System), спрямовані на постійне удосконалення виробничих процесів і постійне підвищення якості програмних продуктів.

Сьогодні тестування стало обов'язковою частиною процесу виробництва. Воно спрямоване на виявлення, виправлення та усунення якомога більшої кількості помилок та недоліків програмного забезпечення. Наслідком такої діяльності є підвищення якості програмного забезпечення за всіма його характеристиками.

Кінцевою метою будь-якого процесу тестування програмного продукту є забезпечення такого сукупного поняття як якість, з урахуванням усіх чи найбільш критичних для даного конкретного випадку складових.

Тестування програмного забезпечення – це процес визначення, чи програма виконує ті дії, котрі від неї очікують. Зазвичай, ніяке тестування неспроможне дати абсолютну гарантію працездатності та коректності роботи програми у майбутньому.

Завдання тестування програмного забезпечення – зменшити вартість розробки шляхом раннього виявлення дефектів. Відповідно тестування програмного забезпечення стало невід'ємною частиною розробки програмних продуктів. Тестування необхідно для того, щоб зрозуміти, чи працює програма, як очікується і чи відповідає вона вимогам, що висуваються до неї. Своєчасне виявлення та виправлення помилок і недоробок має велике значення в процесі розробки програмного продукту, оскільки це зменшує ризики і при цьому відбувається суттєве зниження витрат на розробку програмного забезпечення.

Завдяки тестуванню компанії здатні підтримувати якість своїх продуктів на дуже високому рівні. Часто процес тестування програмного забезпечення може бути автоматизований, що у деяких випадках може позитивно позначиться швидкості проведення та якості тестування, що дозволяє ще більше знизити витрати та підвищити якість кінцевого програмного продукту.

На даний момент суттєва увага приділяється процесам тестування та способам мінімізувати витрати за рахунок автоматизації процесів тестування програмного забезпечення.

Відповідно керуючись вищезгаданим, велика кількість компаній по всьому світу почала інвестувати у підвищення якості програмного забезпечення. У компаніях розробників почали створюватись відділи контролю якості, котрі застосовують новітні технології та засоби автоматизованого тестування програмного забезпечення, що відповідно дозволило компаніям збільшити свою конкурентну перевагу за рахунок підвищення надійності та якості своїх програмних продуктів.

Автоматизоване тестування програмного забезпечення продемонструвало свою ефективність у економії часу, ресурсів і витрат завдяки автоматизації найбільш монотонних завдань, покращенню тестування програмного забезпечення та підвищенню загальної якості програмного забезпечення. Розробка методів та засобів автоматизації тестування програмного забезпечення має потенціал для подальшого покращення його ефективності та результативності, зниження ризику людської помилки та забезпечення більш повного охоплення тестуванням.

1 АНАЛІТИЧНА ЧАСТИНА

1.1 Інструменти автоматизованого тестування програмного забезпечення

Інструменти автоматизованого тестування – це програми, призначені для перевірки функціональних та/або нефункціональних вимог до програмного забезпечення за допомогою сценаріїв автоматизованого тестування. Щоб допомогти прискорити випуск, покращити якість проекту та покращити результати. Інструменти автоматизованого тестування дозволяють без особливих зусиль створювати, запускати та підтримувати тести та підтримувати централізований перегляд аналітики результатів тестування.

Оскільки практики Agile та DevOps є стандартом для тестування програмного забезпечення , створення чіткої стратегії оцінки інструментів тестування автоматизації має важливе значення.

Стратегія вибору інструментарію тестування програмного забезпечення повинна відповісти на такі запитання:

1. Які функції ви шукаєте?
2. Хто буде використовувати інструмент для тестування? Розробники чи команди контролю якості ?
3. Чи можна його інтегрувати з конвеєрами CI/CD або ланцюжками інструментів?
4. Чи легко підтримувати сценарії та повторно використовувати тестові ресурси?
5. Який бюджет команди?
6. Де можна отримати підтримку з технічних питань?

На жаль, насправді не існує універсального інструменту автоматизації; це дійсно зводиться до конкретних потреб команди.

Рисунок 1.1 – Рейтингова іконографіка інструментів автоматизованого тестування програмних продуктів.

1.2 Критерії вибору автоматизованих інструментів тестування

Для оптимального вибору автоматизованих інструментів тестування не потрібно оцінювати та відстежувати результати по готових рішеннях. Натомість потрібно оцінити конкретні потреби команди, – людські ресурси та майбутню масштабованість, щоб обрати найкращий варіант. Наприклад, якщо тестувальники є досвідченими розробниками, відповідно використання Selenium або Appium для автоматизації тестування програмного забезпечення є хорошим варіантом, оскільки вони дозволяють створювати та масштабувати проекти тестування програмного забезпечення з нуля. Однак якщо ваша команда складається здебільшого з ручних тестувальників і вам потрібне рішення з низьким кодом, але таке, яке також може зростати та масштабуватися, тоді готові фреймворки автоматизації тестування програмного забезпечення, такі як Katalon Platform, можуть стати ідеальним рішенням.

Нижче наведено рейтинговий список інструментів автоматизованого тестування, доступних для підтримки ваших потреб у тестуванні.

1.3 Огляд автоматизованих інструментів тестування (з основними функціями)

Product	Katalon	Selenium	appium	TestComplete	cypress
Application Under Test	Web/API/ Mobile/Desktop	Web	Mobile (Android/iOS)	Web/Mobile/ Desktop	Web
Supported platform(s)	Windows/ macOS/ Linux	Windows/ macOS/ Linux/Solaris	Windows/ macOS	Windows	Windows/ macOS/ Linux
Setup & configuration	Easy	Coding Required	Coding Required	Easy	Coding Required
Low-code & Scripting mode	Both	Scripting Only	Scripting Only	Both	Scripting Only
Supported language(s)	Java & Groovy	Java, C#, Python, JavaScript, Ruby, PHP, Perl	Java, C#, Python, JavaScript, Ruby, PHP, Perl	JavaScript, Python, VBScript, JScript, Delphi, C++, C#	JavaScript
Advanced test reporting	✓	✗	✗	✗	✓
Pricing	Free and Paid	Free	Free	Paid	Free and Paid
Ratings & Reviews (Gartner)	4.4/5 740 reviews	4.5/5 443 reviews	4.4/5 90 reviews	4.4/5 45 reviews	4.6/5 27 reviews

Рисунок 1.2 – Порівняльна таблиця систем автоматизації тестування програмного забезпечення за функціональним складом

1.3.1 Платформа Katalon

Рисунок 1.3 – Системна емблема платформи Katalon

Платформа Katalon — це інструмент автоматизованого тестування з низьким кодом і масштабованістю для веб -додатків , API, настільних (Windows) і мобільних додатків . На сьогоднішній день спільнота Katalon перевищила один мільйон користувачів і є надійним рішенням автоматизації для понад 100 000 компаній.

Без необхідності кодувати або створювати структуру автоматизації тестування з нуля, користувачі можуть просто завантажити інструмент і зосередитися виключно на тестуванні. Крім того, Katalon пропонує часті випуски, щоб залишатися сумісними з останніми платформами/браузерами/ОС.

Основні функції:

- Гнучкі методи розробки тестів: запис і відтворення, ручний режим і режим сценарію
- Підтримувані методології тестування: BDD, DDT, тестування на основі ключових слів, крос-браузерне тестування (Headless, Chrome, Edge, Firefox і Safari) і крос-платформне мобільне тестування (iOS, Android). Докладніше про те, як проводити тестування на основі даних за допомогою Katalon, читайте тут .
- Автоматичний повтор невдалих тестів, розумне очікування та механізми самовідновлення.
- Повторно використовувані тестові об’єкти, ключові слова та тестові приклади із спільним використанням тестових артефактів і дизайном об’єктної моделі сторінки
- Власна інтеграція з популярними інструментами CI/CD і ALM (Jira, GitLab, Jenkins, Bitbucket, Azure DevOps тощо)
- Розумний інтерфейс користувача для налагодження та тестування звітів для швидкого усунення несправностей
- Інтеграція з популярними інструментами для співпраці для кращого планування тестування

- Детальна документація інструменту та відеоінструкції про Katalon Academy

  Веб-сайт : https://katalon.com/

1.3.2 Фреймворк Selenium

Рисунок 1.4 – Системна емблема платформи Selenium

Selenium, випущений у 2004 році, є одним із найпопулярніших, якщо не найпопулярніших фреймворків з відкритим кодом для автоматизації веб-тестування. Його набір програмного забезпечення складається з Selenium WebDriver, Selenium Grid і Selenium IDE.

Основні функції:

- Підтримувані мови програмування: Java, C#, Python, JavaScript, Ruby, PHP тощо.
- Підтримувані браузери: Chrome, Firefox, IE, Microsoft Edge, Opera, Safari тощо.
- Тестування на локальних або віддалених машинах через сервер Selenium
- Паралельне та кросбраузерне виконання для скорочення часу виконання та збільшення тестового покриття
- Інтеграція з іншими платформами тестування (наприклад, TestNG для звітності) та інструментами CI/CD

  Веб-сайт : https://www.selenium.dev/

  Ціна : безкоштовно

1.3.3 Інструмент автоматизації тестування програмного забезпечення Appium

Рисунок 1.5 – Системна емблема платформи Appium

Як і Selenium, Appium також є інструментом автоматизації тестування програмного забезпечення з відкритим кодом, але для мобільних додатків. Використовуючи мобільний протокол JSON, Appium дозволяє користувачам писати автоматичні тести інтерфейсу користувача для нативних, веб-і гібридних мобільних додатків як на Android, так і на iOS.

Основні функції:

- Підтримувані мови програмування: Java, C#, Python, JavaScript, Ruby, PHP, Perl
- Кросплатформне тестування з багаторазовими сценаріями тестування та однаковими API
- Виконання на реальних пристроях, симуляторах і емуляторах
- Інтеграція з іншими платформами тестування та інструментами CI/CD

Веб-сайт : https://appium.io/

Ціна : безкоштовно

1.3.4 Функціональне тестування інтерфейсу TestComplete

Рисунок 1.6 – Системна емблема платформи TestComplete

TestComplete може автоматизувати функціональне тестування інтерфейсу користувача для настільних, мобільних і веб-додатків. Завдяки вбудованій підтримці понад 500 елементів керування та фреймворків сторонніх розробників TestComplete може обробляти та ідентифікувати динамічні елементи інтерфейсу користувача в більшості доступних технологій.

Основні функції:

- Підтримувані мови програмування: JavaScript, Python, VBScript, JScript, Delphi, C++, C#
- Гнучкі методи розробки тестів: запис і відтворення, ручний режим і режими сценаріїв із вбудованими ключовими словами
- Ідентифікація об’єктів за допомогою візуального розпізнавання на основі властивостей і ШІ
- Паралельне тестування, тестування між браузерами та різними пристроями
- Інтеграція з іншими платформами тестування, інструментами CI/CD та екосистемою SmartBear

Веб-сайт : https://smartbear.com/product/testcomplete/overview/

Ціна : від 2702 доларів на рік

1.3.5 Система тестування програмного забезпечення Cypress.

Рисунок 1.7 – Системна емблема платформи Cypress

Суто підтримуючи фреймворки JavaScript, Cypress є орієнтованим на розробника інструментом автоматизації для наскрізного веб-тестування. Створений на основі нової архітектури, Cypress може безпосередньо працювати в браузері в тому самому циклі виконання, що й ваша програма,

забезпечуючи власний доступ до елементів і швидше виконання.

Основні функції:

- Підтримувана мова програмування: JavaScript
- Знімки виконання кроку тестування та можливості налагодження зі знайомих інструментів розробника
- Контроль за поведінкою функцій, відповіддю сервера, таймерами та мережевим трафіком
- Підключення до Cypress Cloud для перевірки продуктивності та оптимізації
- Інтеграція з популярними інструментами CI/CD

Веб-сайт : https://www.cypress.io/

Ціна : безкоштовно або від 75 доларів США на місяць для Cypress Cloud

1.3.6 Автоматизоване тестування GUI для веб-додатків Ranorex Studio

Рисунок 1.8 – Системна емблема платформи Ranorex Studio.

Ranorex Studio може автоматизувати тестування GUI для веб-додатків, мобільних і настільних додатків. Оснащений як автоматизацією з низьким кодом, так і повним IDE, фреймворк Ranorex простий у запуску для початківців і продуктивний у використанні для досвідчених тестувальників.

Основні функції:

- Підтримувані мови програмування: VB.Net і C#
- Широка підтримка веб-, мобільних і настільних технологій
- Інструмент Ranorex Spy і RanoreXPath для надійного розпізнавання елементів GUI

- Гнучкі методи розробки тестів: запис і відтворення та режим сценаріїв
- Розподілене або паралельне тестування за допомогою Selenium Grid
- Інтеграція з іншими платформами тестування та інструментами CI/CD

  Веб-сайт : https://www.ranorex.com/

  Ціна : від 2890 євро/рік

1.3.7 Хмарна платформа автоматизованого тестування Perfecto

Рисунок 1.9 – Системна емблема платформи Perfecto

Perfecto — це хмарна платформа автоматизованого тестування веб- і мобільних додатків. Perfecto робить безперервне тестування більш досяжним для команд DevOps, починаючи з автоматизованого виконання між середовищами, спеціальних можливостей, аналізу тестів і широкої інтеграції.

Основні функції:

- Створення тестів без сценаріїв для веб-додатків інтерфейсу користувача
- Стимуляція реального користувача для мобільного тестування: візуалізація мережі та інших умов середовища
- Паралельні та міжплатформні виконання
- Розширена аналітика тестів із централізованою інформаційною панеллю та фільтрацією шуму AI
- Інтеграція з іншими платформами тестування та інструментами CI/CD

  Веб-сайт : https://www.perfecto.io/products/platform/overview

  Ціна : від $125/міс

1.3.8 Хмарна платформа автоматизованого тестування LambdaTest

Рисунок 1.10 – Системна емблема платформи LambdaTest

LambdaTest забезпечує автоматизоване тестування в хмарі. Його хмарний сервіс дозволяє командам розширювати охоплення тестами за допомогою швидкого паралельного тестування, тестування між браузерами та різними пристроями.

Основні функції:

- Онлайн Selenium Grid у хмарі з понад 2000 пристроїв, браузерів і ОС
- Підтримка паралельного та міжбраузерного виконання сценаріїв тестування Cypress
- Веб-тестування геолокації в понад 27 країнах
- Інтеграція з іншими платформами тестування та інструментами CI/CD

Веб-сайт : https://www.lambdatest.com/automation-testing

Ціна : від 99 доларів США на місяць

1.3.9 Інструмент автоматизованого тестування API Postman

Рисунок 1.11 – Системна емблема платформи Postman

Postman є одним із найпоширеніших інструментів автоматизованого тестування API . Це дозволяє користувачам писати різні види тестів, від

функціональних та інтеграційних до регресійних тестів, і автоматично виконувати їх у конвеєрах CI/CD за допомогою командного рядка.

Основні функції:

- Зручний і простий у використанні інтерфейс, оснащений фрагментами коду
- Підтримка кількох методів HTML, форматів Swagger і RAML
- Широка підтримка схем API для створення колекцій і елементів API
- Створення набору тестів, виконання з параметризацією та налагодження
- Інтеграція з популярними інструментами CI/CD

Веб-сайт : https://www.postman.com/

Ціна : безкоштовно або від 12 доларів США за користувача на місяць

1.3.10 Інструмент тестування API SoapUI

Рисунок 1.12 – Системна емблема платформи Katalon

Цей інструмент тестування API із відкритим вихідним кодом розроблено для веб-служб REST і SOAP. Деякі життєво важливі функції включають автоматизоване тестування функцій, продуктивності, регресії та безпеки. Користувачі також можуть орієнтуватися на комерційну версію ReadyAPI (раніше SoapUI Pro) для більш розширених можливостей.

Основні функції:

- Перетягуйте, щоб створювати тести навіть зі складними сценаріями
- Стимулювання сервісу для зменшення зусиль побудови виробничих систем для тестування
- Швидке та просте повторне використання тестового сценарію

- Більше підтримки протоколів, можливостей та інтеграції CI/CD із ReadyAPI

  Веб-сайт : https://www.soapui.org/

  Ціна : безкоштовно або від 749 доларів США на рік за ReadyAPI

1.3.11 Інструмент автоматизації тестування GUI Eggplant, Eggplant Functional

Рисунок 1.13 – Системна емблема платформи ggplant, Eggplant Functional.

Як частина екосистеми Eggplant, Eggplant Functional є інструментом автоматизації тестування GUI для мобільних, настільних і веб-додатків. Завдяки підходу на основі зображень він дозволяє автоматизувати різні комбінації платформ і технологій одним сценарієм.

Основні функції:

- Гнучкі методи розробки тестів: запис, ручний режим і допоміжний сценарій
- Унікальна англомовна мова сценаріїв: SenseTalk
- Підключення до екосистеми Eggplant для більших можливостей тестування та моніторингу
- Інтеграція з популярними інструментами CI/CD

Веб-сайт : https://www.eggplantsoftware.com/product-downloads

1.3.12 Комплексний інструмент автоматизації для веб-тестування

Tricentis Tosca

Рисунок 1.14 – Системна емблема платформи Tricentis Tosca

Tricentis Tosca — це комплексний інструмент автоматизації для веб-тестування, тестування API, мобільних пристроїв і настільних комп'ютерів. Він має унікальний підхід до тестування на основі моделі, що дозволяє користувачам сканувати користувальницький інтерфейс програми або API, щоб створити звичну бізнес-модель для створення та обслуговування тестів.

Основні функції:

- Створення тестів без коду та висока можливість повторного використання тестових активів із підходом на основі моделі
- Оптимізація тестування на основі ризиків із розумним дизайном тестування та пріоритезацією вимог
- Віртуалізація послуг для стимулювання та створення реалістичних середовищ тестування
- Сканування API з розширеною підтримкою технологій API
- Паралельне та міжплатформне тестування
- Інтеграція з іншими платформами тестування та інструментами CI/CD

Веб-сайт : https://www.tricentis.com/products/automate-continuous-testing-tosca/

1.3.13 Інструмент автоматизованого тестування продуктивності

Рисунок 1.15 – Системна емблема платформи Apache JMeter

Цей інструмент із відкритим вихідним кодом призначений для автоматизованого тестування продуктивності, насамперед для веб-додатків. Це може стимулювати велику кількість користувачів до доступу до веб-служб і аналізу продуктивності AUT. JMeter також можна використовувати для функціонального тестування API.

Основні функції:

- Зручний і простий у використанні інтерфейс
- Створення плану тестування з функцією запису
- Виконання тесту в режимах GUI та CLI
- Підтримка багатьох різних серверів, програм і типів протоколів
- Інтеграція з популярними інструментами CI/CD

Веб-сайт : https://jmeter.apache.org/

Ціна : безкоштовно

1.3.14 Платформа автоматизації тестування програмного забезпечення Robot Framework

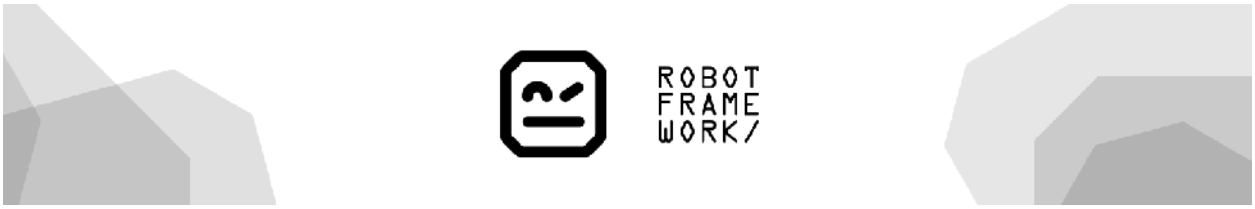


Рисунок 1.16 – Системна емблема платформи Robot Framework.

Robot Framework — це загальна платформа з відкритим кодом для

автоматизації тестування, особливо для приймального тестування та розробки на основі приймального тестування. Створений на основі підходу, що керується ключовими словами, він не потребує коду для запуску, а також розширюється за допомогою багатої екосистеми інструментів і бібліотек.

Основні функції:

- Створення тестів із простим табличним синтаксисом
- Підтримка тестування на основі ключових слів і даних
- Змінні для тестування в різних середовищах
- Підтримка великої кількості зовнішніх бібліотек та інтеграції з іншими інструментами

Веб-сайт : https://robotframework.org/

Ціна : безкоштовно

1.3.15 Автоматизований інструмент візуального тестування Applitools

Рисунок 1.17 – Системна емблема платформи Applitools.

Applitools виділяється як автоматизований інструмент візуального тестування, що революціонізує ландшафт тестування веб-додатків і мобільних додатків. Включення візуального тестування має важливе значення для виявлення помилок інтерфейсу користувача на вашій веб-сторінці, і Applitools перевершує в цьому відношенні.

Ключові особливості:

- **Розумне виявлення помилок:** Applitools використовує інтелектуальну технологію для точного виявлення візуальних помилок і відмінностей, забезпечуючи точне візуальне зіставлення.

- **Візуальне тестування на різних платформах:** переконайтеся, що ваш веб-сайт або програма однаково виглядають у різних браузерах і на різних пристроях, щоб забезпечити стабільну взаємодію з користувачем.
- **Обробка динамічного вмісту:** Applitools вміло керує динамічним вмістом, таким як нові дані чи оновлення, мінімізуючи ризик помилкових спрацьовувань під час тестування.
- **Комплексна візуальна аналітика:** отримуйте доступ до детальних звітів і аналітичних даних, що дозволить вам зрозуміти візуальні зміни з часом і підвищити ефективність тестування.
- **Автоматичне вирішення проблем:** Applitools автоматично визначає першопричини візуальних відмінностей, спрощуючи процес вирішення та усунення будь-яких проблем.

Веб-сайт : Applitools

2 НАУКОВО-ДОСЛІДНА ЧАСТИНА

Порівняння інструментів автоматизованої генерації тестів на рівні модуля

Дані про проекти по всьому світу вказують на те що багато програмних проектів зазнають невдач і більшість із них завершуються із запізненням або перевищують бюджет через відсутність або недосконаліть процесу тестування та оцінювання якості розроблюваного програмного забезпечення. Модульне тестування – це простий, але ефективний метод покращення програмного забезпечення з точки зору якості, гнучкості та часу виходу на ринок.

Ключова ідея модульного тестування полягає в тому, що кожна частина коду потребує власних тестів, і найкращим розробником цих тестів є саме розробник, котрий написав програмне забезпечення. Однак генерування тестів для кожного блоку вручну є дуже дорогим, можливо, непомірно затратним. Автоматична генерація тестових даних має важливе значення для підтримки модульного тестування, і оскільки модульне тестування привертає все більше уваги, розробники все частіше використовують автоматизовані інструменти для створення даних модульного тестування. Однак у розробників дуже мало інформації про ефективність інструментів. У цьому розділі кваліфікаційної проведено порівняння добре відомих загальнодоступні інструменти генерації даних модульного тестування, – TestGen4j , JCrasher та JUB. Їх застосували до класів Java та оцінили ефективність на основі шкали значеннь їхніх мутації.

Для порівняння створено два додаткові набори тестів для кожного класу. Один тестовий набір містив випадкові значення, а інший містив значення, щоб задовольнити охоплення краю. Результати показали, що автоматичні інструменти генерації тестових даних створили тести з майже такими самими оцінками мутацій, як і випадкові тести.

2.1 Вибір інструментів автоматизованої генерації тестів

Важливою метою модульного тестування є перевірка правильності роботи кожної одиниці програмного забезпечення. Модульне тестування дозволяє виявити багато проблем на ранніх стадіях розробки програмного забезпечення. Комплексний набір модульних тестів, який працює разом із щоденними збірками, є важливим для успішного проекту програмного забезпечення [1]. Оскільки обчислювальна сфера використовує більш гнучкі процеси, більше покладається на розробку, керовану тестуванням, і висуває вищі вимоги до надійності програмного забезпечення, модульне тестування продовжуватиме зростати.

Однак, деякі розробники все ще не проводять багато модульного тестування. Однією з можливих причин є те, що вони не повністю розраховують час і зусилля, вартісні затрати [2]. Інша можлива причина полягає в тому, що модульні тести повинні підтримуватися на протязі життєвого циклу програмного забезпечення, а технічне обслуговування модульних тестів часто не передбачено бюджетом. Третя причина полягає в тому, що розробники можуть не знати, як розробити та реалізувати модульні тести високої якості.

Використання автоматизованих засобів модульного тестування замість ручного тестування може допомогти вирішити всі три проблеми. Інструменти автоматизованого модульного тестування можуть скоротити час і зусилля, необхідні для розробки та впровадження модульних тестів, вони можуть полегшити підтримку тестів у міру змін програми, і вони можуть інкапсулювати знання про те, як проектувати та впроваджувати високоякісні тести.

Розглянемо найбільш технічно складну частину модульного тестування, створення тестових даних. Обрано інструменти для емпіричної оцінки на основі наступних трьох факторів:

- інструмент повинен автоматично генерувати тестові значення з невеликим введенням або взагалі без нього,

- інструмент повинен тестувати класи Java,
- інструмент має бути безкоштовним і легкодоступним (наприклад, через Інтернет).

Таким вимогам відповідають три добре відомі, загальнодоступні автоматизовані інструменти:

- JCrasher, інструмент випадкового тестування, який спричиняє «збій» тестованого класу.
- TestGen4J, основним завданням якого є перевірка граничних значень аргументів, переданих до методів.
- JUB (JUnit Test Case Builder), який є фреймворком на основі шаблону Builder [7].

Використаємо ці інструменти для автоматичного створення тестів для набору класів Java.

Для контролю вручну створено два додаткових набори тестів. Набір чисто випадкових тестів був згенерований для кожного класу як порівняння «мінімальних зусиль», а також були згенеровані тести, щоб задовольнити охоплення краю на графі потоку керування.

Окремо, здійснено операцію впровадження помилок за допомогою інструменту аналізу мутацій muJava. MuJava – це автоматизований клас системи мутацій, яка автоматично генерує мутанти для класів Java і оцінює набори тестів шляхом обчислення кількості вбитих мутантів.

Крім цього виконана процедура самоконтролю, ми застосували тести до muJava і порівняли оцінки мутацій (відсоток убитих мутантів).

2.2 Розробка інструментів експериментального дослідження

Спочатку опишемо окремо кожен використовуваний інструмент генерації даних модульного тесту, а потім представимо процес, який використовується для ручного створення додаткових тестів. Відповідно

визначимо класи Java, використані у дослідженні, та сконфігуруємо інструмент тестування мутацій muJava. На звершення потрібно представити процеси, які використовувались для проведення експерименту, при врахуванні можливих загроз для достовірності.

2.3 Суб'єкти та інструменти модульного тестування

У таблиці 2.1 узагальнено три інструменти, розглянуті в цьому дослідженні. Кожен інструмент детально описаний нижче.

JCrasher – це інструмент автоматичного тестування на стійкість класів Java. JCrasher вивчає інформацію про типи методів у класах Java і створює фрагменти коду, які створюватимуть екземпляри різних типів для перевірки поведінки загальнодоступних методів із випадковими даними. JCrasher явно намагається виявити помилки, викликаючи збій тестованого класу, тобто викидаючи неоголошений виняток часу виконання. Незважаючи на те, що цей підхід обмежений випадковістю вхідних значень, він має перевагу в тому, що він повністю автоматичний. Від розробника не потрібні жодні вказівки.

TestGen4J – автоматично генерує тестові випадки JUnit з файлів класів Java. Основна його мета полягає у виконанні граничних значень аргументів, переданих методам. Він використовує правила, записані в настроюваний користувачем файл XML, який визначає граничні умови для типів даних. Тестовий код відокремлюється від тестових даних за допомогою JTestCase1.

JUB (конструктор тестових випадків JUnit) – тестовий приклад JUnit, структура генератора, що супроводжується низкою специфічних розширень IDE. Ці розширення (інструменти, плагіни тощо) викликаються з IDE і повинні зберігати згенерований тестовий код у репозиторії вихідного коду, який адмініструється IDE.

2.4 Додаткові набори тестів

Для контрольного порівняння створено два додаткові набори тестів для кожного класу вручну з обмеженою підтримкою інструментів. Тестування з випадковими значеннями широко розглядається стратегія тестування «найслабших зусиль», і найбільш очікувано, що інструмент генератора даних модульного тестування принаймні працюватиме краще, ніж генерація випадкових значень.

Розроблено спеціальний інструмент, який генерував випадкові тести у два етапи. Для кожного тесту інструмент довільно вибирав метод із класу для перевірки. (Кількість методів у кожному класі наведено в таблиці 2.2. Потім інструмент випадковим чином генерував значення для кожного параметра для цього методу. Інструмент не аналізував класи – методи та параметри були жорстко закодовані в таблиці в інструменті. Створено таку саму випадкову кількість тестів для кожного предметного класу, як інструмент, який створив найбільше тестів для цього класу. Для всіх предметних класів JCrasher створив найбільшу кількість тестів, тому в дослідженні була така ж кількість випадкових тестів, як у JCrasher.

Обрано тестовий критерій у якості форми для другого контролю. Формальні критерії тестування широко розповсюджені, але лише частково використовуються в промисловості. Обрано один із найслабших і найосновніших критеріїв тестування: покриття країв на графах потоку керування. Вручну створено графи потоку керування для кожного методу в кожному класі, а потім розроблено базу вхідних даних, щоб охопити кожне ребро на графіках.

2.5 Перевірка Java класів.

У таблиці 2.2 перелічено класи Java, використані в цьому експерименті. BoundedStack – це невелика реалізація стека фіксованого розміру з веб-сайту інструменту Eclat. Інструмент взято з проекту MuClipse, версії модуля muJava для Eclipse. Вузол — це змінний набір рядків, який є невеликою

частиною системи публікації/підписки. Черга – це змінювана, обмежена структура даних FIFO фіксованого розміру. Recipe також взято з проекту MuClipse; це клас javabean, який представляє реальний об’єкт Recipe. Twelve – інший зразок рішення. Він намагається поєднати три цілих числа за допомогою арифметичних операторів, щоб обчислити рівно дванадцять. VendingMachine – моделює простий автомат для продажу шоколадних цукерок.

Таблиця 2.1 – Інструменти автоматизованого модульного тестування

Ім'я	Версія	Вхідні дані	Інтерфейс
JCrasher	0.1.9 (2004)	Вихідний файл	Плагін Eclipse
TestGen4J	0.1.4-альфа (2005)	Файл Jar	Командний рядок (Linux)
JUB	0.1.2 (2002)	Вихідний файл	Плагін Eclipse

- Таблиця 2.2 – Використані класи предметів

Ім'я	LOC	методи
BoundedStack	85	11
Інвентар	67	11
Вузол	77	9
Черга	59	6
Рецепт	74	15
TrashAndTakeOut	26	2
Дванадцять	94	1
Торговий автомат	52	6
Всього	534	61

2.6 Інструмент тестування мутацій MuJava

Нашим основним критерієм вимірювання тестових наборів у цій роботі є їх здатність знаходити недоліки. MuJava використовується для

впровадження, засівання дефектів (мутантів) у класи та оцінки кількості мутантів, які вбиває кожен тестовий набір.

MuJava – система мутації для класів Java. Він автоматично генерує мутанти як для традиційного тестування на мутації, так і для тестування на мутації на рівні класу. MuJava може тестувати окремі класи та пакети кількох класів. Тести надаються користувачами як послідовності викликів методів до тестованих класів, інкапсульованих у методи в окремих класах.

MuJava створює об'єктно-орієнтовані мутанти для класів Java відповідно до 24 операторів, які включають об'єктно-орієнтовані оператори. Мутанти рівня методу (традиційні) засновані на селективному операторі, встановленому Offutt_et_al. Після створення мутантів muJava дозволяє тестувальнику вводити та запускати тести, а також оцінює охоплення мутаціями тестів. У muJava тести для класу, що тестується, закодовані в окремих класах, які викликають методи в класі, що тестується. Мутанти створюються та страчуються автоматично.

2.7 Порівняльне експериментальне дослідження інструментів JCrasher, TestGen4J, JUB.

Рисунок 2.1 ілюструє експериментальний процес. Класи Java представлені крайнім лівим полем P. Кожен із трьох автоматизованих інструментів (JCrasher, TestGen4J, JUB) використовувався для створення наборів тестів (Test Set JC, ...). Потім вручну створюються випадкові тести, а потім тести покриття країв.

MuJava була розроблена та розроблена до JUnit та інших широко використовуваних інструментів тестування, тому вона має власні синтаксичні вимоги. Кожен тест muJava має бути реалізований відкритим методом, який повертає рядок, котрий інкапсулює результат тесту. Таким чином, багато тестів із трьох інструментів модифіковані для виконання з muJava. Ця модифікація проілюстрована на малюнку 2.2.

Потім muJava була використана для генерації мутантів для кожного

класу суб’єктів і проведення всіх п’яти тестових наборів проти мутантів. Це призвело до п’яти оцінок мутацій для кожного предметного класу (всього 40 оцінок мутацій). Оскільки muJava відокремлює оцінки для традиційних операторів мутації від операторів мутації класу, ці оцінки зберегли окремо.

Таблиця 2.3 – Класи та мутанти

Класи	Мутанти		
	Традиційний	Клас	Всього
Обмежений стек	224	4	228
Інвентар	101	50	151
Вузол	18	4	22
Черга	117	6	120
Рецепт	101	26	127
Вивіз сміття	104	0	104
Дванадцять	234	0	234
Торговий автомат	77	7	84
Всього	976	97	1073

2.8 Загрози достовірності результатів

Як і в будь-якому дослідженні, яке передбачає конкретні програми або класи, немає гарантії, що використані класи є репрезентативними для загальної картини. Не існує загальної теорії того, як вибрати репрезентативні класи для емпіричних досліджень, або скільки класів нам може знадобитися. Ця загальна проблема негативно впливає на здатність застосовувати інструменти статистичного аналізу та знижує зовнішню валідність багатьох емпіричних досліджень програмної інженерії, включно з цим.

Інше питання полягає в тому, чи є три використовувані інструменти репрезентативними. Використані безкоштовні, загальнодоступні інструменти, які генерували б тести, їхня кількість очікувано мала.

Інша можлива проблема з внутрішньою валідністю може полягати в

ручних кроках, які потрібно було застосувати. Тести з трьох інструментів потрібно було перекласти на тести muJava. Ці зміни стосувалися лише структури тестових методів і не вплинули на значення, тому малоймовірно, що ці переклади вплинули на результати. Випадкові тести та тести покриття країв були створені вручну за допомогою певного інструменту. Ці тести були створені без знання мутантів, щоб уникнути будь-якої упередженості.

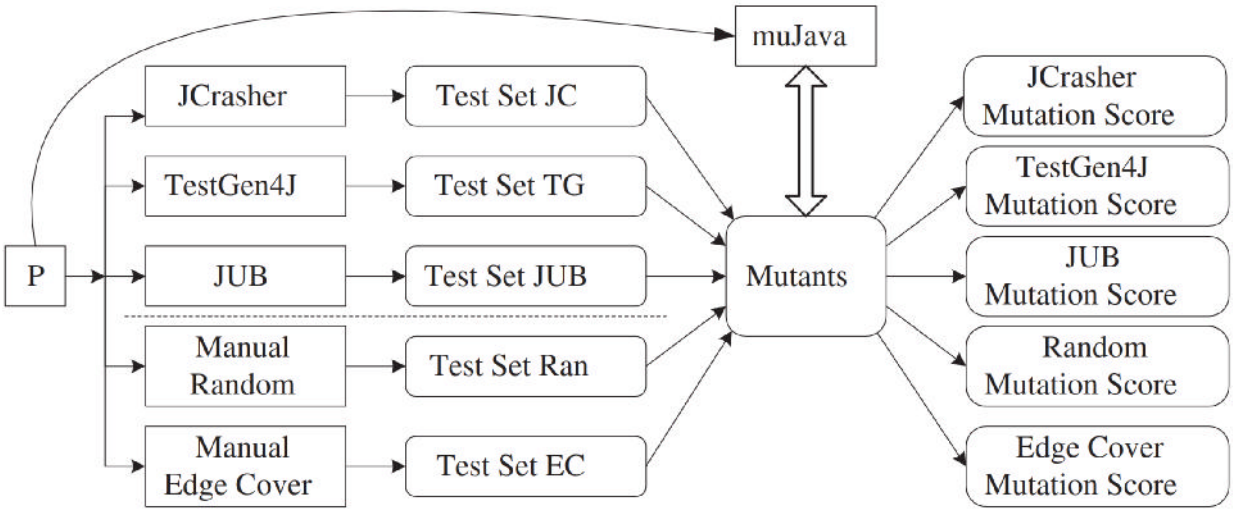


Рисунок 2.1 – Реалізація експерименту

```
public void test242() throws Throwable
{
    try {
        int i1 = 1;
        int i2 = 1;
        int i3 = -1;
        Twelve.twelve (i1, i2, i3);
    }
    catch (Exception e) }
          { dispatchException (e); }
}
```

⟹

```
public String test242() throws Throwable
{
    try {
        int i1 = 1;
        int i2 = 1;
        int i3 = -1;
        return Twelve.twelve (i1, i2, i3);
    }
    catch (Exception e) }
          { return e.toString(); }
}
```

Рисунок 2.2 – Перетворення в тести MuJava

2.9 Результати порівняльних досліджень

Кількість мутантів для кожного класу показано в таблиці 2.3. MuJava генерує традиційні (рівень методу) і класові мутанти окремо. Традиційні оператори зосереджені на окремих операторах, а оператори класів зосереджені на зв’язках між класами, особливо в ієрархіях успадкування. Вузол виглядає аномально через малу кількість мутантів. Node має лише 18 традиційних мутантів на 77 рядків коду. Проте твердження розподілені між

дев’ятьма, здебільшого дуже невеликими, методами. Більшість методів дуже короткі (чотири мають лише один оператор), вони не використовують арифметичні, зсувні чи логічні оператори, у них немає присвоєння та лише кілька операторів прийняття рішення. Таким чином, Node має кілька місць, де можна застосувати оператори мутації muJava.

Результати виконання п’яти наборів тестів під muJava наведені в таблицях 2.4–2.7. Таблиця 2.4 показує кількість тестів у кожному наборі тестів і показники мутацій традиційних мутантів у відсотках убитих мутантів. Рядок «Усього» містить суму тестів для предметних класів і оцінку мутації для всіх предметних класів. Як видно, випадкові тести показали кращі результати, ніж TestGen і тести JUB, але не так добре, як тести JCrasher. Проте тести на покриття країв вбивають на 24% більше мутантів, ніж накращий інструмент, за меншої кількості тестів.

Таблиця 2.5 показує ті самі дані для мутантів рівня класу. Оцінки трьох інструментів відрізняються більше, і JCrasher убив на 12% більше мутантів рівня класу, ніж у випадкових тестах. Однак тести на охоплення країв все ще набагато ефективніші, вбиваючи на 21% більше мутантів, ніж найпотужніший інструмент JCrasher.

Таблиця 2.6 поєднує бали з таблиць 2.5 і 2.4 як для традиційних, так і для мутантів рівня класу. Знову ж таки, тести JUB є найслабшими, JCrasher, TestGen і випадкові тести досить близькі, а тести покриття меж вбивають набагато більше мутантів; На 24% більше, ніж у тестах JCrasher.

У таблиці 2.7 зведені дані для всіх предметних класів для кожного комплекту тестів. Оскільки кількість тестів сильно відрізнялася, потрібно вирішити наскільки ефективним є кожен метод генерації тестів. Щоб наблизити ефективність, обчислили кількість мутантів, убитих за тест. Відповідно, що тести на покриття країв були найвищими з результатом 6,5. TestGen згенерував найменшу кількість тестів і став другим за ефективністю, 5.2.

Таблиця 2.4 – Вбиті традиційні мутанти

Клас	JCrasher Тести/ % убитих		TestGen Тести/ % убитих		Test JUB Тести/ % убитих		Покриття країв Тести/ % убитих		Випадковий Тести/ % убитих	
Обмежений стек	21	22,8	10	23,7	11	17,4	17	46,9	21	22,3
Інвентар	20	68,3	10	60,4	11	68,3	19	60,4	20	65,3
Вузол	16	50,0	16	50,0	9	33,3	14	61,1	16	50,0
Черга	9	39,3	5	34,2	9	41,0	10	51,3	9	15,4
Рецепт	29	78,2	10	70,3	15	35,6	22	75,2	29	57,4
TrashAndTakeOut	13	64,4	2	30,8	2	30,8	5	70,2	13	59,6
Дванадцять	27	35,0	1	0,0	1	0,0	12	86,3	27	35,0
Торговий автомат	14	10,4	5	9,1	6	9,1	10	75,3	14	10,4
Всього	149	42,1	59	28,0	64	24,3	109	66,2	149	36,2

Таблиця 2.5 – Мутанти класів вбиті

Клас	JCrasher Тести/ % убитих		TestGen Тести/ % убитих		Test JUB Тести/ % убитих		Покриття країв Тести/ % убитих		Випадковий Тести/ % убитих	
Обмежений стек	21	25,0%	10	25,0%	11	25,0%	17	75,0%	21	25,0%
Інвентар	20	44,0%	10	36,0%	11	36,0%	19	76,0%	20	40,0%
Вузол	16	25,0%	16	25,0%	9	25,0%	14	25,0%	16	25,0%
Черга	9	33,3%	5	33,3%	9	33,3%	10	33,3%	9	16,7%
рецепт	29	73,1%	10	53,8%	15	11,0%	22	80,8%	29	38,5%
TrashAndTakeOut	13	N/A	2	N/A	2	N/A	5	N/A	5	N/A
Дванадцять	27	N/A	1	N/A	1	N/A	12	N/A	27	N/A
Торговий автомат	14	0,0%	5	0,0%	6	0,0%	10	0,0%	14	0,0%
Всього	149	46,4%	59	37,1%	64	25,6%	109	67,0%	149	34,0%

Таблиця 2.6 – Всього вбито мутантів

Клас	JCrasher Тести/ % убитих		TestGen Тести/ % убитих		Test JUB Тести/ % убитих		Покриття країв Тести/ % убитих		Випадковий Тести/ % убитих	
Обмежений стек	21	22,8%	10	23,2%	11	17,5%	17	47,4%	21	22,4%
Інвентар	20	60,3%	10	52,3%	11	57,6%	19	65,6%	20	57,0%
Вузол	16	45,5%	16	45,5%	9	31,8%	14	54,5%	16	45,5%
Черга	9	37,4%	5	32,5%	9	39,0%	10	48,8%	9	14,6%
Рецепт	29	77,2%	10	66,9%	15	30,6%	22	76,4%	29	53,5%
TrashAndTakeOut	13	64,4%	2	30,8%	2	30,8%	5	70,2%	13	59,6%
Дванадцять	27	35,0%	1	0,0%	1	0,0%	12	86,3%	27	35,0%
Торговий автомат	14	9,5%	5	8,3%	6	8,3%	10	69,0%	14	9,5%
Всього	149	42,5%	59	28,8%	64	24,4%	109	66,3%	149	36,0%

JCrasher і випадкові тести були найменш ефективними; вони згенерували багато тестів без особливої очевидної вигоди, що додає тягаря для розробників, які повинні оцінювати результати кожного тесту.

Малюнок 3 ілюструє загальний відсоток мутантів, знищених кожним набором тестів на стовпчастій діаграмі. Різниця між тестами покриття країв та іншими помітна.

Таблиці 2.8 і 2.9 дають більш детальний погляд на бали для кожного оператора мутації. Оператори мутації описані на сайті muJava. Не було створено жодних мутантів для п’яти традиційних операторів LOR, LOD, AODS, SOR, ASRS, або для більшості операторів класу IOP,IHI, IHD, IOR, PCC, PCI, PPD, ISI, ISD, PMD, IPC, PNC, PCD, PRV, OMR, OMD, OAN, EOA або EOC, тому вони не відображаються. Тести на покриття країв мали найвищі бали для всіх операторів мутації. Жоден із наборів тестів не показав особливого успіху на мутантах арифметичних операторів, оператори, імена яких починаються з літери «A».

2.10 Аналіз результатів дослідження інструментів автоматичного тестування

Є неофіційні докази, що під час ручного створення тестів на знищення мутантів дуже легко вбити від 40% до 50% мутантів. Ця примітка підтверджується цими даними, де випадкові значення досягли середньої оцінки мутацій у 36%. Ці три інструменти спрацювали трохи краще, ніж випадкове тестування. Результат JCrasher був трохи кращим, загалом 6,5%, TestGen був гіршим, на 7,2% нижчий показник мутацій, а JUB був ще гіршим, на 11,6% нижчий показник мутацій.

Слід зауважити по результатам з таблиці 2.6, що бали для «Торгового автомату» набагато нижчі для всіх наборів тестів, за винятком покриття країв. Інші чотири оцінки мутацій нижчі за 10%. Ймовірно, причина полягає у відносній складності «Торгового автомату». Він має кілька предикатів з кількох речень, які визначають більшість його поведінки:

```
(монета!=10 && монета!=25 && монета!=100) (кредит >= 90)
(кредит < 90 || stock.size() <= 0) (stock.size() >= MAX)
```

MuJava створює десятки мутантів на основі цих предикатів, і здебільшого випадкові значення, створені трьома інструментами в цьому дослідженні, мають невеликий шанс вбити цих мутантів.

Ще одне цікаве спостереження з таблиці 2.6 полягає в тому, що бали для «Обмеженого стеку» були другими найнижчими для всіх наборів тестів, за винятком покриття країв, де воно було найнижчим. Різниця в цьому класі полягає в тому, що лише два з одинадцяти методів мають параметри. Три інструменти тестування значною мірою залежать від сигнатури методу, тому менша кількість параметрів може означати слабші тести.

Ще один висновок полягає в тому, що JCrasher отримав найвищий бал мутації серед трьох інструментів. Ми перевірили тести створені JCrasher, і дійшли висновку, що це тому, що JCrasher використовує недійсні значення для спроби «збити» клас, як показано на малюнку 2.4. JUB генерує лише

тести, які використовують 0 для цілих чисел і нуль для загальних предметів, а TestGen4J генерує «звичайні» вхідні дані, такі як пробіли та порожні рядки. JCrasher, звичайно, також створив набагато більше тестів, ніж два інших інструменти.

```
public void test18() throws Throwable
{
    try
    {
        String s1 =
            "~!@$$%^&*()_+{}|[]';:/.,<>?'-=";
        Node n2 = new Node();
        n2.disallow (s1);
    }
    catch (Exception e)
        {dispatchException (e);}
}
```

Рисунок 2.4 – Тест JCrasher

Показник ефективності в таблиці 2.7 дещо зміщений через мутацію, оскільки багато мутантів дуже легко вбити. Досить часто перші кілька тестів вбивають багато мутантів, а наступні тести вбивають менше мутантів, що призводить до свого роду зменшення прибутку.

Розділивши дані для традиційних і класових мутантів у таблицях 2.4 і 2.5. Тести JCrasher мали дещо вищу оцінку мутацій для класових мутантів, а тести TestGen, JUB і випадкові тести були трохи нижчими. Існувала невелика різниця в оцінках мутацій для тестів на охоплення країв. Однак ми не можемо зробити якісь загальні висновки з цих даних.

Також знайдено кореляцію між кількістю тестів для кожного класу та балом мутації. За допомогою тестів JCrasher і Random тестів найбільша кількість тестів (для Recipe в обох випадках) призвела до найвищих оцінок мутацій. Однак інші три набори тестів не показали такої кореляції, і не спостерігається кореляція з найменшою кількістю тестів. Насправді, граничне покриття дало найменшу кількість тестів для класу «Дванадцять», але мало найвищий бал мутації (83,6%). Знову ж таки, не можна зробити загальних висновків із цих даних.

2.11 Емпіричне порівняння методів автоматизованої генерації та класифікації

Застосуємо емпіричне порівняння методів автоматизованої генерації та класифікації для об’єктно-орієнтованого модульного тестування [4]. У дослідженні порівнювалися пари методів генерації тестів, заснованих на випадковій генерації або символьному виконанні, і методів класифікації тестів, заснованих на неперехоплених винятках або операційних моделях.

Зокрема, порівняємо два інструменти, які реалізують автоматизовану генерацію тестів, – Eclat, який використовує випадкову генерацію, і їхній власний інструмент Symclat, котрий використовує символічну генерацію. Інструменти також забезпечують класифікацію тестів на основі операційної моделі та моделі неперехоплених винятків [15].

Результати показали, що два інструменти доповнюють один одного у виявленні недоліків.

Таблиця 2.7 – Зведені дані

Інструмент	Тести	% Убитих			Ефективність
		Традиційний	Клас	Всього	Убитий / Тести
JCrasher	149	42,1%	46,4%	42,5%	3.1
TestGen	59	28,0%	37,1%	28,8%	5.2
JUB	64	24,3%	25,6%	24,4%	4.1
Покриття країв	109	66,2%	67,0%	66,3%	6.5
Випадковий	149	36,2%	34,0%	36,0%	2.6

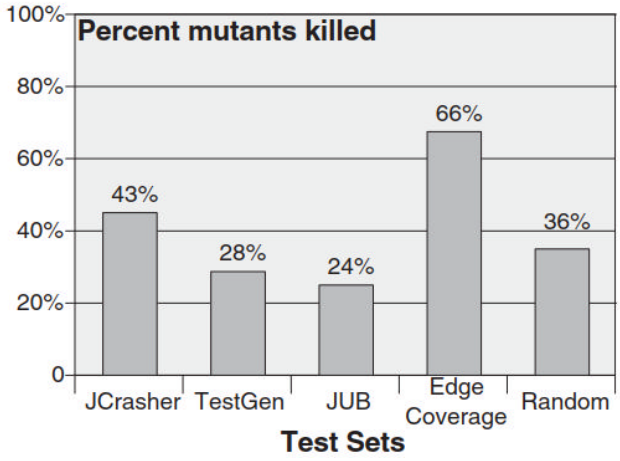


Рисунок 2.3 – Загальний відсоток мутантів, знищених кожною тестовою серією

У подібному дослідженні інструментів тестування статичного аналізу Rutar_et_al. [17] порівняли п’ять тестових інструментів статичного аналізу з п’ятьма проектами з відкритим кодом. Їхні результати показали, що жоден із п’яти інструментів не враховує жодного іншого, з точки зору можливості пошуку несправностей. Запропоновано мета-інструмент для об’єднання та кореляції можливостей цих п’яти інструментів. Існують тематичні дослідження [21], котрі застосовують три статичні інструменти пошуку несправностей, а також перевірку коду та ручне тестування до кількох промислових проектів. Такі дослідження показали, що статичні інструменти переважно виявляли інші помилки, ніж ручне тестування, але підмножина помилок була виявлена в оглядах. Запропоновано поєднання цих трьох типів технік і методів.

Першим дослідницьким інструментом, який впроваджував автоматизоване модульне тестування, був Godzilla, який був частиною набору інструментів Mothra mutation [5]. Godzilla використовував символічну оцінку для автоматичної генерації тестів на знищення мутантів, а пізніша версія включала динамічну символічну оцінку та процедуру динамічного зменшення домену для створення тестів [12]. Більш сучасним інструментом, який використовує дуже подібні методи, є інваріантний детектор Daikon [6]. Це доповнює тип символічної оцінки, який Godzilla використовував з

інваріантами програми, інновація, яка робить процес створення тесту більш ефективним і масштабованим. Daikon аналізує значення, які програма обчислює під час роботи, і повідомляє про властивості, які були істинними під час спостережуваних виконань.

Eclat, керований моделлю інструмент випадкового тестування, використовує Daikon для динамічного виведення операційної моделі, що складається з набору ймовірних інваріантів програми [15]. Для тестування Eclat потрібні класи, а також приклад їх використання, наприклад програма, яка використовує класи, або невеликий початковий набір тестів. Оскільки результат Eclat може залежати від початкових значень, його не можна порівнювати безпосередньо з іншими інструментами в цьому дослідженні.

Іншим інструментом, заснованим на Daikon, є Jov [22], який представляє підхід операційного порушення для генерації та вибору модульних тестів, підхід чорної скриньки, який не потребує специфікацій. Цей підхід динамічно генерує операційні абстракції від виконання існуючого набору модульних тестів. Ці операційні абстракції керують інструментами генерації тестів для створення тестів, які їх порушують. Підхід вибирає для перевірки тести, які порушують операційні абстракції.

Таблиця 2.8 – Оцінки мутацій для кожного оператора традиційної мутації

Традиційний	Мутанти	JCrasher	TestGen	JUB	Покриття країв	Випадковий
AORB	56	32%	21%	30%	66%	29%
AORS	11	46%	27%	36%	55%	27%
AOIU	66	46%	32%	17%	79%	36%
AOIS	438	28%	24%	22%	53%	22%
AODU	1	100%	100%	100%	100%	100%
ROR	256	61%	25%	17%	79%	57%
КОР	12	33%	25%	25%	58%	33%
COD	6	33%	33%	17%	50%	33%

COI	4	75%	75%	50%	75%	75%
LOI	126	53%	48%	44%	80%	48%
Всього	976	42%	28%	24%	66%	36%

Таблиця 2.9 – Оцінки мутацій за оператором мутації класу

Клас	Мутанти	JCrasher	TestGen	JUB	Покриття країв	Випадковий
IOD	6	50%	50%	50%	50%	50%
JTI	20	95%	50%	25%	100%	55%
JTD	6	100%	100%	0%	100%	50%
JSI	13	0%	0%	0%	23%	0%
JSD	4	0%	0%	0%	50%	0%
JID	2	0%	0%	0%	0%	0%
JDC	6	83%	66%	83%	83%	67%
EAM	28	0%	0%	0%	57%	0%
EMM	12	100%	100%	100%	100%	100%
Всього	97	46%	36%	26%	69%	34%

Ці тести демонструють нову поведінку, яка ще не була вивчена існуючими тестами. JOV інтегрує використання Daikon і Parasoft Jtest [16] (комерційний інструмент тестування Java). Agitar Test Runner — це комерційний інструмент для тестування, який частково базувався на Daikon та Godzilla.

2.12 Висновки порівняльного дослідження

Порівнювалися три безкоштовні, загальнодоступні інструменти модульного тестування на основі їхніх можливостей пошуку несправностей. Помилки були впроваджені в класи Java за допомогою автоматизованого інструменту мутації, а тести інструментів порівнювалися з випадковими тестами, створеними вручну, і тестами покриття меж.

Наші висновки свідчать про те, що ці інструменти створюють тести, виявляють помилки не у повному обсязі. Оскільки очікування користувачів щодо надійного програмного забезпечення продовжують зростати, а гнучкі процеси та розробка, керована тестуванням, продовжують отримувати визнання в галузі, модульне тестування стає все більш важливим. На жаль, у розробників програмного забезпечення є невеликий вибір високоякісних інструментів для створення тестових даних, тоді як тестування на основі критеріїв домінувало.

Вже більше двох десятиліть створення промислових генераторів тестових даних не задовольняє потреб користувачів, а це є основою для створення тестів, котрі у повній мірі задовольняють критерії тестування. Інструменти, які оцінюють покриття, доступні, але вони не вирішують найскладнішої проблеми, – генерації тестових значень. Це дослідження доводить, що настав час для створення програмних засобів генерації тестових даних на основі критеріїв для міграції в інструменти, які розробники можуть використовувати з мінімальними знаннями теорії тестування програмного забезпечення.

Ці інструменти порівнювали лише з одним критерієм тестування, охопленням країв на контрольних графах потоку. Широко відомо, що це один із найпростіших, найдешевших і найменш ефективних критеріїв тестування. Неофіційний досвід із вбивством мутантів вручну вказує на те, що оцінки приблизно в 40% досягти тривіально, а 70% досить легко досягти за допомогою невеликого ручного аналізу структури класу. Це спостереження підтверджується цим дослідженням, у якому випадкові значення досягли рівня 40%, а тести на покриття країв досягли рівня 70%. Однак оцінки мутацій від 80% до 90% часто досить важко досягти за допомогою тестів, створених вручну. Це має бути можливим за більш жорстких критеріїв, таких як аналіз основних маршрутів, використання або покриття на основі логіки.

Незважаючи на те, що тестування споживає більше половини ресурсів

індустрії програмного забезпечення, в галузі інформатики на теперішній час мало уваги приділяють курсам тестування програмного забезпечення. Це дослідження привело нас до висновку, що настав час викладати більше знань про тестування програмного забезпечення для фахівців галузі інформатики та програмної інженерії.

3 ТЕХНОЛОГІЧНА ЧАСТИНА

3.1 Тестування програмного забезпечення – загальний огляд

Тестування програмних продуктів – це процес перевірки якості, функціональності та продуктивності програмного забезпечення перед запуском. Щоб провести тестування програмного забезпечення, тестери або взаємодіють із програмним забезпеченням вручну, або виконують сценарії тестування, щоб знайти помилки та помилки, гарантуючи, що програмне забезпечення функціонує відповідним чином. Тестування програмних продуктів також здійснюється, щоб перевірити, чи виконується бізнес-логіка, чи є прогалини у вимогах, які потребують негайного вирішення.

Тестування програмних продуктів є важливою часткою життєвого циклу розробки програмного забезпечення. Без цього помилки, що ушкоджують програму, можуть залишитися непоміченими та негативно вплинути на прибутки. З часом, коли програми стають все більш складними, діяльність з тестування програмного забезпечення також розвивається, з’являється багато нових методів і підходів.

Шлях розробки програмного забезпечення доволі складний, і продукти завжди можуть бути вразливими до помилок і дефектів. Перед виходом на ринок необхідно переконатися, що програмне забезпечення працює згідно вимог замовника. Саме тому тестування програмного забезпечення є надзвичайно важливим етапом у створенні якісних програмних продуктів.

Кінцевою метою тестування програмного забезпечення завжди є виявлення помилок і дефектів. Сучасне програмне забезпечення складається з високо взаємопов’язаних компонентів, які повинні бездоганно працювати разом, щоб забезпечити заплановану функціональність. Один зламаний компонент може створити ефект хвилі та зламати всю програму. Чим швидше буде виправлено зламаний код, тим менший вплив. Хороший процес тестування забезпечує своєчасну поставку високоякісного та надійнішого продукту.

3.2 Підтримка та покращення якості програмної продукції

Якість продукту — це більше, ніж просто відсутність помилок. Коли говорять про якість програмного продукту, ведуть мову про характеристики, котрі відповідають очікуванню клієнтів, а по можливості їх перевищують. Очікується, що програма виконуватиме ті функції, для яких вона була призначена, але вона може досягти статусу «високої якості», лише у тому випадку якщо вона перевершує ці очікування. З цієї точки зору тестування програмного забезпечення є ключем до якості з таких причин:

Підтримка програмних продуктів:

- Регресійне тестування, щоб переконатися, що якість завжди підтримується незмінним стандартом після додавання нового коду
- Періодичне тестування, – зручності використання, сумісності та безпеки, щоб гарантувати, що ці аспекти програмного забезпечення ретельно перевіряються
- Інформаційне забезпечення про якість продукту, відповідно команда розробників постійно оновлює документацію

Поліпшення програмних продуктів за експлуатаційний період:

- Виявлення та усунення помилок, щоб зробити програмне забезпечення надійнішим
- Визначення сфери для оптимізації та вдосконалення
- Тестування програмного забезпечення для забезпечення його роботи належним чином
- Визначити, що програмне забезпечення відповідає відгукам користувачів
- Забезпечення сумісності програмного забезпечення між платформами та середовищами

Результатом ретельного тестування програмного забезпечення є підвищення довіри клієнтів. Хоча нереалістично очікувати, що програмне забезпечення буде повністю вільним від помилок, наявність стабільного, надійного продукту, який постійно відповідає потребам клієнтів, зрештою

призведе до позитивного досвіду користувача в довгостроковій перспективі. Застосування найкращих методів управління якістю програмного забезпечення гарантує зацікавленим сторонам і клієнтам, що вони можуть покладатися на продукт, який був багаторазово протестований.

Фінансове, медичне, юридичне програмне забезпечення або будь-які типи програмного забезпечення в полі YMYL (Your Money Your Life) працюють з конфіденційною інформацією. Програмні програми, створені для цих сфер, не можуть дозволити собі зазнавати збоїв, пошкодження даних або системних збоїв, навіть у невеликих масштабах, оскільки це вплине на життя багатьох людей. Помилки в цьому програмному забезпеченні можуть завдати незворотної шкоди та поставити компанію під загрозу судового розгляду. Тестування програмного забезпечення покликане захистити компанії від такого ризику.

3.3 Види тестування ПЗ

Різні типи тестування програмного забезпечення можна класифікувати за кількома категоріями на основі цілей тестування, стратегії тестування та результатів. Наразі існує два основних типи тестування програмного забезпечення, які часто використовують спеціалісти із забезпечення якості, зокрема:

- **Функціональне тестування:** це тип тестування програмного забезпечення для перевірки того, чи програма забезпечує очікуваний результат.
- **Нефункціональне тестування:** це тип тестування програмного забезпечення для перевірки того, чи нефункціональні аспекти програми (наприклад, стабільність, безпека та зручність використання) працюють належним чином.

Ці загальні терміни охоплюють широкий діапазон типів тестування, кожен з яких служить лише певній меті. Як приклад, існує 3 основних типи функціонального тестування:

- **Модульне тестування**: тип тестування окремого модуля в програмі
- **Тестування інтеграції**: тип тестування, що виконується на групах програмних одиниць, щоб побачити, як вони працюють разом
- **Приймальне тестування**: тип тестування для оцінки програми за реальними сценаріями

Подібним чином у розділі «Нефункціональне тестування» також існує багато поширених типів тестування, кожен з яких має різні цілі та стратегії:

- **Тестування безпеки**: тестування, яке перевіряє, чи програмне забезпечення є безпечним і захищає від несанкціонованого доступу чи загроз.
- **Тестування продуктивності**: тестування, яке оцінює, наскільки добре працює програмне забезпечення щодо швидкості, стабільності та використання ресурсів.
- **Тестування навантаження**: тип тестування продуктивності, який оцінює, як програмне забезпечення справляється з очікуваними та піковими навантаженнями.
- **Тестування зручності використання**: Тестування, яке визначає, наскільки зручним і простим у використанні є програмне забезпечення.
- **Тестування сумісності (або Крос-браузерне тестування)**: тестування, яке забезпечує правильну роботу програмного забезпечення різні платформи, пристрої чи середовища.

Рішення про використання цих типів тестів програмного забезпечення залежить від сценаріїв тестування, доступності ресурсів і бізнес-вимог.

3.4 Підходи до тестування ПЗ

Фахівці з контролю якості мають 2 різні підходи до тестування програмного забезпечення: ручне тестування та автоматизоване тестування. Кожен підхід має власний набір переваг і недоліків, які тестувальники повинні ретельно враховувати, щоб максимізувати тестові ресурси.

- **Тестування вручну:** тестування програмного забезпечення людьми

вручну без будь-яких засобів автоматизації чи сценаріїв

- **Автоматизоване тестування:** тестування програмного забезпечення за допомогою інструментів або сценаріїв, які автоматично взаємодіють із програмним забезпеченням. Людині-тестеру потрібно лише виконати сценарій і дозволити йому виконувати решту тестування.

3.5 Життєвий цикл тестування програмного забезпечення

Багато ініціатив тестування програмних продуктів дотримуються процесу, широко відомого як життєвий цикл тестування програмного забезпечення – STLC.

STLC складається з 6 ключових заходів для забезпечення досягнення всіх цілей якості програмного забезпечення, як показано нижче:

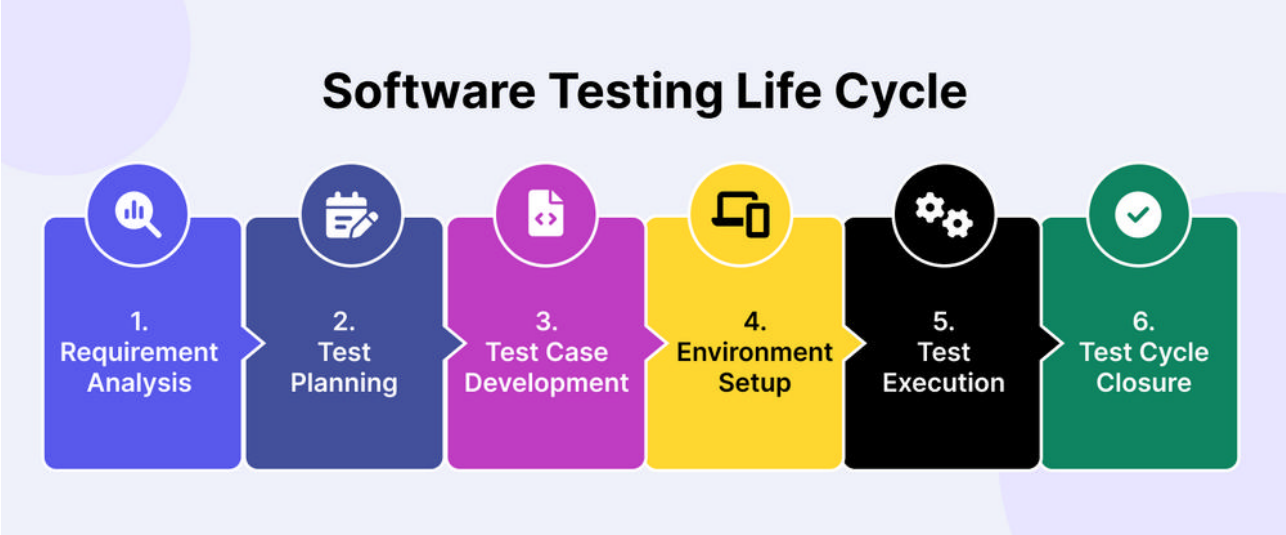


Рисунок 3.1 – Життєвий цикл тестування програмного забезпечення

3.5.1. Аналіз вимог.

На цьому етапі тестувальники програмного забезпечення працюють із зацікавленими сторонами, залученими до процесу розробки, щоб визначити та зрозуміти вимоги до тестування. Ідеї цього обговорення, об’єднані в документ «Матриця відстеження вимог» (RTM), стануть основою для побудови стратегії тестування.

Те, що розробник і тестувальник розуміють на етапі аналізу вимог, відрізняються. Розробник зосереджується на перекладі вимог у код, включаючи архітектуру, методи проектування та технології, тоді як тестер перевіряє можливість перевірки коду. Вони визначають, як код можна розбити на менші частини, сценарії та тестові приклади.

Щоб забезпечити найвищий рівень розуміння між зацікавленими сторонами, групи контролю якості можуть використовувати тестування BDD, гнучкий підхід до тестування програмного забезпечення, де цінується простота. Забезпечення можливості тестування має вирішальне значення на етапі проектування, щоб уникнути неоднозначних вимог, які можуть призвести до недійсних тестів програмного забезпечення.

Після цього тестувальники та розробники повинні співпрацювати, щоб зрозуміти доцільність впровадження бізнес-вимог. Якщо ці вимоги не можуть бути виконані в рамках заданих обмежень або ресурсів, їм потрібно буде обговорити з діловою стороною (бізнес-аналітиком, керівником проекту та/або клієнтом), щоб внести корективи або знайти альтернативні рішення.

3.5.2. Планування тестування

Після ретельного аналізу створюється план тестування. Планування тестування передбачає узгодження з відповідними зацікавленими сторонами стратегії тестування:

- **Цілі тестування**: визнають такі атрибути, як функціональність, зручність використання, безпека, продуктивність і сумісність.
- **Результат і кінцеві результати**: документують тестові сценарії, тестові випадки та тестові дані, які потрібно створювати та контролювати.
- **Обсяг тестування**: визначають, які області та функції програми буде перевірено (в межах), а які ні (поза межами).
- **Ресурси**: Оцінюють витрати на інженерів-випробувачів, засоби ручного/автоматичного тестування, середовища та тестові дані.

- **Часова шкала**: встановлюють очікувані етапи для конкретних дій, пов’язаних із тестуванням, разом із розробкою та розгортанням.
- **Підхід до тестування**: Оцінюють методи тестування (тестування білого/чорного ящика), рівні тестування (модуль, інтеграція та наскрізне тестування) і типи тестів (регресія, перевірка розумності).

Для більшого контролю над проектом тестувальники програмного забезпечення можуть додати план на випадок непередбачених обставин, щоб скоригувати змінні на випадок, якщо проект рухається в неочікуваному напрямку.

3.5.3. Розробка тестового випадку

Після визначення сценаріїв і функцій, які потрібно перевірити, створюють тестові випадки.

Для випадків тестування вручну, інструменти керування, такі як Xray, можна використовувати для запису подробиць того, що було виконано, результатів, висновків і пропозицій для розробників щодо відтворення цих помилок.

Для автоматизованих тестів доступні інтуїтивно зрозумілі інтерфейси, надані такими інструментами, як Katalon, Ranorex або TestComplete. Варіанти з відкритим вихідним кодом, такі як Selenium, Cypress і Playwright, також популярні для створення власних фреймворків.

Якщо невідомо з яких тестів почати, існує список популярних тестів:

1. Тестові випадки для тестування API
2. Тестові випадки для сторінки входу
3. Тестові випадки для сторінки реєстрації
4. Тестові приклади для банківської програми
5. Тестові приклади для веб-сайту електронної комерції
6. Тестові випадки для функціональності пошуку

Вони дають гарну основу того, як підходити до тестувальнику системи:

Фактична кількість тестів, які потрібно виконати, багато в чому

залежить від складності системи, що тестується. Хороший тестер – це той, хто може придумати креативні способи зламати систему, тому, розробляючи свої тестові приклади, спробуйте поставити себе на місце людини, яка абсолютно не уявляє, як ця система працює, і знайдіть якомога більше способів помилитися як це можливо.

3.5.4. Налаштування тестового середовища

Цей крок можна виконати паралельно з розробкою тестового прикладу. Тестове середовище – це конфігурації програмного та апаратного забезпечення, у яких тестується програма, включаючи сервер бази даних, зовнішнє робоче середовище, браузер, мережу, апаратне забезпечення тощо. Команди контролю якості плануватимуть використання ресурсів для розробки тестового середовища. Така практика забезпечує ефективний розподіл ресурсів. Ось короткий контрольний список пунктів, які потрібно враховувати під час налаштування тестового середовища:

- Перевірте апаратні характеристики (ЦП, оперативна пам’ять, накопичувач).
- Перевірте залежності програмного забезпечення (операційної системи, бібліотек, фреймворків).
- Перевірте вимоги до мережі (правила брандмауера, порти, підключення).
- Переконайтеся, що тестове середовище відокремлене від робочого.
- Використовуйте сегрегацію мережі, віртуалізацію або спеціальне обладнання.
- Визначте відповідні сценарії тестування та вимоги до даних.
- Створіть репрезентативні набори тестових даних.
- Зверніть увагу на правила конфіденційності та безпеки даних.
- Встановити необхідні операційні системи та програмне забезпечення.
- Налаштуйте бази даних, веб-сервери та інші необхідні компоненти.
- Налаштуйте параметри мережі, брандмауери та заходи безпеки.

- Встановіть механізми відновлення для відновлення середовища до відомого стану.
- Впроваджуйте регулярне резервне копіювання, щоб запобігти втраті даних або проблемам конфігурації.
- Встановіть механізми відновлення для відновлення середовища до відомого стану.
- Впроваджуйте регулярне резервне копіювання, щоб запобігти втраті даних або проблемам конфігурації.

3.5.5. Виконання тесту

Маючи на увазі чіткі цілі, команда QA пише тестові випадки, тестові сценарії та готує необхідні тестові дані для виконання.

Тести можна виконувати вручну або автоматично. Ручне тестування підходить, коли потрібна людська думка та судження, тоді як автоматизоване тестування є кращим для повторюваних потоків з незначними коригуваннями. Після виконання тестів усі виявлені дефекти відстежуються та повідомляються команді розробників, яка негайно їх усуває.

3.5.6. Закриття випробувального циклу

Це завершальний етап тестування програмного забезпечення. Тестувальники програмного забезпечення збераються, щоб проаналізувати результати тестування, оцінити ефективність і задокументувати ключові висновки для подальшого використання. Дуже важливо регулярно оцінювати процес тестування програмного забезпечення командою з контролю якості, щоб контролювати всі дії з тестування на всіх етапах STLC.

3.6 Популярні моделі тестування програмного забезпечення

Еволюція режиму тестування відбувалася паралельно з еволюцією методології розробки програмного забезпечення.

3.6.1. V-модель

У минулому командам із забезпечення якості доводилося чекати фінального етапу розробки, щоб почати тестування. Якість тестування зазвичай була низькою, і розробники не могли вчасно усунути неполадки до випуску продукту.

V-модель вирішує цю проблему, залучаючи тестувальників до кожної фази розробки. Кожному етапу розробки призначається відповідний етап тестування. Ця модель добре працює з майже застарілим методом водоспаду.

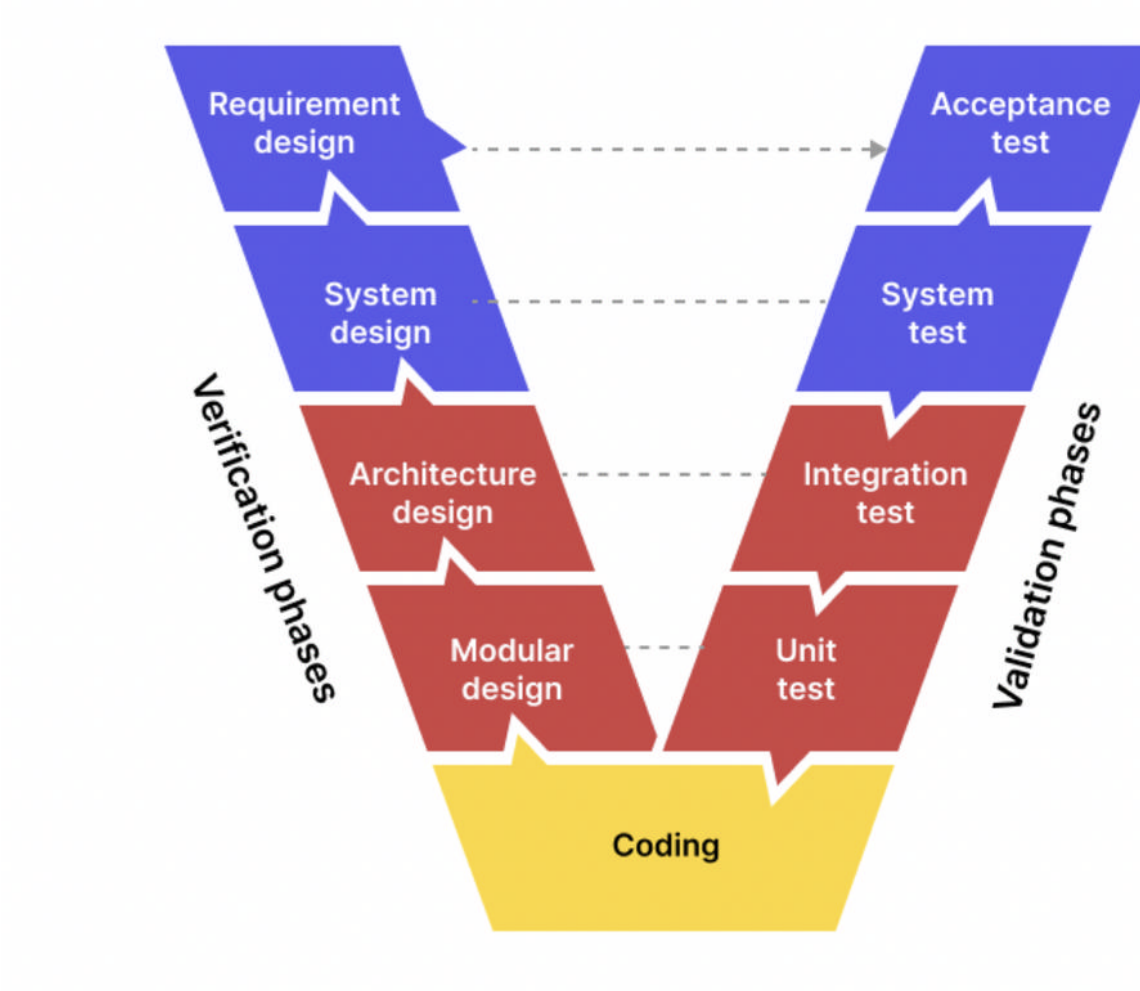


Рисунок 3.2 – V-модель

3.6.2. Тестова модель піраміди

З розвитком технологій модель Waterfall поступово поступається місцем широко використовуваному гнучкому методу тестування. Отже, V-модель також розвинулася до моделі тестової піраміди, яка візуально

представляє стратегію тестування з трьох частин.

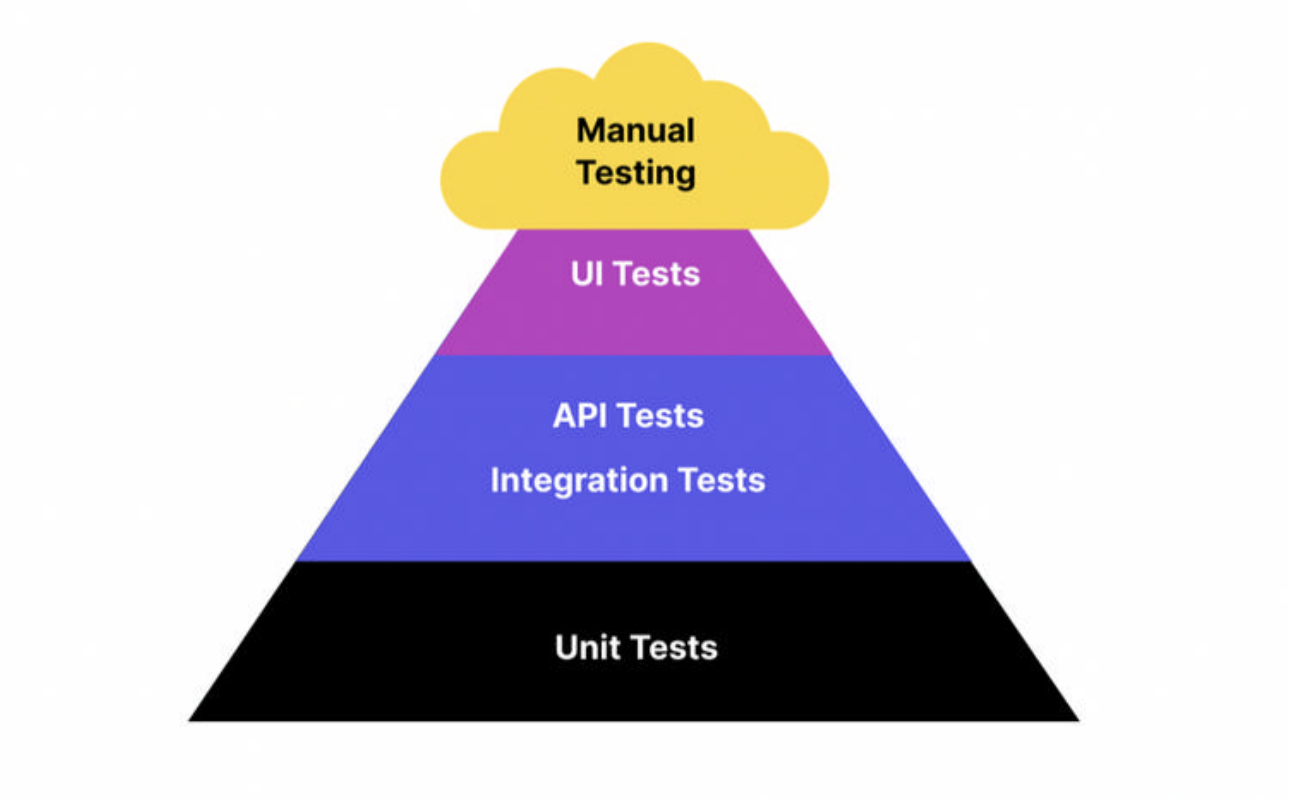


Рисунок 3.3 – Модель піраміди

Більшість тестів є модульними, спрямованими на перевірку лише окремих компонентів. Далі тестери групують ці компоненти та перевіряють їх як єдине ціле, щоб побачити, як вони взаємодіють. На цих етапах можна використовувати автоматизоване тестування для досягнення оптимальної ефективності.

Зрештою, на етапі тестування інтерфейсу користувача тестувальники зосереджуються на UX та інтерфейсі користувача програми.

3.6.3. Стільникова модель

Стільникова модель — це сучасний підхід до тестування програмного забезпечення, в якому основна увага приділяється інтеграційному тестуванню, тоді як модульному тестуванню (подробиці реалізації) та тестуванню інтерфейсу користувача (інтегрованому) приділяється менше уваги. Ця модель тестування програмного забезпечення відображає орієнтовану на API архітектуру системи, оскільки організації рухаються до

хмарної інфраструктури.

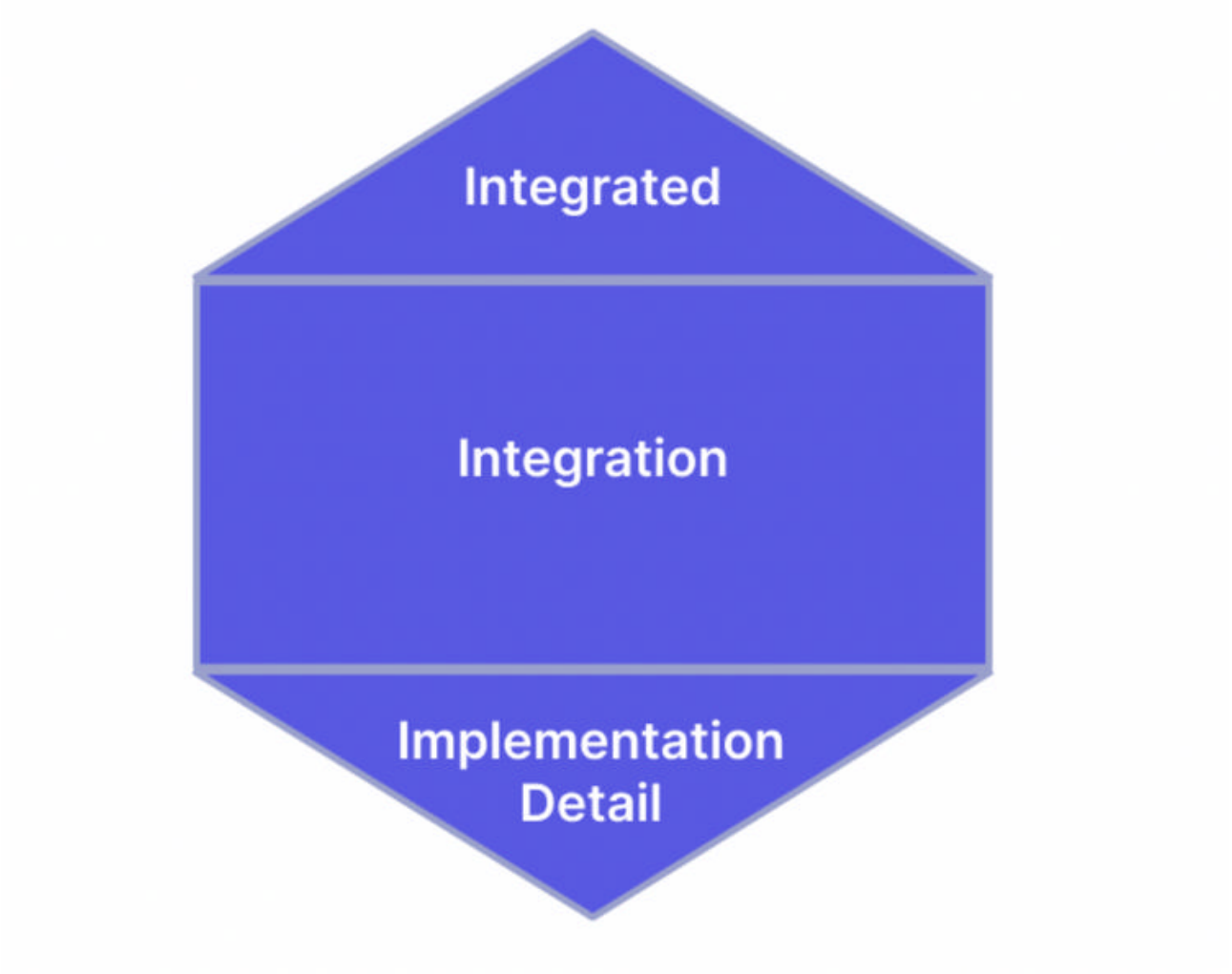


Рисунок 3.4 – Стільникова модель

3.7 Автоматизація в рамках тестування програмного забезпечення.

Автоматизоване тестування програмного забезпечення можна використовувати для тестування великих обсягів програмних компонентів, коли ручне тестування стає контрпродуктивним. Автоматизація забезпечує високу рентабельність інвестицій і дозволяє командам із забезпечення якості виконувати важливіші дії з тестування. Тестовий приклад придатний для автоматизації, якщо відповідають таким критеріям:

- Операція тестування повторюється, і етапи тестування навряд чи зміняться;
- Операція тестування займає багато часу, її неможливо виконати вручну;

- Операція тестування працює на кількох програмних/апаратних платформах;
- Операція тестування монотонна і схильна до людських помилок.

На практиці регресійне тестування зазвичай має вищий пріоритет для автоматизації, оскільки його природа полягає в повторному виконанні тих самих тестів. Автоматизація дозволяє швидше виконувати регресійний тест, дозволяючи командам із забезпечення якості зосередитися на інших критичних завданнях.

Навіть якщо тестувальникам насправді не потрібно автоматизувати тести, інструменти автоматизованого тестування можуть підтримувати їх у багатьох інших сферах, наприклад формування аналітики для прийняття рішень на основі даних. Вони також пропонують централізоване керування тестуванням, над яким команди можуть співпрацювати на багатьох етапах розробки та тестування. Інструменти автоматизованого тестування допомагають формувати звіти про тестування

3.8 Порівняння автоматизованого тестування проти ручного

Починаючи будь-який проект з тестування програмного забезпечення, команда тестувальників і команда розробників повинні зібратися разом і розробити план тестування, окреслюючи, які області тестувати вручну, а які – використовувати автоматизоване тестування. Гібридний підхід має надати тестувальникам переваги обох типів, як показано в порівняльній таблиці нижче:

	Manual Testing	Automated Testing
Definition	Manual testing is an approach in which a human tests software by manually using it like a user would.	Automated testing is an approach in which software testers use automated testing tools/scripts to speed up the process.
Benefits	• Easily capture bugs that sometimes automated scripts cannot detect • Lower cost • High flexibility	• Fast and effective • Great long-term ROI • Transparent results for the entire QA team
Drawbacks	• Prone to human error • Time-consuming and tedious • Not everything can be tested manually	• Require upfront investment • Machines cannot provide insight into how the app's UX or design appeal to the human user • May lack flexibility
Types of Testing to use	• Exploratory testing • Usability testing • Ad-hoc testing	• Regression testing • Load & Performance testing • Unit testing & Integrated testing
When to use	• During the very first rounds of testing to have a general understanding of the application • When testing infrequent, specific, and ad-hoc scenarios • When assessing the UI/UX of the application	• When continuous feedback on application quality is required • When there are features that have to be tested repeatedly after each release cycle

Рисунок 3.5 – Порівняння автоматизованого тестування проти ручного

Автоматизоване тестування виводить тестування програмного забезпечення на новий рівень, дозволяючи командам із забезпечення якості тестувати швидше та ефективніше. Отже, чи це робить ручне тестування справою минулого?

Ручне тестування все ще займає своє місце у світі тестування програмного забезпечення. Ручне тестування програмних продуктів забезпечує на даний час наступні операції:

- **Дослідницьке тестування**: тестувальники розробляють і виконують тести одночасно на основі свого досвіду та знань.

- **Спеціальне тестування**: тестування виконується без попередньо визначених тестів або сценаріїв, покладаючись на тестувальників, інтуїція та креативність.
- **Тестування зручності використання**: оцінює зручність використання та загальну взаємодію з користувачем, щоб виявити проблеми зручності використання та покращити зручність використання програмного забезпечення.

Відповідно також потрібні люди для оцінки UX програми, контролю за тестуванням автоматизації та втручання, коли це необхідно. Однак технології ШІ поступово змінюють ландшафт. Функції інтелектуального тестування додано до багатьох інструментів автоматизованого тестування програмного забезпечення, щоб значно зменшити потребу в людському втручанні.

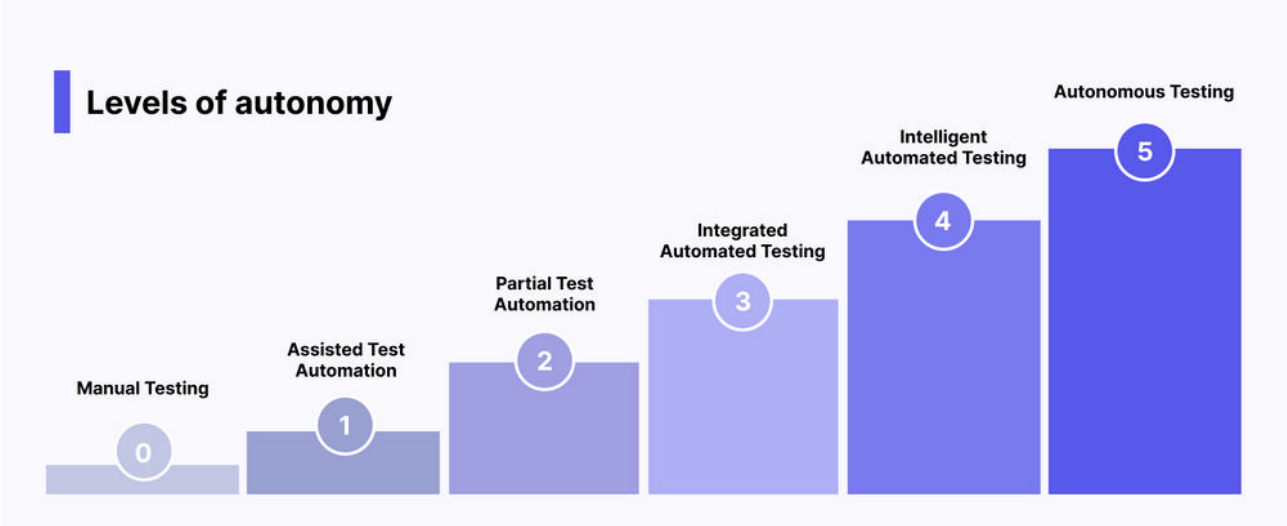


Рисунок 3.6 – Рівні автономності тестування програмного забезпечення

У майбутньому ми можемо розраховувати на досягнення автономного тестування, де машини повністю візьмуть під контроль і виконають усі дії тестування. Людей абсолютно не буде потрібно, окрім розробки алгоритмів тестування. Інструменти тестування програмного забезпечення тепер інтегрували ChatGPT – новаторський чат-бот зі штучним інтелектом – щоб наблизити нас до цієї автономної функції.

4 ПРОЕКТНА ЧАСТИНА

4.1 Автоматизоване тестування веб-сайту

Для імплементації та демонстрації автоматизованого тестування було обрано тренувальний веб-сайт https://www.demoblaze.com/, який представляє собой інтернет-магазин з backend частиною (є АПІ endpoints), що дозволятись нам провести UI і API автоматизоване тестування

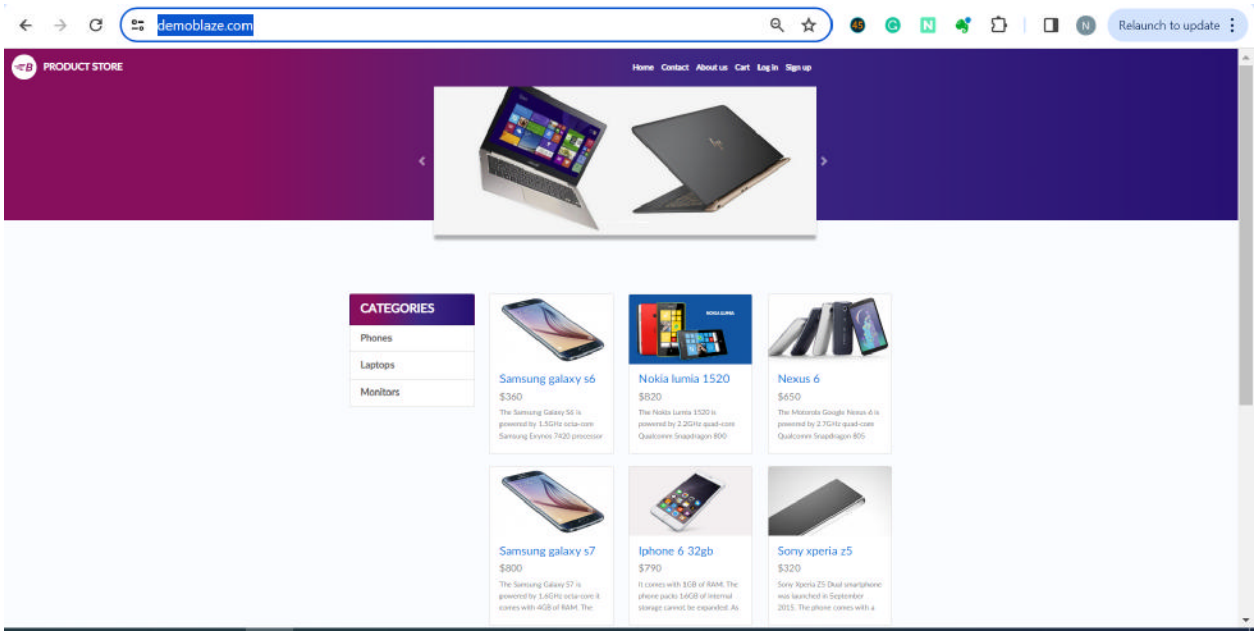


Рисунок 4.1. – Веб-сторінка для автоматизації

4.1.1. Стратегія тестування:

Перед тим, як починати розробку автоматизованого фреймворку, спочатку необхідно обрати стратегію та планування нашого автоматизованого тестування.

В нашому випадку буде 2 типи тестування - UI і API. API тестування буде мати вищий пріоритет, так як ці тести легші до написання, займають менше часу і швидше виконуються. Також деякі API запити можна буде використовувавати як переудмови до запуску UI тестів.

Другий етап - збір результатів в репорт і наповнення необхідною інформацією.

Третій етап - впровадження CI/CD процесу, що дозволить нам запускати тести на кожну зміну, регулярно чи при виконанні інших умов.

4.1.2. Підхід до написання автоматизованих тестів:

Відповідно до піраміди тестування, більшість тестів буде складати API тести, решта - UI тести.

Для написання і підтримки тестів ми будемо використовувати підхід ‘Always Clean & Always Green’:

‘Always Cleen’ - означає, що перед запуском тестів ми створюємо тестові дані, які нам потрібні для тестів і після виконання всіх тестів, ми повертаємо стан системи до вихідного. Тобто, після виконання всіх тестів, ніяких тестових даних не має залишитись в системі

‘Always Green’ - ми запускаємо і аналізуємо лише стабільні тести. Тести повинні бути створені так, щоб вони були стабільні і їх потрібно оновлювати лише коли якісь критичні зміни, чи вони впали коли в системі дефект. Якщо якісь тести нестабільні, ми їх помічаємо спеціальним тегом і запускаємо окремо, до тих пір поки тест не буде знову стабільним.

4.1.3. API тестування:

API (Application Programming Interface) тестування — це процес перевірки правильності та надійності взаємодії програмних компонентів через API. Такі тести дозволяють впевнитися в тому, що програмний інтерфейс працює правильно, навіть якщо різні частини програми були розроблені різними командами або компаніями.

Основні етапи API тестування включають:

Планування тестування:

- Визначення функціональності, яку потрібно протестувати.
- Створення тестового плану та визначення критеріїв успішності.

Визначення тестових сценаріїв:

- Створення сценаріїв тестування для різних API-методів та операцій.
- Урахування різних можливих вхідних даних і сценаріїв використання.

Налаштування середовища тестування:

- Забезпечення доступу до тестового середовища або відокремленого тестового сервера.
- Налаштування тестових даних та оточення.

Виконання тестів:

- Виконання тестових сценаріїв та реєстрація результатів.
- Перевірка відповідей API на відповідність очікуваним результатам.

Автоматизація тестів:

- Розробка автоматизованих тестових скриптів для покриття широкого спектру тестових сценаріїв.
- Використання інструментів автоматизації тестування API, таких як Postman, RestAssured, або Selenium.

Моніторинг та відлагодження:

- Моніторинг даних в реальному часі для виявлення помилок та аномалій.
- Відлагодження та вирішення виявлених проблем.

Документація:

- Створення та оновлення документації API для користувачів.

Безпека:

- Перевірка вразливостей та забезпечення безпеки API.

Для автоматизації тестів можна використовувати різноманітні інструменти, такі як Postman, Swagger, JUnit, TestNG, тощо.

Важливо враховувати різні типи тестів, такі як тести на правильність, відмовостійкості, навантаження та безпеки, для повного охоплення функціональності API.

4.1.4. Ручне API тестування:

Ручне API тестування є важливою складовою процесу забезпечення якості програмного забезпечення. Під час цього виду тестування, людський

тестувальник виконує серію кроків для перевірки правильності та надійності взаємодії програмних компонентів через API. Основні етапи ручного API тестування включають розуміння API, підготовку тестового середовища, створення тестових сценаріїв, виконання тестів, перевірку даних, тестування відмовостійкості та безпеки, а також документування результатів.

Під час тестування важливо перевіряти різні сценарії використання та різноманітні вхідні дані, щоб забезпечити вичерпне охоплення функціональності API. Також слід акцентувати увагу на відмовостійкості API, перевіряючи його поведінку в умовах помилок або неправильного використання.

Ручне тестування має перевагу у випадках, коли автоматизація неможлива або неефективна, а також у випадках, коли необхідно провести нестандартні або інтерактивні тести. Цей підхід дозволяє тестувальникам здійснювати глибше розуміння функціональності та інтерфейсу API, а також ефективно виявляти проблеми, які можуть залишитися непоміченими під час автоматизованого тестування.

У підсумку, ручне API тестування відіграє важливу роль у забезпеченні якості програмного забезпечення, доповнюючи автоматизовані підходи та забезпечуючи високу якість та надійність API.

4.1.5. Автоматизоване API тестування:

Автоматизоване API тестування є ключовим елементом забезпечення якості програмного забезпечення, дозволяючи автоматизувати процеси тестування та забезпечуючи ефективність та швидкість виявлення помилок. За допомогою спеціалізованих інструментів, таких як Postman, RestAssured, або Selenium, розробники та тестувальники можуть автоматизувати виконання тестових сценаріїв, перевірку відповідей API, тестування регресії та багато іншого.

Автоматизоване тестування забезпечує широкий спектр переваг, включаючи швидше виявлення та виправлення помилок, вартість зниження

тестування у довгостроковій перспективі, можливість повторного використання тестових скриптів та інтеграцію з системами збірки та CI/CD. Цей підхід допомагає забезпечити стабільність та надійність API протягом усього циклу розробки, а також прискорює процес впровадження нового функціоналу.

Завдяки автоматизації, команди розробників можуть швидше та ефективніше виконувати тестування, скорочуючи час циклу розробки та забезпечуючи більш високу якість продукту.

4.1.6. Переваги автоматизованого API тестування:

Автоматизоване тестування API визнається ключовим елементом сучасного процесу розробки програмного забезпечення, і його переваги є важливими для забезпечення високої якості продуктів та оптимізації процесу розробки. Основною перевагою автоматизованого API тестування є підвищення ефективності та швидкості в порівнянні з ручним тестуванням.

Використання автоматизованих тестових скриптів дозволяє автоматизувати повторювані завдання, що раніше вимагали б великої кількості часу та зусиль. Відсутність необхідності у вручному введенні даних чи перевірці реакції системи дозволяє прискорити цикл тестування, виявляти помилки та недоліки швидше, що призводить до збільшення продуктивності розробницького процесу.

Додатково, автоматизоване API тестування дозволяє забезпечити більш широке покриття функціональності, оскільки тести можуть бути легко масштабовані для включення нових функцій чи змінених компонентів системи. Це сприяє вчасному виявленню та виправленню помилок, а також покращує стабільність програмного продукту.

Загалом, автоматизоване API тестування не тільки сприяє підвищенню якості та надійності програмного забезпечення, але й раціоналізує процес розробки, забезпечуючи ефективніший використання ресурсів та швидше введення продукту на ринок.

4.1.7. Процес тестування API:

Процес автоматизованого тестування API включає кілька етапів, від планування до виконання та аналізу результатів. Перш за все, планується тестування, визначаються мети та вибираються інструменти. Далі аналізується API-документація, створюються тестові сценарії та підготовлюється тестове середовище.

Після цього розробляються тестові скрипти, використовуючи обрані інструменти автоматизації. Тестові сценарії перетворюються на конкретні тести з використанням API-запитів. Слідом за цим запускаються автоматизовані тести, результати реєструються, а виявлені проблеми документуються.

Аналіз результатів включає виявлення та документування проблем, а також генерацію звітів про результати тестування для команди розробки. Завершальним етапом є підтримка та оновлення тестових скриптів, включаючи їхню актуалізацію для врахування змін у API або вимогах проекту. Цей процес дозволяє забезпечити ефективність, стабільність та високу якість API, покращуючи цикл розробки та сприяючи швидкому виявленню та виправленню помилок.

4.1.8. Реалізація API тестування:

Для нашого проекту було вибрано інструмент SuperTest для API тестування. Для запуску тестів використовуємо Jest.

Отож, у створеному проекті додаємо до package.json потрібні бібліотеки, що будуть використовуватись для тестування:

```
"devDependencies": {
  "@types/express": "^4.17.13",
  "@types/jest": "*",
  "@types/node": "^16.4.12",
  "@types/supertest": "^2.0.11",
  "@types/uuid": "^8.3.4",
  "@typescript-eslint/eslint-plugin": "^5.27.1",
  "@typescript-eslint/parser": "^5.27.1",
  "eslint": "^8.17.0",
  "jest": "*",
  "jest-html-reporters": "^3.0.9",
  "jest-junit": "^14.0.0",
  "nodemon": "^2.0.12",
  "supertest": "^6.1.4",
  "ts-jest": "*",
  "ts-node": "^10.1.0",
  "typescript": "^4.3.5"
}
```

Рисунок 4.2 – Package.json для АПІ тестів

Після цього створюємо архітектуру папок, в яких будуть лежати відповідні частини проету, це дозволить розуміти де що лежить і як використовувати.

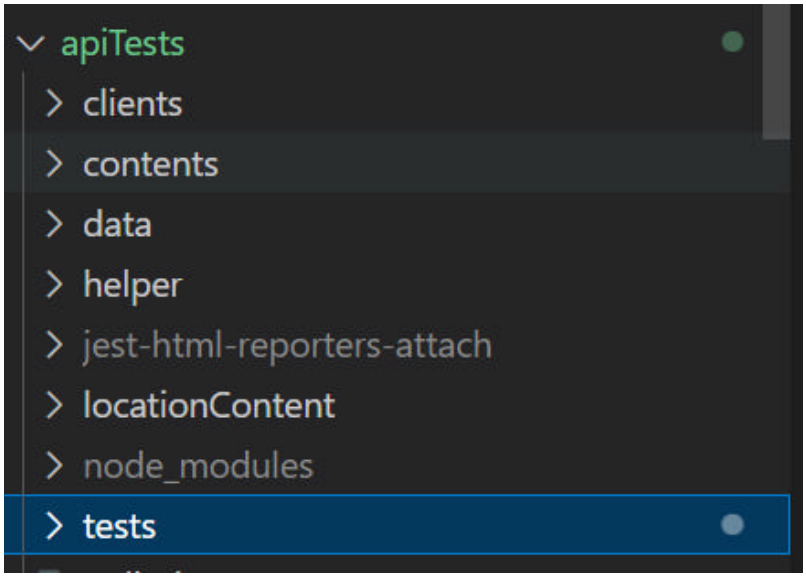


Рисунок 4.3 – Структура АПІ фреймворку

В папці clients ми будемо тримати всі наші API endpoints з якими ми будемо працювати. В кожному клієнті ми будемо підключати SuperTest

request і response обєкти для того, щоб відправити API запит та отримати результат і опрацювати його.

Приклад, як працюємо з SuperTest.

```
import request, {Response} from 'supertest'
import {requestToEnv} from "../helper/CommonUtils";
import {commonHeaders, commonHeadersWithoutAcceptField} from "../helper/Headers";

const req = requestToEnv('titles/');

export async function getTitleById(id: string): Promise<request.Response> {
    const resp = await req.get(id)
        .set(commonHeadersWithoutAcceptField);
    return resp;
}

export async function createTitle(title: object): Promise<request.Response> {
    const resp = await req.post("")
        .send(title)
        .set(commonHeadersWithoutAcceptField);
    return resp;
}
```

Рисунок 4.4 – Приклад використання SuperTest для виклику запиту

Папки data/contents містять json та інші файли, що нам потрібні для генерації даних.

Helper папка містить допоміжні функції, що використовуються для перевірки обєкту що повернувся у запиті та для генерації випадкових даних.

```
export function checkResponseStatusCode(response: any, ids: Array<string>, statusCode = 201) {
    if (response.statusCode == statusCode) {
        ids.push(response.body.id)
    } else if (response.statusCode == 409) {
        console.log(response.body.error);
        if("storageInformation" in response.body.extensions) ids.push(response.body.extensions.storag
        else ids.push(response.body.extensions.assetId)
    }
    expect(response.statusCode).toEqual(statusCode);
    return ids;
}
```

Рисунок 4.5 – Приклад допоміжної функції

В папці tests містяться безпосередньо наші тести. За допомогою jest ми описуємо назву функціоналу який тестуємо і тоді створюємо кожен тест з конкретною назвою, що ми хочемо протестувати.

Всередині тесту ми використовуємо наші API клієнти та допоміжні методи для генерації даних та для перевірки статусів та обєктів.

Приклад, як виглядає один з таких тестових класів

```
describe("Test /title endpoint", () => {

    async function cleanUp(): Promise<void> {
        for (let i = 0; i < titles.requestArray.length; i++) {
            const titleId = titles.requestArray[i];
            const deleteTitleResponse = await titles.deleteTitle(titleId);
            expect(deleteTitleResponse.statusCode).toEqual(204);
        }
    }

    afterAll(() => {
        return cleanUp()
    })

    it("Create title", async () => {
        const req = titles.getTitleBody("sq", "sq", 2001, "autoTestName");
        const resp = await titles.createTitle(req);
        checkResponseStatusCode(resp, titles.requestArray);
        expect(resp.statusCode).toEqual(201);
        expect(resp.body.note).toEqual(req.note)
        expect(resp.body.releaseYear).toEqual(req.releaseYear)
    });
```

Рисунок 4.6 – Приклад тестового класу

Якщо не додавати ніякого стороннього репорту, то після запуску тестів ми зможемо лише в консолі побачити які тести впали а які пройшли, що не є дуже зручно. Тому ми додали сторонню бібліотеку, яка показує тести в доволі зручному і красивому форматі:

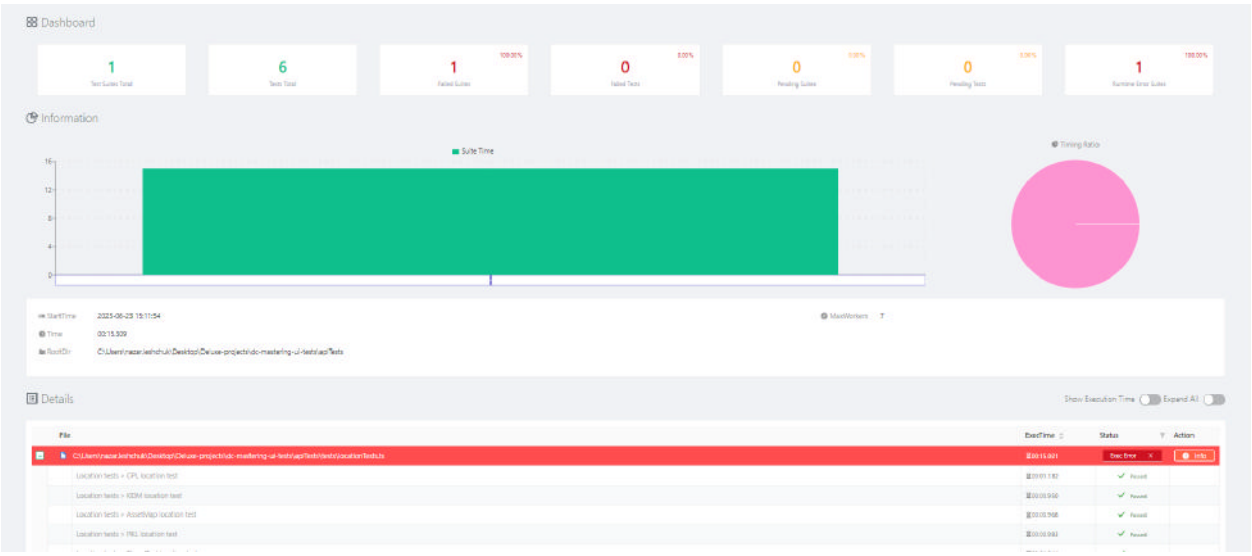


Рисунок 4.7 – Тестовий репорт

4.1.9. Реалізація UI тестування:

Для тестування UI частини нашого веб-сайту, було обрано інструмент Playwright, який має дуже багато можливостей.

Для UI тестів була створена окрема папка, в якій ми працюємо і додаємо всі бібліотеки. Вигляд має наступний:

Рисунок 4.8 – Package.json для UI тестів

Після додавання бібліотеки в наш проект, аналогічно створюємо структуру папок, для зручного подальшого використання. В першу чергу застосовуємо PageObject шаблон, який вказує, що кожну сторінку сайту потрібно представляти як окремий клас в нашому проекті. Тому структура має вигляд

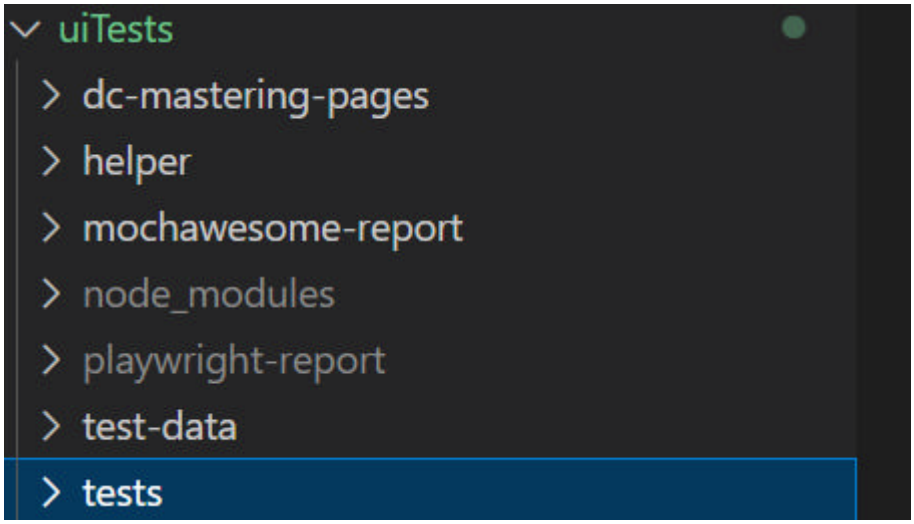


Рисунок 4.9 – Структура UI фреймворку

В pages ми якраз і тримаємо всі наші сторінки, які в свою чергу містять елементи сторінки, з якими ми будемо взаємодіяти.

Приклад написання локаторів, які будуть шукати елементи на сторінці

```
export class CountriesPage {
    readonly page: Page;
    private readonly homePage: BasePage;
    private title = '//h2[text()="Countries"]';
    private addButton = '//button[text()="Add Country"]';
    private addCountryDialog = '//h2[text()="New Country"]';
    // TODO Find a way to verify a state of the toggle
    private countryAreaToggle = '//span[contains(text(),"Country / Area")]/preceding-sibling::span';
    private countryTagRfc5646Input = '//input[@name="countryTagRfc5646"]';
    private dcncNameInput = '//input[@name="dcncName"]';
    private dcncCodeInput = '//input[@name="dcncCode"]';
    private countryCodeIso3166_1Alpha2Input = '//input[@name="countryCodeIso3166_1Alpha2"]';
    private countryCodeIso3166_1Alpha3Input = '//input[@name="countryCodeIso3166_1Alpha3"]';
    private createButton = '//button[text()="Create"]';
    private areaTagRfc5646Input = '//input[@name="countryTagRfc5646"]';
    private areaUnM49Input = '//input[@name="areaUnM49"]';
```

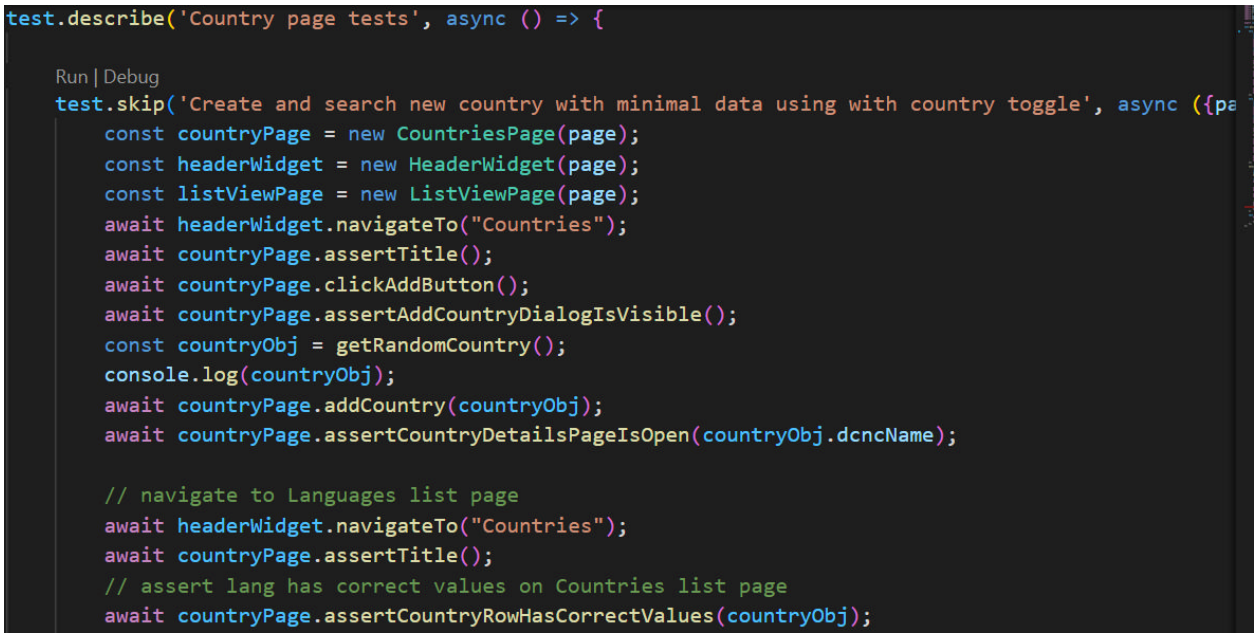
Рисунок 4.10 – Приклад PageObject класу з локаторами

Приклад взаємодії з цими елементами за допомогою Playwright

```
async assertTitle() {
    return await waitForVisible(this.page, this.title);
}

async assertAddButton() {
    return await actions.waitForVisible(this.page, this.addButton);
}

async clickAddButton() {
    return await actions.click(this.page, this.addButton);
}

async assertAddCountryDialogIsVisible() {
    return await actions.waitForVisible(this.page, this.addCountryDialog);
}

async assertCountryDetailsPageIsOpen(dcncName: string) {
    return await actions.waitForVisible(this.page, actions.locatorWithSpecificText(this.countryDe
}
```

Рисунок 4.11 – Взаємодія з елементами на сторінці

Приклад того як виглядає тестовий клас, де ми викликаємо методи з pages



```
test.describe('Country page tests', async () => {

    Run | Debug
    test.skip('Create and search new country with minimal data using with country toggle', async ({pa
        const countryPage = new CountriesPage(page);
        const headerWidget = new HeaderWidget(page);
        const listViewPage = new ListViewPage(page);
        await headerWidget.navigateTo("Countries");
        await countryPage.assertTitle();
        await countryPage.clickAddButton();
        await countryPage.assertAddCountryDialogIsVisible();
        const countryObj = getRandomCountry();
        console.log(countryObj);
        await countryPage.addCountry(countryObj);
        await countryPage.assertCountryDetailsPageIsOpen(countryObj.dcncName);

        // navigate to Languages list page
        await headerWidget.navigateTo("Countries");
        await countryPage.assertTitle();
        // assert lang has correct values on Countries list page
        await countryPage.assertCountryRowHasCorrectValues(countryObj);
```

Рисунок 4.12 – Приклад тестового класу

4.2 Архітектура автоматизованого фреймворку

На картинці внизу показано який вигляд має наш автоматизований фреймворк. Як відбувається комунікаціями між обєктами та шарами проекту.

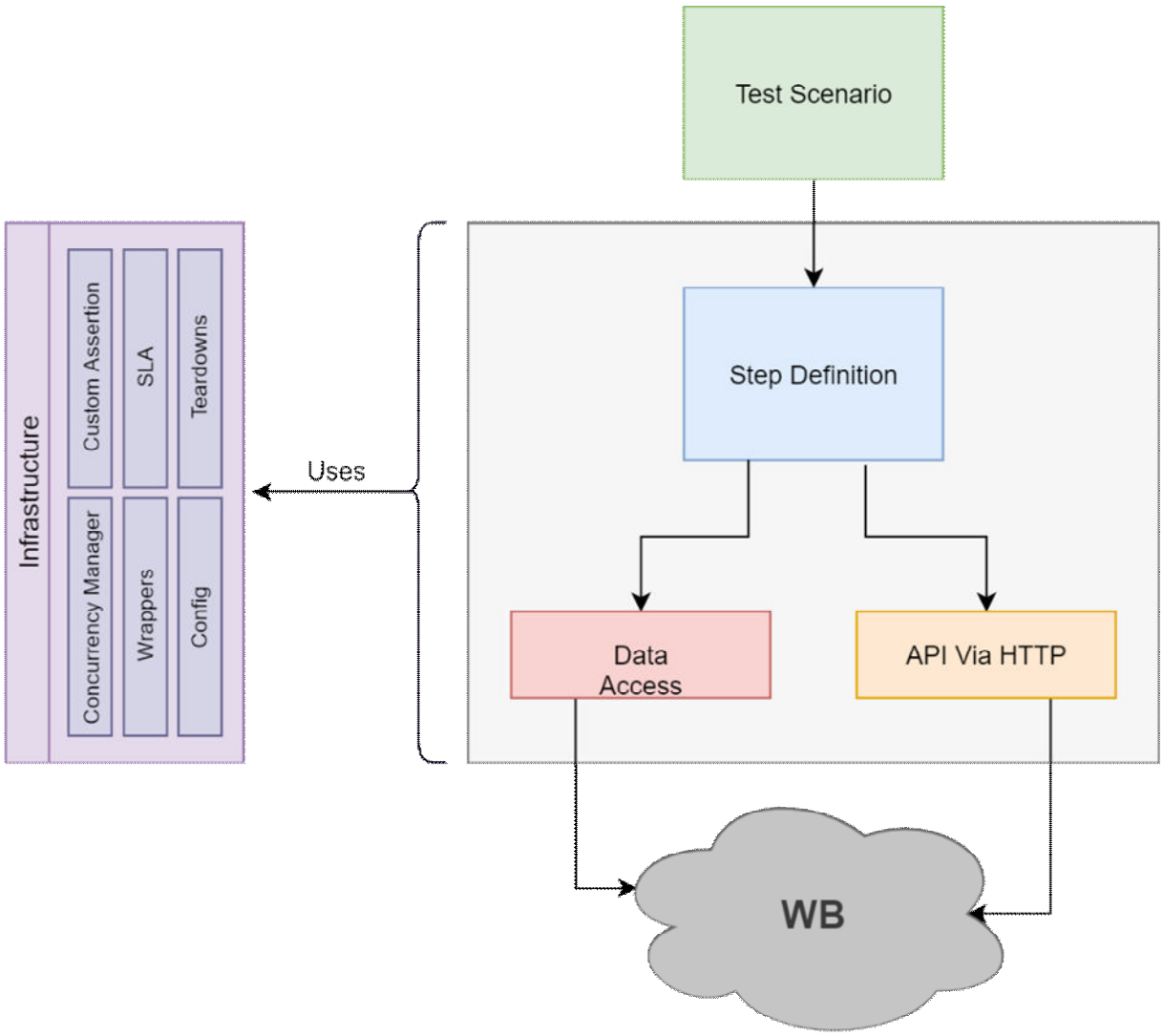


Рисунок 4.13 – Архітектура фреймворку

4.2.1. Написання тестів

Написання тестів з верху в низ:

Під час написання тестів слід памятати наступні правила, так як це зробить проект стійким і його буде легше розширяти.

З верху в низ означає що ми починаємо описувати тест з кінця, тобто описуємо які саме кроки нам треба зробити для даного тесту, після цього опускаємось нижче - створюємо АПІ запити, знаходимо потрібні елементи на сторінці та взаємодіємо з ними.

Не створюємо інфрастуктуру що не потрібна буде в межах цього тесту

Ми будемо мати цілісну картину тесту коли будемо писати низькорівневі методи і логіку.

4.2.2. Процес написання тестів:

1. Верхній шар тестів пишемо зрозумілою англійською мовою, що дасть змогу легко зрозуміти що саме тестуємо.
2. Імплементуємо steps для кожного тестового сценарію. Ця частина буде містити всю логіку тестових дій та методів перевірки.
3. Створюємо необхідні обєкти що будуть використовуватись для тестового сценарію.
4. Не забуваємо створювати загальні методи, які дадуть змогу наповнити систему тестовими даними (користувачами, продуктами і тд).

4.2.3. Тестові Layers (шари):

Наш фреймворк поділений на 3 шари

1. Tests - містить тестові класи, ці класи виконують тести.
2. Logic - на цьому шарі міститься логічний код: API, обєкти, функції для тестів
3. Infra - зєднання з іншими системами, endpoints, репорти і тд.

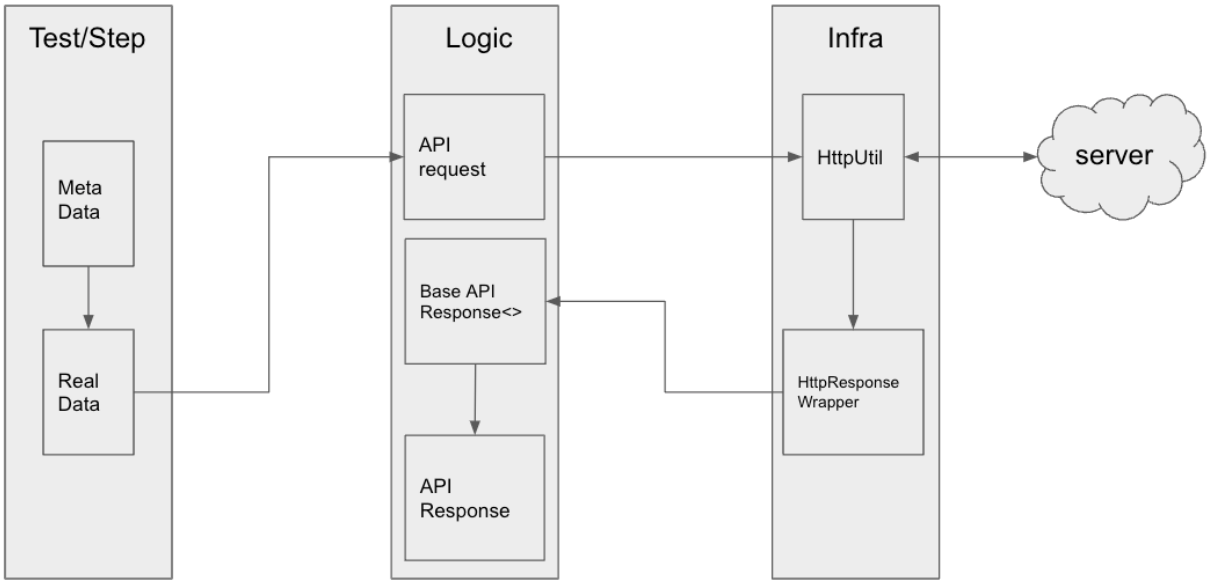


Рисунок 4.14 – Test layers

4.3 CI/CD + репортинг

Важливим елементом тестового фреймворку є читабельний тест репорт, який містить всю необхідну інформацію про запуск тестів. Маючи цей репорт, зручно інтегрувати в CI/CD процес, так як це дозволить легко перевіряти причину падіння тестів.

4.3.1.Тестові репорти:

Оскільки в нас є два типи тестів UI і API і для цього використовуються різні інструменти, то відповідні генеруються різні тестові репорти, для API це окрема бібліотека яка на основі Jest runner агрегує дані і генерує репорт, для UI простіше, адже Playwright має хороший вбудований ріпорт.

Також для зручності, ми додали ще один репорт - Allure, який агрерує всі дані і формує один загальний репорт.

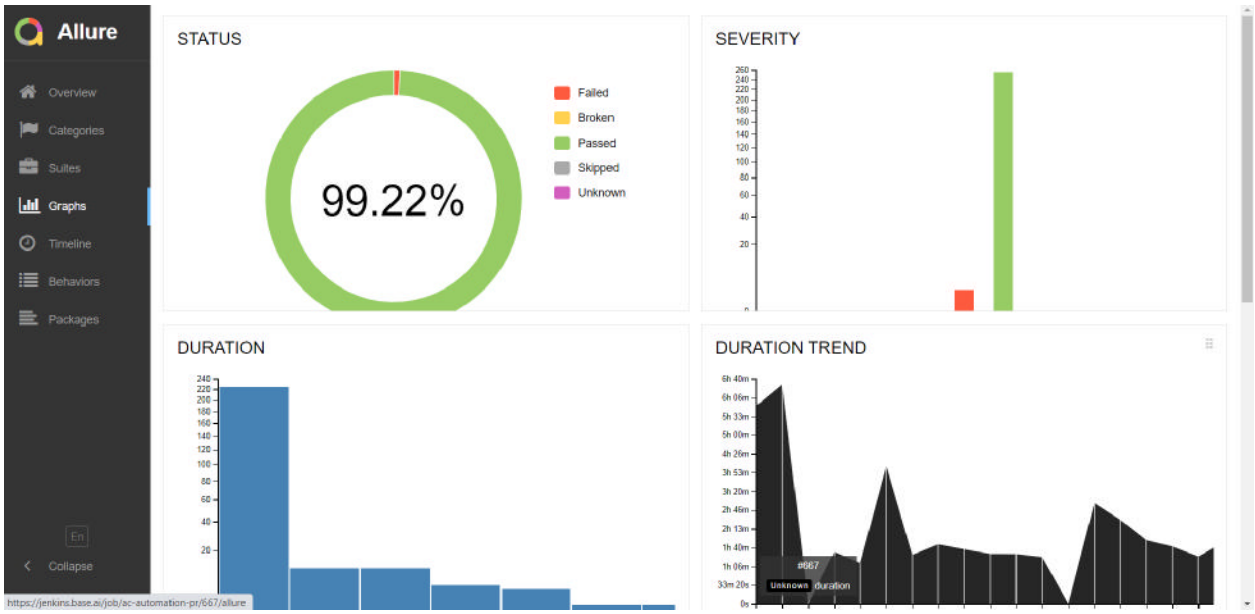


Рисунок 4.15 – Allure репорт

4.3.2. CI/CD:

Для того, щоб автоматизоване тестування приносило більше користі, ми налаштували CI/CD Jenkins, що дає нам змогу запускати тести при необдіхності чи виконанні певних умов.

Зокрема можна запускати тести після того, як розробником було зроблено якісь зміни, також налаштували, щоб тести бігли кожен в певній годині.

Після того, як пробігають тести, нам стає в пригоді наш згенерований репорт, який можна зручно відкрити в Jenkins і переглянути результати.

Налаштувати Jenkins не складно:

Перш за все, встановлюємо і запускаємо як сервіс.

Після того можемо створювати jobs, в яких вказуємо як запускати тести, коли запускати і які тести запускати.

Також додаємо плагін, який дасть змогу переглядати репорт прямо в Jenkins.

Виглядає це наступний чином:

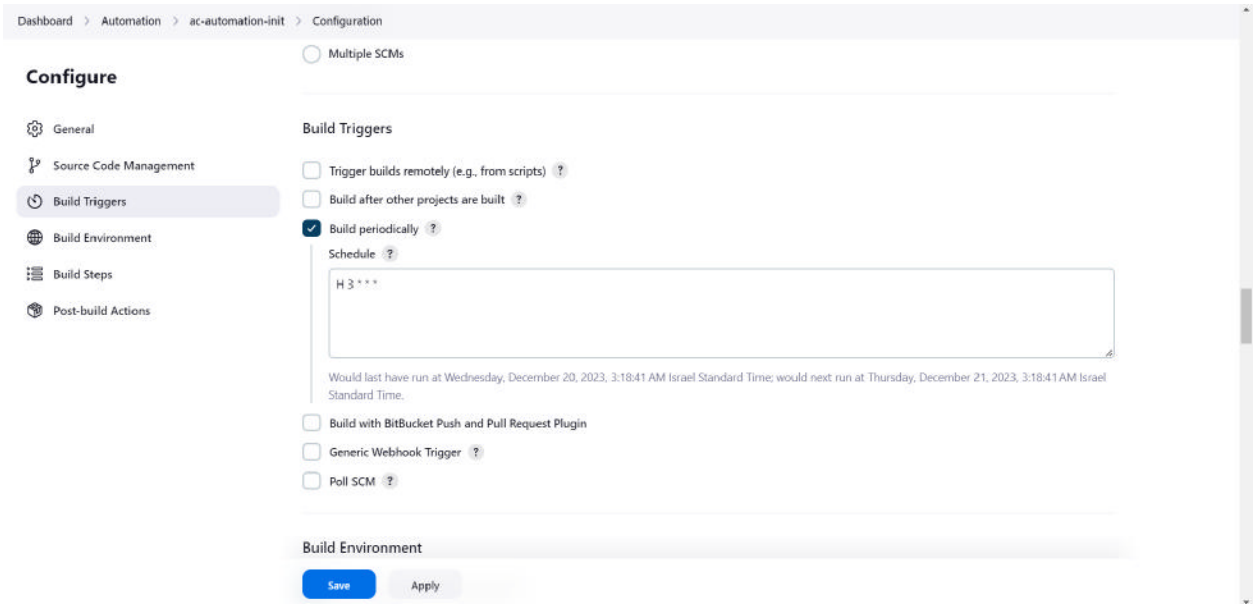


Рисунок 4.16 – Приклад конфігурації Jenkins job(періодичність запуску)

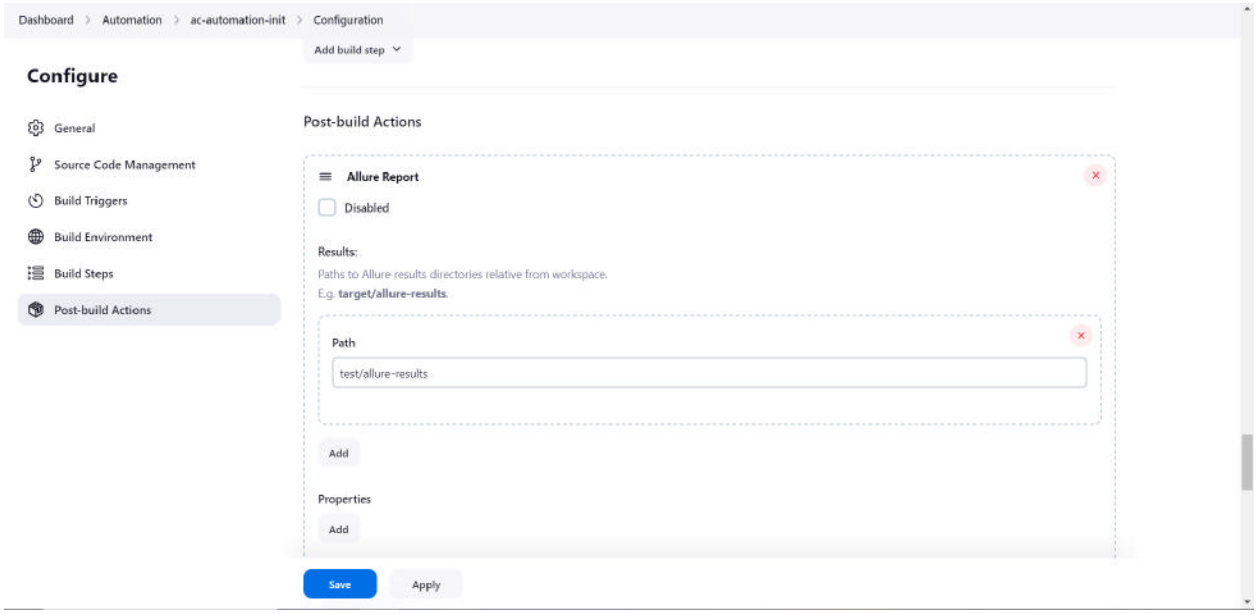


Рисунок 4.17 – Приклад конфугурації Jenkins job(генерація репорту)

5 СПЕЦІАЛЬНА ЧАСТИНА

5.1 Фреймворк для тестування JavaScript Jest.

Jest — це фреймворк для тестування JavaScipt, розроблений для забезпечення правильності будь-якого JavaScript коду. Він дозволяє писати тести з використанням доступного, знайомого і багатофункціонального API, що дає швидкий результат.

5.1.1. Синтаксис та низький поріг вхідного бар'єру.

Jest відомий своїм простим синтаксисом, який дозволяє легко створювати тести. Встановлення та конфігурація також є максимально простою, що робить його доступним для новачків при цьому залишаючись хорошим та гнучким інструментом для досвідчених розробників.

```
javascriptCopy code
test('приклад тесту', () => { expect(1 + 2).toBe(3); });
```

5.1.2. Автоматичне визначення тестових файлів.

Jest автоматично визначає тестові файли у вашому проекті, що полегшує організацію тестів і дозволяє швидше розпочати написання тестів.

5.1.3. Можливості мокування та підробки (Mocking).

Jest має вбудовану підтримку для створення mock-об'єктів, функцій та модулів. Це дозволяє ефективно тестувати компоненти, що залежать від зовнішніх ресурсів чи служб. Jest використовує власний резолер для імпортів у ваших тестах, що дозволяє легко створювати об’єкти-імітації для будь якого модуля за межами видимості тесту. Ви можете використовувати імітований імпорт з багатими функціями API для відслідковування викликів функцій зі зручним синтаксисом тестів.

```
javascriptCopy code
const mockFunction = jest.fn(); mockFunction();
expect(mockFunction).toHaveBeenCalled();
```

5.1.4. Розширена підтримка асинхронного коду.

Jest робить роботу з асинхронним кодом простою. Він автоматично очікує завершення асинхронних операцій перед завершенням тесту.

```
javascriptCopy code
test('асинхронний тест', async () => { const data = await
fetchData(); expect(data).toEqual('очікувані дані'); });
```

5.1.5. Зручний інтерфейс виведення результатів.

Jest надає інтерактивний інтерфейс виведення результатів, який чітко вказує на те, які тести пройшли, а які не вдалося виконати.

Рисунок 5.1 – Інтерактивний інтерфейс виведення результатів

5.1.6. Сприяння тестам з великим обсягом даних.

Jest розроблений так, щоб ефективно вирішувати завдання тестування додатків з великим обсягом даних чи складних структур. Забезпечуючи для ваших тестів унікальний глобальний контекст, Jest може надійно виконувати тести паралельно. Щоб пришвидшити роботу, Jest спочатку запускає тести, які завершалися з помилками і постійно реорганізує порядок виконання тестів на основі того, як довго вони виконуються.

5.1.7. Підтримка Snapshot-Тестування.

Jest дозволяє зберігати "знімки" (snapshots) вихідних даних та порівнювати їх під час подальших тестів, що полегшує виявлення змін від версії до версії. Це дозволяє відслідковувати великі об'єкти в тестах без зайвих зусиль. Знімки живуть поруч з вашими тестами або вбудовуються прямо в них.

```
javascriptCopy code
test('знімковий тест', () => { const component = render();
expect(component).toMatchSnapshot(); });
```

5.1.8. Підтримка таймерів.

Jest має вбудовану підтримку таймерів, що дозволяє контролювати та тестувати функції, які використовують таймери.

```
javascriptCopy code
test('тест з таймером', () => { jest.useFakeTimers(); //
Виклик функції, яка використовує таймер jest.runAllTimers();
// Очікування результатів тесту });
```

5.1.9. Розширюваність та плагіни.

Jest підтримує розширюваність через плагіни, що дозволяє розробникам розширювати його функціонал та використовувати власні розширення.

5.1.10. Підтримка серверного та клієнтського коду.

Jest може використовуватися для тестування як серверного, так і клієнтського коду, що робить його універсальним для різних типів проектів.

Це лише декілька ключових можливостей фреймворка Jest. Враховуючи всі ці функції, Jest стає важливим інструментом для розробників, що допомагає створювати надійний і високоякісний код.

5.1.11. Параметризовані тести:

Jest підтримує параметризовані тести, що дозволяє вам використовувати один і той же тестовий випадок для різних наборів вхідних даних.

```
javascriptCopy code
test.each([ [1, 1, 2], [1, 2, 3], [2, 2, 4], ])('додавання %i
до %i дорівнює %i', (a, b, expected) => { expect(a +
b).toBe(expected); });
```

5.1.12. Тайм-аути та Retry.

Jest дозволяє налаштовувати тайм-аути для тестів та має вбудовану підтримку повторення (retry), що дозволяє вам повторювати фейлені тести з заданими параметрами.

```
javascriptCopy code
test('тест з тайм-аутом', () => { // Код тестування з великим
часовим обмеженням }, 10000); // Тайм-аут у мілісекундах
```

5.1.13. Глобальні функції beforeAll, afterAll, beforeEach, afterEach.

Jest дозволяє вам визначати глобальні функції beforeAll, afterAll, beforeEach, afterEach, які виконуються перед або після виконання тестових блоків. Якщо функція повертає проміс, або генератор, Jest очікує на виконання цього промісу, перш ніж продовжити. Додатково, ви можете

вказати timeout (у мілісекундах) на те як довго чекати перед перериванням. За замовчуванням, тайм-аут становить 5 секунд.

```
javascriptCopy code
beforeEach(() => { // Перед кожним тестом }); afterEach(() =>
{ // Після кожного тесту }); beforeAll(() => { // Один раз
перед всіма тестами }); afterAll(() => { // Один раз після
всіх тестів });
```

5.1.14. Тестування Async/await.

Jest надає зручні функції для тестування асинхронних операцій, таких як async і await.

```
javascriptCopy code
test('асинхронний тест', async () => { const result = await
fetchData(); expect(result).toBe('очікуваний результат'); });
```

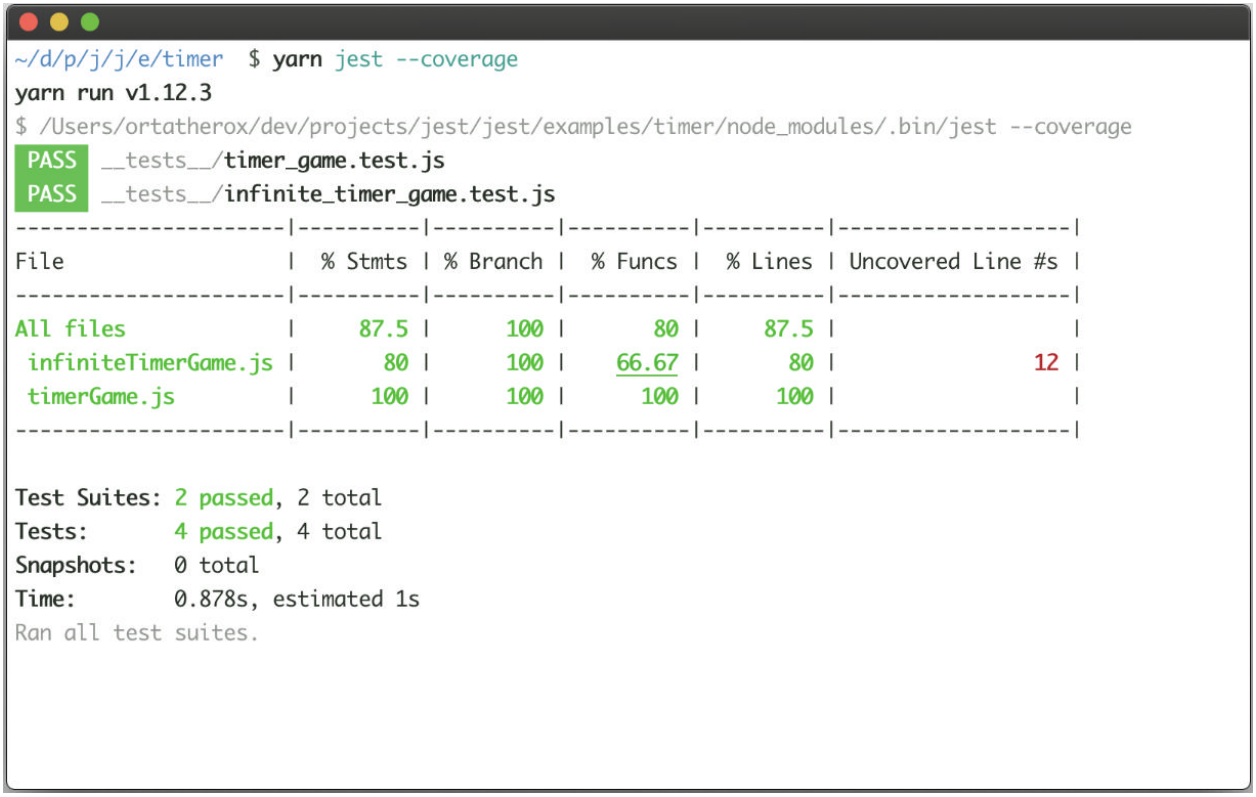
5.1.15. Динамічні Тести.

Jest дозволяє створювати динамічні тести з використанням функцій, що повертають блоки тестів.

```
javascriptCopy code
const data = [1, 2, 3]; data.forEach((value) => { test(`тест
зі значенням ${value}`, () => {
expect(value).toBeGreaterThan(0); }); });
```

5.1.16. Виведення покриття коду.

Jest має вбудовану можливість генерації звітів про покриття коду, що допомагає визначити, які частини вашого коду були протестовані, а які - ні.




```
bashCopy code
# Запуск тестів з виведенням звіту про покриття коду npx jest
--coverage
```

Рисунок 5.2 – Генерація звітів про покриття коду

5.1.17. Вбудований інтерфейс тестування React.

Для тестування React-компонентів Jest має вбудований пакет @testing-library/react, який дозволяє легко тестувати різні аспекти React-застосунків.

```
javascriptCopy code
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent'; test('тест
компоненту', () => { render(<MyComponent />); const
linkElement = screen.getByText(/Hello, World/i);
expect(linkElement).toBeInTheDocument(); });
```

Це лише кілька з можливостей фреймворка Jest. Завдяки своїм розширеним функціоналом, Jest став одним із найпопулярніших інструментів для тестування JavaScript-додатків, забезпечуючи швидкість, надійність та зручний синтаксис.

5.2 Детальний розбір бібліотеки SuperTest.

SuperTest - це бібліотека для тестування API в середовищі Node.js. Вона базується на іншій популярній бібліотеці SuperAgent і надає зручний та ефективний інтерфейс для виконання HTTP-запитів та перевірки відповідей сервера. Нижче подано детальний розбір можливостей SuperTest.

5.2.1. Синтаксис та інтеграція.

SuperTest використовує простий та інтуїтивно зрозумілий синтаксис, що дозволяє легко виконувати HTTP-запити. Вона може легко інтегруватися з іншими фреймворками для тестування, такими як Mocha чи Jest.

```
javascriptCopy code
const request = require('supertest'); const app =
require('../app'); request(app) .get('/api/users')
.expect(200) .end((err, res) => { // Обробка відповіді сервера
});
```

5.2.2. Підтримка різних HTTP-методів.

SuperTest підтримує різні HTTP-методи, включаючи GET, POST, PUT, DELETE, PATCH і багато інших. Це дозволяє вам легко взаємодіяти з різними ендпоінтами API.

```
javascriptCopy code
request(app) .post('/api/users') .send({ name: 'John Doe' })
.set('Accept', 'application/json') .expect(201) .end((err,
res) => { // Обробка відповіді сервера після POST-запиту });
```

5.2.3. Перевірка заголовків та тіла відповіді.

SuperTest дозволяє вам перевіряти різні аспекти відповіді сервера, такі як HTTP-статус, заголовки та тіло відповіді.

```
javascriptCopy code
request(app) .get('/api/users/1') .expect('Content-Type',
/json/) .expect(200) .end((err, res) => { // Перевірка
заголовків та тіла відповіді });
```

5.2.4. Можливість використання Cookies та Сесій.

SuperTest дозволяє робити HTTP-запити з використанням кукісів та управляти сесіями, що дуже важливо для тестування веб-додатків, які вимагають автентифікації.

```
javascriptCopy code
request(app) .get('/api/private') .set('Cookie',
'token=your_auth_token') .expect(200) .end((err, res) => { //
Обробка відповіді сервера для захищених ресурсів });
```

5.2.5. Асинхронна підтримка.

SuperTest підтримує асинхронний код, що робить його ідеальним варіантом для тестування веб-додатків, які використовують асинхронні операції.

```
javascriptCopy code
request(app) .get('/api/data') .expect(200) .then((res) => {
// Обробка асинхронного коду після виконання запиту })
.catch((err) => { // Обробка помилок });
```

5.2.6. Підтримка JSON та форм-даних.

SuperTest легко взаємодіє з JSON-даними та формами для передачі даних на сервер.

```
javascriptCopy code
request(app) .post('/api/users') .send({ name: 'John Doe' })
.set('Accept', 'application/json') .expect(201) .end((err,
res) => { // Обробка відповіді сервера після POST-запиту });
```

5.2.7. Виведення тестових запитів.

SuperTest може виводити тестові запити в консоль для подальшого аналізу та відлагодження.

```
javascriptCopy code
const testRequest = request(app) .get('/api/data')
.expect(200); console.log(testRequest); // Виведення тестового
запиту в консоль
```

5.2.8. Легка інтеграція з іншими фреймворками тестування.

SuperTest легко інтегрується з різними фреймворками тестування, такими як Mocha, Jest, або іншими, що дозволяє вам використовувати його в вашому улюбленому середовищі.

```
javascriptCopy code
const request = require('supertest'); const app =
require('../app'); describe('API тести', () => { it('повинен
повернути статус 200 для GET-запиту', (done) => { request(app)
.get('/api/data') .expect(200, done); }); });
```

5.2.9. Підтримка таймаутів та повторів.

SuperTest дозволяє встановлювати таймаути для тестових запитів і надає можливість повторювати запити в разі необхідності.

```
javascriptCopy code
request(app) .get('/api/data') .timeout(5000) // Таймаут у
мілісекундах .retry(3) // Кількість повторів в разі невдалого
тестування .expect(200) .end((err, res) => { // Обробка
відповіді сервера });
```

Це лише декілька ключових можливостей бібліотеки SuperTest. Завдяки своїй простоті та потужності, вона стала популярним інструментом для тестування веб-серверів та API в середовищі Node.js.

5.2.10. Організація тестових середовищ.

SuperTest надає можливість легко організовувати та використовувати тестові середовища для різних частин вашого додатку чи API. Це дозволяє вам ефективно виконувати тести в різних конфігураціях.

```
javascriptCopy code
const request = require('supertest'); const app =
require('../app'); const testEnvironment = request.agent(app);
testEnvironment .get('/api/data') .expect(200) .end((err, res)
=> { // Обробка відповіді сервера в рамках тестового
середовища });
```

5.2.11. Підтримка організації тестових сценаріїв.

SuperTest дозволяє організовувати тести в окремі тестові сценарії, що полегшує їхнє управління та робить код тестів більш зрозумілим.

```
javascriptCopy code
const request = require('supertest'); const app =
require('../app'); describe('API Тести', () => { it('повинен
повернути статус 200 для GET-запиту', (done) => { request(app)
.get('/api/data') .expect(200, done); }); it('повинен
повернути статус 404 для неправильного шляху', (done) => {
request(app) .get('/api/invalid') .expect(404, done); }); });
```

5.2.12. Виведення деталей запиту та відповіді.

SuperTest дозволяє виводити детальну інформацію про кожний тестовий запит та отриману відповідь, що значно полегшує відлагодження.

```
javascriptCopy code
const testRequest = request(app) .get('/api/data')
.expect(200); console.log(testRequest.req); // Деталі
тестового запиту console.log(testRequest.res); // Деталі
отриманої відповіді
```

5.2.13. Використання Promise та Async/await.

SuperTest підтримує використання Promise та може бути легко інтегрована з async/await, що робить код більш читабельним та зручним для роботи з асинхронним кодом.

```
javascriptCopy code
const response = await request(app).get('/api/data');
expect(response.status).toBe(200);
```

5.2.14. Підтримка тестового збільшення навантаження.

SuperTest може бути використана для тестування навантаження (load testing), дозволяючи вам виміряти, як добре ваш сервер витримує навантаження при великій кількості одночасних запитів.

```
javascriptCopy code
const numRequests = 100; const requests = Array.from({ length:
numRequests }, () => request(app).get('/api/data') );
Promise.all(requests).then((responses) => { // Обробка
відповідей для тестового збільшення навантаження });
```

5.2.15. Розширюваність та плагіни.

SuperTest може бути розширена за допомогою плагінів, дозволяючи розробникам додавати власний функціонал та підтримку для різних аспектів тестування.

```
javascriptCopy code
const plugin = (req) => { // Додавання власного функціоналу до
тестового запиту req.customFunction = () => { // Логіка вашого
плагіну }; }; request(app) .get('/api/data') .use(plugin)
.end((err, res) => { res.req.customFunction(); // Використання
функціоналу плагіну });
```

SuperTest - це потужний інструмент для тестування API в середовищі Node.js, який надає зручний та ефективний інтерфейс для виконання HTTP-запитів та перевірки відповідей сервера. Його можливості дозволяють

створювати структуровані, надійні та зручні тести для вашого веб-додатку чи API.

5.3 Детальний розбір Playwright.

Playwright - це сучасний, потужний та багатофункціональний фреймворк для автоматизації браузерів. Розроблений компанією Microsoft, Playwright надає засоби для автоматизації веб-додатків у Chrome, Firefox та WebKit (Safari) браузерах. Нижче подано детальний розбір можливостей Playwright.

5.3.1. Багатофункціональність.

Playwright надає повний доступ до браузера, дозволяючи автоматизувати велику кількість сценаріїв, включаючи навігацію, маніпуляції з DOM, обробку подій, перетягування та введення даних. Це дозволяє максимально покрити можливі сценарії використання програмного продукту та надає широкий та легкий інструментарій для написання тестів.

```
javascriptCopy code
const { chromium } = require('playwright'); (async () => {
const browser = await chromium.launch(); const context = await
browser.newContext(); const page = await context.newPage();
await page.goto('https://example.com'); await page.click('a');
// Інші дії з браузером... await browser.close(); })();
```

5.3.2. Підтримка різних браузерів.

Playwright підтримує три популярні браузери: Chromium, Firefox та WebKit (Safari). Це дозволяє розробникам перевіряти та впевнюватися в сумісності своїх додатків з різними браузерами.

```
javascriptCopy code
const { chromium, firefox, webkit } = require('playwright');
const browserType = process.env.BROWSER || 'chromium'; const
browser = await { chromium, firefox, webkit
}[browserType].launch();
```

5.3.3. Асинхронна підтримка та Promise API.

Playwright повністю асинхронний та використовує Promise API. Це дозволяє зручно використовувати async/await для управління асинхронним кодом.

```
javascriptCopy code
const { chromium } = require('playwright'); (async () => {
const browser = await chromium.launch(); const context = await
browser.newContext(); const page = await context.newPage();
await page.goto('https://example.com'); await
page.screenshot({ path: 'example.png' }); await
browser.close(); })();
```

5.3.4. Робота з багатьма сторінками, мультиплікація.

Playwright надає можливість створювати та працювати з багатьма сторінками браузера в одному контексті, а також відстежувати їхню активність.

```
javascriptCopy code
const context = await browser.newContext(); const page1 =
await context.newPage(); const page2 = await
context.newPage(); await
page1.goto('https://example.com/page1'); await
page2.goto('https://example.com/page2');
```

5.3.5. Засоби для тестування продуктивності.

Playwright має вбудовані засоби для вимірювання продуктивності веб-додатків, такі як засоби збору метрик про завантаження сторінки, емуляція

швидкості мережі та різних типів пристроїв.

```
javascriptCopy code
const metrics = await page.metrics(); console.log(metrics);
```

5.3.6. Обробка подій та введення користувача.

Playwright дозволяє емулювати різні події та введення користувача, такі як кліки, натискання клавіш, рухи миші тощо.

```
javascriptCopy code
await page.click('button'); await page.type('input', 'Hello,
Playwright!');
```

5.3.7. Взаємодія з фреймами та IFrames.

Playwright дозволяє легко взаємодіяти з фреймами та IFrames на сторінці, що полегшує тестування складних веб-додатків.

```
javascriptCopy code
const frame = await page.frame({ url: /example/ }); await
frame.click('button');
```

5.3.8. Тестування мобільних веб-додатків.

Playwright підтримує тестування мобільних веб-додатків, дозволяючи емулювати різні пристрої та конфігурації екрану.

```
javascriptCopy code
const context = await browser.newContext({ ...devices['iPhone
11'] }); const page = await context.newPage(); await
page.goto('https://example.com');
```

5.3.9. Синхронізація та очікування умов.

Playwright автоматично встановлює з'єднання та синхронізує дії з веб-додатком, забезпечуючи надійність тестів, особливо при роботі з асинхронним кодом.

```
javascriptCopy code
await page.waitForSelector('button'); await
page.click('button');
```

5.3.10. Скріншоти та відеозаписи тестів:.

Playwright дозволяє легко створювати скріншоти сторінок та записувати відео тестів для подальшого аналізу.

```
javascriptCopy code
await page.screenshot({ path: 'screenshot.png' }); await
page.video().startRecording(); await
page.goto('https://example.com'); const video = await
page.video().stopRecording(); await
video.saveAs('recording.mp4');
```

Це лише кілька з ключових можливостей та функцій, які надає фреймворк Playwright. Завдяки своїй високій продуктивності, гнучкості та зручному API, Playwright став одним із найпопулярніших інструментів для автоматизації веб-додатків та тестування їх функціональності.

5.3.11. Інтеграція з іншими інструментами.

Playwright може бути легко інтегрований з різними інструментами та фреймворками тестування, такими як Jest, Mocha, чи іншими.

```
javascriptCopy code
const { test, expect } = require('@playwright/test');
test('приклад тесту Playwright', async ({ page }) => { await
page.goto('https://example.com'); const title = await
page.title(); expect(title).toBe('Example Domain'); });
```

5.3.12. Обмеження та підтримка конфігурацій.

Playwright дозволяє налаштовувати та обмежувати роботу браузера в різних конфігураціях, таких як обмеження швидкості мережі, часу відповіді сервера, розмір екрану тощо.

```
javascriptCopy code
const context = await browser.newContext({ viewport: { width:
1920, height: 1080 }, permissions: ['geolocation'] });
```

5.3.13. Відлагодження та інспектор браузера.

Playwright дозволяє використовувати відлагодження з браузером, а також використовувати вбудований інспектор для аналізу сторінок та їхнього стану.

```
javascriptCopy code
await page.evaluate(() => { debugger; // Зупинка виконання
коду для відлагодження });
```

5.3.14. Сценарії тестування інтерактивності.

Playwright може бути використаний для тестування інтерактивності веб-додатків, таких як перетягування, зміна розмірів вікна, клік та інші взаємодії користувача.

```
javascriptCopy code
const element = await page.$('button'); await
element.dragAndDropTo('input');
```

5.3.15. Мови та підтримка тестування на віддалених серверах.

Playwright підтримує різні мови програмування, такі як JavaScript, TypeScript, Python, та інші. Крім того, він може бути використаний для тестування на віддалених серверах, що полегшує тестування у розподілених команд.

```
typescriptCopy code
const { test, expect } = require('@playwright/test');
test('приклад тесту Playwright', async ({ page }) => { await
page.goto('https://example.com'); const title = await
page.title(); expect(title).toBe('Example Domain'); });
```

5.3.16. Взаємодія з локальним сервером та мокованими даними.

Playwright може взаємодіяти з локальним сервером та мокованими даними, що робить його ідеальним для тестування на різних етапах розробки.

```
javascriptCopy code
await page.route('**/*.png', route => route.abort()); await
page.goto('http://localhost:3000');
```

5.3.17. Підтримка регулярних виразів для селекторів.

Playwright підтримує використання регулярних виразів для вибору елементів на сторінці, що полегшує навігацію та взаємодію з DOM.

```
javascriptCopy code
await page.click('text=Click me');
```

5.3.18. Підтримка темниць та робота з Cookies.

Playwright дозволяє працювати з темницями (браузерні контексти) та маніпулювати кукісами для тестування різних сценаріїв.

```
javascriptCopy code
const context = await browser.newContext(); const page = await
context.newPage(); await page.goto('https://example.com');
await context.addCookies([{ name: 'example', value: '123' }]);
```

Playwright - це потужний та гнучкий фреймворк для автоматизації тестування веб-додатків. Його багатофункціональність, підтримка різних браузерів та широкі можливості для взаємодії з веб-додатками роблять його одним з популярних інструментів для автоматизації та тестування веб-додатків.

5.4 Установка Jest.

Для встановлення Jest у проект виконайте:

```
npm install --save-dev jest
```

Якщо ви використовуєте yarn:

```
yarn add --dev jest
```

Після інсталяції можете оновити секцію scripts вашого package.json:

```
“scripts” : {
     “test”: “jest”
}
```

За допомогою такого простого виклику ми вже можемо запустити тести (насправді jest вимагатиме існування хоча б одного тесту).

Також можна встановити глобально:

```
npm install jest --global
```

І відповідно для yarn:

```
yarn global add jest
```

Після цього ви можете використовувати jest безпосередньо з командного рядка.

За допомогою виклику команди jest --init в корені проекту, відповівши на кілька запитань, ви отримаєте файл із налаштуваннями jest.config.js. Або можна додати конфігурацію прямо у ваш package.json. Для цього додайте в корінь json ключ «jest» і у відповідному об'єкті можете додавати необхідні вам налаштування.

Самі налаштування ми розберемо пізніше. На цьому етапі в цьому немає потреби, оскільки jest можна використовувати «відразу», без додаткових конфігурацій.

5.5 Запуск тестування Jest.

Створимо файл first.test.js та напишемо перший тест:

```
//first.test.js
test('My first test', () => {
    expect(Math.max(1, 5, 10)).toBe(10);
});
```

Запустимо тести за допомогою npm run test або безпосередньо командою jest (якщо вона встановлена глобально). Після запуску побачимо звіт про проходження тестів.

```
 <b>PASS</b>  ./first.test.js
  ✓ My first test (1 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.618 s, estimated 1 s
```

Давайте "зламаємо" тест і запустимо jest повторно:

```
//first.test.js
test('My first test', () => {
    expect(Math.max(1, 5, 10)).toBe(5);
});
```

Як бачимо, тепер тест не проходить перевірки. Jest відображає детальну інформацію про те, де виникла проблема, який був очікуваний результат, і що ми отримали замість нього.

Розберемо код тесту. Функція test використовується для створення нового тесту. Вона приймає три аргументи (у прикладі ми використали виклик із двома аргументами). Перший — рядок із назвою тесту, його jest відобразить у звіті. Другий – функція, яка містить логіку нашого тесту. Також можна використовувати 3-й аргумент – тайм-аут. Він не є обов'язковим, а його значення за замовчуванням становить 5 секунд.

Задається у мілісекундах.

Цей параметр необхідний, коли ми працюємо з асинхронним кодом і повертаємо з функції тесту проміс. Він вказує, як довго jest повинен чекати дозволу промісу. Після закінчення цього часу, якщо проміс не був дозволений - jest вважатиме тест не пройденим. Докладніше про роботу з асинхронними дзвінками буде в наступних частинах. Також замість test() можна використовувати it(). Різниці між такими викликами немає. it() це просто аліас на функцію test().

Розберемо основні з цих методів:.toBe повертає об'єкт «обгортку», яка має низку методів для зіставлення отриманого значення з очікуваним. Один із таких методів ми й використовували — expect(). Math.max(1, 5, 10). Йому ми передаємо значення, яке хочемо перевірити. У нашому випадку це результат виклику expect()

Всередині функції тесту ми спочатку викликаємо

- **toBe()** — підходить, якщо потрібно порівнювати примітивні значення чи є передане значення посиланням на той самий об'єкт, що вказаний як очікуване значення. Порівнюються значення за допомогою Object.is(). На відміну від === це дозволяє відрізняти 0 від -0, перевірити рівність NaN с NaN.
- **toEqual()** —якщо необхідно порівняти структуру складніших типів. Він порівняє всі поля переданого об'єкта з очікуваним. Перевірить кожний елемент масиву. І зробить це рекурсивно по всій вкладеності.

```
test('toEqual with objects', () => {
    expect({ foo: 'foo', subObject: { baz: 'baz' } })
        .toEqual({ foo: 'foo', subObject: { baz: 'baz' } });  // Ок
    expect({ foo: 'foo', subObject: { num: 0 } })
        .toEqual({ foo: 'foo', subObject: { baz: 'baz' } });  // А вот так ошибка.
});
test('toEqual with arrays', () => {
    expect([11, 19, 5]).toEqual([11, 19, 5]); // Ок
    expect([11, 19, 5]).toEqual([11, 19]); // Ошибка
});
```

- **toContain()** — перевіряють чи містить масив чи ітерований об'єкт значення. Для порівняння використовується оператор ===.

```
const arr = ['apple', 'orange', 'banana'];
expect(arr).toContain('banana');
expect(new Set(arr)).toContain('banana');
expect('apple, orange, banana').toContain('banana');
```

- **toContainEqual()** — перевіряє чи містить масив елемент із очікуваною структурою.

```
expect([{a: 1}, {b: 2}]).toContainEqual({a: 1});
```

- **toHaveLength()** — перевіряє чи властивість length об'єкта відповідає очікуваному.

```
expect([1, 2, 3, 4]).toHaveLength(4);
expect('foo').toHaveLength(3);
expect({ length: 1 }).toHaveLength(1);
```

- **toBeNull()** — перевіряє на рівність з null.
- **toBeUndefined()** — перевіряє на рівність з undefined.

- **toBeDefined()** — протилежність toBeUndefined. Перевіряє чи значення !== undefined.
- **toBeTruthy()** — перевіряє чи в бульовому контексті значення відповідає true. Тобто будь-які значення крім false, null, undefined, 0, NaN та порожніх рядків.
- **toBeFalsy()** — протилежність toBeTruthy(). Перевіряє чи в булевому контексті значення відповідає false.
- **toBeGreaterThan()** і **toBeGreaterThanOrEqual()** — перший метод перевіряє чи передане числове значення більше, ніж очікуване >, другий перевіряє більше або дорівнює очікуваному >=.
- **toBeLessThan()** і **toBeLessThanOrEqual()** — протилежність toBeGreaterThan() та toBeGreaterThanOrEqual()
- **toBeCloseTo()** — зручно використовувати для чисел із плаваючою комою, коли вам не важлива точність і ви не хочете, щоб тест залежав від незначної різниці в дробі. Другим аргументом можна передати до якогось знака після коми необхідна точність при порівнянні.

```
const num = 0.1 + 0.2; // 0.30000000000000004
expect(num).toBeCloseTo(0.3);
expect(Math.PI).toBeCloseTo(3.14, 2);
```

- **toMatch()** — перевіряє відповідність рядка регулярному виразу.

```
expect('Banana').toMatch(/Ba/);
```

- **toThrow()** — використовується у випадках, коли треба перевірити виняток. Можна перевірити як факт помилки, так і перевірити на викид виключення певного класу, або за повідомленням помилки, або за відповідністю повідомлення регулярному виразу.

```
function funcWithError() {
    throw new Error('some error');
}
expect(funcWithError).toThrow();
expect(funcWithError).toThrow(Error);
expect(funcWithError).toThrow('some error');
expect(funcWithError).toThrow(/some/);
```

- **not** — це властивість дозволяє перевірити нерівність. Вона надає об'єкт, який має перераховані вище методи, але працюватимуть вони навпаки.

```
expect(true).not.toBe(false);
expect({ foo: 'bar' }).not.toEqual({});
function funcWithoutError() {}
expect(funcWithoutError).not.toThrow();
```

Напишемо кілька простих тестів. Для початку створимо простий модуль, який міститиме кілька методів для роботи з колами.

```
// src/circle.js
const area = (radius) => Math.PI * radius ** 2;
const circumference = (radius) => 2 * Math.PI * radius;
module.exports = { area, circumference };
```

Далі додамо тести:

```
// tests/circle.test.js
const circle = require('../src/circle');
test('Circle area', () => {
    expect(circle.area(5)).toBeCloseTo(78.54);
    expect(circle.area()).toBeNaN();
});
test('Circumference', () => {
    expect(circle.circumference(11)).toBeCloseTo(69.1, 1);
    expect(circle.circumference()).toBeNaN();
});
```

У цих тестах ми перевірили результат роботи 2-х методів –area та circumference. За допомогою методу toBeCloseTo ми звірилися з очікуваним результатом.

У першому випадку ми перевірили чи обчислювана площа кола з радіусом 5 приблизно дорівнює 78.53, при цьому різниця з отриманим значенням (воно становитиме 78.53871633973483) не велика і тест буде зарахований.

У другому ми зазначили, що нас цікавить перевірка з точністю до 1 знака після коми. Також ми викликали наші методи без аргументів та перевірили результат за допомогою toBeNaN. Оскільки результат виконання буде NaN, то й тести будуть пройдені успішно.

Розберемо ще один приклад. Створимо функцію, яка фільтруватиме масив продуктів за ціною:

```
// src/productFilter.js
const byPriceRange = (products, min, max) =>
        products.filter(item => item.price >= min && item.price
<= max);
module.exports = { byPriceRange };
```

І додамо тест:

```
// tests/product.test.js
const productFilter = require('../src/producFilter');
const products = [
    { name: 'onion', price: 12 },
    { name: 'tomato', price: 26 },
    { name: 'banana', price: 29 },
    { name: 'orange', price: 38 }
];
test('Test product filter by range', () => {
    const FROM = 15;
    const TO = 30;
    const  filteredProducts  =  productFilter.byPriceRange(products,
FROM, TO);
    expect(filteredProducts).toHaveLength(2);
    expect(filteredProducts).toContainEqual({    name:    'tomato',
price: 26 });
    expect(filteredProducts).toEqual([{  name:  'tomato',  price:  26
}, { name: 'banana', price: 29 }]);

expect(filteredProducts[0].price).toBeGreaterThanOrEqual(FROM);
    expect(filteredProducts[1].price).toBeLessThanOrEqual(TO);
    expect(filteredProducts).not.toContainEqual({  name:  'orange',
price: 38 });
});
```

У цьому тесті ми перевіримо результат роботи функції byRangePrice. Спочатку ми перевірили відповідність довжини отриманого масиву очікуваної - 2. Наступна перевірка вимагає, щоб у масиві знаходився елемент - {name: 'tomaato', price: 27}. Об'єкт у масиві та об'єкт переданий toContainEqual — це два різні об'єкти, а не посилання на один і той же. Але toContainEqual звірить кожну властивість. Оскільки обидва об'єкти ідентичні, перевірка пройде успішно.

Далі використовуємо для перевірки структури всього масиву та його елементів. Методи toBeGreaterThanOrEqual та toBeLessThanOrEqual допоможуть нам перевірити price першого та другого елемента масиву. І,

нарешті, виклик not.toContainEqual зробить перевірку, чи не міститься в масиві елемент — { name: 'orangee', price: 39 }, якого за умовою там бути не повинно. У цих прикладах приведено кілька простих тестів, використовуючи функції перевірки описані вище.

5.6 Установка Playwright.

Для початку роботи, необхідно мати встановленим npm або yarn. Далі в командному рядку виконуємо наступну команду, щоб встановити пакет:

```
npm init playwright@latest
```

Рисунок 5.3 – Установка Playwright

Під час встановлення необхідно буде зробити декілька виборів:

- Вибрати між TypeScript та JavaScript (по замовчування використовується TypeScript)
- Назва папки для тестів
- Чи встановлювати підтримувані Playwright браузери (по замовчуванню так)

В результаті в обраній локації будуть створені наступні файли

```
playwright.config.ts
package.json
package-lock.json
tests/
  example.spec.ts
tests-examples/
  demo-todo-app.spec.ts
```

Рисунок 5.4 – Конфігурування Playwright

Playwright.config це файл де ви можете конфігурувати виконання тестів, включаючи вибір браузерів для тестування. Якщо ви запускаєте тести в уже існуючому проекті, тоді залежності будуть додані безпосередньо до вашого package.json.

Папка tests містить приклади тестів для швидкого ознайомлення з функціоналом. Для більших деталей можна заглянути в папку tests-examples , яка містить більш складні тести в якості прикладів.

6 ОХОРОНА ПРАЦІ

Охорона праці - система законодавчих актів, соціально-економічних, організаційних, технічних, гігієнічних і лікувально-профілактичних заходів і засобів, що забезпечують безпеку, збереження здоров'я й працездатності людини в процесі праці. Охорона здоров'я працівників, забезпечення безпеки умов праці, ліквідація професійних захворювань і виробничого травматизму становить одну з головних турбот людського суспільства. Звертається увага на необхідність широкого застосування прогресивних форм наукової організації праці, зведення до мінімуму ручної, малокваліфікованої праці, створення обстановки, що виключає професійні захворювання й виробничий травматизм.

Даний розділ дипломного проекту присвячений розгляду наступних питань:

- визначення оптимальних умов праці оператора системи;
- розрахунок вентиляції;

6.1 Визначення оптимальних умов праці інженера-оператора системи мікроклімату

Проектування робочих місць, обладнаних відеотерміналами, відносять до числа найважливіших проблем ергономічного проектування в галузі обчислювальної техніки.

Робоче місце й взаємне розташування всіх його елементів повинне відповідати антропометричним, фізичним і психологічним вимогам. Велике значення має також характер роботи. Зокрема, при організації робочого місця оператора повинні бути дотримані наступні основні умови:

- оптимальне розміщення устаткування, що входить до складу робочого місця;
- достатній робочий простір, що дозволяє здійснювати всі необхідні рухи й переміщення;

- необхідно природне й штучне освітлення для виконання поставлених завдань;
- рівень акустичного шуму не повинен перевищувати припустимого значення.
- достатня вентиляція робочого місця;

Ергономічними аспектами проектування відеотермінальних робочих місць, зокрема, є: висота робочої поверхні, розміри простору для ніг, вимоги до розташування документів на робочому місці (наявність і розміри підставки для документів, можливість різного розміщення документів, відстань від очей користувача до екрана, документа, клавіатури й т.д.), характеристики робочого крісла, вимоги до поверхні робочого стола, регульованість робочого місця і його елементів.

Головними елементами робочого місця оператора є стіл і крісло. Основним робочим положенням є положення сидячи. Робоча поза сидячи викликає мінімальне стомлення оператора. Раціональне планування робочого місця передбачає чіткий порядок і сталість розміщення предметів, засобів праці й документації. Те, що потрібно для виконання робіт частіше, розташовано в зоні легкої досяжності робочого простору. Моторне поле - простір робочого місця, у якому можуть здійснюватися рухові дії людини.

Максимальна зона досяжності рук - це частина моторного поля робочого місця, обмеженого дугами, описуваними максимально витягнутими руками при русі їх у плечовому суглобі.

Оптимальна зона - частина моторного поля робочого місця, обмеженого дугами, описуваними передпліччями при русі в ліктьових суглобах з опорою в крапці ліктя й з відносно нерухливим плечем.

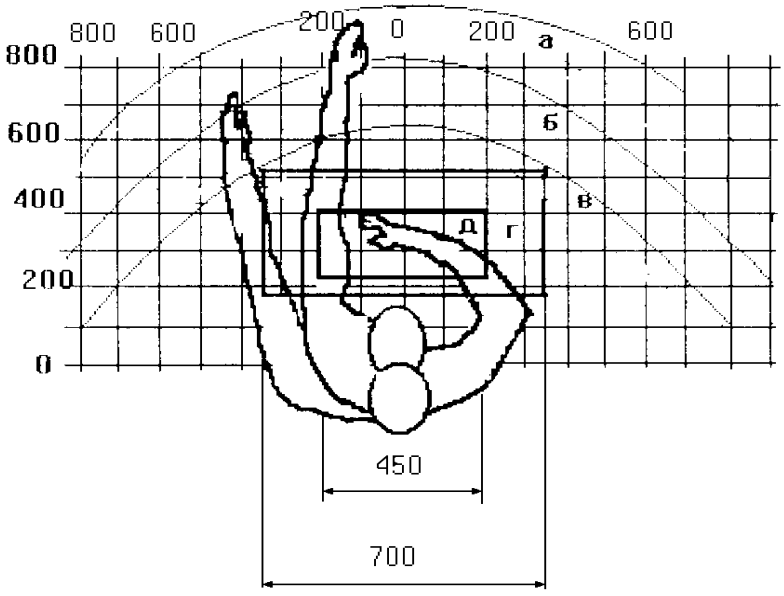


Рисунок 6.1 – Зони досяжності рук у горизонтальній площині.

а - зона максимальної досяжності;

б - зона досяжності пальців при витягнутій руці;

в - зона легкої досяжності долоні;

г - оптимальний простір для грубої ручної роботи;

д - оптимальний простір для тонкої ручної роботи.

Розглянемо оптимальне розміщення предметів праці й документації в зонах досяжності рук:

ДИСПЛЕЙ розміщається в зоні а (у центрі);

КЛАВІАТУРА - у зоні г/д;

СИСТЕМНИЙ БЛОК розміщається в зоні б (ліворуч);

ПРИНТЕР перебуває в зоні а (праворуч);

ДОКУМЕНТАЦІЯ

1) у зоні легкої досяжності долоні - в (ліворуч) - література й документація, необхідна при роботі;
2) у висувних ящиках стола - література, не використовувана постійно.

При проектуванні робочого стола варто враховувати наступне:

- висота стола повинна бути обрана з урахуванням можливості сидіти вільно, у зручній позі, при необхідності опираючись на підлокітники;

- нижня частина стола повинна бути сконструйована так, щоб програміст міг зручно сидіти, не був змушений підтискати ноги;
- поверхня стола повинна мати властивості, що виключають появу відблисків у полі зору оператора;
- конструкція стола повинна передбачати наявність висувних ящиків (не менш 3 для зберігання документації, лістингів, канцелярських засобів, особистих речей).

Висота робочої поверхні рекомендується в межах 680-760 мм. Висота робочої поверхні, на яку встановлюється клавіатура, повинна бути 650 мм. Велике значення надається характеристикам робочого крісла. Так, рекомендується висота сидіння над рівнем підлоги повинна бути в межах 420-550 мм. Поверхня сидіння рекомендується робити м'якої, передній край закругленим, а кут нахилу спинки робочого крісла - регульованою.

Необхідно передбачати при проектуванні можливість різного розміщення документів: збоку від відеотермінала, між монітором і клавіатурою й т.п. Крім того, у випадках, коли відеотермінал має низьку якість зображення, відстань від очей до екрана роблять більше (близько 700 мм), відстань від ока до документа (300-450 мм). Взагалі при високій якості зображення на відеотерміналі відстань від очей користувача до екрана, документа й клавіатури можуть бути рівним.

Положення екрана визначається:

- відстанню зчитування (0.60 + 0.10 м);

- кутом зчитування, напрямком погляду на 20 нижче горизонталі до центра екрана, причому екран перпендикулярний цьому напрямку.

Повинна передбачатися можливість регулювання екрана:

- по висоті +3 см;

- по нахилу від 10 до 20 щодо вертикалі;

- у лівому і правом напрямках.

Зоровий комфорт підкоряється двом основним вимогам:

- чіткості на екрані, клавіатурі й у документах;

- освітленості й рівномірності яскравості між навколишніми об’єктами й різними ділянками робочого місця;

Велике значення також надається правильній робочій позі користувача. При незручній робочій позі можуть з'явитися болі в м'язах, суглобах і сухожиллях. Вимоги до робочої пози користувача відеотермінала наступні: шия не повинна бути нахилена більш ніж на 20˚ (між віссю " голова-шия" і віссю тулуба), плечі повинні бути розслаблені, лікті - перебувати під кутом 80˚ - 100˚, передпліччя й кисті рук - у горизонтальному положенні.

Характеристики використовуваного робочого місця:

- висота робочої поверхні стола 750 мм;
- висота простору для ніг 650 мм;
- висота сидіння над рівнем підлоги 450 мм;
- поверхня сидіння м'яка із закругленим переднім краєм;
- передбачена можливість розміщення документів праворуч і ліворуч;
- відстань від ока до екрана 700 мм;
- відстань від ока до клавіатури 400 мм;
- відстань від ока до документів 500 мм;
- можливе регулювання екрана по висоті, по нахилу, у лівому і в правому напрямках;

Створення сприятливих умов праці й правильне естетичне оформлення робочих місць на виробництві має велике значення як для полегшення праці, так і для підвищення його привабливості, що позитивно впливає на продуктивність праці. При розробці оптимальних умов праці оператора необхідно враховувати освітленість, шум і мікроклімат.

6.2 Розрахунок освітленості робочого місця

Раціональне освітлення робочого місця є одним з найважливіших факторів, що впливають на ефективність трудової діяльності людини, що попереджають травматизм і професійні захворювання. Правильно організоване освітлення створює сприятливі умови праці, підвищує

працездатність і продуктивність праці. Освітлення на робочому місці оператора повинне бути таким, щоб працівник міг без напруги зору виконувати свою роботу. Стомлюваність органів зору залежить від ряду причин:

- недостатність освітленості;
- надмірна освітленість;
- неправильний напрямок світла.

Недостатність освітлення приводить до напруги зору, послабляє увагу, приводить до настання передчасної стомленості. Надмірно яскраве освітлення викликає осліплення, роздратування й різь в очах. Неправильний напрямок світла на робочому місці може створювати різкі тіні, відблиски, дезорієнтувати працюючого.

Розрахунок освітленості робочого місця зводиться до вибору системи освітлення, визначенню необхідного числа світильників, їхнього типу й розміщення. Процес роботи оператора в таких умовах, коли природне освітлення недостатнє або відсутнє. Виходячи із цього, розрахуємо параметри штучного освітлення.

Штучне освітлення забезпечується за допомогою електричних джерел світла двох видів: ламп накалювання й люмінесцентних ламп. Будемо використовувати люмінесцентні лампи, які в порівнянні з лампами накалювання мають істотні переваги:

- по спектральному составі світла вони близькі до денного, природного освітлення;
- володіють більше високим ККД (в 1.5-2 рази вище, ніж ККД ламп накалювання);
- мають підвищену світловіддачу (в 3-4 рази вище, ніж у ламп накалювання);
- більше тривалий термін служби.

Розрахунок освітлення виробляється для кімнати площею 36 $м^2$, ширина якої 4.9 м, висота - 4.2 м. Скористаємося методом світлового потоку.

Для визначення кількості світильників визначимо світловий потік, що падає на поверхню за формулою:

$$F = \frac{E \cdot K \cdot S \cdot Z}{n}$$, де

F - світловий потік, що розраховується, Лм;

E - нормована мінімальна освітленість, Лк (визначається по таблиці). Роботу оператора, відповідно до цієї таблиці, можна віднести до розряду точних робіт, отже, мінімальна освітленість буде E = 300 Лк при газорозрядних лампах;

S - площа освітлюваного приміщення (у нашім випадку S = 36 м^2);

Z - відношення середньої освітленості до мінімального (звичайно приймається рівною 1.1-1.2 , нехай Z = 1.1);

K - коефіцієнт запасу, що враховує зменшення світлового потоку лампи в результаті забруднення світильників у процесі експлуатації (його значення визначається по таблиці коефіцієнтів запасу для різних приміщень і в нашім випадку K = 1.5);

n - коефіцієнт використання, виражається відношенням світлового потоку, що падає на розрахункову поверхню, до сумарного потоку всіх ламп і обчислюється в частках одиниці; залежить від характеристик світильника, розмірів приміщення, колір стін і стелі, характеризованих коефіцієнтами відбиття від стін (Pc) і стелі (Pп), значення коефіцієнтів Pc і Pп визначимо по таблиці залежностей коефіцієнтів відбиття від характеру поверхні: Pc=30%, Pп=50%. Значення n визначимо по таблиці коефіцієнтів використання різних світильників. Для цього обчислимо індекс приміщення за формулою:

$$I = \frac{S}{h(A+B)}$$, де

S - площа приміщення, S = 36 м^2;

h - розрахункова висота підвісу, h = 3.39 м;

A - ширина приміщення, A = 4.9 м;

У - довжина приміщення, В = 6.35 м.

Підставивши значення одержимо:

$$I = \frac{3.6}{3.39 \cdot (4.9+7.35)} = 0.8$$

Знаючи індекс приміщення I, Рс і Рп, по таблиці знаходимо n = 0.28

Підставимо всі значення у формулу для визначення світлового потоку F:

$$F = \frac{300 \cdot 1.5 \cdot 36 \cdot 1.1}{0.28} = 63642.857$$ Лм

Для освітлення вибираємо люмінесцентні лампи типу ЛБ40-1, світловий потік яких F = 4320 Лк.

Розрахуємо необхідну кількість ламп за формулою:

$$N = \frac{F}{F_л}$$, де

N - обумовлене число ламп;

F - світловий потік, F = 63642,857 Лм;

$F_л$- світловий потік лампи, $F_л$ = 4320 Лм.

$$N = \frac{63642.857}{4320} = 15$$ шт.

При виборі освітлювальних приладів використовуємо світильники типу ОД. Кожний світильник комплектується двома лампами. Розміщаються світильники двома рядами, по чотирьох у кожному ряді.

7 БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

Цивільна оборона являє собою систему загальнодержавних заходів, які здійснюються у мирний та воєнний час для захисту населення і народного господарства від зброї масового знищення та інших засобів ураження, наслідків аварій, катастроф, пожеж, а також для проведення рятувальних і невідкладних аварійно-відновлювальних робіт в осередках ураження та в районах стихійних лих (повені, землетрусів, зсувів ґрунту та ін.).

7.1 Основні вражаючі фактори ядерних вибухів, їхні параметри і наслідки впливу на людей

Основними вражаючими факторами ядерного вибуху є: повітряна ударна хвиля, світлове випромінювання, проникаюча радіація, радіоактивне зараження та електромагнітний імпульс. Всі ці вражаючі фактори можуть в різній мірі впливати на функціонування об’єкта.

При ядерному вибуху можуть виникати вторинні вражаючі фактори: пожежі, вибухи, зараження отруйними і сильно діючими отруйними речовинами (СДОР) місцевості, атмосфери та водойм, катастрофічне затоплення в зонах, розміщених нижче дамб гідровузлів, і т. п. Вторинні вражаючі фактори ядерного вибуху в ряді випадків можуть здійснити значний вплив на роботу об’єкта.

Ударна хвиля – основний вражаючий фактор ядерного вибуху. При вибуху в атмосфері на долю ударної хвилі припадає 50 % енергії вибуху. Ударна хвиля представляє собою область сильно стиснутого повітря, що розповсюджується у всі сторони від центра вибуху із надзвуковою швидкістю.

Ударна хвиля має фазу стискування і фазу розрідження. Найбільший тиск повітря спостерігається на зовнішній межі фази стискування – у фронті хвилі.

Ударна хвиля уражає людей, руйнує або пошкоджує будинки, споруди, обладнання, техніку та майно. Ударна хвиля уражає незахищених людей в результаті безпосереднього (прямого), а також непрямого впливу, спричинюючи травми різного ступеня.

При безпосередньому впливі ударної хвилі причиною ураження є надлишковий тиск. При непрямому впливі – люди уражаються шматками зруйнованих будинків, уламками скла та іншими предметами, що переміщаються під дією швидкісного напору.

Світлове випромінювання ядерного вибуху представляє собою електромагнітне випромінювання в ультрафіолетовій, видимій та інфрачервоній областях спектру.

На долю світлового випромінювання припадає 30…40 % всієї енергії атомного або термоядерного вибуху. На відкритій місцевості світлове випромінювання володіє великим радіусом дії порівняно з ударною хвилею і проникаючою радіацією.

Вражаюча дія світлового випромінювання визначається поглинутою частиною енергії світлового імпульсу, яка, перетворюючись в теплову, нагріває опромінюваний об’єкт. Світлове випромінювання, впливаючи на незахищених людей, викликає опіки відкритих ділянок тіла і уражає очі.

В результаті впливу світлового випромінювання на матеріали може відбутися їх короблення, розтріскування, плавлення, обвуглювання або загоряння.

Ступінь ураження будь-якого матеріалу під дією світлового випромінювання при одному і тому ж світловому імпульсі залежить від коефіцієнта поглинання, фізичних властивостей (густини, теплоємності, теплопровідності), товщини матеріалу та інших факторів.

Вплив світлового випромінювання ядерного вибуху на будинки та споруди об’єктів народного господарства проявляється у виникненні загорянь та пожеж, що викликають руйнування і знищення матеріальних

цінностей, в ряді випадків переважаючі за масштабами руйнування від ударної хвилі.

Вторинними вражаючими факторами ядерного вибуху є вибухи, пожежі, затоплення, зараження атмосфери та місцевості, падіння пошкоджених конструкцій будівель, виникаючі в результаті руйнувань та пожеж, викликаних ядерним вибухом.

Ядерний вибух супроводжуються сильними радіоактивними випромінюваннями. Радіоактивними називаються випромінювання, що виникають при радіоактивному розпаді ядер атомів.

Основним параметром, що характеризує вражаючу дію ядерних випромінювань, є поглинена доза радіації (доза опромінення).

Доза радіації – це кількість енергії радіоактивних випромінювань, поглинена одиницею маси опромінюваної речовини.

Другою характеристикою степені впливу випромінювань є потужність дози (рівень радіації) – доза, приведена до одиниці часу (доза, що накопичується протягом одиниці часу).

Радіоактивне випромінювання, яке утворюється безпосередньо при ядерному вибуху, називається проникаючою радіацією.

Ядерні випромінювання, іонізуючи молекули живих тканин, здійснюють шкідливу біологічну дію на людину, що приводить до порушення життєвих функцій окремих органів та систем і до розвитку променевої хвороби.

Радіоактивне зараження – це зараження поверхні землі, атмосфери, водойм і різних предметів радіоактивними речовинами, що випали з хмари ядерного вибуху.

Ядерний вибух супроводжується електромагнітним випромінюванням у вигляді потужного короткого імпульсу, вражаючого головним чином електричну та електронну апаратуру.

Основні параметри електромагнітного імпульсу (ЕМІ).

Основними параметрами ЕМІ, визначаючими вражаючу дію, є характер зміни напруженості електричного та магнітного полів в часі (форма імпульсу) і максимальна напруженість поля (амплітуда імпульсу).

Вражаюча дія ЕМІ.

На утворення ЕМІ витрачається невелика частина ядерної енергії, проте він здатний викликати високі імпульси струмів та напруг в дротах і кабелях повітряних і підземних ліній зв'язку, сигналізації, керування, електропередачі, в антенах радіостанцій і т. п.

Дія ЕМІ може спричинити згоряння чутливих електронних та електричних елементів, зв'язаних з великими антенами або відкритими проводами (електромережа), а також до серйозних порушень в цифрових і контрольних пристроях, зазвичай без безповоротних змін. Отже, вплив ЕМІ необхідно враховувати для всіх електричних і електронних систем. Для найбільш важливих пристроїв потрібно застосовувати засоби захисту і підвищувати їх стійкість до ЕМІ.

Особливістю ЕМІ як вражаючого фактора є його здатність розповсюджуватись на десятки і сотні кілометрів в навколишньому середовищі по різноманітних комунікаціях (мережах електро- та водопостачання, провідного зв'язку і т. п.). Тому ЕМІ може здійснити вплив на об'єкти там де ударна хвиля, світлове випромінювання і проникаюча радіація втрачають своє значення як вражаючі фактори.

7.2 Методи захисту та безпека підприємств промисловості, відновлення інженерно-технічного комплексу цеху (заводу)

Підвищення надійності об'єкту по суті досягається шляхом підсилення найбільш слабких елементів і ділянок об'єкту. Для цього на кожному об'єкті заздалегідь на основі досліджень планується і проводиться великий об'єм робіт, який включає в себе виконання організаційних і інженерно технічних заходів. Особливо важливе значення має проведення інженерно технічних заходів. Досягнення сучасної науки і техніки дозволяють впроваджувати такі

рішення, при яких підприємство буде надійно до дії на нього навіть великих тисків. Але це пов’язано з великими затратами, які можуть бути оправдані тільки гострою необхідністю охорони унікальних, особливо важливих елементів об’єкту.

Для відпрацювання заходів підприємства по підвищенню надійності потрібно підходити обдумано, всесторонньо оцінюючи їх технічну і економічні сторони. Заходи будуть економічно обґрунтовані в тому випадку, якщо вони максимально пов’язані з задачами, які вирішуються в мирний час з ціллю забезпечення безаварійної роботи об'єкту поліпшення умов праці, удосконаленні виробничого процесу. Приміром таких рішень можуть служити використання сховищ для охоронних цілей і обслуговування населення, будівництво підземних ємностей для горючих, отруйних і агресивних рідин і газів. Особливо велике значення має розробка інженерно-технічних заходів при новому будівництві, так як при новому будівництві, так як в процесі проектування в багатьох випадках можна досягнути логічного поєднання загальних інженерних рішень з охоронними заходами ГО, що знизить затрати на їх реалізацію. На діючих об’єктах заходи по підвищенню стійкості їх роботи потрібно проводити в процесі реконструкції або виконанні других ремонтно-будівних робіт.

7.3 Висновки розділу

Основні заходи для рішення задач підвищення безпеки роботи підприємств:

- захист робочих і службовців від зброї масового ураження;
- підвищення надійності важливих елементів об’єктів і вдосконалення технологічного процесу;
- підвищення стійкості матеріально технічного поставок;
- підвищення стійкості управління об’єктом;
- розробка заходів по зменшенню виникнення вторинних факторів враження і наслідків після них;

– підготовка до відновлення виробництва після ураження об’єкту.

Розробка і впровадження заходів по підвищенню стійкості роботи об’єкта в більшості випадків проводиться в мирний час.

Підвищення стійкості технологічного процесу

Насичення сучасних технологічних ліній засобами автоматики, телемеханіки, електронної і напівпровідникової техніки в значній мірі сприяє вдосконаленню технологічних процесів, але в цей час робить ці процеси більш вразливі до вражаючих факторів ядерного вибуху.

Необхідні вимоги надійності технологічного процесу – стійкість системи управління і безперебійне забезпечення всіма видами енергозабезпечення.

Санітарні профілактичні заходи

Санітарна обробка – це комплекс заходів по ліквідації зараження робітників, населення радіоактивними, отруйними речовинами або бактеріальними засобами – складова частина спеціальної обробки. Своєчасне і якісне проведення санітарної обробки: знезаражування поверхні тіла і поверхневих слизистих оболонок, одягу і взуття значно знижують можливість ураження людей, які знаходились в зонах ураження, і в цілому попереджують розповсюдження інфекції за межі зони бактеріологічного зараження. Поділяються вони на частинну і повну.

Під частинною санітарною обробкою розуміється механічна очистка і обробка відкритих ділянок тіла, зовнішніх поверхонь одягу, взуття, засобів індивідуальної безпеки або протирання за допомогою індивідуальних протихімічних пакетів. Вона проводиться в епіцентрі ураження в ході проведення СНАВР, і носить характер недовготривалої і переслідує ціль попередити небезпеку вторинного інфекціонування людей.

Повна санітарна обробка – знезаражування тіла людини дезінфікуючою рецептурою, обмивка людини зі зміною білизни і одягу, дезінфекція (дезінсекція) знятого одягу. Ціль обробки – повне знезаражування від радіоактивних, отруйних речовин одягу, взуття, засобів індивідуального

захисту, поверхнею тіла і слизових оболонок. Повною санітарною обробкою підлягають робочі, службовці і евакуйоване населення після виходу з вогнищ ураження. Усі обмивочні пункти потрібно розміщувати по єдиній схемі, відповідно до якої будуть ставитися інші приміщення : регулювальний пост, площадка зрошення верхнього одягу і взуття, роздягальня, обмивочна, гардероб, а також допоміжні приміщення для складування мішків з зараженим одягом, обмінний фонд одягу і взуття, медичний пункт, кімната матері і дитини, кімната працівників обмивочного пункту, комірка, туалет. Приміщення повинні розділятися на “брудну” і “чисту” половини.

Знезараження одягу, взуття і засобів індивідуального захисту в залежності від ситуації і можливостей проводиться: камерним методом; газовим способом в пристосуваннях камерах, ємностях, приміщеннях; замочуваннях в розчинах дизинфектантів; під час прання в пральних машинах. Можливо також знезаражування одягу парами формальдегіду в поліетиленових мішках при кімнатній температурі. Найбільш реальний метод знезаражування документів – газовий : дією суміші окису етилену і бромистого метилу в поліетиленових мішках при дозуванні 2 мкл препарату на 1 л об’єму при температурі 35 градусів на протязі 1 години.

ВИСНОВОК

Дана магістерська кваліфікаційна стосується розробки засобів автоматизованого тестування програмного забезпечення для підвищення його ефективності, економії часу, ресурсів і витрат.

Завдання автоматизованого тестування програмного забезпечення – зменшити вартість розробки шляхом раннього виявлення дефектів. Відповідно тестування програмного забезпечення є невід'ємною частиною розробки програмних продуктів. Своєчасне виявлення та виправлення помилок і недоробок має велике значення в процесі розробки програмного продукту, оскільки це зменшує ризики і при цьому відбувається суттєве зниження витрат на розробку програмного забезпечення.

Інструменти автоматизованого тестування – це програми, призначені для перевірки функціональних та/або нефункціональних вимог до програмного забезпечення за допомогою сценаріїв автоматизованого тестування.

В даній роботі проаналізовано аналоги платформ тестування програмного забезпечення, визначено їх функціональні можливості. Розглянуто різні сучасні інструменти автоматизованого тестування програмного забезпечення, а саме:

- Платформа Katalon
- Фреймворк Selenium
- Інструмент автоматизації тестування програмного забезпечення Appium
- Функціональне тестування інтерфейсу TestComplete
- Система тестування програмного забезпечення Cypress.
- Автоматизоване тестування GUI для веб-додатків Ranorex Studio
- Хмарна платформа автоматизованого тестування Perfecto
- Хмарна платформа автоматизованого тестування LambdaTest

- Інструмент автоматизованого тестування API Postman
- Інструмент тестування API SoapUI
- Інструмент автоматизації тестування GUI Eggplant, Eggplant Functional
- Комплексний інструмент автоматизації для веб-тестування Tricentis Tosca
- Платформа автоматизації тестування програмного забезпечення Robot Framework
- Автоматизований інструмент візуального тестування Applitools

Розроблено програмне забезпечення для автоматизованого тестування веб-сайтів за допомогою двох типів тестування, – UI & API та API тестування. Окремо розроблено архітектуру автоматизованого фреймворку.

Для тестування UI частини нашого веб-сайту, було використано інструмент Playwright, для API тестування вибрано інструмент SuperTest. Для запуску тестів використовуємо Jest приєднаними відповідними бібліотеками, котрі додаємо до package.json.

Розроблено систему збору аналітичних даних по результатам проведених тестувань. Для цього використані тестові репорти двох типи тестів UI і API і для цього використовуються різні інструменти. Для API це окрема бібліотека яка на основі Jest_runner, котра агрегує дані і генерує репорт. Для UI тестування інструмент Playwright має хороший вбудований репорт. Для зручності додано ще один репорт – Allure, котрий агрерує всі дані і формує один загальний репорт.

Досліджено ефективність інструментів автоматичного тестування з використанням автоматичної генерації тестових даних, що має важливе значення для підтримки модульного тестування. У кваліфікаційній роботі проведено порівняння добре відомих загальнодоступні інструменти генерації даних модульного тестування, – TestGen4j , JCrasher та JUB. Їх застосували до класів Java та оцінили ефективність на основі шкали значень їхніх мутації.

Інструменти автоматизованого тестування дозволяють без особливих

зусиль створювати, запускати та підтримувати тести та підтримувати централізований перегляд аналітики результатів тестування.

Автоматизації підлягають найбільш монотонні завдання що покращує результати тестування та підвищенню загальної якості програмного забезпечення. Розробка методів та засобів автоматизації тестування програмного забезпечення має потенціал для подальшого покращення його ефективності та результативності, зниження ризику людської помилки та забезпечення більш повного охоплення тестуванням.

ПЕРЕЛІК ПОСИЛАНЬ

1. Apache Jena. A free and open source Java framework for building Semantic Web and Linked Data applications.URL: https://jena.apache.org/index.html.
2. Baader F. The Description Logic Handbook: Theory, Implementation, Applications / F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel- Schneider. Cambridge, 2003. 574 P.
3. Bacon Jono. Reviews: Qt. A multi-platform graphical toolkit. 2004. URL: http://preserve.mactech.com/articles/mactech/Vol.20/20.03/QtReview/index.html .
4. Belshe M., Peon R., Thomson M. Hypertext Transfer Protocol version 2. 2015. URL: https://tools.ietf.org/html/draft-ietf-httpbis-http2-17
5. Blomqvist E., Hammar K., Presutti V. Engineering Ontologies with Patterns: The eXtreme Design Methodology, In Ontology Engineering with Ontology Design Patterns. Studies on the Semantic Web, Eds., Hitzler, P., and A. Gangemi, K. Janowicz, A. Krisnadhi, V. Presutti, IOS Press, pp: 23-50, 2016.
6. Crockford D. The application/json Media Type for JavaScript Object Notation (JSON) — Internet Engineering Task Force, 2006. — 10 p.
7. FreeMind – free mind mapping software – URL: http://freemind.sourceforge.net/wiki/index.php/Main_Page .
8. Fuseki: serving RDF data over HTTP. – URL: http://jena.apache.org/documentation/serving_data/
9. Gudgin M., Hadley M., Mendelsohn N., Moreau J.-J., Nielsen H.F., Karmarkar A., Lafon Y. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). URL:
10. jColibri: CBR Framework jColibri 2.1 Tutorial. URL: http://gaia.fdi.ucm.es/research/colibri/
11. Protocol Buffers. Developer Guide — // Google Developers, April 2, 2012. URL: https://developers.google.com/protocol-buffers/docs/overview

12. Tagliaferri Lisa. An Introduction to JSON, 2016 URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-json
13. Paul Ammann and Jeff Offutt. Introduction to Soft- ware Testing. Cambridge University Press, Cam- bridge, UK, 2008. ISBN 0-52188-038-1.
14. Automated QA. Testcomplete. Online, 2008. http://www.automatedqa.com/ products/ testcomplete.
15. Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. Software: Practice and Experience, 34:1025–1050, 2004.
16. Parasoft. Jtest. Online, http://www.parasoft.com/jsp/products/home.jsp-product=Jtest .
17. Riasat, Nazir, Zubair. (2021) “Critical Analysis of Software Testing Techniques and Automation Testing Tools. International Journal of Scientific and Engineering Research”. Volume: 12, Issue: 2. ISSN 2229-5518.
18. Albert, Mark (2016-01-01), “Standard Tool Classification for better Data Communication”, Modern Machine Shop (ISSN 0026-8003).
19. Dobslaw, R. Feldt, D. Michaelson, P. Haar, F. G. de Olivera Neto, R. Rorkar, (2019) “Estimating Return On Investment for GUI Test Automation Frameworks”. 1907.03475v2.
20. Automated Testing. (n.d.). [Blog] Katalon Studio versus Selenium-based open source frameworks. Available at: https://www.katalon.com/resources-center/blog/katalon-studio-vs-selenium-based-open-source-frameworks.
21. Проектування мікропроцесорних систем керування: навчальний посібник/ І.Р. Козбур, П.О. Марущак, В.Р. Медвідь, В.Б. Савків, В.П. Пісьціо.–Тернопіль: Вид-во ТНТУ імені Івана Пулюя, 2022.–324с.
22. Я.І. Проць, В.Б. Савків,О.К. Шкодзінський, О.Л. Ляшук. Автоматизація виробничих процесів. Навчальний посібник для технічних спеціальностей вищих навчальних закладів. – Тернопіль: ТНТУ ім. І.Пулюя, 2011. – 344с.
23. Основи наукових досліджень і теорія експерименту : Навчальний

посібник / укл. Ю. Б. Капаціла, П. О. Марущак, В. Б. Савків, О. П. Шовкун. Тернопіль : ФОП Паляниця В.А., 2023. 186 с.». http://elartu.tntu.edu.ua/handle/lib/40843.

24. Пилипець М. І. Правила заповнення основних форм технологічних документів : навч.-метод. посіб. / Уклад. Пилипець М. І., Ткаченко І. Г., Левкович М. Г., Васильків В. В., Радик Д. Л. Тернопіль : ТДТУ, 2009. 108 с. https://elartu.tntu.edu.ua/handle/lib/42995.
25. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання «Безпека в надзвичайних ситуаціях» / В.С. Стручок –Тернопіль: ФОП Паляниця В. А., –156 с. https://elartu.tntu.edu.ua/handle/lib/39196.
26. Навчальний посібник «Техноекологія та цивільна безпека. Частина «Цивільна безпека»» / автор-укладач В.С. Стручок – Тернопіль: ФОП Паляниця В. А., – 156 с. http://elartu.tntu.edu.ua/handle/lib/39424