Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

**ДОСЛІДЖЕННЯ СЕРВІСІВ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ КОДУ**
(тема)

Виконав:
студент 2 курсу, групи ІНФм-22-1
Бабакін П.С.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник доц. Тітова О.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _______________
(підпис)
«____»_______________ 2024 р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Бабакіну Павлу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження сервісів безперервної інтеграції для автоматизації тестування коду

затверджена наказом по університету від 3 листопада 2023 року № 1280Ст

2. Термін подання студентом роботи до екзаменаційної комісії 25 грудня 2023 р.

3. Вихідні дані до роботи Теоретичні відомості про методи безперервної інтеграції, приклади та моделі сервісів для безперервної інтеграції. Теоретичні та практичні приклади автоматизації тестування коду.

4. Перелік питань, що потрібно опрацювати в роботі

1. Огляд основних теоретичних підходів до безперервної інтеграції та автоматизації тестування коду.
2. Вплив сервісів безперервної інтеграції на автоматизацію тестування коду.
3. Розгляд можливості оптимізації тестів для їхнього швидкого виконання.
4. Вивчення впливу різних видів тестів (одиничні, інтеграційні, функціональні) на час виконання тестування коду.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність автоматизації тестування коду, постановка задачі, вплив сервісів безперервної інтеграції на автоматизацію тестування коду.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

| Найменування розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 03.11.2023 | |
| 2 | Аналіз завдання, підбір літератури | 03.11.23-05.11.23 | |
| 3 | Аналіз літератури з досліджуваної проблеми | 05.11.23-20.11.23 | |
| 4 | Аналіз технічних засобів | 20.11.23-30.11.23 | |
| 5 | Розробка методу | 01.12.23-07.12.23 | |
| 6 | Програмна реалізація | 07.12.23-10.12.23 | |
| 7 | Оформлення пояснювальної записки | 10.12.23-15.12.23 | |
| 8 | Перевірка на плагіат | 16.12.2023 | |
| 9 | Рецензування | 18.12.2023 | |
| 10 | Підготовка презентації та доповіді | 21.12.2023 | |
| 11 | Занесення роботи в електронний архів | 03.12.2024 | |
| 12 | Попередній захист кваліфікаційної роботи | 04.12.2024 | |

Дата видачі завдання 3 листопада 2023 р.

Студент ______________________________
(підпис)

Керівник роботи ______________________ доц. Тітова О.В.
(підпис) (посада, прізвище, ініціали)

**РЕФЕРАТ/ABSTRACT**

Пояснювальна записка до кваліфікаційної роботи: 65 с., 2 табл., 23 рис., 1 дод., 34 джерела.

АВТОМАТИЗАЦІЯ, ФРЕЙМВОРК, ПАТЕРН, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, SELENIUM.

У кваліфікаційній роботі реалізована тема: «Дослідження сервісів безперервної інтеграції для автоматизації тестування коду», метою якої є зменшення часу, необхідного на виконання та аналіз тестів тестувальником.

У розділі загальні положення були описані діяльності моделі з боку тестувальника та з боку користувача та поставлена задача із визначеними цілями та метою. У розділі з інформаційного забезпечення були визначені вхідні та вихідні дані, була показана схема вхідних даних. У розділі з математичного забезпечення було розраховано ефективність впровадження автоматизації у процес розробки вебпорталу. У розділі з програмного забезпечення описані основні засоби розробки комплексу задач, висунуті вимоги до технічного забезпечення, обрано та обґрунтовано архітектуру програмного забезпечення. У технологічному розділі описана інструкція користувача, проведені випробування та показані результати з висновками.

AUTOMATION, FRAMEWORK, PATTERN, CONTINUOUS INTEGRATION, SELENIUM.

The theme of the diploma project is implemented: "Research of continuous integration services for code testing automation", which aims to reduce the time required to execute and analyse tests by a tester. The general provisions section described the model's activities on the part of the tester and the user and set the task with defined goals and objectives. In the section on information support, the input and output data were defined, and the input data scheme was shown. The section on mathematical support calculated the effectiveness of implementing automation in the web portal development process.

The software section describes the main means of developing a set of tasks, sets forth the requirements for technical support, and selects and justifies the software architecture.

The technological section describes the user manual, the tests performed, and the results with conclusions.

# ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів ........... 5
Вступ ........ 6
1 Огляд основних методів тестування для перевірки функціональності, якості та надійності програмного забезпечення ........ 9
1.1 Основні методи тестування для перевірки функціональності, якості та надійності програмного забезпечення ........ 9
1.2 Модульне тестування ........ 10
1.3 Статичні методи ........ 12
1.4 Динамічні методи ........ 12
1.5 Методи шляхів, що реалізовуються ........ 13
1.6 Інтеграційне тестування ........ 13
1.7 Функціональне тестування ........ 17
1.8 Постановка задачі дослідження ........ 19
2 Поняття безперервної інтеграції та автоматизованого тестування і їх значення для розробки програмного забезпечення ........ 21
2.1 Основні принципи безперервної інтеграції ........ 21
2.2 Аналіз існуючих сервісів безперервної інтеграції ........ 26
2.3 Рівні автоматизації. Місце тестування користувальницького інтерфейсу в загальному процесі автоматизованого тестування ........ 30
2.4 Популярні фреймворки для автоматизації тестування коду ........ 31
2.5 Існуючі підходи до автоматизації ........ 41
2.6 Опис процесу діяльності ........ 43
2.7 Огляд наявних аналогів ........ 48
3 Розробка тестів та інтеграція з системою TeamCity ........ 49
3.1 Керівництво користувача ........ 49
3.1.1 Автоматизація тестів ........ 50
3.1.2 Система параметризації ........ 54
3.1.3 Інтергація з системою керування версіями Git ........ 55

3.1.4 Інтергація з системою неперервної інтеграції TeamCity .... 57
3.2 Випробування програмного продукту .............................................. 58
3.3 Висновок до розділу ........................................................................ 59
Висновки ................................................................................................ 60
Перелік джерел посилання .................................................................... 64

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CI – Continuous Integration (спосіб розробки програмного забезпечення, який полягає у виконанні частих автоматизованих складових проекту для якнайшвидшого виявлення та вирішення інтеграційних функціональних проблем)

ROI – Return On Investments (фінансовий коефіцієнт, який ілюструє рівень прибутку або збитку бізнесу, враховуючи суму зроблених інвестицій)

XML – eXtensible Markup Language (стандарт побудови мов розмітки ієрархічно структурованих даних для обміну між різними застосунками)

SQL – Structured Query Language (мова програмування для взаємодії користувача з базами даних, що застосовується для формування запитів)

КГП – керуючий граф програми

# ВСТУП

Важливою темою в сфері розробки програмного забезпечення є автоматизація тестування коду за допомогою сервісів безперервної інтеграції.

Велика кількість програмного продукту випускається та використовується у будь яких сферах нашого життя: будь-то індустрія розваг, наукова сфера, бізнес, медицина, машинобудування чи навіть індустрія харчової промисловості: усюди використовується програмне забезпечення, яке, насамперед, поліпшує ефективність виробництва та зменшує людський вплив на процес виконання певної роботи.

З часом важче й важче стає слідування вимогам щодо функціональної складової систем, адже майже у всіх сферах життя людській роботі віддають перевагу роботизованим системам та системам, де вплив людини є мінімальним. Від роботи таких систем залежить працездатність та ефективність роботи і прибуток великих підприємств, а іноді й життя людей. Ці фактори висувають наперед важливі питання щодо відмово стійкості, надійності та якості програмних забезпечень. З плином часу сам процес розробки програм змінювався: з’явилось багато нових методологій та інструментів, які дозволяють керувати, структурувати та прискорювати процеси розробки, удосконалювати оцінювання часу та ресурсів, призначених для отримання результату.

При розробці продукту необхідно, щоб після випуску на ринок, він працював коректно, без збоїв, а також відповідав вимогам замовника. Також важливо, аби у процесі розробки команда розробників мала частково функціонуючий продукт з надійно працюючим та відповідним до вимог функціоналом, який з часом розширюється та удосконалюється. Це дозволяє, насамперед, вже з ранніх етапів розробки випускати продукт на ринок, що має прибуткову перевагу у зрівнянні з тими системами, які є повноцінно функціонуючими лише у кінці розробки. Це завдання допомагають вирішити

такий процеси тестування.

Автоматизоване тестування дозволяє значно знизити витрати компаній замовників, зекономити ресурси та час, які використовуються для тестування та підтримки високої якості продукту.

При великих проектах зі значною кількістю коду та частими змінами може стати непросто вручну перевірити всі аспекти програми. Це може призводити до помилок та проблем в роботі програмного продукту, що в свою чергу впливає на якість та надійність програмного забезпечення. Тому виникає потреба в автоматизації тестування коду для забезпечення ефективності та якості розробки програмного забезпечення.

В сучасній розробці програмного забезпечення акцент зростає на швидкості розробки та впровадження нового функціоналу, а також на забезпеченні якості продукту. Безперервна інтеграція є одним з основних підходів, який допомагає вирішити ці проблеми. Використання сервісів безперервної інтеграції дозволяє автоматизувати процеси збірки, тестування та розгортання програмного забезпечення.

Актуальність дослідження полягає в тому, що існуючі сервіси безперервної інтеграції пропонують різні можливості та функціонал, і важливо дослідити, які з них найбільш ефективні для автоматизації тестування коду. Це дозволить розробникам програмного забезпечення вибрати найкращі інструменти та методики для своїх проектів і покращити якість та продуктивність своїх розробок.

Проведення дослідження сервісів безперервної інтеграції для автоматизації тестування коду має велику актуальність у сучасній розробці та може привести до покращення процесу розробки програмного забезпечення та підвищення якості продукту. Кваліфікаційна робота присвячена дослідженню сервісів безперервної інтеграції для автоматизації тестування коду з метою визначення їх ефективності та внесення рекомендацій щодо їх використання в процесі розробки програмного забезпечення.

# 1 ОГЛЯД ОСНОВНИХ МЕТОДІВ ТЕСТУВАННЯ ДЛЯ ПЕРЕВІРКИ ФУНКЦІОНАЛЬНОСТІ, ЯКОСТІ ТА НАДІЙНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1 Основні методи тестування для перевірки функціональності, якості та надійності програмного забезпечення

Методи тестування коду використовуються для перевірки функціональності, якості та надійності програмного забезпечення. Існує декілька методів тестування коду, які широко використовуються:

Юніт-тести. Такі тести виконуються на найменшій одиниці коду, такій як функція, метод або клас. Вони перевіряють правильність роботи окремих частин програми і зазвичай створюються розробниками. Юніт-тести допомагають виявляти помилки та допуски в коді на ранніх етапах розробки.

Інтеграційні тести – тести, які перевіряють взаємодію між різними компонентами системи. Вони виконуються для переконання, що окремі частини програми працюють разом належним чином. Інтеграційні тести можуть бути виконані на рівні модулів, сервісів або вебінтерфейсу.

Функціональні тести. Функціональні тести перевіряють, чи працює програмне забезпечення згідно з вимогами та очікуваннями користувачів. Вони перевіряють, чи виконуються очікувані функції та результати для різних введених даних або взаємодії з програмою. Також можуть включати тестування введення та виведення, обробки помилок, роботи з базою даних та інші аспекти функціональності програми.

Відмовостійкі тести – тести, які перевіряють, як програмне забезпечення поводиться у випадку неправильного функціонування або незвичайних умов. Вони виконуються для перевірки стійкості програми до помилок, відмов та непередбачуваних ситуацій. Відмовостійкі тести можуть включати обробку винятків, відновлення системи після відмови, тестування навантаження та інші аспекти надійності програми.

Регресійні тести. Регресійні тести перевіряють, чи не впливають нові зміни або виправлення на раніше працюючі функціональності програми. Вони виконуються для виявлення регресійних помилок, які можуть виникати під час розробки нових функцій або виправлення багів. Регресійні тести можуть бути автоматизовані, щоб забезпечити стабільність функціональності під час змін у програмі.

Ці методи тестування коду можуть бути використані окремо або в поєднанні, залежно від потреб проекту та типу програмного забезпечення. Використання різних методів допомагає забезпечити високу якість та надійність програмного забезпечення.

## 1.2 Модульне тестування

Однією з важливих практик у розробці програмного забезпечення є модульне тестування. Модульне тестування відіграє ключову роль у забезпеченні якості коду, підвищенні його надійності та покращенні швидкості розробки.

Модульне тестування – це тестування програми на рівні окремо узятих модулів, функцій або класів.

Модульне тестування проводиться за принципом «білого ящика», тобто грунтується на знанні внутрішньої структури програми, і часто включає ті або інші методи аналізу покриття коду.

Модульне тестування зазвичай має на увазі створення кожного модуля певного середовища, що включає заглушки для всіх інтерфейсів тестованого модуля. Деякі з них можуть використовуватися для подачі вхідних значень, інші для аналізу результатів, присутність третіх може бути продиктована вимогами, що накладаються компілятором і компоновщиком.

На рівні модульного тестування найпростіше виявити дефекти, пов’язані з алгоритмічними помилками і помилками кодування алгоритмів,

наприклад, роботи з умовами і лічильниками циклів, а також з використанням локальних змінних і ресурсів. Помилки, пов’язані з невірним трактуванням даних, некоректною реалізацією інтерфейсів, сумісністю, продуктивністю і т.д. зазвичай пропускаються на рівні модульного тестування і виявляються на пізніших стадіях тестування.

За способом виконання структурним тестуванням або тестуванням «білого ящика», модульне тестування характеризується тим, що тести виконують або покривають логіку програми (вихідний текст). Тести, пов’язані із структурним тестуванням, будуються за наступними принципами:

– на основі аналізу потоку управління. У цьому випадку елементи, які мають бути покриті під час проходження тестів, визначаються на основі структурних критеріїв тестування С0, С1, С2. До них відносяться вершини, дуги, шляхи керуючого графа програми (КГП), що управляє умовами, комбінацією умов і т.д.;

– на основі аналізу потоку даних, коли елементи, які мають бути покриті, визначаються за допомогою потоку даних, тобто інформаційного графу програми;

Тестування на основі потоку даних. Цей вигляд тестування направлений на виявлення аномалій потоку даних. Запропонована там стратегія вимагала тестування всіх взаємозв’язків, тобто потрібне покриття дуг інформаційного графа програми.

Процес побудови набору тестів при структурному тестуванні прийнято ділити на три фази:

– конструювання КГП;

– вибір тестових шляхів;

– генерація тестів, відповідних тестовим шляхам.

Перша фаза відповідає статичному аналізу програми, завдання якого полягає в графі програми і залежного від нього і від критерію тестування множини елементів, які необхідно покрити тестами.

Друга фаза забезпечує вибір тестових шляхів. Виділяють три підходи до побудови тестових шляхів:

– статичні методи;
– динамічні методи;
– методи шляхів, що реалізовуються.

## 1.3 Статичні методи

Найпростіший метод – побудова кожного шляху за допомогою поступового його подовження за рахунок додавання дуг, поки не буде досягнута вихідна вершина графа програми. Ця ідея може бути посилена в так званих адаптивних методах, які кожного разу додають лише один тестовий шлях (вхідний тест), використовуючи попередні шляхи (тести) як керівництво для вибору подальших доріг відповідно до деякої стратегії. Найчастіше адаптивні стратегії застосовуються по відношенню до критерію С1. Основний недолік статичних методів полягає в тому, що не враховується можливість або не можливість реалізації побудованих тестових шляхів.

## 1.4 Динамічні методи

Такі методи передбачають побудову повної системи тестів, що задовольняють заданому критерію, шляхом одночасного рішення задачі побудови покриваючої безлічі шляхів і формування тестових даних. При цьому можна автоматично враховувати можливість реалізації або не реалізації раніше розглянутих шляхів або їх частин. Основною ідеєю динамічних методів є під’єднування до початкових реалізованих відрізків шляхів їх подальших частин так, щоб не втрачати при цьому знов отриманих доріг та покрити необхідні елементи структури програми.

## 1.5 Методи шляхів, що реалізовуються

Дана методика полягає у виділенні з безлічі шляхів підмножини всіх шляхів, що реалізовуються. Після чого покриваюча безліч шляхів будується з отриманої підмножини шляхів, що реалізовуються.

Перевага статичних методів полягає в порівняно невеликій кількості необхідних ресурсів, як під час використання, так і під час розробки. Проте їх реалізація може містити непередбачуваний відсоток браку (шляхів, що не реалізовуються). Крім того, в цих системах перехід від покриваючої великої кількісті шляхів до повної системи тестів користувач повинен здійснити вручну, а ця робота досить трудомістка.

Динамічні методи вимагають значно більших ресурсів як під час розробки, так і під час експлуатації, збільшення витрат відбувається, в основному, за рахунок розробки і експлуатації апарату визначення шляхів, що реалізуються (символічний інтерпретатор, розв’язувач нерівностей).

На третій фазі за відомими шляхами тестування здійснюється пошук відповідних тестів, що реалізовують проходження цих шляхів.

Перевага цих методів полягає в тому, що їх продукція має деякий якісний рівень, який реалізовується за допомогою шляхів. Методи шляхів, що реалізовуються, дають найкращий результат.

## 1.6 Інтеграційне тестування

Інтеграційне тестування – це тестування частини системи, що складається з двох і більше модулів.

Основне завдання інтеграційного тестування – пошук дефектів, пов’язаних з помилками в реалізації і інтерпретації інтерфейсної взаємодії між модулями.

З технологічної точки зору інтеграційне тестування є кількісним

розвитком модульного, оскільки так само, як і модульне тестування, оперує інтерфейсами модулів і підсистем і вимагає створення тестового оточення, включаючи заглушки (Stub) на місці відсутніх модулів. Основна різниця між модульним і інтеграційним тестуванням полягає в цілях, тобто в типах дефектів, що виявляються, які, у свою чергу, визначають стратегію вибору вхідних даних і методів аналізу. Зокрема, на рівні інтеграційного тестування часто застосовуються методи, пов’язані з покриттям інтерфейсів, наприклад, викликів функцій або методів, або аналіз використання інтерфейсних об’єктів, таких як глобальні ресурси, засоби комунікацій, що надаються операційною системою.

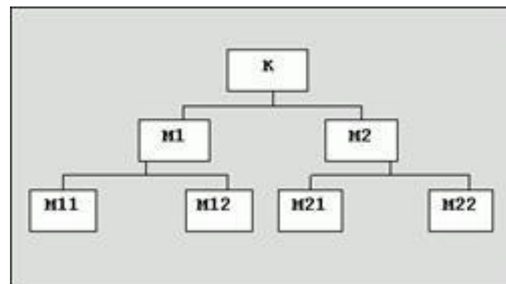


Рисунок 1.1 – Приклад структури комплексу програм

На рис. 1.1 приведена структура комплексу програм *K*, що складається з модулів M1, M2, M11, M12, M21, M22, які були відтестовані на етапі модульного тестування. Завдання, що вирішується методом інтеграційного тестування – тестування міжмодульних зв’язків, що реалізовуються під час виконання програмного забезпечення комплексу *K*. Інтеграційне тестування використовує модель «білого ящика» на модульному рівні. Оскільки тестувальникові текст програми відомий з детальністю до виклику всіх модулів, що входять в тестований комплекс, вживання структурних критеріїв на даному етапі можливо і виправдано.

Інтеграційне тестування застосовується на етапі складання модулів, що модульно протестовані в єдиний комплекс.

Існує два методи складання модулів:

– монолітний, такий, що характеризується одночасним об'єднанням всіх модулів в тестований комплекс;

– інкрементальний, що характеризується кроковим (помодульним) нарощуванням комплексу програм з кроковим тестуванням складеного комплексу. В інкрементальном методі виділяють дві стадії додавання модулів:

– «зверху вниз» і відповідне йому висхідне тестування;

– «від низу до верху» і відповідно низхідне тестування;

Особливості монолітного тестування полягають в наступному: для заміни неопрацьованих до моменту тестування модулів, окрім самого верхнього (рис. 1.1), необхідно додатково розробляти драйвери (test driver) і заглушки (stub), які заміняють відсутні протестовані модулі нижніх рівнів.

Таким чином, порівняння монолітного та інкрементального підходів дає наступне:

– монолітне тестування вимагає великих трудовитрат, пов'язаних з додатковою розробкою драйверів і заглушок і зі складністю ідентифікації помилок, що виявляються в просторі зібраного коду;

– покрокове тестування пов'язане з меншою трудомісткістю ідентифікації помилок за рахунок поступового нарощування об'єму тестованого коду і відповідно локалізації доданої області тестованого коду;

Монолітне тестування надає великі можливості розпаралелювання робіт особливо на початковій фазі тестування. При цьому, особливості низхідного тестування полягають в наступному:

– організація середовища для виконуваної черговості викликів модулями тестованих модулів, що вже протестували;

– постійна розробка і використання заглушок;

– організація пріоритетного тестування модулів, що містять операції обміну з оточенням, або модулів, критичних для тестованого алгоритму.

Наприклад, порядок тестування комплексу *K* (рис. 1.1) при низхідному тестуванні може бути таким, як показано в Прикладі 1, де тестовий набір,

розроблений для модуля $M_i$, позначений як $XY_i = (X, Y)_i$

Приклад 1. Можливий порядок тестів при низхідному тестуванні.

- $K \rightarrow XY_k$
- $M_1 \rightarrow XY_1$
- $M_{11} \rightarrow XY_{11}$
- $M_2 \rightarrow XY_2$
- $M_{22} \rightarrow XY_{22}$
- $M_{21} \rightarrow XY_{21}$
- $M_{12} \rightarrow XY_{12}$

Низхідне тестування має свої недоліки, а саме:

Проблема розробки достатньо «інтелектуальних» заглушок, тобто заглушок, придатних до використання під час моделювання різних режимів роботи комплексу, необхідних для тестування.

Складність організації і розробки середовища для реалізації виконання модулів в потрібній послідовності.

Паралельна розробка модулів верхніх і нижніх рівнів призводиться до не завжди ефективної реалізації модулів із-за спеціалізації ще не тестованих модулів нижніх рівнів до модулів верхніх рівнів, що вже відтестували.

Особливості висхідного тестування, на відміну від низхідного, полягають в організації порядку збірки і переходу до тестування модулів, відповідного порядку їх реалізації. Наприклад, повернувшись до рис. 1.1, порядок тестування комплексу *K* при висхідному тестуванні може бути наступним (Приклад 2).

Приклад 2. Можливий порядок тестів при висхідному тестуванні.

- $M_{11} \rightarrow XY_{11}$
- $M_{12} \rightarrow XY_{12}$
- $M_1 \rightarrow XY_1$
- $M_{21} \rightarrow XY_{21}$
- $M_2(M_{21}, Stub(M_{22})) \rightarrow XY_2$
- $K(M_1, M_2(M_{21}, Stub(M_{22})) \rightarrow XY_k$

– $M_{22} \rightarrow XY_{22}$

– $M_2 \rightarrow XY_2$

– $K \rightarrow XY_k$

Недоліками висхідного тестування є – запізнювання перевірки концептуальних особливостей тестованого комплексу та необхідність в розробці і використанні драйверів.

Таким чином, інтеграційне тестування є важливою складовою процесу розробки програмного забезпечення. Воно допомагає виявляти та усувати проблеми взаємодії між компонентами та забезпечує належну якість продукту. Правильно підібрана стратегія інтеграційного тестування разом з інструментами автоматизації дозволяє забезпечити ефективне та надійне тестування програмного забезпечення.

## 1.7 Функціональне тестування

Одним із найважливіших етапів у процесі розробки програмного забезпечення є функціональне тестування. Ця практика допомагає переконатися, що програма виконує свої функції вірно та відповідає вимогам замовника. Розглянемо, що таке функціональне тестування, які його переваги та як воно впливає на якість програмного забезпечення.

Функціональне тестування – це процес перевірки функціональності програми або системи з метою визначення, чи відповідає вона вимогам та специфікаціям. Це включає перевірку правильності реалізації функцій, операцій та взаємодії між ними. Головною метою функціонального тестування є виявлення помилок та невідповідностей між очікуваним та фактичним результатом.

Як і інші види тестування, функціональне має свою низку переваг, а саме, функціональні тести допомагають:

– забезпечити, що програма виконує свої функції правильно та згідно вимог;

– виявити та усунути помилки на ранніх етапах розробки, забезпечуючи високу якість продукту;

– виявити невідповідності між очікуваними та фактичними результатами взаємодії з програмою;

– переконатися, що програма відповідає специфікаціям та вимогам;

– знизити ризики неправильної роботи програми в реальному середовищі.

Функціональне тестування, в свою чергу, поділяється на види:

– тестування чорного ящика (Black Box Testing) – виконується без знання внутрішньої реалізації програми, тестується зовнішня поведінка.

– тестування білого ящика (White Box Testing) – тестується внутрішня логіка програми, виконується знанням про її структуру;

– тестування інтерфейсів (UI Testing) – перевірка коректності реакції програми на взаємодію користувача;

– тестування взаємодії (Integration Testing) – перевірка взаємодії між різними компонентами програми.

Для того, щоб виконати функціональне тестування, необхідно:

– ознайомитися з вимогами та специфікаціями до програми;

– створити тестові сценарії, які перевіряють різні аспекти функціональності;

– запустіти тести, зафіксувати результати та порівняти їх з очікуваними;

– оцінити результати тестів, виявлені помилки та невідповідності;

– після виправлення помилок повторно виконайте тести, щоб переконатися, що проблеми виправлено.

Отже, функціональне тестування є критично важливою практикою у розробці програмного забезпечення. Воно допомагає перевірити, чи відповідає програма вимогам та специфікаціям, забезпечуючи високу якість

та надійність продукту. Правильно організоване функціональне тестування дозволяє виявити та усунути помилки на ранніх етапах розробки та зекономити час та ресурси.

## 1.8 Постановка задачі дослідження

Метою даної роботи є розробка автоматизованих фреймворків для підвищення ефективності тестування за рахунок зменшення часу необхідного для виконання та аналізу тестів. Для досягнення мети необхідно вирішити такі завдання:

- розробка шаблонів автоматичних тестів;
- аналіз існуючих сервісів безперервної інтеграції;
- створення фреймворку та інтеграція з системою управління версій Git;
- впровадження у систему безперервної інтеграції TeamCity процесу тестування та нотифікації.

Також варто зазначити, що досягти ідеально якісного продукту неможливо, згідно з принципами тестування. Тому дана робота призначена для досягнення зменшення на 80% часу, необхідного на виконання тестів.

# 2 ПОНЯТТЯ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ ТА АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ І ЇХ ЗНАЧЕННЯ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 2.1 Основні принципи безперервної інтеграції

Безперервна інтеграція (Continuous Integration, CI) – це методологія розробки програмного забезпечення, яка передбачає часте та автоматизоване об’єднання зміненого коду різних розробників у спільний репозиторій, а також автоматичну перевірку цього коду за допомогою автоматизованих тестів та засобів перевірки якості. Головна ідея безперервної інтеграції полягає у тому, щоб забезпечити постійну відправку інтегрованого коду в центральний репозиторій, зменшуючи час та ризики, пов’язані з об’єднанням коду в кінці розробки.

Основні принципи безперервної інтеграції включають:

– часті коміти: розробники регулярно вносять свої зміни в репозиторій, інтегруючи їх з загальним кодом;

– автоматизовану збірку: при кожному коміті виконується автоматична збірка проекту для перевірки правильності компіляції та інтеграції змін;

– автоматичні тести: наявність набору автоматизованих тестів, які автоматично запускаються під час збірки для перевірки функціональності та якості коду;

– спільний репозиторій: всі зміни вносяться в один централізований репозиторій, що дозволяє забезпечити єдність кодової бази та швидке виявлення конфліктів;

– застосування засобів автоматичного тестування та перевірки якості: використання автоматичних тестів для перевірки функціональності, продуктивності, безпеки та інших характеристик програмного забезпечення.

Таким чином, безперервна інтеграція має певні переваги. По-перше,

знижує ризики: раннє виявлення проблем у коді та їх виправлення сприяє зменшенню ризиків виникнення помилок та проблем під час розробки. По-друге, збільшує якість систематичного проведення автоматичних тестів та перевірку якості коду, що допомагає покращити якість програмного забезпечення. По-третє, безперервна інтеграція дозволяє розробникам працювати над проектом паралельно та спілкуватися за допомогою засобів контролю версій, що прискорює процес розробки. Також, раннє виявлення проблем та помилок у коді спрощує їх відладку та полегшує процес розробки, чим сприяє постійній комунікації та співпраці між членами розробницького колективу.

Ці переваги роблять безперервну інтеграцію цінним інструментом у розробці програмного забезпечення, дозволяючи підвищити ефективність розробки, якість та швидкість доставки продукту на ринок.

Критично важливою є роль автоматизації тестування коду в процесі безперервної інтеграції (CI).

Тестування являється вагомою частиною розробки програмного забезпечення. Якістю у сфері розробки програмного забезпечення це не тільки зручність користування програмою або її надійність. Якість програмного засобу – це сукупність його властивостей, які є сукупність його придатності та задовольняють усі потреби за його призначенням [3]. Отже, якісним є продукт, який відповідає чітким критеріям якості, що пред’явлені до нього зацікавленими особами, користувачами або замовниками даного продукту. Щоб програмний продукт вважався якісним він повинен відповідати певним очікуванням іі стандартам. За визначенням Гленфорд Майерс, тестування – це процес дослідження програм з метою знаходження в них помилок [4].

У цьому визначенні під помилками, або дефектами програмного забезпечення, розуміють недоліки в розробленого програмного продукту, наслідком яких є їх невідповідність очікуваним результатам виконання програмного продукту і фактично отриманим результатам [5]. Тестування

здійснюється шляхом проведення певних дій у тестованому додатку, результати виконання цих дій у подальшому звіряють з еталонними даними.

Також необхідно вказати, що існує поняття «принципів тестування». Основних принципів тестування 7, вони представлені нижче. Принципи тестування були розроблялися останні 50 років і є загальним збірником правил для тестування у цілому.

– тестування виявляє наявність дефектів;

– тестування може виявити наявність дефектів в програмі, але не може довести їх відсутність. Важливо складати тест-кейси, які будуть знаходити максимальну кількість дефектів. Отже, при правильному тестовому покритті, тестування допоможе знизити ймовірність не виявлення дефектів в програмному забезпеченні. Якщо ж дефекти не були виявлені під час тестування, не можна з упевненістю стверджувати, що їх немає;

– повне тестування неможливе;

– неможливо провести повне тестування, яке б враховувало усі комбінації вводу призначеного для користувача та станів системи, за винятком дуже примітивних випадків. Замість спроб виявлення всіх недоліків слід провести аналіз ризиків та розстановити пріоритети, що допоможе якомога більш ефективно розподілити зусилля по забезпеченню якості програмного забезпечення;

– раннє тестування;

– тестування варто проводити напочатку життєвого циклу розробки програмного забезпечення. зусилля тестування необхідно концентрувати на певних цілях;

– накопичення дефектів;

– різні елементи системи можуть містити в собі різну кількість дефектів – тобто, частота виявлення дефектів в різних модулях програми може різнитися. Зусилля по тестуванню необхідно розподіляти пропорційно щільності дефектів. як правило, найбільша кількість критичних дефектів розподілена в обмеженій кількості елементів системи, що є проявом

принципу парето: 80% дефектів містяться в 20% модулів;

– парадокс пестициду;

– виконуючи одні і ті самі тести знову і знову, ви виявите те, що вони знаходять все меншу кількість нових помилок. Через те, що система еволюціонує, велика кількість раніше виявлених дефектів виправляються та старі тест-кейси більше не працюють;

– щоб запобігти виникненню цього парадоксу, необхідно час від часу вносити поправки в набори тестів, що використовуються, рецензувати і виправляти їх з метою досягнення їх відповідності новому стану системи і допомагали знаходженню якомога більшої кількості дефектів;

– тестування залежить від контексту;

– вибір способу, техніки і виду тестування напряму залежить від самого програмного забезпечення. Наприклад, програмне забезпечення для військових потреб потребує більш суворої та ретельної перевірки, аніж, скажімо, комп'ютерна гра. З тих же міркувань, відповідальний сайт вимагає серйозного тестування на продуктивність, щоб перевірити можливість його роботи при високому навантаженні;

– помилка відсутності помилок;

– факт того, що тестування не виявило помилок, ще не означає, що програма готова до користування. Пошук і виправлення дефектів будуть не важливими, якщо виявиться, що система незручна у використанні, і не задовольняє потреб користувача [6].

Тестування може проводитися як вручну, спеціалістами з тестування, так і автоматично, з використанням спеціальних програмних засобів. Процес верифікації програмного продукту, в ході якого основні етапи та функції тесту, такі як запуск, ініціалізація, виконання, аналіз і видача результату, проводяться автоматично з допомогою інструментів автоматизованого тестування, і називається автоматизованим тестуванням програмного забезпечення [7].

У автоматизованого тестування є як свої переваги, так і недоліки. До

сильних сторін автоматизованого тестування відносять:

– висока швидкість виконання, набагато перевершує можливості людини;

– відсутність впливу «людського фактора» (неуважність, втома);

– можливість багаторазового виконання тестів і зниження витрат ресурсів через це;

– виконання тест-кейсів особливої складності, недоступних людині;

– здатність зберігати, аналізувати в зручній формі великі обсяги даних;

– здатність виконувати низько-рівневі дії з додатком, з операційною системою тощо.

До недоліків автоматизованого тестування відносяться:

– необхідність залучення висококваліфікованого персоналу для автоматизації замість можливості використовувати працю ручних тестувальників;

– витрати на засоби автоматизації, на розробку і підтримку тестів;

– фінансові витрати та ризики, пов’язані з наявністю великої кількості наявних інструментів автоматизації і складністю вибору;

– старіння тестів у випадках змін вимог, переробки інтерфейсів продуктів, що тестуються.

Автоматизація тестування не дозволяє зовсім відмовитися від використання ручного тестування при розробці програмного забезпечення, але дозволяє суттєво знизити його частку, допомагає уникнути рутинних дій у процесі тестування, зменшити вартість і суттєво покращити якість програмного забезпечення, що розроблюється. Тестування є однією з найбільш трудомістких фаз розробки програмного забезпечення, що займає багато часу. При цьому вартість виправлення помилки, не тільки матеріальна, а й фізична, як правило, безпосередньо залежить від часу, який проходить з моменту її виникнення до моменту виявлення. Вважається, що ідея автоматизації процесу тестування дозволить вирішити питання з

нестачею часу для здійснення якісного тестування, ставала все більш актуальною та важливою по мірі збільшення складності розроблюваних програмних продуктів і зростання вимог до їх якості.

Зараз складно уявити собі розробку масштабного програмного проекту без використання автоматизованого тестування в процесі розробки. Етап розвитку тестування характеризується широкою інтеграцією тестування з процесом розробки в цілому, а також використанням автоматизації, великим набором технологій і інструментальних засобів та бібліотек для автоматизації [8].

## 2.2 Аналіз існуючих сервісів безперервної інтеграції

Аналіз існуючих сервісів безперервної інтеграції (CI) є важливим етапом дослідження для визначення доступних рішень та вибору найбільш підходящих сервісів для автоматизації тестування коду. І тут слід звернути увагу на наступні аспекти:

– функціональні можливості. Це можуть бути засоби для автоматичної збірки, запуску автоматичних тестів, інтеграції з системами контролю версій, повідомлення про помилки та інше. Тобто сервіси, які відповідають конкретним потребам;

– інтеграція з іншими інструментами. Для цього слід перевірити, як добре сервіси CI можуть інтегруватися з іншими інструментами, які вже використовуються. Це можуть бути системи контролю версій, системи зберігання артефактів, інструменти для управління завданнями тощо. Важливо, щоб сервіси CI можна було легко і безпроблемно інтегрувати з існуючою інфраструктурою;

– масштабованість. Звісно ж, потрібно враховувати можливості масштабування сервісів CI. Для цього потрібно переконатися, що сервіси можуть впоратися зі зростаючою кількістю розробників, проектів та кодових

баз. Також важливо з'ясувати, чи можуть сервіси працювати з різними типами проектів, мовами програмування та платформами;

– безпека. Не менш особливу увагу треба приділити питанням безпеки, оскільки сервіси CI працюють з кодом та іншими конфіденційними даними. Обов'язково необхідно вивчити, які заходи безпеки вживаються сервісами, чи є можливість налаштування рівня доступу, шифрування даних та аутентифікації;

– спільнота та підтримка. Також необхідно оцінити активність спільноти та рівень підтримки від розробників сервісів CI, тобто перевірити, чи надаються достатній обсяг документації, форуми підтримки, оновлення та поліпшення сервісів;

– вартість та ліцензування. Важливо розглянути вартість використання сервісів CI та їхні ліцензійні умови. Деякі сервіси можуть мати безкоштовні плани для невеликих проектів, але вимагати плату за додаткові функції або використання на більшому масштабі;

Після проведення детального аналізу існуючих сервісів безперервної інтеграції ми можемо вибрати ті, які найкраще відповідають потребам.

Сервіси безперервної інтеграції є одною з ключових практик у сучасній розробці програмного забезпечення. Ця практика відіграє важливу роль у забезпеченні якості продукту, покращенні ефективності розробки та зниженні ризиків.

Сервіси безперервної інтеграції є важливим питанням в контексті розробки програмного забезпечення. Ця практика стала необхідною у сучасному IT-світі для досягнення високої якості, швидкості та надійності у процесі розробки. Головною метою CI є виявлення та усунення помилок на ранніх етапах, що забезпечує стабільність, швидкість та надійність процесу розробки.

Після кожного коміту в репозиторій, CI-сервер автоматично стягує оновлений код. Далі CI-сервер виконує процес збирання, під час якого компілюється програма та створюються необхідні компоненти. Після

збирання, CI-сервер автоматично запускає набір тестів для перевірки коду на відповідність стандартам. Коли тести виконані, CI-сервер аналізує результати, виявляє проблеми та генерує звіт про результати тестування та можливі проблеми, який отримує розробник.

Застосовувати сервіси безперервної інтеграції можна як для невеликих проектів, де CI допомагає швидше виявляти проблеми, так і для великих проектів, де CI допомагає забезпечити стабільність, підтримувати велику кодову базу та зберігати високу якість. Також можна застосовувати CI для розподілених команд, що сприяє спільній розробці, незалежно від географічного розташування.

Далі, в таблиці, наведено різні сервіси безперервної інтеграції, їх особливості, переваги і недоліки, сфери застосування (табл. 2.1).

Таблиця 2.1 – Сервіси безперервної інтеграції

| **Інструмент** | **Опис** | **Основні особливості** | **Переваги** | **Недоліки** |
|---|---|---|---|---|
| Jenkins | Відкрите ПЗ з великим спільнотним забезпеченням | Розширенні можливості налаштовувати складні процеси CI/CD | Гнучкість, велика кількість плагінів | Вимагає багато налаштувань. |
| Travis CI | Хмарний сервіс для автоматичної інтеграції | Легка інтеграція з GitHub, велика підтримка ЯП | Простий, безкоштовний для OS | Обмежені опції для великих проектів |
| CircleCI | Хмарний сервіс для автоматичної інтеграції | Підтримка Docker, можливість паралельного виконання завдань | Хороше інтегрування з Docker, швидкість | Вищі витрати для більших команд |
| GitLab CI/CD | Частина інтегрованої платформи GitLab | Повна інтеграція з GitLab, автоматичне створення інфраструктури, YAML-конфігурація | Зручний для проектів на GitLab, єдність платформи | Обмежена гнучкість для не-GitLab проектів |

Продовження таблиці 2.1

| TeamCity | Інтегрована система збирання та інтеграції | Висока продуктивність, гнучка конфігурація, підтримка різних ЯП та фреймворків | Завширшені можливості налаштування | Вимагає налаштування сервера, вартість ліцензій |
|---|---|---|---|---|

Ця таблиця допомагає нам отримати більш чітке уявлення про переваги та недоліки кожного сервісу безперервної інтеграції та обрати найкращий варіант. Як бачимо, будь-який з цих сервісів може бути вибраний залежно від специфіки проекту, вимог до конфігурації та побажань команди розробників.

## 2.3 Рівні автоматизації. Місце тестування користувальницького інтерфейсу в загальному процесі автоматизованого тестування

На сьогоднішній день існують різні підходи до автоматизації тестування.

Загальна стратегія тестування включає в себе 3 рівні автоматизації:

– рівень функціонального тестування (Functional Test Layer, or Service / API Tests Layer) – тестування через доступ до функціонального прошарку, минаючи призначений для користувача інтерфейс;

– рівень тестування користувальницького інтерфейсу, або через призначений для користувача інтерфейс (GUI Test Layer) – дає можливість тестувати не тільки інтерфейс користувача, а й реалізовувати функціональне тестування, тобто виконувати операції, що використовують бізнес логіку програми.

Для забезпечення найкращої якості додатку рекомендується автоматизувати всі три рівня. Класична піраміда тестування (рис. 2.2) [9].

Рисунок 2.2 – Піраміда тестування

Дана піраміда показує збалансоване співвідношення тестів різного рівня для середньостатистичного проекту. Тести верхнього рівня вважаються більш складними в розробці і потребують більших матеріальних витрат. Співвідношення може змінюватися в залежності від специфіки продукту, що тестується.

## 2.4 Популярні фреймворки для автоматизації тестування коду

Фреймворки для автоматизації тестування коду – це набір інструментів та бібліотек, які дозволяють розробникам створювати, виконувати та керувати автоматичними тестами для перевірки функціональності, надійності та якості програмного забезпечення. Кожен з них має свої особливості та переваги, і вибір залежатиме від наших потреб, мов програмування та платформи, на якій ми працюємо.

Розглянемо деякі популярні фреймворки для автоматизації тестування коду більш детально.

Selenium є однією з найпопулярніших та потужних автоматизованих тестувальних платформ для вебдодатків. Він дозволяє автоматизувати взаємодії з вебсайтами та вебдодатками, що допомагає забезпечити стабільність та якість програмного забезпечення. Selenium розроблений для

автоматизації вебтестування із застосуванням різних мов програмування. Він може ефективно взаємодіяти з браузерами, симулюючи дії користувача та перевіряючи правильність роботи вебдодатків.

Основними компонентами Selenium є:

Selenium WebDriver. Це головна частина фреймворка, яка надає програмний інтерфейс для взаємодії з різними браузерами. WebDriver забезпечує можливість виконувати різні дії, такі як натискання кнопок, введення тексту, перехід за посиланням тощо.

Selenium IDE (Integrated Development Environment). Це розширення для браузерів, що дозволяє записувати та відтворювати дії користувача на вебсторінці. Хоча цей компонент менше потужний порівняно з WebDriver, він є корисним для швидкого створення базових тестів.

Selenium Grid. Цей компонент дозволяє паралельно запускати тестові скрипти на різних браузерах та операційних системах, що поліпшує продуктивність та скорочує час виконання тестів.

Перевагою Selenium є мовна незалежність, що дає змогу підтримувати різні мови програмування, такі як Java, Python, C#, Ruby тощо, що дозволяє розробникам використовувати ту мову, з якою зручніше.

Також перевагами є:

– підтримка багатьох браузерів: Selenium підтримує популярні браузери, такі як Chrome, Firefox, Safari, Edge та інші;

– загальнодоступність: Selenium є open–source проектом, що означає, що його можна безкоштовно використовувати та модифікувати відповідно до наших потреб;

– велика спільнота: Існує велика спільнота користувачів Selenium, яка надає підтримку, документацію та навчальні ресурси.

Таким чином, для роботи з Selenium, ми зазвичай використовуєм обрану нами мову програмування (наприклад, Java) та відповідні бібліотеки. Ми створюємо тестові скрипти, які взаємодіють з WebDriver, моделюючи дії користувача.

Selenium – це потужний фреймворк для автоматизації тестування вебдодатків, який дозволяє створювати стабільні та надійні тести. Його гнучкість, мовна незалежність та підтримка багатьох браузерів роблять його популярним інструментом у світі розробки ПЗ.

Наступний фреймворк – Appium – це відкритий та безкоштовний фреймворк для автоматизованого тестування мобільних додатків на різних платформах, таких як iOS, Android та Windows. Цей фреймворк дозволяє створювати тести, які взаємодіють з мобільними додатками, симулюючи дії користувача та перевіряючи їх функціональність та стабільність.

Основними рисами Appium є:

Крос-платформовість.

Appium підтримує автоматизацію мобільних додатків для різних операційних систем, таких як iOS, Android та Windows. Це дозволяє створювати однакові тести для різних платформ, зменшуючи накладні витрати на розробку та тестування.

Мовна незалежність.

Appium підтримує різні мови програмування, такі як Java, Python, Ruby, JavaScript та інші. Це дозволяє використовувати ту мову, з якою вони зручніше.

Підтримка різних технологій.

Appium дозволяє тестувати як нативні, так і гібридні мобільні додатки, а також вебдодатки, що запускаються у вбудованих браузерах. Ця універсальність робить його зручним інструментом для різноманітних проектів.

Інтеграція з тестовими рамками.

Appium може бути інтегрований з різними популярними тестовими рамками, такими як JUnit, TestNG, PyTest тощо. Це дозволяє легко організовувати тести, групувати їх та аналізувати результати.

Хмарний тестувальний сервіс.

Appium може бути використаний з хмарними тестувальними сервісами,

такими як Sauce Labs, BrowserStack, які надають можливість виконувати тести на різних пристроях та операційних системах безпосередньо з хмари.

Компоненти Appium.

Appium Server – це центральний компонент, який приймає команди для взаємодії з мобільними додатками та пристроями. Він здатен розпочати сесії на різних платформах та керувати взаємодією з додатками.

Appium Client Libraries – ці бібліотеки доступні на різних мовах програмування та надають інтерфейс для написання тестових скриптів. Вони взаємодіють з Appium Server через Appium WebDriver.

Appium WebDriver – це компонент, який здійснює комунікацію між тестовими скриптами та Appium Server. Він приймає команди зі скриптів, взаємодіє з пристроями та додатками, і повертає результати в тестові скрипти.

Таким чином, можемо сказати, що Appium – це теж потужний інструмент для автоматизованого тестування мобільних додатків, який забезпечує крос-платформовість, мовну незалежність та підтримку різних технологій. Він дозволяє створювати ефективні тести для різних платформ, забезпечуючи стабільність та якість мобільних додатків.

Фреймворк JUnit – це популярний фреймворк для тестування програмного забезпечення на платформі Java. Він надає зручний спосіб створювати та виконувати тести, що допомагає забезпечити якість та надійність коду. JUnit став стандартом у світі тестування Java-програм і активно використовується в розробці забезпечення.

Основні рисами JUnit є:

Простота використання.

JUnit має простий та зрозумілий синтаксис, який дозволяє легко створювати тести для різних частин програми.

Аннотації.

JUnit використовує аннотації (спеціальні мітки перед методами) для визначення тестових методів та їх параметрів. Це спрощує структуру

тестових класів та вказує фреймворку, які методи потрібно виконати як тести.

Тестові фікстури.

JUnit дозволяє визначити методи, які виконуються перед і після виконання кожного тестового методу або перед і після виконання всього класу тестів. Це дозволяє налаштовувати початковий та кінцевий стан для тестів.

Асерти.

JUnit надає багато методів-асертів (перевірок), які допомагають перевіряти очікувані результати під час виконання тестів. Це включає методи для порівняння значень, перевірки на наявність null, інші асерти, які спрощують перевірку результатів.

Інтеграція з іншими інструментами.

JUnit добре інтегрується з різними інструментами розробки та засобами збірки, такими як Maven, Gradle, Jenkins та інші. Це допомагає автоматизувати процес виконання тестів у різних середовищах.

Розширюваність.

JUnit можна розширити за допомогою власних розширень та правил (rules), що дозволяє впроваджувати специфічні для проекту функціональності.

Основними аннотаціями JUnit є:

– `@Test` – вказує, що метод є тестовим та повинен бути виконаний.

– `@Before` – вказує на метод, який виконує підготовку перед виконанням кожного тесту.

– `@After` – вказує на метод, який виконує підготовку після виконання кожного тесту.

– `@BeforeClass` – вказує на метод, який виконує підготовку перед виконанням всього класу тестів.

– `@AfterClass` – вказує на метод, який виконує підготовку після виконання всього класу тестів.

З вище зазначеного ми бачимо, що популярний фреймворк JUnit надає зручні засоби для створення, виконання та аналізу тестів. Його простота використання, аннотації, тестові фікстури та асерти роблять його незамінним інструментом у розробці програмного забезпечення.

Далі розглянемо PyTest – потужний та популярний фреймворк для автоматизованого тестування програм на мові програмування Python. Він надає простий та ефективний спосіб створення та виконання тестів, допомагаючи забезпечити якість та надійність свого коду.

PyTest має інтуїтивний та зрозумілий синтаксис. Багато тестів можна створити за короткий час, і він пропонує багато зручних можливостей для тестування, він автоматично знаходить та виконує всі функції, які називаються `test`_ або розташовані у файлах з назвами, що починаються з `test`_. Це зменшує потребу в ручному налаштуванні.

PyTest надає багато вбудованих асертів (перевірок), які допомагають впевнитися, що результати тестів відповідають очікуванням та дозволяє передавати параметри до тестових функцій, що дозволяє проводити одні й ті ж тести з різними вхідними даними.

PyTest має підтримку для фікстур – підготовлених станів або об'єктів, які можуть бути використані у тестових функціях. PyTest надає потужний механізм для створення та використання фікстур.

PyTest дозволяє параметризувати тести, що дозволяє виконувати одну та саму тестову функцію з різними наборами параметрів. Він може розширюватись за допомогою плагінів, що дозволяє додавати додаткову функціональність до фреймворка, а також добре інтегруватися з іншими інструментами, такими як мова програмування Python, засоби збірки, IDE, забезпечення Continuous Integration та інші.

Основними аннотаціями та функціями PyTest є:

– `@pytest.fixture`. Визначає фікстуру – підготовлений стан або об'єкт, який може бути використаний у тестових функціях.

– `@pytest.mark.parametrize`. Параметризує тестову функцію, дозволяючи виконувати її з різними наборами параметрів.

– `@pytest.mark.skip`. Позначає тест як пропущений.

– `@pytest.mark.xfail`. Позначає тест як очікувано провалений.

– `pytest.raises(exception)`.Перевіряє, що виклик функції викликає виняток `exception`.

Таким чином, ми бачимо, що PyTest – це потужний та зручний фреймворк для автоматизованого тестування програм на мові Python. Його простий синтаксис, фікстури, асерти, параметризація тестів та інші можливості роблять його популярним інструментом для розробників, які бажають забезпечити якість свого коду.

Популярним фреймворком для автоматизованого тестування на платформі Java є TestNG (Test Next Generation). Він надає розробникам зручні засоби для створення, організації та виконання тестів, що допомагає забезпечити високу якість програмного забезпечення.

Основними рисами TestNG є:

- аннотаційна підтримка. TestNG використовує аннотації для визначення тестових методів, налаштувань, фікстур та інших елементів. Це робить код більш читабельним та дозволяє швидко створювати та організовувати тести.

– гнучка організація тестів. TestNG надає можливість групувати тести за атрибутами, які ви визначаєте, такими як функціональність, пріоритет, рівень важливості тощо. Це допомагає легко вибирати та виконувати конкретні групи тестів;

– залежності між тестами. TestNG дозволяє визначати залежності між тестами, що означає, що ви можете вказати, які тести мають бути виконані перед іншими тестами. Це допомагає виконувати тести у певній послідовності;

– дата-провайдери. TestNG надає можливість передавати дані у тести через аннотовані методи, які повертають масиви об'єктів. Це корисно для виконання тестів з різними наборами даних;

– фікстури та життєвий цикл. TestNG надає можливість визначати фікстури (методи, які виконуються перед і після тестового методу) та налаштовувати життєвий цикл тесту (створення, ініціалізація, виконання, знищення);

– паралельне тестування. TestNG підтримує паралельне виконання тестів на різних потоках, що допомагає прискорити виконання тестової суіти;

– репортінг. TestNG надає зручний віджет звітів, який включає інформацію про виконані тести, їх статуси, час виконання та інші показники;

– інтеграція з іншими інструментами. TestNG добре інтегрується з іншими інструментами розробки, такими як засоби збірки (Maven, Gradle), засоби Continuous Integration (Jenkins, Travis CI), засоби для генерації звітів тощо;

Основні аннотації та атрибути TestNG:

– `@Test`: вказує, що метод є тестовим;

– `@BeforeMethod`: вказує на метод, який виконує підготовку перед виконанням кожного тесту;

– `@AfterMethod`: вказує на метод, який виконує підготовку після виконання кожного тесту;

– `@BeforeClass`: вказує на метод, який виконує підготовку перед виконанням класу тестів;

– `@AfterClass`: вказує на метод, який виконує підготовку після виконання класу тестів;

Інші аннотації для налаштування життєвого циклу тесту, параметризації та іншого.

TestNG – гнучкий фреймворк для автоматизованого тестування на мові програмування Java. Його аннотаційна підтримка, можливості групування

тестів, залежності між тестами, паралельне виконання та інші функції роблять його популярним інструментом для забезпечення високої якості програмного забезпечення.

І ще один, не менш цікавий фреймворк – це Cucumber – фреймворк для автоматизованого тестування, який дозволяє представляти тести у вигляді живого опису відповідно до специфікацій, що називається «живою документацією». Фреймворк дозволяє зберігати тести у вигляді текстових файлів, які можуть бути зрозумілі та перевірені не тільки розробниками, але і бізнес–аналітиками, менеджерами проекту та іншими учасниками процесу розробки.

Розглянемо основні риси цього фреймворку.

Cucumber дозволяє описувати поведінку програми у вигляді чітко структурованих текстових файлів, які можуть бути використані як специфікації. Це забезпечує зрозумілість та зв’язок між функціональністю та тестами.

Тести в Cucumber виражаються у вигляді «живих» сценаріїв, які описують, як програма повинна вести себе у різних ситуаціях. Це дозволяє бізнес-користувачам та розробникам зблизитися на спільну мову щодо вимог та функціональності.

Cucumber підтримує різні мови програмування, такі як Java, Ruby, Python, JavaScript та інші, що дозволяє розробникам використовувати ту мову, з якою вони зручніше. Він може бути інтегрований з різними інструментами автоматизації, такими як Selenium, Appium, RestAssured, що дозволяє перевіряти функціональність програми через виконання тестів у відповідності до сценаріїв.

Cucumber дозволяє використовувати таблиці та вирази для передачі даних у сценарії та параметризації тестів та надає зручний звіт після виконання тестів, який включає інформацію про пройдені та провалені сценарії, їх статуси, час виконання та інші показники.

Основні компоненти Cucumber:

– Feature Files. Текстові файли, які містять опис сценаріїв та поведінки програми;

– Step Definitions. Методи на мові програмування, які виконуються під час виконання кожного кроку сценарію;

– Hooks. Спеціальні методи, які виконуються перед або після кожного сценарію чи фічі;

– Runners. Класи на мові програмування, які виконують тести з feature-файлів та вказують, які step definitions використовувати;

Синтаксис Gherkin – це мова, яку використовує Cucumber для опису сценаріїв. Вона має простий та зрозумілий синтаксис. Основні ключові слова в Gherkin: Given, When, Then, And, But, Scenario, Feature, Background, Scenario Outline, Examples.

Cucumber – це потужний фреймворк, який дозволяє описувати та виконувати тести у вигляді «живої документації». Його бізнес-орієнтований підхід, мовна незалежність, можливість автоматизованої верифікації та інші функції роблять його популярним інструментом для забезпечення якості програмного забезпечення.

Таким чином ми бачимо, що популярні фреймворки для автоматизованого тестування, такі як Selenium, Appium, JUnit, PyTest, TestNG та Cucumber, представляють значний вибір для розробників, які прагнуть забезпечити високу якість свого програмного забезпечення. Кожен з цих фреймворків має свої унікальні особливості та переваги.

## 2.5 Існуючі підходи до автоматизації

На поточний момент на автоматизацію тестування вебінтерфейсу користувача існує 4 основні підходи:

– capture / Playback – запис і відтворення. Даний підхід передбачає використання утиліт та програмних продуктів запису і

відтворення, записуючих певну послідовність дій і потім її відтворюють вже без участі людини. Такі тести створюються легко і швидко, але за будь-яких змін функціональності чи інтерфейсу вимагають повного перезапису;

– scripting – написання сценарію. Методологія полягає у написанні тестових сценаріїв на мовах програмування, розроблених спеціально для автоматизації тестування. Даний підхід вирішує проблеми з масштабуванням і підтримкою тестів, відкриває великі можливості. Але при цьому є більш дорогим, що вимагає більшої кількості ресурсів, оскільки розробкою займаються спеціалісти більш високого рівня;

– data-driven testing – управління даними тестування. Методологія створення скриптів і верифікації їх на основі даних, що зберігаються у будь-якому сховищі або базі даних. Використовується у випадках, якщо необхідно реалізовувати однотипні види перевірок для множинних варіантів вхідних даних;

– keyword-based – тестування за ключовими словами. Тест являє собою не програмний код, а послідовність дій з їх параметрами, описану за допомогою ключових слів. Це дозволяє створювати тести людям, які не мають навичок програмування.

Для довгострокових, складних проектів найбільш ефективними є друга і третя техніка (третя – при необхідності тестування на великому обсязі даних), вони вимагають більше ресурсів для реалізації, але при цьому окупаються в майбутньому, дозволяючи багаторазово використовувати як написаний код, так і самі створені тести (для регресійного тестування, включати їх в процес безперервної інтеграції (CI), використовувати для навантажувального тестування, тощо). При написанні тестових скриптів зазвичай застосовують технологію фреймворків.

Як ми вже розглядали вище, фреймворк – це організація проекту, що дозволяє спростити розробку, модифікацію і підтримку програмного коду. У разі організації проекту для автоматизації тестування вебінтерфейсу доводиться зважати на те, що тести потрібно підтримувати, а можливо і

істотно переписувати в разі змін в інтерфейсі тестованої програми. Основне завдання, яке вирішується через використання фреймворків – зниження кількості змін, які потрібно внести в тести при змінах в тестованому додатку.

На даний момент в області автоматизації тестування існує один найбільш відомий шаблон проектування – Page Object. Подібна організація проекту дозволяє значно спростити підтримку тестів і знизити дублювання коду. Основні ідеї цього шаблону проектування:

– чіткий розподіл між безпосередньо кодом тестів і кодом, що спеціалізуються на роботі з локаторами (шляхами до елементів інтерфейсу) та веб елементами;

– наявність єдиного сховища для всіх служб і операцій, представлених на сторінці, а не безлічі розкиданих по всьому тестовому набору [10].

Патерн Page Object був розроблений для тестування вебдодатків, але його ідеї використовуються сьогодні у всіх інших видах автоматизованого тестування, існують різні різновиди реалізації даного шаблону. Основною перевагою його використання є те, що при змінах в інтерфейсі досить модифікувати код тестів тільки в одному місці (у розташуванні локаторів).

Створення тест-кейсів може призвести до непідтримуваного проекту. Однією з причин є те, що використовується дуже багато дублювання коду. Дубльований код може бути викликаний дублюванням функціональності, і це призведе до дублювання використання локаторів. Недоліком дубльованого коду є те, що проект є менш ремонтопридатним та підтримуваним. Якщо якийсь локатор зміниться, потрібно пройти весь код, щоб налаштувати локатори, де це необхідно. Використовуючи патерн Page Object разом з Page Factory, ми можемо зробити тестовий код, що є гнучким, і зменшити або усунути дублікати тестового коду. Реалізація об'єктної моделі сторінки може бути досягнута шляхом розділення абстракції тестового об'єкта і тестових скриптів.

Клас Page Factory в Selenium – це розширення патерну дизайну Page

Object Page. Він використовується для ініціалізації елементів об’єкта сторінки або для створення самого об’єкта сторінки. Анотації до елементів також можуть бути створені (і рекомендовані), оскільки описуючі властивості можуть не завжди бути достатньо змістовними, щоб викликати один об’єкт з іншого.

## 2.6 Опис процесу діяльності

Фреймворк для автоматизованого тестування розроблюється для тестування бізнес логіки додатку та виконання певних тестів спеціалістом. Діяльність тестувальника у межах фреймворку направлене на створення тестових наборів даних, які подаються до системи у вигляді параметрів, описаних у текстовому документі та розширення тестових наборів у тих випадках, коли необхідно перевірити нову функціональність або додати нову валідацію, якщо попередня логіка була змінена.

Класично тестувальнику необхідно виконати також ряд дій для виконання тестів: локально запустити тести та витрачати певний час на виконання нібито автоматизованого процесу і вже після цього при отриманні результатів або йти далі до наступних тестів, або лагодити тести, які не пройшли та аналізувати причину. Схематично цей процес можна відобразити на діаграмі послідовності (рис. 2.3). Звісно, такий процес не дуже спрощує сам процес тестування, а спеціаліст витрачає майже стільки ж часу, скільки б було витрачено на ручне тестування. Даний проект дозволяє вирішити цю проблему за допомогою введення у процес тестування безперервної інтеграції та паралелізації виконання тестів. Це спрощує саме тестування та зменшує кількість витраченого часу, а також вартість виконання роботи. Користувач такої системи буде аналізувати пройдені тести за результатами, які подаються як вихідні дані процесу у вигляді звіту з параметрами пройдених тестів та тестів, які не пройшли.

Загалом, весь процес діяльності можна відобразити на діаграмі послідовності (рис. 2.4).

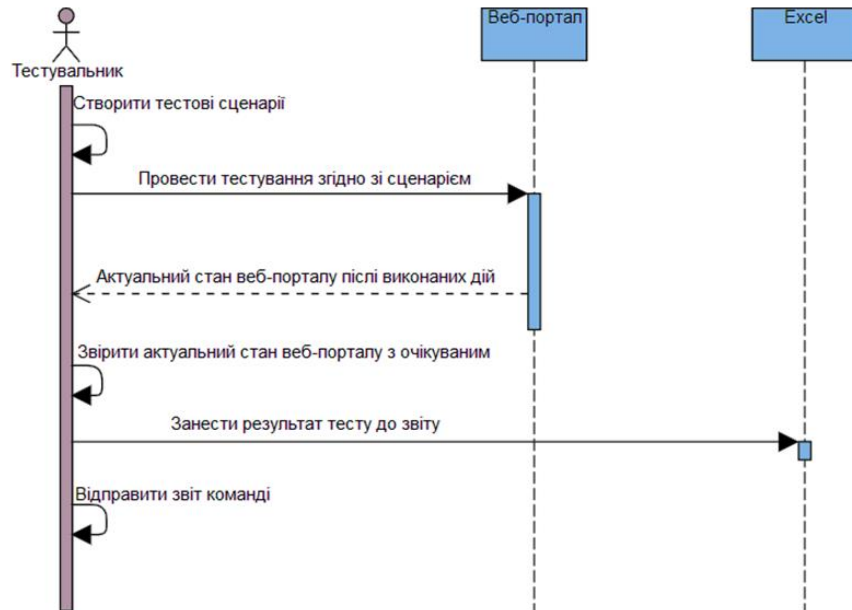


Рисунок 2.3 – Класична діаграма послідовності тестувальника.

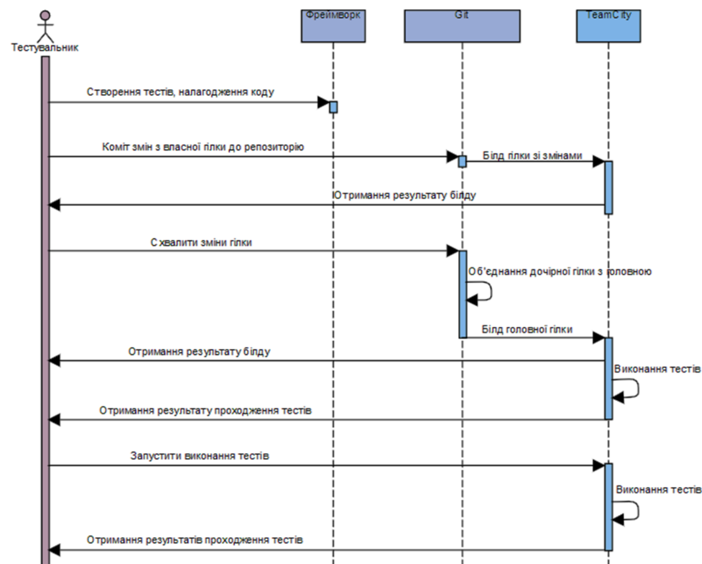


Рисунок 2.4 – Проектна діаграма послідовності.

Розглянемо приклад діяльності вебпорталу «IT knowledge. Освітній IT кластер», який знаходиться на стадії розробки.

Рисунок 2.5 – Форма для ведення профайлу

Перш за все, вебресурс призначений для об’єднання студентів, представників компаній та викладачів, задіяних в ІТ сфері для спілкування, навчання, надання та пошуку вакансій у різних технічних компаніях. Загально, кожен користувач може вести власний профайл (рис. 2.5), та створювати і переглядати пости (рис. 2.6). Таким чином, маємо можливість спілкуватись з іншими користувачами, проходити кваліфікаційні тести, отримувати нові знання не лише про технології, а й про різні ведучі компанії та відгуки щодо вищих навчальних закладів. Важливою функціональністю вебпорталу є створення резюме студентами для проходження практики та пошуку роботи у компаніях, які також зацікавлені у пошуку нових кадрів. Для представників компаній реалізована можливість створювати списки спеціальностей та відкритих вакансій, вони можуть відповідати на запити на підготовчу практику або на роботу, а також переглядати відкриті резюме студентів та запрошувати їх до компанії.

У викладача університету є також можливість проходити практику та підвищувати кваліфікацію у різних ІТ компаніях, також створювати освітні тести по спеціальностям та створювати переліки тем та допоміжних матеріалів для курсових та дипломних проектів. Звісно, кожен з користувачів може, перш за все, переглядати різні матеріали на різні теми та створювати власні статті на наукову тему.

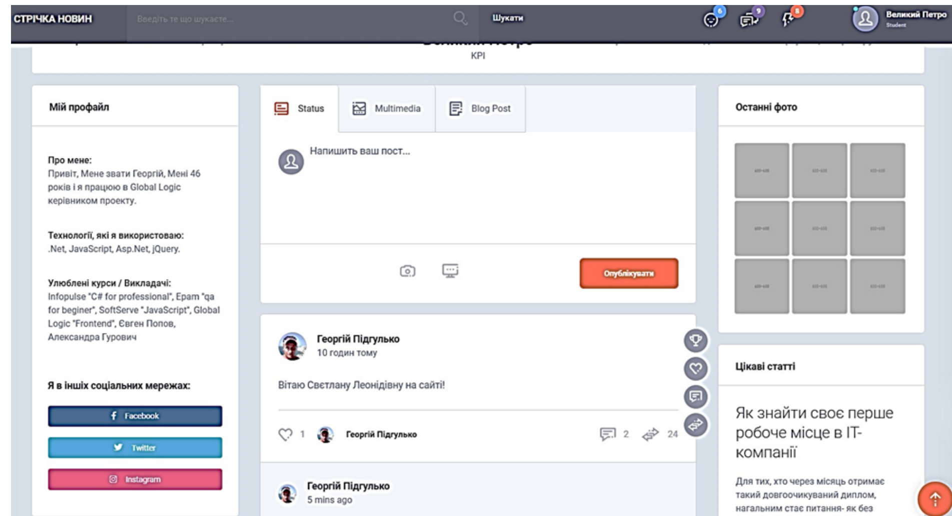


Рисунок 2.6 – Форма перегляду та створення постів

Як підсумок, вебзастосунок, який піддається тестуванню, має такий функціональні пункти:

– реєстрація та вхід до порталу;
– ведення власного профайлу;
– створення та перегляд постів та коментарів;
– створення та перегляд наукових статей та інших матеріалів;
– проходження освітніх курсів;
– створення резюме, вакансій;
– ведення переліку тем для курсових та дипломних проектів.

Це графічно описується згідно з діаграмою прецедентів (рис. 2.7).

Рисунок 2.7 – Діаграма прецедентів

Даний перелік основних функцій є важливим для створення бази для функціонального тестування, який буде реалізовано у проектній роботі.

## 2.7 Огляд наявних аналогів

Найбільш відомими комерційними продуктами для автоматизації тестування користувальницького інтерфейсу веб додатків є на сьогодні такі інструменти, як Serenity, Selenide, Cypress, Robot Framework, RedwoodHQ, Sahi, Gauge, Galen Framework, Citrus Framework.

В якості оціночних критеріїв при виборі інструментів рекомендується використовувати наступні [9]:

– підтримують кросплатформність (розпізнавання необхідних

елементів інтерфейсу, технології взаємодії з ними, як реалізований процес взаємодії – задані параметри необхідні бути виконані на різних платформах, які є вимогою від замовника);

– підтримують типи додатків (нативні, веб, гібридні);

– підтримка запуску тестів на емуляторах, реальних пристроях;

– IDE (підтримують мови, зручність користування, інструменти налагодження);

– менеджери тестів (можливості по конфігурації тестових наборів, можливість паралельного запуску тестів, настройка параметрів запуску);

– логування, створення звітів (можливість надання звітів, налаштувань, інформативність).

При виборі інструмента фахівці радять брати до уваги специфіку додатки, поставлені завдання, кваліфікацію розробників та інженерів з тестування, технічні та фінансові ресурси, плани на майбутнє, ризики [11].

# 3 РОЗРОБКА ТЕСТІВ ТА ІНТЕГРАЦІЯ З СИСТЕМОЮ TEAMCITY

## 3.1 Керівництво користувача

У процесі розробки основна увага приділялась чотирьом основним функціям проекту, а саме:

– автоматизація тестів. Сенс даного пункту є створення гнучкого фреймворку, за яким спеціалісту буде зручно орієнтуватись та розширювати функціонал та створювати нові тести та тестові набори;

– система параметризації. Особливо важливо було створити функціональний набір, мета якого оперувати вхідними даними у вигляді файлу з xml-розміткою та конвертувати його у модель;

– інтеграція з системою керування версіями Git. Це дозволяє створювати віддалений доступ до коду та надає можливість роботи над кодом декільком спеціалістам одразу.

– інтеграція з системою неперервної інтеграції TeamCity, що дозволяє виконувати тести віддалено та без участі людини.

### 3.1.1 Автоматизація тестів

Тестувальник має можливість у межах фреймворку створювати тестові сценарії та тестові дані. Для того, щоб створити новий тестовий набір або фічу, необхідно створити новий файл класу у проекті Test (рис. 3.8).

```
[TestFixture]
[Category("Smoke")]
0 references | Юлия Устенко, 4 days ago | 4 authors, 11 changes | 4 work items
public class Smoke : BaseTestFixture
{
    private readonly CommonSteps _loginSteps = new CommonSteps();
    private readonly NewsAndProfileSteps _newsAndProfileSteps = new NewsAndProfileSteps();

    [Test]
    0 references | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void ITPortal_Login()
    {
        var expectedHeader = "Стрічка новин";

        _loginSteps.LoginToPortalAsDefaultUser();
        var actualHeader = ITPortalContext.IsNewsPageOpen();

        Assert.That(actualHeader, Is.EqualTo(expectedHeader.ToUpperInvariant()), "The News Page isn't opened!");
    }

    [Test]
    0 references | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void ITPortal_Logout()
    {
        _loginSteps.LoginToPortalAsDefaultUser();
        _newsAndProfileSteps.LogoutFromPortal();

        var actualIsOpened = ITPortalContext.IsLoginPageOpened();

        Assert.That(actualIsOpened, "The Login Page isn't opened!");
    }
}
```

Рисунок 3.8 – Тестовий набір

Для того, щоб оголосити клас як тестовий набір необхідно додати атрибут [TextFixture] (рис. 3.9):

```
[TestFixture]
0 references | 0 changes | 0 authors, 0 changes
public class NewTests
{

}
```

Рисунок 3.9 – Створення тестового набору

Для того, щоб оголосити метод у тестовому класі як тест, необхідно додати атрибут [Test] (рис. 3.10):

```
[TestFixture]
0 references | 0 changes | 0 authors, 0 changes
public class NewTests
{
    [Test]
    0 references | 0 changes | 0 authors, 0 changes
    public void Test_Something()
    {

    }
}
```

Рисунок 3.10 – Додавання тесту до тестового набору

Для створення нового вебелементу або його зміни, необхідно звернутися до проекту Context тек Page, Form або Section. Наприклад, зміст логін сторінки у фреймфорку (рис. 3.11).

```
1 reference | Юлия Устенко, 4 days ago | 2 authors, 6 changes | 2 work items
public class LoginPage : WebPage
{
    [FindBy(XPath = ".//div[@class='registration-login-form']")]
    public RegistrationLoginForm RegistrationLoginForm;

    [FindBy(XPath = "//a[contains(text(),'Про проект')]")]
    public Button AboutProjectLink;

    [FindBy(XPath = ".//div[@class='header--standard-wrap']")]
    public LoginHeaderSection HeaderSection;

    [FindBy(XPath = ".//input[@type='submit']")]
    [Name("Submit")]
    public Button SubmitButton;

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public Button GetControlPanelButton(string buttonName)...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public string GetErrorMessageForRegistration(string type)...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public string GetErrorMessageForLogin(string type)...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void ClickOnExpandCollapseButton()...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public bool IsLoginMenuDisplayed()...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void ClickOnGoToLoginPage()...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void SetTextToRegisterField(string fieldName, string text)...
}
```

Рисунок 3.11 – Зміст логін сторінки

Тут присутні елементи, які існують лише в рамках існування логін сторінки у браузері, наприклад, елемент форми для реєстрації та логіну (рис. 3.12).

```
1 reference | Юлия Устенко, 2 days ago | 1 author, 2 changes | 2 work items
public class RegistrationLoginForm : Form
{
    [FindBy(XPath = ".//form[@action='/Account/Login']")]
    public LoginForm LoginForm;

    [FindBy(XPath = ".//form[@action='/Account/Register']")]
    public RegisterForm RegisterForm;

    [FindBy(XPath = ".//a[contains(text(),'Реєстрація')]")]
    [Name("Register")]
    public Button RegisterTab;

    [FindBy(XPath = ".//a[contains(text(),'Вхід')]")]
    [Name("Login")]
    public Button LoginTab;

    2 references | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void SelectForm(string form)
    {
        var button = GetElementClass.GetButton(form);
        if (!button.GetAttribute(name: "class").Contains(value: "active"))
            button.Click();
    }
}
```

Рисунок 3.12 – Зміст форми для логіну чи реєстрації

Ця форма містить у собі два важливих елементи: вкладки логіну та реєстрації. Для керування цими елементами було створено метод, який натискає на вкладку, якщо необхідно відкрити якусь необхідну.

```
1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
public class LoginForm : Form<User>
{
    [FindBy(XPath = ".//input[@type='email']")]
    [Name("Email")]
    public TextField EmailField;

    [FindBy(XPath = ".//input[@type='password']")]
    [Name("Password")]
    public TextField PasswordField;

    [FindBy(XPath = ".//input[@type='submit']")]
    [Name("Submit")]
    public Button SubmitButton;

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public List<Label> EmailErrorLabels...

    2 references | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public string IsErrorPresent()...

    1 reference | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
    public void SetUserData(User user)...
}
```

Рисунок 3.13 – Зміст логін форми

Як видно з рис. 3.11 – 13, у кожному вебкласі створено методи для взаємодії з елементами, це методи Context рівня, які називаються контекстними методами.

Для створення нового вебелементу, необхідно створити нове поле з типом веб елементу та додати до нього атрибути для пошуку цього елементу за локатором. Додатково, можна присвоїти цьому об’єкту назву.

На рівні Steps можна створювати методи, які взаємодіють з вебелементами шляхом використання контекстних методів. Взаємодію цього проекту та контексту можна побачити у використанні простору імен Context, (рис. 3.14).

```
namespace Steps
{
    using Context;

    4 references | Юлия Устенко, 4 days ago | 1 author, 2 changes | 2 work items
    public class CommonSteps
    {
        2 references | Юлия Устенко, 4 days ago | 1 author, 1 change | 1 work item
        public void LoginToPortalAs(User user)
        {
            var loginPage = WebPortal.LoginPage.RegistrationLoginForm;
            loginPage.SelectForm("Login");
            loginPage.LoginForm.SetUserData(user);
            SessionService.Current.CurrentUser = user;
        }
```

Рисунок 3.14 – Взаємодія контекстного прошарку та Steps

### 3.1.2 Система параметризації

Для створення та керування існуючими вхідними параметрами тестувальнику необхідно, перш за все, створити файл з розширенням *.xml* та наповнити його даними у вигляді xml-розмітки та логічним розділенням даних згідно з моделлю об'єкту. Приклад створеного файлу (рис. 3.15).

```xml
<?xml version="1.0" encoding="utf-8"?>
<ListExtendedUsersWithMessage xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <User>
    <Role>Student</Role>
    <FirstName>First</FirstName>
    <LastName>Last Name</LastName>
    <Password>password1</Password>
    <Email />
    <Institution>institu</Institution>
    <Object>
      <Item1>Email</Item1>
      <Item2>Це поле є обов'язкове.</Item2>
    </Object>
  </User>
  <User>
    <Role>Student</Role>
    <FirstName />
    <LastName>Last Name</LastName>
    <Password>password1</Password>
    <Email>1test@test.com</Email>
    <Institution>institu</Institution>
    <Object>
      <Item1>FirstName</Item1>
      <Item2>Це поле є обов'язкове.</Item2>
    </Object>
  </User>
  <User>...</User>
  <User>...</User>
  <User>...</User>
</ListExtendedUsersWithMessage>
```

Рисунок 3.15 – Приклад файлу з вхідними даними

Для десеріалізції файлу з вхідними параметрами наобхідно у класі TestExtendedData створити новий об'єкт (будь який об'єкт, реалізуючий IEnumerable), у якому необхідно викликати метод FromXML() та в параметрі вказати назву файлу (рис. 3.16).

```
public class TestExtendedData
{
    public static IEnumerable<ExtendedUser> TestDataExtendedUserLogin =
        new ListExtendedUsersWithMessage()
            .FromXML("TestListExtendedUsersWithMessageForLogin.xml").Users;
```

Рисунок 3.16 – Приклад створення об’єкту з вхідними даними

Теперь тестувальник має можливість використати створений об’єкт у тестовому методі, написавши атрибут [TestCaseSource] (рис. 3.17).

```
[Test]
[TestCaseSource(typeof(TestExtendedData), "TestCasesLogin")]
0 references | Павло Бабакін , 3 days ago | 1 author, 3 changes | 3 work items
public void LoginForm_EnterWrongCredentials(ExtendedUser user)
{
    _loginSteps.LoginToPortalAs(user);
    var actualMessage = _loginSteps.GetErrorMessageForFielsInForm(field: user.Object.Item1, form: "Login");
    var expectedMessage = user.Object.Item2;

    Assert.That(actualMessage, Is.EqualTo(expectedMessage), "Actual message does not equal to expected!");
}
```

Рисунок 3.17 – Використання об’єкту з вхідними параметрами у тесті

### 3.1.3 Інтеграція з системою керування версіями Git

Тестувальник має можливість створювати відгалудженні гілки та локально працювати з кодом, але зберігати усі дані віддалено. Для того, щоб створити нову гілку, необхідно у сервісі Bitbucked у ролі Admin перейти до вкладки «Branches» та натиснути кнопку «Create branch (рис. 3.18)

Branches

Create branch

Search branches | Active branches | Branch type

| Branch | Behind | Ahead | Updated | Pull request | Builds |
|---|---|---|---|---|---|
| master MAIN DEVELOPMENT | | | 3 days ago | | ✓ |
| 0206 | | | 3 days ago | #36 OPEN | ✓ |
| XMLSourceExample | | | 3 days ago | Create | ✓ |
| 0106 | | | 3 days ago | Create | ✓ |
| Creating_Tests | | | 4 days ago | Create | ✓ |

Рисунок 3.18 – Створення нової гілки

Для того, щоб відправити нові зміни до віддалної гілки або зібрати локально зміни, треба у Visual Studio у вкладці Team Explorer обрати пункт Changes, закомітити усі зміни, додавши назву та натиснути кнопку «Commit all and push», (рис. 3.19).

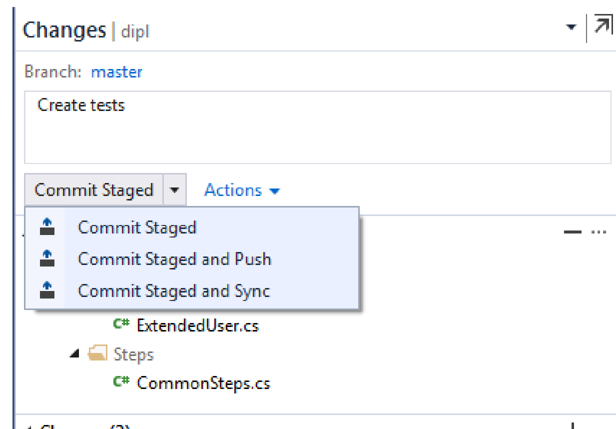


Рисунок 3.19 – Відправка змін до віддаленого репозиторію

Для того, щоб маги змогу з’єднати нові зміни з головною гілкою, треба створити запит на об’єднання та отримати схвалення одного з тестувальників. Запит створюється у сервісі Bitbucked. Необхідно обрати вкладку Pull request та натиснути кнопку Create pull request. Далі необхідно написати назву запиту та приєднати гілку, яку необхідно об’єднати (рис. 3.20).

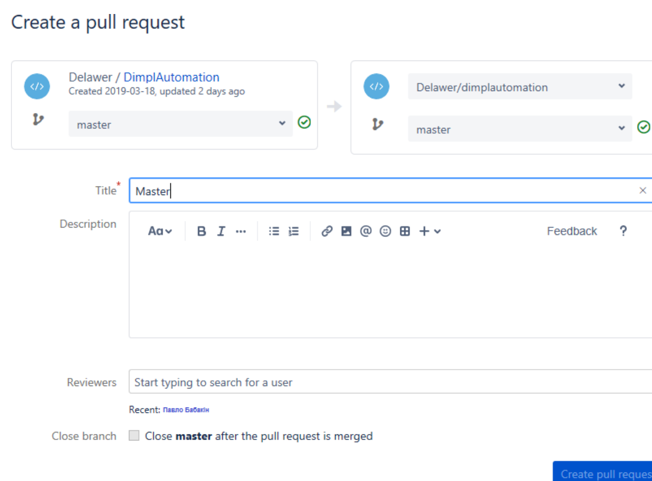


Рисунок 3.20 – Створення запиту

Для схвалення об'єднання необхідно у ролі Admin натиснути у запиті кнопку Approve, вже після цього можна з'єднати гілки, натиснувши кнопку Approve, знаходження цих кнопок (рис. 3.21).

Рисунок 3.21 – Розміщення основних кнопок запиту

3.1.4 Інтеграція з системою неперервної інтеграції TeamCity

Зараз тестувальнику не треба докладати зусиль для того, щоб тести виконувались. У сервісі TeamCity цей процес проходить автоматично, тестувальник отримує кожного разу повідомлення про результат проходження на поштову скриньку.

Для тих випадків, коли тестувальнику необхідно додатково запустити виконання тестів, йому необхідно написнути на кнопку Run у джобі (обсяг ранів чи білдів) Test Run (рис. 3.22).

Рисунок 3.22 – Запуск виконання тестів вручну

Після цього запуск стає у чергу виконань, та виконується автоматично.

Кожного разу, коли тестувальник створює новий запит на об’єднання, у сервісі TeamCity автоматично проходить білд проекту та надсилається повідомлення про результат на поштову скриньку команди. Загальний вигляд таких білдів (рис. 3.23).

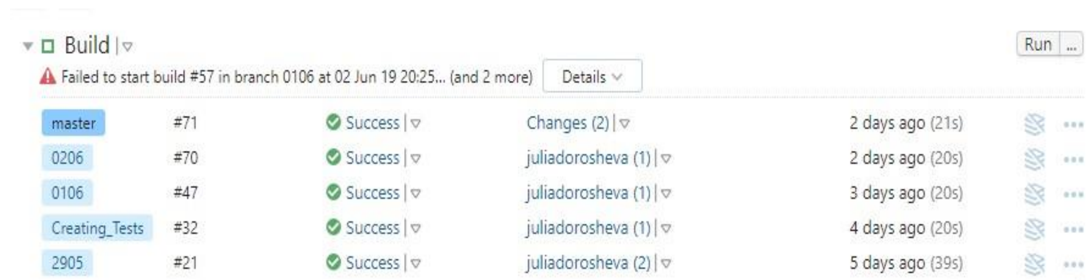


Рисунок 3.23 – Джоба для білду гілок

## 3.2 Випробування програмного продукту

Для випробування проекту буде перевірено такі функції:

- пошук веб елементу на сторінці;
- проходження тесту;
- перевірка виклику методів та дотримання рівнів автоматизації;
- відправлення локальних змін до віддаленого репозиторію;
- створення запиту на об’єднання;
- перевірка відображення результату білду на Bitbucket;
- запуск тестів;
- нотифікація тестувальника.

### 3.2.1 Мета та загальні положення випробувань

Метою випробувань є перевірка відповідності функцій комплексу задач системи автоматизації тестування вебпорталу вимогам технічного завдання.

Випробування проводяться на основі наступних документів:

– ГОСТ 34.603-92. Інформаційна технологія.

Види випробувань автоматизованих систем:

– ГОСТ РД 50-34.698-90. Автоматизовані системи вимог до змісту документів.

### 3.2.2 Результати випробувань

Таблиця 3.2 – Результати випробувань

| **Ситуація** | **Очікуваний результат** |
|---|---|
| Пошук веб елементу під час виконання тесту | Веб елемент знайдений та з ним можна виконувати дії |
| Виклик методів згідно рівням | Методи викликаются та виконуються послідовно |
| Відправка локальних змін до віддаленого репозиторію через Visual Studio | Дані відравляються та відображаються у Bitbucket Git |
| Створення запиту на об'єднання коду та перевірка білду гілки | Гілка збілдилась та відображаються у запиті результати |
| Налагоджене автоматичне виконання тестів у TeamCity | Тести виконуються на агенті |
| Білд гілки та отримання нотифікації | Тестувальник отримав лист на поштову скриньку з результатами білду |
| Проходження тестів та отримання нотифікації | Тестувальник отримав лист на поштову скриньку зі звітом про виконання тестів |

## 3.3 Висновок до розділу

У даному розділі було створено детальну інструкцію користувача для роботи з фреймворком та додатковими системами для автоматизації та управління. Були наведені скріншоти послідовності дій тестувальника для різних варіантів використань. Був виконаний набір тестів, таких як:

– пошук веб елементу на сторінці;
– проходження тесту;
– перевірка виклику методів та дотримання рівнів автоматизації;
– відправлення локальних змін до віддаленого репозиторію;
– створення запиту на об’єднання;
– перевірка відображення результату білду на Bitbucket;
– запуск тестів;
– нотифікація тестувальника.

# ВИСНОВКИ

У даній кваліфікаційній роботі було проведено дослідження сервісів безперервної інтеграції для автоматизації тестування коду. Було розглянуто поняття безперервної інтеграції, її значення та роль автоматизації тестування коду у процесі безперервної інтеграції.

В ході дослідження були проведені аналіз існуючих сервісів безперервної інтеграції, включаючи Jenkins, Travis CI, CircleCI, GitLab CI/CD та TeamCity. Були розглянуті їх особливості, можливості та варіанти інтеграції з різними інструментами тестування.

Також були розглянуті популярні фреймворки для автоматизації тестування коду, включаючи Selenium, Appium, JUnit та PyTest. Були представлені їх основні функціональні можливості та способи використання.

Для покращення ефективності тестування було описано методи тестування коду, включаючи модульне тестування, функціональне тестування та інтеграційне тестування. Були наведені приклади та описано особливості кожного методу.

Для автоматизації тестування були розглянуті різні інструменти, такі як Selenium WebDriver, Appium, Robot Framework та Postman. Були надані приклади їх використання та описані переваги та обмеження.

Окремо були розглянуті різні сервіси безперервної інтеграції, включаючи Jenkins, Travis CI, CircleCI, GitLab CI/CD та TeamCity. Були надані детальні описи їх особливостей, можливостей та варіантів використання.

Також, було розглянуто та запропоновано спосіб зменшення часу, необхідного для виконання та аналізу тестів та виконано усі поставлені завдання, а саме:

– проведено аналіз існуючих аналогів та знайдено переваги та недоліки кожного з них;

– проведено аналіз існуючих засобів, які допомогають вирішити поставлену мету;

– сформульовано постановку та задачі проекту та основі функції, які повинні бути реалізовані;

– розроблено шаблони для автоматизованих тестів;

– створено фреймворк для автоматизованого тестування та інтегровано із системою управління версій Git;

– впроваджено у систему безперервної інтеграції TeamCity автоматичного процесу тестування.

У результаті виконання кваліфікаційної роботи було поглиблено теоретичні та практичні навички зі створення гнучкої інфраструктурної системи автоматичного фреймворку, були проаналізовані можливості та засоби налагодження системи управління версіями Git та системи безперервної інтеграції TeamCity.

Були проаналізовані результати виконання, які показали, що впровадження автоматизації тестування у процес розробки – ефективний варіант для економії часових та фінансових ресурсів, а використання інфраструктурних рівнів автоматизованого фреймворку дозволяє створювати гнучкий та легко підтримуваний код.

Таким чином, на основі проведеного дослідження, були зроблені висновки щодо ефективності використання сервісів безперервної інтеграції та автоматизації тестування коду. Було встановлено, що вибір конкретного сервісу або фреймворку залежить від потреб проекту, його характеристик та вимог. Належне використання цих інструментів може значно полегшити та прискорити процес розробки та тестування програмного коду.

Дослідження також показало, що важливо враховувати особливості проекту, команди розробників та їхні вимоги до інструментів. Правильний вибір сервісу безперервної інтеграції та фреймворку для автоматизації тестування може позитивно вплинути на якість та надійність розроблюваного програмного продукту.

В цілому, дане дослідження показало, що безперервна інтеграція та автоматизація тестування коду є важливими етапами розробки програмного

продукту. Правильний вибір сервісів, фреймворків та інструментів допоможе покращити якість та швидкість розробки, забезпечити стабільність продукту та зменшити кількість помилок.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. DEVOPS. (2022). International Research Journal of Modernization in Engineering Technology and Science. https://doi.org/10.56726/irjmets30906

2. Люта, М. В., Розломій, І. О., & Новикова, К. В. (2017). Аналіз методів тестування програмного забезпечення [Thesis, Міжнародний центр наукових досліджень]. erKNUTD – Електронний архів Київського національного університету технологій та дизайну. https://er.knutd.edu.ua/handle/123456789/8421

3. Gorokhovatskyi V., Tvoroshenko I., Kobylin O., and Vlasenko N. (2023) Search for visual objects by request in the form of a cluster representation for the structural image description, Advances in Electrical and Electronic Engineering, 21(1), pp. 19-27.

4. Watson, M. P. (2016). Continuous Integration with Teamcity. Createspace Independent Publishing Platform.

5. Singh, A. (2021). A Comparison on Continuous Integration and Continuous Deployment (CI/CD) on Cloud Based on Various Deployment and Testing Strategies. International Journal for Research in Applied Science and Engineering Technology, 9(VI), 4968–4977. https://doi.org/10.22214/ijraset. 2021. 36038

6. Designs, d. a. (2020). Devops Engineer Notebook. Independently Published.

7. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. IEEE Software, 33(3), 94–100. https://doi.org/10.1109/ms.2016.68

8. Timothy Fitz. (2021, 15 листопада). What is Continuous Deployment? https://www.agilealliance.org/. https://www.agilealliance.org/glossary/continuous-deployment/.

9. Dinkem, S. (2019). DevOps: The Quickstart DevOps Handbook. Independently Published.

10. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, Сучасні інформаційні системи, 6(3), С. 5- 12.

11. Дювалл, П., Матіас, С., & Гловер, А. (2007). Постійна інтеграція: покращення якості програмного забезпечення та зменшення ризику. Addison–Wesley Professional.

12. Гамбл, Дж., & Фарлі, Д. (2010). Постійна доставка: надійні випуски програмного забезпечення за допомогою автоматизації збирання, тестування та розгортання. Addison–Wesley Professional.

13. Гесс, Д., & Лайт, А. (2006). Автоматизоване тестування програмного забезпечення: вступ, управління та продуктивність. Auerbach Publications.

14. Стольберг, Т., & Басілі, В. Р. (2008). Налаштування методики вимірювання цілей та питань для тестування. У Процедурні 2008 року міжнародної симпозіуму з тестування і аналізу програмного забезпечення (ISSTA '08), ACM.

15. Діамантопулос, Т., & Спінелліс, Д. (2016). Постійна інтеграція для розробки мобільних додатків для різних платформ. IEEE Software, 33(1), 83–89.

16. Янг, Х., & Ксу, Б. (2017). Порівняльне дослідження Jenkins і Travis CI з JUnit в GitHub. У 2017 році Міжнародна конференція з якості програмного забезпечення, надійності та безпеки (QRS), IEEE.

17. Вассалло, К., & Голл, Х. С. (2018). TravisTorrent: Синтез Travis CI та GitHub для дослідження постійної інтеграції на всіх рівнях. Емпірична інженерія програмного забезпечення, 23(4), 2195–2236.

18. Babakin P.S. (2023) Distance learning in modern conditions and new technologies.

19. Шейх, А. Ф., Рост, Н., & Стірволт, Р. Е. (2016). Постійна інтеграція: емпіричне дослідження прийняття та ключові передбачувачі. У Процедурні

38–ої міжнародної конференції з інженерії програмного забезпечення (ICSE '16), ACM.

20. Хуанг, Л., Василеску, Б., & Серебренік, А. (2019). Вплив постійної інтеграції на інші практики розробки програмного забезпечення: великомасштабне емпіричне дослідження. У Процедурні 2019 року 27–ої ACM спільної зустрічі з європейською конференцією з інженерії програмного забезпечення та симпозіумом з основ інженерії програмного забезпечення (ESEC/FSE '19), ACM.

21. Джейатілака, Д. А. С., & Ліянаж, Л. А. Р. Л. (2018). Автоматизація процесу тестування програмного забезпечення за допомогою постійної інтеграції: випадковий зразок. У 2018 році Міжнародна конференція з інтелектуальних роботів та систем (IROS), IEEE.

22. Halstenberg, J., Pfitzinger, B., & Jestädt, T. (2020). DevOps. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-31405-7

23. Loukides, M. (2012). What Is DevOps? O'Reilly Media, Incorporated.

24. Beetz, F., Kammer, A., Scheungrab, S., & Harrer, S. (2021). GitOps: Cloud-native Continuous Deployment. innoQ Deutschland GmbH.

25. Press, D. E. B. (2020). Don't Panic! I'm a Professional Deployment Engineer : Customized 100 Page Lined Notebook Journal Gift for a Busy Deployment Engineer: Far Better Than a Throw Away Greeting Card. Independently Published.